FACULTAD DE INFORMÁTICA

TESINA DE LICENCIATURA

TÍTULO: Generación de expresiones algebraicas en formato LaTeX a partir del procesamiento del lenguaje natural

AUTORES: Suelgaray, Franco

DIRECTOR/A: Dr. Hasperué, Waldo

CODIRECTOR/A:

ASESOR/A PROFESIONAL:

CARRERA: Licenciatura en Sistemas

Resumen

Considerando el impacto de la inteligencia artificial en la actualidad, particularmente de los modelos neuronales de lenguaje en el ámbito educativo, resultó inminente la adopción de estas tecnologías dentro de las aulas. La falta de precisión de estos modelos de aprendizaje profundo en la resolución de tareas particulares como aquellas del área de la matemática, permite argumentar que el fácil acceso a éstas herramientas no implica necesariamente la correctitud en las respuestas producidas por ellas. A raíz de esto, en el presente trabajo se desarrolló un sistema orientado a la generación de expresiones algebraicas en formato LaTeX a partir del procesamiento de descripciones en idioma español, capaz de extraer entidades semánticas embebidas en ellas y extrapolarlas a la estructura jerárquica de las fórmulas generadas.

Palabras Clave

Tesina de grado, Aprendizaje profundo, Arquitectura Transformer, Procesamiento de Lenguaje Natural, Procesamiento secuencia a secuencia, Expresiones Algebraicas, LaTeX, Aprendizaje Multitarea, Fine-tuning

Trabajos Realizados

Como punto de partida se planteó entrenar un modelo Transformer para producir secuencias en un formato apodado pseudoLaTeX, a partir del procesamiento de textos descriptivos. Luego, con un enfoque multitarea, se incorporaron tareas auxiliares de reconocimiento de entidades nombradas y generación de árboles de operadores para apoyar la tarea principal. Para la recopilación de datos se confeccionó la web LEDProject, destinada a estudiantes de la UNLP, donde podían describir fórmulas en español. Posteriormente, los datos se usaron para entrenar el modelo final y evaluar el impacto de las tareas auxiliares en el proceso. Finalmente, el modelo se aplicó en un prototipo interactivo que genera expresiones algebraicas en LaTeX a partir de descripciones.

Conclusiones

A pesar de no lograr una comprobación de la capacidad de generalización del modelo Transformer entrenado, el análisis realizado ha podido confirmar la efectividad de la técnica de fine-tuning y el enfoque multitarea utilizados para mejorar la convergencia del error cometido durante el entrenamiento e inferencia. Esta mejora podría atribuirse al hecho que el aprendizaje simultáneo sobre las tareas auxiliares permitió capturar elementos sintácticos y semánticos de las estructuras de los lenguajes natural y matemático, para luego utilizarlos como información adicional en la resolución de la tarea final de generación de las secuencias de pseudoLaTeX.

Trabajos Futuros

Se identificó una serie de trabajos futuros que podrían extender el análisis e implementación de esta tesina, de entre los cuales se destaca, en primer lugar, la inclusión de otros elementos de la matemática de áreas como el cálculo o geometría y utilizar la página LEDProject confeccionada para recolectar los datos de entrenamiento. Por otro lado, la comparación de desempeño contra otros modelos de lenguaje permitiría identificar nuevas mejoras a implementar en modelos futuros. Finalmente se resalta la posibilidad de utilizar el modelo obtenido dentro de aplicaciones de uso educativo como herramientas de creación de exámenes para los profesionales docentes o páginas interactivas de estudio para alumnos.

Fecha de la presentación: *Septiembre de 2024*

Generación de expresiones algebraicas en formato LaTeX a partir del procesamiento del lenguaje natural

UNIVERSIDAD NACIONAL DE LA PLATA

Facultad de Informática

Alumno: Suelgaray, Franco

Director: Dr. Hasperué, Waldo

TESINA DE GRADO

Para obtener el grado de Licenciado en Sistemas

SEPTIEMBRE 2024

Índice

1 Introducción **4**

1.1 Resumen 4

1.2 Motivación 5

1.3 Objetivos 6

1.4 Organización del documento 7

2 Conceptos Preliminares **8**

2.1 Expresiones Algebraicas 8

2.1.1 Polinomios 9

2.1.2 Sistemas de Ecuaciones 10

2.2 Aprendizaje Profundo 11

2.2.1 Redes neuronales artificiales 12

2.2.2 Redes neuronales recurrentes 25

2.3 LaTeX 28

2.4 Estado del arte 30

3 Procesamiento de Lenguaje Natural **32**

3.1 Modelos de lenguaje 33

3.1.1 Modelos basados en n-gramas 34

3.1.2 Modelos neuronales de lenguaje 36

3.2 Preprocesamiento y semántica vectorial 40

3.2.1 Tokenización 40

3.2.2 Aprendizaje de características - Embeddings 43

3.3 Transformers 47

3.3.1 Modelo Encoder-Decoder 47

3.3.2 Mecanismo de atención 49

3.3.3 Arquitectura Transformer 51

3.4 Grandes modelos de lenguaje (LLMs) 59

3.4.1 Transfer learning y Fine-tuning 59

3.4.2 Modelo BERT 60

3.4.3 Modelos GPT 62

3.4.4 Modelo T5 63

3.5 Reconocimiento de entidades nombradas 65

3.6 Aprendizaje multitarea 66

3.7 Métricas de evaluación de modelos 68

4 Datos de entrenamiento **70**
4.1 Diseño conceptual de los datos 71
4.1.1 PseudoLaTeX 71
4.1.2 Entidades Nombradas 72
4.1.3 Árboles de Operadores 74
4.2 Recolección de datos - LEDProject 79
4.2.1 Generación de expresiones 80
4.2.2 Modelo físico de la base de datos 87
4.2.3 Tecnologías de desarrollo de LEDProject 88
4.2.4 Resultados de la recolección 92
4.3 Preprocesamiento de datos 92
4.3.1 Curado de datos recolectados 93
4.3.2 Afinación de árboles de operadores 94
4.3.3 Etiquetado de descripciones 94
4.3.4 Aumento de volúmen de datos 95
5 Desarrollo del modelo **97**
5.1 Hipótesis y modelos 97
5.1.1 Modelo inicial 98
5.1.2 Modelos intermedios 98
5.1.3 Modelo final 100
5.2 Implementación 101
5.2.1 Modelo de lenguaje base 101
5.2.2 Herramientas y desarrollo 102
5.3 Entrenamiento 105
5.3.1 Hiperparámetros 105
5.3.2 Hardware utilizado y tiempos de ejecución 106
6 Resultados obtenidos **108**
6.1 Comparación de modelos 108
6.1.1 Errores de entrenamiento VS errores de validación 109
6.1.2 Resultados de validación y prueba 110
6.2 Observaciones de rendimiento 112
7 Conclusiones y Trabajos futuros **117**
7.1 Conclusiones generales 117
7.2 Líneas futuras de trabajo 119
Bibliografía **121**

CAPÍTULO 1

Introducción

1.1 Resumen

Considerando el gran impacto de la inteligencia artificial en la actualidad, particularmente de los modelos neuronales de lenguaje en el ámbito educativo, resultó inminente la rápida adopción de estas tecnologías dentro de las aulas por parte de los alumnos. A pesar de presentarse formas accesibles para el uso de estos modelos basados en aprendizaje profundo, constituyen en su mayoría herramientas de propósito general como lo es el caso del chatbot ChatGPT. La falta de precisión en tareas particulares como aquellas del área de la matemática, permite argumentar que el fácil acceso a éstas herramientas y los grandes volúmenes de datos sobre los que fueron entrenadas, no implican necesariamente la correctitud en las respuestas producidas por ellas. A raíz de esto, en el presente trabajo se desarrolló un sistema orientado específicamente a la generación de polinomios y sistemas de ecuaciones en formato LaTeX a partir del procesamiento de descripciones en idioma español, capaz de extraer entidades semánticas embebidas en ellas y extrapolarlas a la estructura jerárquica de las fórmulas matemáticas generadas.

La implementación del sistema se centró principalmente en la construcción y entrenamiento de un modelo neuronal basado en la arquitectura *Transformer*, capaz de resolver tres clases de tareas: una principal de generación de expresiones algebraicas en un formato similar al LaTeX, denominado pseudoLaTeX, y dos tareas secundarias, el reconocimiento de entidades nombradas y la generación de árboles de operadores. Para la confección del conjunto de datos de entrenamiento se implementó una página web apodada LEDProject que permitió a alumnos de la universidad UNLP aportar descripciones propias en español de subconjuntos aleatorios de expresiones algebraicas. Luego, al partir del modelo de lenguaje preentrenado T5S, y aplicando las técnicas de *fine-tuning* y *aprendizaje multitarea*, se obtuvo un modelo capaz de generar satisfactoriamente polinomios y sistemas de ecuaciones que respetaran las características indicadas por medio de descripciones en español.

1.2 Motivación

A raíz del auge producido en el uso cotidiano del procesamiento de lenguaje natural a partir de herramientas de inteligencia artificial, su investigación se está desarrollando de forma más exhaustiva en el marco del aprendizaje automático. En los últimos años, el amplio incremento de la utilización de modelos preentrenados para la resolución automatizada de tareas ha cambiado sustancialmente la perspectiva que se tiene de la inteligencia artificial en la vida diaria.

En este sentido, la investigación sobre el procesamiento y generación de lenguaje natural se ha profundizado con una orientación hacia los modelos denominados Transformers (Vaswani et al., 2017). Estos representan el estado del arte en dicho contexto, ya que son los modelos neuronales que consiguen mejores resultados (Brown et al., 2020), (Radford et al., 2018), abarcando un variado abanico de tareas como la traducción de texto de un lenguaje a otro, la generación de resúmenes y síntesis de documentos, el autocompletado de código fuente, e incluso la generación de imágenes o videos desde una solicitud en formato texto. En la actualidad existen sistemas bien conocidos basados en aprendizaje profundo, como BERT (Devlin et al., 2018), T5 (Raffel et al., 2023) y GPT (Radford et al., 2018) junto a su ampliamente utilizado chatbot ChatGPT. Todos ellos presentan una arquitectura subyacente centrada en los ya mencionados Transformers, que utilizan el denominado "mecanismo de self-attention", el cual les permite procesar y aprender grandes volúmenes de datos de una forma altamente eficiente.

Los sistemas de procesamiento de lenguaje se han ido insertando de forma acelerada en la cotidianidad de los estudiante de distintos niveles educativos, principalmente con la aparición del sistema ChatGPT. A raíz de ello surge el interrogante de cuál es el nivel de confianza que se puede depositar en estos modelos informáticos (Frieder et al., 2023). Para tomar un acercamiento a esta nueva realidad en la educación, el presente trabajo de tesina plantea el desarrollo de un modelo especializado que no solo funcione como un sistema de automatización, sino como una herramienta complementaria de enseñanza para los profesionales docentes. Se busca construir un modelo de aprendizaje profundo focalizado en la rama de la ciencia matemática, el cual aportará un mayor grado de creatividad y practicidad en la producción de nuevo material de clases.

Con la introducción de los transformers como una innovadora arquitectura de redes neuronales de procesamiento secuencia a secuencia, se han conseguido resultados prometedores en lo que respecta al aprendizaje profundo aplicado a la matemática (Lample & Charton, 2019), (Charton et al., 2021), (Wang et al., 2021). Modelos especializados en el área se han creado para poner a prueba los límites que estos sistemas tienen a la hora de comprender el contenido matemático, resolver cálculos complejos y aprender los patrones subyacentes de las fórmulas que procesan.

Por otro lado, existen modelos de procesamiento de lenguaje natural con un propósito general, como GPT y su aplicación chatbot ChatGPT, que presentan una mayor facilidad de acceso por parte del público general y amplias capacidades para tareas de generación de texto. Estos sistemas han sido probados sobre datos específicos de matemática para evaluar la extensión de su entendimiento sobre esta clase de contenidos, pero se han encontrado una considerable cantidad de errores a la hora de enfrentarse al cálculo y manejo simbólico de las expresiones matemáticas (Frieder et al., 2023).

Tomando lo anterior como punto de partida, el foco de este trabajo final de grado se centra en demostrar la capacidad de los modelos de redes neuronales para hallar patrones en los elementos simbólicos de la ciencia matemática. De esta forma se busca que el modelo logre una generalización de los conceptos más allá de los adquiridos durante el entrenamiento sobre los datos experimentales. El desarrollo del modelo a construir en esta tesina supone un desafío interesante de traducción, donde una simple oración descriptiva en español debe traducirse a código que no solo represente una expresión algebráica, sino que, a su vez, lleve embebidas las características esenciales de dicha descripción. A continuación mostramos un ejemplo sencillo:

Entrada: "Polinomio de grado 3" → **Salida**: "$VNx^3 + VNx + VR$".

Por último, cabe destacar que se utilizará un formato casi idéntico al LaTeX, apodado en este contexto pseudoLaTeX, como lenguaje de salida para lograr una mayor utilidad del modelo. Éste define un conjunto de *operandos variables* representados por tokens de dos caracteres como "VN" y "VR" mencionados en el ejemplo anterior, los cuales son utilizados como marcadores de posición para un futuro reemplazo por valores numéricos específicos, como naturales o reales. La variabilidad aportada por estos operandos, sumada al amplio reconocimiento y uso del lenguaje tipográfico LaTeX por parte de la comunidad científica y educativa universitaria, es la principal motivación de su incorporación en el proceso de traducción.

1.3 Objetivos

El objetivo principal de la presente tesina consiste en la construcción de un modelo basado en técnicas de aprendizaje profundo que genere expresiones algebraicas, incluyendo polinomios y sistemas de ecuaciones, en un formato apodado pseudoLaTeX (similar al LaTeX) a partir de una descripción escrita en lenguaje natural. Se busca que dicho modelo especializado tenga capacidades de procesamiento de lenguaje natural y generación de código, adquiridas por medio del entrenamiento sobre un conjunto de datos orientado al aprendizaje multitarea. La generación de las mencionadas expresiones matemáticas se realizará a través de un prototipo de sistema web que busca simular la futura utilización por parte de profesionales docentes. Como objetivos específicos se definen:

- Estudiar en profundidad las técnicas de procesamiento de lenguaje natural, particularmente los Transformers y los componentes de su arquitectura como "Embeddings", "Positional Encodings" y "Self-Attention".

- Investigar sobre modelos neuronales previamente definidos con orientaciones específicas en el área de la ciencia matemática. A su vez profundizar en los conceptos y las arquitecturas utilizadas en los modelos de lenguaje de propósito general ya conocidos como GPT, BERT y T5.

- Confeccionar una base de datos con ejemplos para el entrenamiento del modelo neuronal con un enfoque de aprendizaje multitarea, que le permita lograr una correcta generación de expresiones algebráicas a partir del procesamiento de descripciones en español.
- Desarrollar y entrenar diferentes versiones del modelo neuronal final, junto a sus respectivos análisis y comparaciones de rendimiento. Se busca hacer hincapié en las fallas encontradas, para así dar lugar a trabajos futuros.
- Incorporar el funcionamiento del modelo óptimo construido dentro de un prototipo de sistema web que, a partir de la entrada de textos descriptivos por parte de los usuarios, provea las respuestas generadas por dicho modelo.

1.4 Organización del documento

Este documento se encuentra organizado en siete capítulos que describirán de forma progresiva los conceptos utilizados y actividades desarrolladas durante toda la extensión de la tesina de grado:

Continuando la introducción de este Capítulo 1, el Capítulo 2 abordará los conceptos preliminares que fueron investigados en una primera instancia de estudio, incluyendo las definiciones de las clases de expresiones algebraicas seleccionadas, las nociones introductorias al aprendizaje profundo y los modelos de redes neuronales, y finalmente, el uso y parte de la sintaxis del lenguaje LaTeX. Luego, a modo de obtener un mayor refinamiento de los conceptos de aprendizaje profundo para el alcance del presente trabajo, el Capítulo 3 buscará profundizarlos aportando un mayor enfoque en los elementos y técnicas utilizadas en el procesamiento de lenguaje natural y la arquitectura Transformers de redes neuronales.

Una vez introducidos los pilares teóricos de la tesina, en el Capítulo 4 se detallarán las actividades desarrolladas para la confección del conjunto de datos de entrenamiento y las etapas de análisis, recolección y preprocesamiento requeridas para ello. En particular se describirá la implementación del subproyecto LEDProject utilizado para reunir eficientemente un conjunto de volumen significativo de ejemplos de descripciones en español para expresiones algebraicas. Posteriormente, conociendo las distintas representaciones utilizadas en los datos de entrenamiento, en el Capítulo 5 se detallarán los modelos neuronales intermedios desarrollados como fases evolutivas del modelo final obtenido. Utilizando los resultados reunidos en las etapas de entrenamiento, validación y prueba de estos modelos, se mostrarán en el Capítulo 6 un conjunto de gráficos que describen el rendimiento de cada uno de ellos y las mejoras aportadas por la inclusión del aprendizaje multitarea durante el proceso de entrenamiento.

Finalmente, el Capítulo 7 expondrá las conclusiones obtenidas de las actividades desarrolladas a lo largo de la presente tesina y proveerá un listado de las posibles líneas futuras de trabajo que podrían extender la investigación e implementaciones aplicadas en el contexto de este trabajo de grado.

CAPÍTULO 2

Conceptos Preliminares

A modo de introducción teórica para el lector, este capítulo presentará los conceptos básicos fundamentales relacionados con los tres temas centrales del presente trabajo de grado. La primera sección 2.1 introducirá la noción de expresiones algebraicas como objeto de la ciencia matemática y definirá formalmente las dos clases de expresiones consideradas en el alcance de esta tesina, los polinomios y los sistemas de ecuaciones. Por otro lado, en la sección 2.2 se ahondará en los conceptos principales de la arquitectura de los modelos de redes neuronales y los detalles de su funcionamiento. Finalmente, se presentará en 2.3 al sistema de composición tipográfica LaTeX, cuyo formato será similar al objetivo de generación del modelo neuronal desarrollado en este trabajo.

2.1 Expresiones Algebraicas

Esta sección tiene por objetivo presentar el concepto de expresiones algebraicas y cómo éstas componen los elementos matemáticos centrales del presente trabajo, los *polinomios* y los *sistemas de ecuaciones*.

Una *expresión algebraica* está constituida por cualquier combinación de números y letras que se vinculan entre sí por medio de operaciones aritméticas como la suma, resta, producto, división, potenciación o radicación. Según la composición de dichas expresiones, puede proveerse una clasificación que se ilustra en la siguiente Figura 2.1

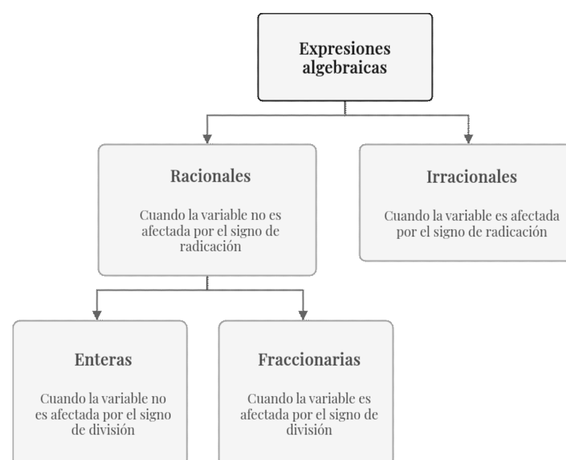

Figura 2.1: *Jerarquía de clasificación de expresiones algebráicas*

Por otro lado, es posible combinar expresiones algebraicas con el fin de obtener otras más complejas, añadiendo operaciones entre ellas o incluso relaciones como la igualdad. Las denominadas *ecuaciones algebraicas* representan igualdades entre expresiones algebraicas donde uno o múltiples de los elementos que las componen, denominados variables, son incógnitas que deben ser halladas para verificar dicha relación de igualdad.

En esta primera sección del capítulo se presentará inicialmente una de las clases de expresiones más utilizadas en la ciencia matemática, conocidas como *polinomios*. Luego, a partir de ella y tomando la noción de ecuación, se introducirá el concepto de *sistema de ecuaciones*, que son utilizados desde problemas de optimización hasta la resolución de ecuaciones diferenciales que explican fenómenos físicos.

2.1.1 Polinomios

Un *polinomio en una indeterminada* x es una expresión de la forma:

$$P(x) = a_n x^n + a_{n-1} x^{n-1} + \cdots + a_2 x^2 + a_1 x + a_0$$

para $n \in \mathbb{N}$ y $a_1, a_2, \ldots, a_n$ elementos de un cuerpo conmutativo K (como los reales $\mathbb{R}$ o los complejos $\mathbb{C}$), denominados *coeficientes* del polinomio (Ferre et al., 2018). En este trabajo se tratará con los polinomios de $\mathbb{C}[x]$ que denota el conjunto de todos los polinomios en la indeterminada x con coeficientes complejos.

Se debe notar que los exponentes de la indeterminada en los distintos términos del polinomio son números estrictamente naturales, por lo que aquellas expresiones donde la indeterminada se encuentre elevada a una potencia negativa o fraccionaria no será considerado como un polinomio.

Dos de los elementos que caracterizan a un polinomio son su *grado* y su *coeficiente principal*. El grado de un polinomio $P(x)$ es la mayor potencia a la cual se encuentra elevada la indeterminada x tal que está multiplicada por un escalar no nulo a_i del cuerpo, quien recibe el nombre de coeficiente principal. Luego, si se presenta el siguiente ejemplo:

$$Q(x) = \sqrt{3}x^4 + 8x - x^5$$

Es posible identificar que el grado de $Q(x)$ es 5 y el coeficiente principal es -1. A raíz de este ejemplo es posible destacar que un polinomio puede encontrarse ordenado, por sus potencias en forma creciente o decreciente, o *desordenado*. A su vez se pueden hallar polinomios a los que se denominan *incompletos*, caracterizados por no incluir todos los términos con la indeterminada elevada a cada potencia desde 0 hasta n, sino que algunos de sus términos se omiten ya que se interpreta que los coeficientes en dichos términos son nulos.

Todo polinomio de grado n necesariamente debe incluir en su definición al término $a_n x^n$ donde $a_n \neq 0$, caso contrario, no podría interpretarse a n como el grado del polinomio.

Finalmente, como se trabajará sobre polinomios con coeficientes complejos, resulta imprescindible recordar que los números reales se encuentran incluidos en el cuerpo de los números complejos $\mathbb{C}$, y por ser subconjuntos de ellos, también los racionales o fraccionarios, irracionales, enteros y naturales.

2.1.2 Sistemas de Ecuaciones

Debido a su aplicación en el modelado matemático de sistemas de la realidad para describir fenómenos físicos o comportamientos biológicos, o para definir relaciones de igualdad en Geometría Analítica, los *sistemas de ecuaciones* son extensivamente utilizados en distintas ramas de la matemática.

Un sistema de ecuaciones está definido por una cierta cantidad de ecuaciones que relacionan un conjunto de variables. De esta forma, un sistema de *m* ecuaciones con *n* incógnitas puede tener, por ejemplo, de la siguiente forma:

$$\begin{cases} a_{11}x_1 + a_{12}x_2 + \cdots + a_{1j}x_j + \cdots + a_{1n}x_n = b_1 \\ a_{21}x_1 + a_{22}x_2 + \cdots + a_{2j}x_j + \cdots + a_{2n}x_n = b_2 \\ \quad\vdots \qquad \vdots \qquad \vdots \qquad \vdots \qquad \vdots \\ a_{m1}x_1 + a_{m2}x_2 + \cdots + a_{mj}x_j + \cdots + a_{mn}x_n = b_m \end{cases}$$

donde $x_1, x_2, \ldots, x_n$ son las *incógnitas*. Particularmente el ejemplo anterior es la forma de un sistema de ecuaciones *lineales* (sobre los que haremos enfoque en este trabajo) donde, $a_{11}, a_{12}, \ldots, a_{mn}$ reciben el nombre de *coeficientes* del sistema y $b_1, b_2, \ldots, b_m$ son los *términos independientes*. Tanto los coeficientes como los términos independientes son elementos de un cuerpo conmutativo K.

Cada sistema de ecuaciones puede contar con una, ninguna o múltiples soluciones, que consisten de un conjunto de valores numéricos que satisfacen las ecuaciones de forma simultánea (Ferre et al., 2018). Por ejemplo los siguientes casos son sistemas de ecuaciones, con infinitas, una y ninguna solución respectivamente:

$$\begin{cases} x + y = 1 \\ 2x + 2y = 2 \end{cases} \qquad \begin{cases} y - 5w = 20 \\ -5w + y = 8 \end{cases} \qquad \begin{cases} 3x + y = 1 \\ x - 2y = 2 \end{cases}$$

Es posible destacar algunos sistemas de ecuaciones particulares como los homogéneos que se caracterizan por tener todos sus términos independientes iguales a 0. A su vez, los sistemas pueden ser no lineales al introducir funciones que afecten a las variables como la potenciación.

2.2 Aprendizaje Profundo

Hoy en día la *inteligencia artificial (IA)* es un campo de estudio altamente activo, con múltiples aplicaciones posibles y resultados de investigación satisfactorios. Particularmente, un subconjunto de esta rama recibe la mayor atención, el *aprendizaje automático o machine learning* (ML) cuyo principal objetivo se centra en resolver aquellas actividades que son simples desde la perspectiva de una persona, pero complejas para describirlas formalmente. Entre algunos ejemplos es posible destacar la interpretación de una imágen, la producción de texto o incluso el entendimiento de una oración. Si bien puede resultar intuitivo y evidente distinguir un árbol en una foto de un bosque o interpretar que la oración “Polinomio genérico de grado dos” se puede escribir de la forma $a_2x^2 + a_1x + a_0$, no resulta sencillo hallar una especificación de reglas lógicas o matemáticas que describan la resolución de estas tareas de forma genérica por parte de una computadora.

> Machine learning is a subfield of computer science wherein *machines learn* to perform tasks for which they were *not explicitly programmed*. In short, machines observe a pattern and attempt to imitate it in some way that can be either direct or indirect. [El aprendizaje automático es un subcampo de la ciencia de la computación en el cual las *máquinas aprenden a* realizar tareas para las que no son explícitamente programadas. En resumen, las máquinas observan un patrón e intentan imitarlo de alguna forma que puede ser tanto directa como indirecta.] (Trask, 2019, p. 11)

Cuando dicha “imitación” es directa, es decir, donde se le presenta a la máquina el patrón de asociación que debe aprender, y sobre todo generalizar, se habla del llamado *aprendizaje supervisado*. En esta clase de aprendizaje, se construye un algoritmo que toma de entrada un conjunto de datos iniciales junto a los valores o etiquetas deseados y obtiene como salida una función. Luego del entrenamiento, dada una instancia de los datos de prueba (un ejemplo que no se ha proporcionado previamente) dicha función debe producir la etiqueta deseada. De esta manera se espera que la función resultante represente una regla de asociación generalizada de forma que sea capaz de predecir correctamente etiquetas o valores para datos que no fueron utilizados durante el entrenamiento. (Goldberg, 2017)

Un aspecto importante del aprendizaje automático es que el conjunto de algoritmos que utiliza dependen de una correcta *representación* de los datos a la hora de servirlos como entrada. Por ejemplo en la traducción de una oración de un idioma a otro, el algoritmo podría necesitar como entrada a las palabras de la oración en formato de lista, donde cada una tiene un orden particular. Incluso podría ser de ayuda incluir porciones de cada palabra como sus raíces o sufijos. Proporcionar los atributos correctos que describen los datos de entrada y que aportan información del problema a solucionar puede resultar crítico a la hora de conseguir resultados precisos. El problema es que no siempre es posible determinar qué características son relevantes de aquello que se está observando de forma que se garantice resolver la tarea con alto nivel de exactitud. Una solución a este problema es utilizar el mismo aprendizaje automático tanto para hallar las relaciones de mapeo entre la representación de entrada y la salida

correspondiente, como también para encontrar dicha representación inicial (Goodfellow et al., 2016). A este acercamiento se lo denomina *aprendizaje de atributos*, o también apodado *feature learning*.

El enfoque que se realizará a lo largo de este trabajo será en una particular subrama del Machine Learning, el *aprendizaje profundo* o más conocido como *deep learning*, que soluciona el problema del feature learning mediante el uso de las *redes neuronales artificiales*, permitiendo construir conceptos complejos con múltiples factores de variación a partir de la unión de otros conceptos más simples.

Con el objetivo de comprender el funcionamiento de estos modelos de aprendizaje profundo, en la siguiente sección 2.2.1 se introducirán los principales conceptos y arquitecturas de las redes neuronales artificiales, y donde se brindará una formalización matemática de estos modelos. Finalmente, con lo que se presume un entendimiento básico, se presentará en la sección 2.2.2 una modificación de la arquitectura de redes neuronales de propagación hacia adelante (o *feedforward*) para obtener modelos más complejos como las redes neuronales recurrentes.

2.2.1 Redes neuronales artificiales

Tomando la analogía de la neurociencia con la que fueron concebidas, una *red neuronal artificial* (RNA) o *artificial neural network* (ANN) puede interpretarse como una abstracción de una red de neuronas del sistema nervioso humano, pensando a la neurona como la célula funcional base de dicha red. La neurona artificial, toma como referencia el funcionamiento característico de dichas células que reciben señales de entrada a través de las sinapsis de sus dendritas, las procesan en su núcleo y determinan si disparar o no una corriente eléctrica a través de su axón, dependiendo de si la corriente recibida es superior a un cierto umbral (Hernández Orallo, 2004). En términos más formales, una neurona artificial toma *datos de entrada* $x_1, x_2, \ldots, x_n$, cada uno de los cuales es multiplicado por un valor $w_1, w_2, \ldots, w_n$ llamados *pesos* que determinan la importancia de los respectivos datos de entrada a la salida de la neurona. Tantos los datos como los pesos son procesados en conjunto computando la suma pesada $\sum_i x_i.w_i$ cuyo resultado finalmente sirve de entrada a una función de activación f que lo transforma en la salida final de la neurona artificial.

Reescribiendo los parámetros de entrada y los pesos en términos de vectores $x = (x_1, \ldots, x_n)$ y $w = (w_1, \ldots, w_n)$ respectivamente, se puede expresar el funcionamiento y salida de la neurona en la siguiente línea:

$$y = f\left(\sum_{i=0}^{n} w_i x_i\right) = f(w \cdot x) = f(w^\top x)$$

Cabe destacar que frecuentemente se añade a la neurona una entrada con valor fijo -1 y un peso variable θ denominado umbral que se resta a la suma pesada del resto de datos entrantes, alterando el argumento de la función de activación de la siguiente forma:

$$y = f\left(\sum_{i=0}^{n} w_i x_i - \theta\right) = f((w \cdot x) - \theta) = f((w^\top x) - \theta)$$

Es necesario resaltar que la capacidad de aprendizaje de esta clase de modelos radica en la modificación gradual de los parámetros internos de las neuronas. Dado que la función de activación es completamente determinística, la única posibilidad de cambiar su resultado final es mediante la modificación de los pesos y el umbral θ. Luego, en el contexto del aprendizaje supervisado, el "entrenador" proporciona un conjunto de datos de entrada y las respuestas correctas esperadas, siendo posible comparar los resultados sucesivos de la red neuronal con los valores deseados. De esta forma, ante una discrepancia entre el resultado real y el esperado se aplica una modificación a los pesos intentando corregir el comportamiento de las neuronas y disminuyendo el error de inferencia.

Se introducirá a continuación un caso de este mecanismo de aprendizaje utilizado por el primer modelo neuronal conocido como Perceptrón.

Perceptrón Simple

Es posible pensar al Perceptrón como el bloque fundamental de las redes neuronales más complejas, debido a que todas ellas se construyen combinando una serie de perceptrones en disposiciones diversas que dan lugar a comportamientos más complejos.

De la misma manera en que se definieron previamente las neuronas artificiales, el Perceptrón simple toma un vector de datos de entradas $x = (-1, x_1, \ldots, x_n)$ y un vector de pesos $w = (\theta, w_1, \ldots, w_n)$, en los cuales se incluyen una entrada fija $x_0 = -1$ y el valor θ como su respectivo peso. Finalmente, la función de activación utilizada es la función escalón, definida de la forma:

$$H\left(\sum w_i x_i - \theta\right) = \begin{cases} 0 & \text{si } \sum w_i x_i - \theta \leq 0 \\ 1 & \text{si } \sum w_i x_i - \theta > 0 \end{cases} = \begin{cases} 0 & \text{si } \sum w_i x_i \leq \theta \\ 1 & \text{si } \sum w_i x_i > \theta \end{cases}$$

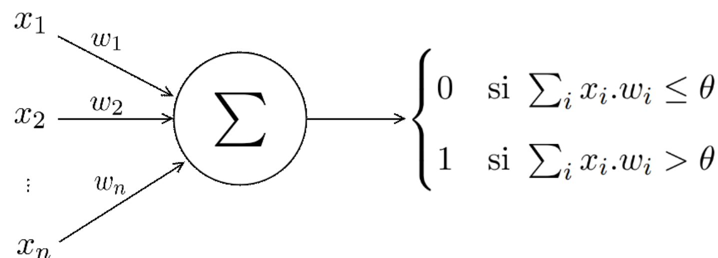

Figura 2.2: *Estructura del Perceptrón simple con función de activación escalón*

A continuación se presenta un ejemplo del uso de un perceptrón para el cálculo de la operación lógica AND: Dado un vector de datos de entrada con tres componentes $x = (-1, x_1, x_2)$, donde $x_1, x_2 \in 0, 1$ y el vector de pesos $w = (1.5, 1, 1)$ (notar que se incorpora $\theta = 1.5$), es posible obtener una neurona artificial que se comporte de la siguiente forma:

- **Caso 1:** $x_1 = 0,\ x_2 = 0 \quad \longrightarrow \quad w \cdot x = 0 < 1.5 \quad \longrightarrow \quad \text{Salida} = 0$
- **Caso 2:** $x_1 = 0,\ x_2 = 1 \quad \longrightarrow \quad w \cdot x = 1 < 1.5 \quad \longrightarrow \quad \text{Salida} = 0$

- **Caso 3:** $x_1 = 1,\ x_2 = 0 \quad \longrightarrow \quad w \cdot x = 1 < 1.5 \quad \longrightarrow \quad \text{Salida} = 0$
- **Caso 4:** $x_1 = 1,\ x_2 = 1 \quad \longrightarrow \quad w \cdot x = 2 > 1.5 \quad \longrightarrow \quad \text{Salida} = 1$

Resulta evidente que, dadas todas las combinaciones posibles de entradas, el perceptrón se comporta como una compuerta lógica AND. Luego, tomando este resultado, es posible modificar levemente la definición de la función escalón de este caso para denotar una igualdad en lugar de una desigualdad. De esta manera, se puede obtener una función lineal discriminante que separa gráficamente los casos donde la salida de la neurona es 0 y aquellos donde es 1.

Tomando $x \cdot w = x_1.w_1 + x_2.w_2 = x_1 + x_2 = \theta$, se interpreta gráficamente como la recta que separa los datos de entrada, como se muestra en la Figura 2.3. Aquí es posible observar que la recta divide al plano cartesiano en dos partes, separando aquellos puntos de entrada que dieron como resultado 0 y aquellos que resultaron en 1. Esta función lineal se conoce como función discriminante ya que separa o discrimina los datos según la clase a la que pertenecen (en el presente caso 0 o 1).

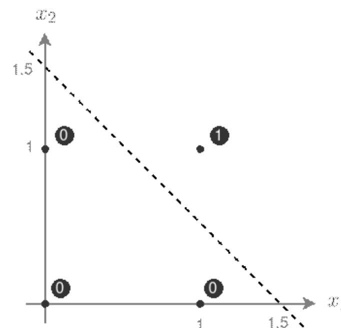

***Figura 2.3**: Función discriminante lineal que separa entradas de una compuerta lógica AND*

Si bien en el caso presentado anteriormente los pesos de cada entrada a la neurona fueron elegidos de forma tal de obtener la salida exacta que imita el comportamiento de la compuerta lógica AND, esto no es posible generalizarlo a todas las redes neuronales. No siempre que se construye un modelo basado en redes neuronales es posible elegir manualmente los pesos de cada entrada. Esto es debido, en primer lugar, al gran número de neuronas y entradas que puede alcanzar un modelo de aprendizaje de esta clase, y segundo, por la complejidad inherente del problema a resolver. Es por ello que se introduce la idea de *entrenamiento* de la red, con el fin de que ésta *aprenda los* pesos correctos de las neuronas sin intervención del desarrollador.

El Perceptrón simple, en particular, puede ser entrenado al actualizar sus parámetros internos como se muestra en el Algoritmo 1. De esta forma, seleccionando un valor para el coeficiente de aprendizaje η, es posible modificar los pesos de la neurona en las distintas iteraciones del algoritmo hasta que se obtenga el valor de salida esperado para todos los datos de entrada.

Algoritmo 1: Entrenamiento de un Perceptrón

Datos

- Coeficiente de aprendizaje η. Cumple $0 < \eta \leq 1$
- Vector de pesos w con coordenadas w_j con $j \in \{0, 1, \ldots, m\}$
- Vectores de z_i $i \in \{0, 1, \ldots, n\}$ entrada con x_j y $j \in \{0, 1, \ldots, m\}$ coordenadas t_i con
- Valores esperados con $i \in \{0, 1, \ldots, n\}$

Inicializar el coeficiente de aprendizaje η con un valor deseado.

Inicializar los pesos de las w_j conexiones con valores random.

while (no se clasifiquen correctamente todos los ejemplos)

  foreach (vector de entrada z_i)

    $y_i \leftarrow$ Resultado del Perceptrón con entrada z_i

    if ($y_i \neq t_i$)

      foreach (coordenada w_j del vector de pesos)

        $w_j = w_j + \eta(t_i - y_i)x_j$

      end

    end

  end

end

Multiperceptrón

Una gran limitación de los perceptrones es que no son capaces de realizar tareas de separación no lineal, como puede ser la clasificación de imágenes o resolver una simple función XOR. Esto condujo a la introducción del Multiperceptrón que combina el comportamiento de múltiples perceptrones (o incluso otras clases de neuronas como la sigmoidea) en varias capas para resolver problemas más complejos.

El Multiperceptrón, también conocido como red neuronal multicapa, posee una primera fila de neuronas llamada *capa de entrada* que recibe secuencialmente los datos de cada ejemplo que ingresa a la red. Luego, cada neurona de la capa de entrada se conecta a todas las neuronas de una capa intermedia, conocida como *capa oculta*. Las capas ocultas pueden ser múltiples en toda la red, haciéndola *profunda* (por ello recibe el nombre *aprendizaje profundo*), y las neuronas de cada una de ellas se conectan a todas las de la capa oculta siguiente. Finalmente, la red combina los resultados de la última capa oculta en una *capa de salida* que computa los resultados finales de la red (Nielsen, 2015).

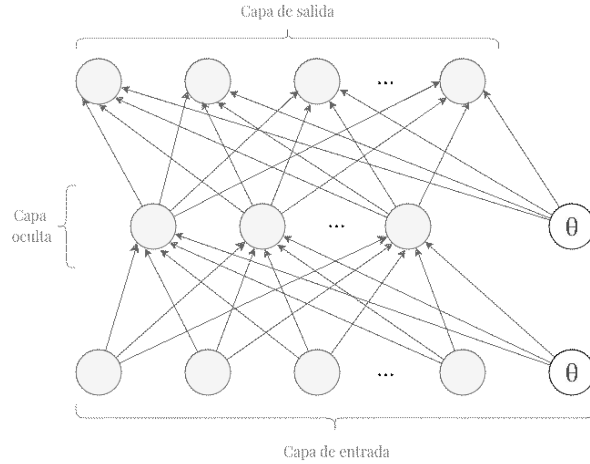

Figura 2.4: *Arquitectura general del multiperceptron con una capa oculta*

Para presentar una formalización matemática de un multiperceptrón se debe notar que cada capa se encuentra formada por numerosas neuronas, las cuales cuentan, a su vez, con múltiples conexiones de salida a las neuronas de la siguiente capa. Luego, en lugar de contar con vectores de pesos, es necesario definir matrices $W^l \in \mathbb{R}^{d_l \times d_{(l-1)}}$ donde l indica que contiene los pesos de las conexiones de la capa $l-1$ a la l, y $d_{(l-1)}$ y d_l representan el número de neuronas de la capa $l-1$ y l, respectivamente. Esta notación fue extraída del libro de (Nielsen, 2015), quien identifica a lo elementos de estas matrices como , w^l_{jk} esto es, el peso de la conexión de la k-ésima neurona en la capa $l-1$ con la j-ésima neurona de la capa l.

Utilizando ahora estas nuevas matrices W^l, y recordando la suma pesada de cada neurona y su respectiva función de activación σ_l, podemos expresar la salida de la j-ésima neurona de la capa l de la siguiente forma:

$$a^l_j = \sigma_l \left(\sum_{k=1}^{d_{(l-1)}} w^l_{jk} a^{l-1}_k + b^l_j \right)$$

Los vectores b^l son llamados *términos bias que* representan el opuesto del umbral previamente mencionado en la introducción de la neurona artificial. Los elementos de estos vectores se denotan como b^l_j, que indica el valor de bias de la j-ésima neurona de la capa l.

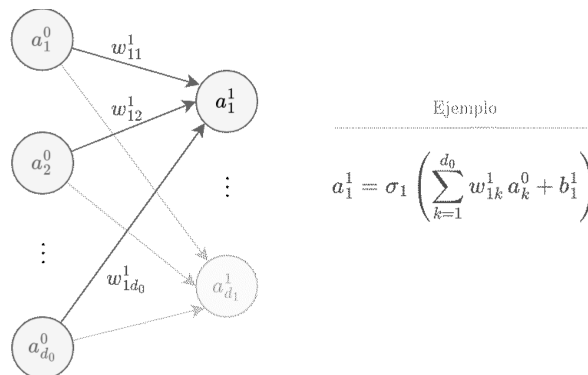

Figura 2.5: *Ejemplo de cálculo de valor de activación de la primera neurona de la capa 1.*

Finalmente, es posible expresar la salida de toda una capa de la red utilizando el producto entre la matriz de pesos y el vector de salidas de la capa anterior (Nielsen, 2015):

$$a^l = \sigma_l \left(w^l a^{l-1} + b^l \right)$$

La arquitectura de multiperceptrón no solamente permitió evadir la limitación del perceptrón y combinador lineal de resolución de problemas linealmente no separables, sino que fue demostrado que un multiperceptrón con una sola capa oculta (o más) es un aproximador universal de cualquier familia de funciones continuas en un subconjunto cerrado de $\mathbb{R}^n$ (Goldberg, 2017). En otras palabras, es posible construir una multiperceptrón que imite el comportamiento de cualquier situación, lo cual no implica, en lo absoluto, que sea fácil determinar los parámetros de la red para solucionarlo, como los pesos de las conexiones, la cantidad de capas ocultas y las neuronas que debería incluir cada una de ellas.

Funciones de activación

Hasta el momento solo se ha mencionado la función escalón como activación de las neuronas, utilizada por el Perceptrón simple, pero a lo largo del estudio y desarrollo de las redes neuronales artificiales se introdujo el uso de otras clases de funciones que tienen el aspecto en común de ser diferenciables. Para algunas de ellas resulta menos complejo hallar su derivada, la cual juega un rol importante durante el proceso de entrenamiento.

La elección correcta de la función de activación depende mayormente de las pruebas empíricas que se realicen sobre el problema en cuestión. No existe actualmente una buena forma de determinar la función no lineal de activación precisa que resuelva una situación particular (Goldberg, 2017). Entre las funciones no lineales de activación más utilizadas es posible resaltar:

- Sigmoidea: La función sigmoidea fue originalmente una de las más utilizadas en los inicios del desarrollo de redes neuronales artificiales, pero actualmente no es incluida en las capas ocultas ya que las siguientes funciones probaron ser empíricamente mejores (Goldberg, 2017). Esta función está definida de la siguiente forma:

  $$\sigma(x) = \frac{1}{1 + e^{-x}} \qquad \sigma : \mathbb{R} \longrightarrow (0, 1)$$

  Su forma puede observarse en la Figura 2.6.

- Tangente Hiperbólica: La función tangente hiperbólica tiene una forma de S y está definida como:

  $$tanh(x) = \frac{e^{2x} - 1}{e^{2x} + 1} \qquad tanh : \mathbb{R} \longrightarrow (-1, 1)$$

  Puede verse su gráfico en la Figura 2.7.

- <u>Lineal a Trozos:</u> Esta función es similar a la Tangente Hiperbólica y es conocida en inglés como *"Hard" Hyperbolic Tangent* ya que no es tan suave como la anterior. Es una función definida a trozos de la forma:

$$hardtanh(x) = \begin{cases} -1 & x < -1 \\ 1 & x > 1 \\ x & \text{caso contrario} \end{cases} \qquad hardtanh : \mathbb{R} \longrightarrow [-1, 1]$$

  Su forma puede observarse en la Figura 2.8 .

- <u>Función Rectificadora ReLU:</u> Esta función presentada por (Glorot et al., 2011), comúnmente conocida como función ReLU (*Rectifier Linear Unit*), está definida de la siguiente forma:

$$ReLU(x) = \begin{cases} 0 & x < 0 \\ x & \text{caso contrario} \end{cases} \qquad ReLU : \mathbb{R} \longrightarrow \mathbb{R}^+$$

  Esta función es actualmente la más utilizada debido al bajo cómputo necesario para su cálculo. El gráfico de la función se muestra en la Figura 2.9.

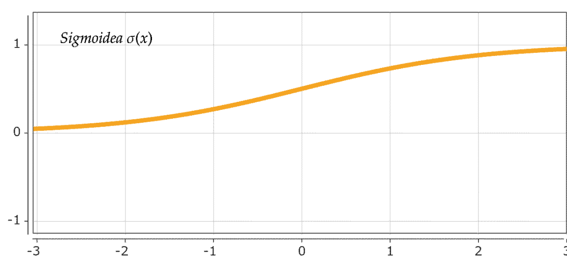

Figura 2.6: *Función Sigmoidea*

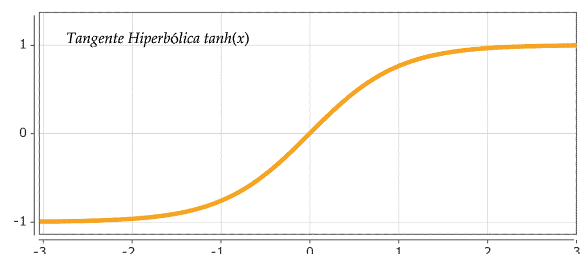

Figura 2.7: *Función Tangente Hiperbólica*

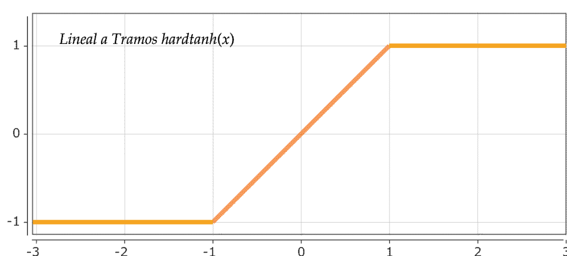

Figura 2.8: *Función lineal a trozos*

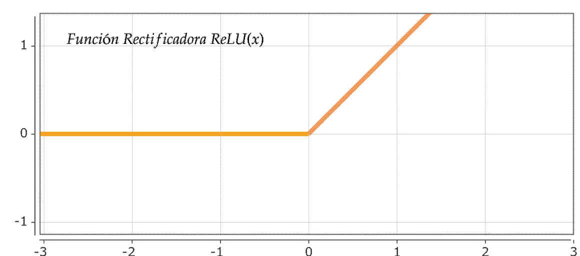

Figura 2.9: *Función Rectificadora ReLU*

Funciones de error

Con una noción de la arquitectura general de una red neuronal artificial, como la del multiperceptrón, y el funcionamiento interno de las neuronas de cada capa, es posible introducir las funciones de error con las que estos modelos de aprendizaje profundo identifican si los resultados que generan son correctos y la magnitud del error que cometen.

En el caso del perceptrón no se contaba con ninguna función de error sofisticada para propiciar el entrenamiento, sino que simplemente se realizaba la comparación entre el resultado final de la red y el valor deseado.

En general, el objetivo del entrenamiento es minimizar una función de error teniendo en cuenta el conjunto de ejemplos que se presentan de entrada al modelo. Dicha función provee una medida de la magnitud del error cometido entre el resultado final de la red y el valor deseado. Formalmente, una función de error $L(\hat{y}, y)$ asigna un valor numérico a una salida predicha $\hat{y}$ dado que el valor esperado verdadero es y , que debe estar inferiormente acotada y cuyo mínimo sólo puede obtenerse cuando la predicción es correcta (Goldberg, 2017). Dado que el valor predicho $\hat{y}$ se calcula en función de los parámetros internos (pesos y bias), la propia función de error es una función de múltiples variables, las cuales deben establecerse de forma que se minimice el error cometido L a lo largo del entrenamiento de modelo.

Usualmente la función de costo o error sobre todo el conjunto de ejemplos se calcula como el promedio del error de cada una de las instancias de ejemplo presentadas. Dado un conjunto de $\{(x_1, y_1), \ldots, (x_n, y_n)\}$ entrenamiento, una función de error L por instancia, y la función $f(x, \Theta)$ paramétrica que determina la salida de la red, se puede definir a la función de costo o error total como:

$$\mathcal{L}(\Theta) = \frac{1}{n} \sum_{i=1}^{n} L(f(x_i, \Theta), y_i) \tag{2.1}$$

Luego, el objetivo de entrenamiento es definir los parámetros Θ de forma tal que se minimice la ecuación 2.1 anterior:

$$\hat{\Theta} = \underset{\Theta}{argmin}\ \mathcal{L}(\Theta) = \underset{\Theta}{argmin}\ \frac{1}{n} \sum_{i=1}^{n} L(f(x_i, \Theta), y_i) \tag{2.2}$$

Entre las funciones de error L más utilizadas se pueden mencionar a las siguientes:

- Entropía Cruzada Binaria: Utilizada para resolver problemas de clasificación binaria (de dos clases). Tiene la forma:

$$L_{ECB}(\hat{y}, y) = -y\, log(\hat{y}) - (1 - y) log(1 - \hat{y})$$

- Entropía Cruzada Binaria: Esta función de error se utiliza para la clasificación multiclase de datos. Dado un ejemplo de entrenamiento x , se define el vector de distribución multinomial

$y = (y_1, ..., y_n)$ donde y_i indica la probabilidad de $P(y = i|x)$ que el dato sea de la clase i dado que la entrada sea x simbólicamente $\hat{y} = (\hat{y}_1, ..., \hat{y}_n)$

Luego, el modelo de clasificación retorna como resultado una predicción de la distribución y sirve de entrada a la función de entropía cruzada definida de la forma:

$$L_{ECC}(\hat{y}, y) = -\sum_i y_i \, log(\hat{y}_i)$$

Entrenamiento de redes neuronales

Conociendo ya cómo se propaga la información a través de la RNA y cómo se computa la función de costo $\mathcal{L}$, en este apartado se introducirá al funcionamiento de los modelos de aprendizaje profundo presentando el mecanismo general de entrenamiento *backpropagation*. No se debe perder el foco del objetivo principal que es minimizar la mencionada función de costo modificando progresivamente los parámetros de la red, como muestra la ecuación 2.2.

La estrategia de optimización por excelencia que configura las bases de todo entrenamiento de redes neuronales artificiales es el Descenso del Gradiente. Este método numérico permite hallar los extremos globales (y en ciertos casos extremos locales) de funciones de múltiples variables a partir de la implementación de un algoritmo iterativo que utiliza el gradiente de la función. En particular, una variante del método del gradiente utilizada en aprendizaje profundo es el *descenso del gradiente estocástico* que será introducido más adelante en este apartado (Goldberg, 2017).

DESCENSO DEL GRADIENTE: Dada una función f de n variables, y su vector gradiente ∇f formado por las derivadas parciales de f con respecto a cada una de las variables, es posible fijar un punto inicial x_0 en un espacio n-dimensional y modificar iterativamente sus coordenadas utilizando un escalado del gradiente de la función evaluado en dicho punto, de forma que en cada iteración i el punto x_i se encuentre más cerca del mínimo de la función. Este comportamiento es posible debido a que se respeta el siguiente teorema:

TEOREMA: *Dada una función diferenciable f de n variables, su vector gradiente ∇f y un punto $x \in \mathbb{R}^n$, si $\nabla f(x) \neq 0$ entonces el vector $\nabla f(x)$ indica la dirección de máximo crecimiento de la función en el punto x. Análogamente $-\nabla f(x)$ determina la dirección de mínimo crecimiento*

(Marsden & Tromba, 2012)

Luego, dada una aproximación del punto mínimo x_i en la iteración i del algoritmo y una *tasa de aprendizaje* η, si $\nabla f(x_i) \neq 0$, se define el siguiente punto de la iteración de la forma

$$x_{i+1} = x_i - \eta \nabla f(x_i)$$

El hiperparámetro η del modelo, definido por el usuario, es un valor generalmente pequeño entre 0 y 1 que escala el vector gradiente de forma que la actualización de los puntos de cada iteración se efectúe de forma progresiva, evitando que las sucesivas aproximaciones "reboten" alrededor del extremo mínimo de la función, divergiendo hacia valores lejanos.

En términos del aprendizaje profundo y en el contexto de entrenamiento de los modelos, la función de múltiples variables que se considera es la función de costo $\mathcal{L}(\Theta)$ y su vector gradiente, cuyas coordenadas son las derivadas parciales de $\mathcal{L}$ con respecto a cada parámetro de Θ.

Profundizando más en las variantes de las aplicaciones del descenso del gradiente, LeCun et al., 1998 menciona que los algoritmos de aprendizaje basados en el gradiente pueden utilizar una de dos clases de métodos de actualización de parámetros: El primero, apodado *Gradiente por lotes* (*Batch Gradient*), es el método clásico donde los gradientes se acumulan sobre todo el conjunto de entrenamiento y los parámetros son actualizados luego del cómputo exacto del gradiente. En el segundo método, llamado *Gradiente Estocástico*, se evalúa un gradiente parcial tomando solo una instancia de entrenamiento (o un subconjunto pequeño) y los parámetros se actualizan utilizando un gradiente aproximado.

> An empirical result of considerable practical importance is that on tasks with large redundant data sets the stochastic version is considerably faster than the batch version sometimes by orders of magnitude. [Un resultado empírico de considerable importancia práctica es que en tareas con grandes conjuntos de datos redundantes la versión estocástica es considerablemente más rápida que la versión por lotes, a veces por órdenes de magnitud] (LeCun et al., 1998, p. 41)

En el caso del descenso del gradiente estocástico, al utilizar una única instancia de entrenamiento para actualizar los parámetros Θ, el gradiente calculado es sobre la función de error L en lugar de $\mathcal{L}$. De esta forma se puede definir en términos simples el algoritmo SGD (Stochastic Gradient Descent) como se muestra a continuación (Goldberg, 2017):

Algoritmo 2: Descenso del gradiente estocástico

Datos

- Ratio de aprendizaje η
- Conjunto de entrenamiento con entradas $x_1, \ldots, x_n$ y salidas deseadas $y_1, \ldots, y_n$
- Función de error por instancia L
- Función parametrizada de la RNA $f(x; \Theta)$

while (no se cumpla la condición de corte)

Seleccionar un ejemplo de entrenamiento x_i, y_i

Calcular el error $L(f(x_i; \Theta), y_i)$

$\hat{g} \leftarrow$ gradiente de $L(f(x_i; \Theta), y_i)$ con respecto a Θ

$\Theta \leftarrow \Theta - \eta\, \hat{g}$

end

Considerar solo un ejemplo en el cálculo del gradiente implica una actualización imprecisa de los parámetros de la red y por lo tanto una estimación pobre del mínimo de la función $\mathcal{L}$. Es por ello que existe otra variante del algoritmo SGD llamado *minibatch* ("mini lote") que, en lugar de tomar una instancia de entrada, toma un subconjunto de m ejemplos de entrenamiento, obteniendo una aproximación más precisa del gradiente. Cuanto mayor sea el tamaño m del lote mayor precisión se conseguirá pero a expensas de menos actualizaciones que toman mayor tiempo de cómputo.

Tomando en cuenta este método de optimización es momento de introducir el mecanismo de entrenamiento utilizado por los modelos de aprendizaje profundo, la *propagación hacia atrás*. Cuando se utiliza una red neuronal de propagación hacia adelante como un multiperceptrón que recibe una entrada x y produce una salida $\hat{y}$, la información fluye hacia adelante a través de la red, computándose sumas pesadas y funciones de activación que continúan hasta producir un escalar final de error $\mathcal{L}(\Theta)$ .El algoritmo de propagación hacia atrás (Rumelhart et al., 1986), o más bien conocido como *backpropagation*, permite que la información de la función de error fluya hacia atrás a través de la red, con el fin de calcular el gradiente.

> Back-propagation is an algorithm that computes the chain rule, with a specific order of operations that is highly efficient. [Back-propagation es un algoritmo que calcula la regla de la cadena, con un orden específico de operaciones que es altamente eficiente.] (Goodfellow et al., 2016)

El concepto matemático fundamental detrás del algoritmo de Back-propagation es la propiedad de la *regla de la cadena* en la derivación de funciones (de una o múltiples variables):

TEOREMA: Sean $U \in \mathbb{R}^n$ y $V \in \mathbb{R}^m$ conjuntos abiertos. Dadas las funciones $g : U \subset \mathbb{R}^n \to \mathbb{R}^m$ y $f : V \subset \mathbb{R}^m \to \mathbb{R}^p$ donde g mapea valores de U a V, se define la composición $f \circ g = f(g(x))$. Suponiendo que g es diferenc- en un punto x_0, y f es diferenciable en $y_0 = g(x_0)$ entonces $f \circ g$ también es diferenciable en x_0 y se cumple

$$\mathbf{D}(f \circ g)(x_0) = \mathbf{D}f(y_0)\mathbf{D}g(x_0)$$

donde $\mathbf{D}f$ indica la matriz Jacobiana de f y $\mathbf{D}g$ el vector gradiente de g .

(Marsden & Tromba, 2012)

Luego, utilizando la regla de la cadena es posible calcular la derivada parcial de la función de costo $\mathcal{L}$ con respecto a cualquiera de los pesos w_{jk} de la red, esto es, la medida de cuánto cambia el valor de $\mathcal{L}$ cuando se realiza una modificación al parámetro w_{jk} . Finalmente, con las derivadas parciales de $\mathcal{L}$ con respecto a cada uno de los pesos es posible aplicar el descenso del gradiente, actualizando cada parámetro según el valor de la respectiva derivada. Back-propagation provee de una forma eficiente de realizar todo el cómputo del gradiente de $\mathcal{L}$, evitando el cálculo repetido de resultados previamente obtenidos y optimizando el uso de la memoria. (Goodfellow et al., 2016)

Sobreajuste y regularización

Con el uso de la ecuación 2.2 se busca minimizar el error de todo el conjunto de ejemplos con los que se entrena la red, pero esto, de la forma en la que se presenta, puede ocasionar *sobreajuste*. Incluso si el modelo produjese resultados extremadamente precisos para el conjunto de datos de entrenamiento, no necesariamente implicaría que es un buen modelo. Por el contrario, podría indicar que la red ajusta los parámetros sólo a los ejemplos presentados, de forma de minimizar la función de costo o error, pero sin capturar ninguna información valiosa del fenómeno tratado.

> When that happens the model will work well for the existing data, but will fail to generalize to new situations. The true test of a model is its ability to make predictions in situations it hasn't been exposed to before. [Cuando ello ocurre el modelo funcionará bien para los datos existentes, pero fallará al generalizar para nuevas situaciones. La verdadera prueba de un modelo es su habilidad para realizar predicciones en situaciones a las cuales no ha sido expuesto antes.] (Nielsen, 2015)

Por ejemplo, podría ocurrir un caso donde se observa una mejora continua de un modelo durante su entrenamiento en un gráfico de Error-Épocas pero, a la vez, identificar un crecimiento del error en la exactitud de predicción con nuevos ejemplos de prueba. Esta situación podría indicar que los parámetros se están sobreajustando a los datos de entrenamiento y el modelo está "sobreentrenado".

Una forma de solucionar el problema de sobreajuste es la incorporación de un *término de regularización* en la función de error a minimizar. La técnica de regularización consiste en utilizar un nuevo hiperparámetro λ conocido como *parámetro de regularización* y una función $R(\Theta)$ que toma como entrada los parámetros de la red neuronal y retorna un escalar que indica, en cierta forma, su

complejidad, la cual se desea mantener baja (Goldberg, 2017). De esta forma, el problema de optimización tiene la forma:

$$\hat{\Theta} = \underset{\Theta}{argmin} \left(\underbrace{\frac{1}{n}\sum_{i=1}^{n} L(f(x_i, \Phi), y_i)}_{Error\ \mathcal{L}} + \underbrace{\lambda R(\Theta)}_{Regularizacion} \right) \tag{2.3}$$

Con el nuevo término de regularización, la optimización 2.3 consiste en hallar un balance entre bajo error y baja complejidad de parámetros. Intuitivamente, el efecto de la regularización es que la red neuronal busque aprender pesos más pequeños y este nivel de preferencia puede ser modificado mediante el parámetro λ: Cuanto más grande sea su valor, más se incrementará el término de regularización y la optimización tenderá a seleccionar pesos más pequeños, mientras que si λ es un número chico, el modelo preferirá minimizar el error de predicción $\mathcal{L}$.

A continuación se destacan algunas de las estrategias de regularización más utilizadas actualmente (Goldberg, 2017) (Goodfellow et al., 2016). Estas difieren en la definición de la función R:

- Regularización L^1: La regularización L^1 se define formalmente como la suma de los valores absolutos de los pesos de la red, es decir:

  $$R_{L^1}(\Theta) = R_{L^1}(W) = \sum_i |w_i|$$

  Esta regularización penaliza los parámetros (pesos) pequeños y grandes de forma uniforme intentado llevar a todo parámetro no nulo a cero.

- Regularización L^2: La regularización L^2 es similar a la L^1 en términos de que utiliza la suma de la norma de los parámetros. En el caso de esta estrategia se utiliza la suma de los cuadrados de los pesos como función de regularización:

  $$R_{L^2}(\Theta) = R_{L^2}(W) = \sum_i (w_i)^2$$

  Al elevar al cuadrado los pesos se penaliza más a aquellos cuya magnitud sea mayor, mientras que los parámetros que se encuentren entre 0 y 1 no serán tomados tanto en cuenta a la hora de buscar la optimización 2.3.

Por último se debe poner en relieve una última estrategia de regularización relevante para la solución del problema del sobreajuste conocida como *Dropout*. A diferencia de las regularizaciones L^1 y L^2 donde se modifica la función de error, al utilizar dropout se modifica la propia red neuronal. En términos concretos, al comenzar el entrenamiento del modelo con un lote de ejemplos, un porcentaje de las neuronas de las capas intermedias son eliminadas (desactivadas) temporalmente y aquellas de las

capas de entrada y salida se mantienen sin modificaciones. Luego, se realiza la propagación de cada ejemplo hacia adelante a través de toda la red, obteniéndose el valor de error cometido, y se aplica finalmente el proceso de propagación hacia atrás actualizando los pesos de las neuronas presentes de forma de minimizar el error. Una vez finalizado el entrenamiento con un lote se prosigue de la misma forma con el siguiente, primero activando las neuronas previamente inhabilitadas y desactivando otro subconjunto aleatorio de ellas (Nielsen, 2015). En (Hinton et al., 2012) los autores destacan un punto de vista intuitivo de la razón por la que esta estrategia evita el sobreajuste:

> On each presentation of each training case, each hidden unit is randomly omitted from the network with a probability of 0.5, so a hidden unit cannot rely on other hidden units being present. Another way to view the dropout procedure is as a very efficient way of performing model averaging with neural networks. A good way to reduce the error on the test set is to average the predictions produced by a very large number of different networks. [En la presentación de cada caso de entrenamiento, cada unidad oculta es aleatoriamente omitida de la red con una probabilidad del 0.5, de forma que una unidad oculta no puede depender de la presencia de otra. Otra forma de ver el procedimiento de dropout es como una forma muy eficiente de realizar el promedio de modelos con redes neuronales. Una buena forma de reducir el error en el conjunto de prueba es promediar las predicciones producidas por un gran número de redes diferentes] (Hinton et al., 2012)

En efecto, al aplicar dropout, cada configuración diferente de neuronas en las capas ocultas representa una red distinta que se entrena para predecir los resultados en una forma particular. Luego, al considerar todas las unidades ocultas como parte de la red final, se logra una combinación de las predicciones de todas las redes con dropout previamente entrenadas, consiguiendo un modelo generalizador que extrapola las características principales los ejemplos utilizados durante la etapa de entrenamiento a datos de prueba que nunca fueron presentados.

2.2.2 Redes neuronales recurrentes

Dado que el objetivo principal de este trabajo es la generación de una secuencia de símbolos a partir del procesamiento de oraciones en español, la presente sección focaliza la atención en una familia particular de redes neuronales artificiales que permite resolver esta clase de problemas, denominadas *redes neuronales recurrentes* (RNNs). Debido a la propiedad inherente del lenguaje de ser un fenómeno temporal, éste puede ser interpretado como una secuencia de eventos (acústicos o escritos) a lo largo del tiempo. La naturaleza temporal del lenguaje puede ser reflejada en arquitecturas particulares de redes neuronales con el fin de resolver problemas de esta índole.

La clase de redes neuronales recurrentes hace posible el procesamiento de secuencias de valores, particularmente vectores $x_1, \ldots, x_\tau$ con un índice de tiempo entre 1 y τ. En la práctica, las RNNs usualmente operan sobre minilotes de esta clase de secuencias que suelen variar en longitud τ y

presentan nuevas conexiones recurrentes que permiten la representación del contexto previo de valores que pasan por la red. Una red neuronal recurrente es una red que tiene ciclos entre las conexiones de sus neuronas, cuyas salidas sirven luego como entrada en futuros valores a procesar.

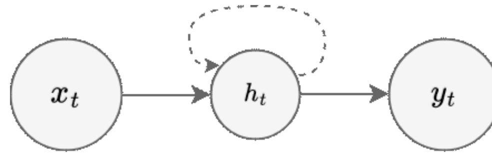

Figura 2.10: *Ejemplo de red neuronal recurrente. La capa oculta incluye una conexión recurrente o cíclica como parte de su entrada.*

La Figura 2.10 demuestra la idea básica de las conexiones cíclicas en las capas ocultas de una RNN. De forma casi idéntica a las redes neuronales con propagación hacia adelante como el multiperceptrón introducido en la sección 2.2.1, la RNN recibe una entrada x_t (donde t indica el paso en el tiempo, siendo entonces x_t el vector x en el tiempo t) que es procesada por una capa oculta, calculando la suma pesada y la función de activación. El resultado de esta capa sirve luego como entrada a la capa de salida para producir el valor final y_t . La diferencia clave entre las redes de propagación hacia adelante y las recurrentes yace en la conexión de realimentación que se muestra en la Figura 2.10 con la línea punteada. Cuando ingresa en la capa oculta un valor futuro de la secuencia x_{t+1} , el procesamiento en esta capa es influenciado tanto por los pesos que unen a la primera capa de entrada con la oculta, y además, por los valores de las salidas de la capa oculta del paso previo en el tiempo t y los pesos de las conexiones cíclicas. (Jurafsky & Martin, 2024)

> The hidden layer from the previous time step provides a form of memory, or context, that encodes earlier processing and informs the decisions to be made at later points in time. [La capa oculta de pasos previos en el tiempo proveen una forma de memoria, o contexto, que codifica procesamiento anterior y brinda información para la toma de decisiones en puntos futuros en el tiempo.] (Jurafsky & Martin, 2024, p.186)

Formalmente, la propagación hacia adelante en una red neuronal recurrente consiste en obtener un valor inferido y_t para un dato de entrada x_t , siendo necesario en el proceso el cómputo de un valor de activación, llamado *estado*, en una capa oculta h_t . Para calcular este último resultado la entrada x_t se multiplica con la matriz de pesos W y la salida de la capa oculta del paso anterior h_{t-1} con una nueva matriz de pesos U. Luego, sumando estos valores y sirviéndolos de entrada a una función de activación g es posible obtener el estado h_t:

$$h_t = g(Uh_{t-1} + Wx_t)$$

donde $W \in \mathbb{R}^{d_{in} \times d_h}$, $U \in \mathbb{R}^{d_h \times d_h}$, y d_{in} y d_h son las dimensiones de las capas de entrada y oculta respectivamente.

Una particularidad a destacar es que las matrices de pesos W y U son compartidas a través del tiempo, mientras que nuevos valores de h y de y son generados en cada paso de tiempo para una secuencia de datos particular. Que los parámetros sean compartidos hace posible extender y aplicar el modelo a secuencias de diferentes longitudes, generalizando sobre ellas. Este aspecto resulta importante cuando información relevante aparece en diferentes posiciones en secuencias distintas. Si se cuenta con una red neuronal de propagación hacia adelante que permite el procesamiento de secuencias de una longitud fija, a cada porción de la secuencia le corresponde un conjunto de parámetros (pesos y bias) distintos para los cuales se realizan actualizaciones durante el entrenamiento, potencialmente aprendiendo las mismas reglas para distintas posiciones de la secuencia (Goodfellow et al., 2016)

El hecho de que el cómputo del estado en el tiempo t dependa de aquel obtenido en el tiempo previo $t-1$ indica una secuencia de eventos que puede expresarse como una conexión recurrente en la red neuronal. El desarrollo en el tiempo de esta conexión puede observarse "desdoblando" la red recurrente en un grafo computacional de la conexión cíclica del estado oculto h. Un ejemplo sencillo de esto puede observarse en la siguiente Figura 2.11:

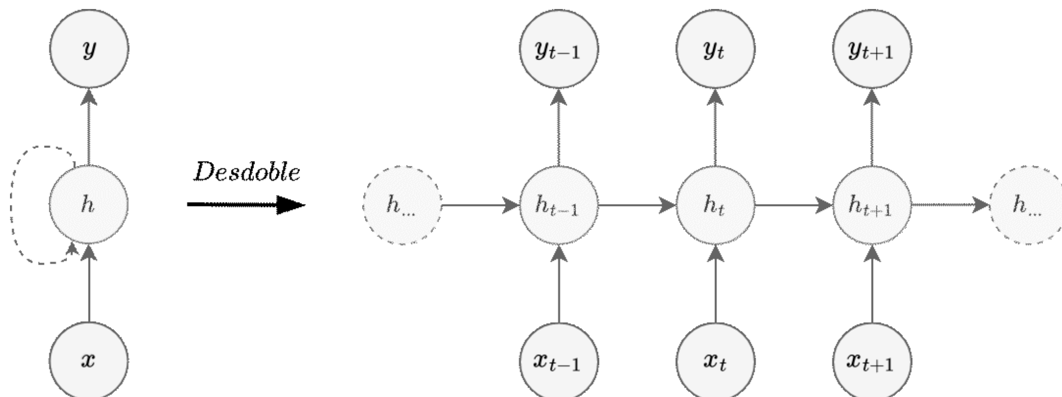

Figura 2.11: *Desdoblamiento de los pasos de propagación de una secuencia de datos en una RNN a través del tiempo*

Finalmente, al igual que con las redes neuronales de propagación hacia adelante presentadas en la sección previa 2.2.1, el entrenamiento de una RNN consiste en utilizar un conjunto de casos de ejemplo que atraviesan la red, calculando al final un valor de error que determina el grado de discrepancia entre el valor resultante y el esperado. Luego, utilizando un algoritmo de propagación hacia atrás se calculan los gradientes de la función de error con respecto a los parámetros en cada paso de una secuencia y en cada secuencia de ejemplo. Un algoritmo particular utilizado en las RNN es la p*ropagación hacia atrás a través del tiempo* (*back-propagation through time)* (Goodfellow et al., 2016) que equivale a aplicar back-propagation sobre el grafo computacional "desdoblado" de la red recurrente.

2.3 LaTeX

En el año 1977 Donald E. Knuth desarrolló el sistema de composición tipográfica TeX orientado a la preparación de una gran variedad de documentos con ciertas necesidades de diseño tipográfico, como libros, manuales, artículos científicos, tesis y cartas. Este sistema permitía describir de forma programática los detalles de formato de dichos documentos por medio de un conjunto de comandos que luego eran interpretados para generar un archivo que describía, a bajo nivel, la estructura de cada página independientemente del dispositivo donde se mostrara. Basado en TeX, Leslie Lamport introdujo en 1985 la primera versión del sistema LaTeX, centrado más en la estructura de los documentos que en los detalles de formato. Éste nuevo sistema representó en su momento, y al día de hoy, una interfaz de descripción de documentos que separaba su contenido de su forma, mostrando una visión lógica de ellos y menos visual. En conjunto con la incorporación de hojas de estilo que permitían definir el formato visual de los documentos, LaTeX se convirtió en un estándar por defecto para la comunicación y publicación de documentos de disciplinas académicas (Mittelbach & Rowley, 1999)

En particular, LaTeX tuvo un gran impacto en las áreas de la ciencia estrechamente relacionadas con la matemática dado al amplio soporte de formatos y visualización estandarizada para los símbolos y elementos matemáticos.

> LaTeX is now extremely popular in the scientific and academic communities, and it is used extensively in industry. It has become a *lingua franca* of the scientific world. [LaTeX es ahora extremadamente popular en las comunidades científicas y académicas, y es usado extensivamente en la industria. Se ha convertido en una *lingua franca* del mundo científico.] (Lamport, 1994, p.12)

LaTeX permite definir la composición tipográfica de los documentos a través de una librería de comandos que son de uso directo y sencillo, a diferencia de otros lenguajes de programación con una gramática más compleja como C o C++. La sintaxis de LaTeX está compuesta principalmente por dos clases de elementos (Datta, 2017):

- Los *comandos:* Representan instrucciones independientes que permiten producir algo nuevo o cambiar la forma de un elemento preexistente, por ejemplo añadir el símbolo β con el comando `\beta` o cambiar el estilo de la palabra “itálica” a “*itálica*” con `\textit{itálica}`. La sintaxis de los comandos se caracteriza por siempre comenzar con el carácter de barra invertida “\” y estar seguido por el nombre del comando. A su vez es posible enviarles parámetros obligatorios y opcionales, de forma que puedan utilizarlos como argumentos de entrada para llevar a cabo su tarea.
- Los *ambientes:* Estas son estructuras que se caracterizan por comenzar y finalizar con dos comandos aparejados `\begin` y `\end` que delimitan el alcance del contenido que debe ser parte de la tarea a resolver, como definir una lista de elementos, insertar una imagen o escribir una ecuación matemática.

En conjunto, estos dos elementos del lenguaje permiten especificar un impresionante abanico de instrucciones y contenido con una calidad tipográfica excepcional. Considerando el contexto del presente trabajo, se destaca la gran utilidad de LaTeX para el manejo de símbolos y fórmulas matemáticas. Por medio de la definición de ambientes específicos como `equation`, `cases` o `pmatrix`, es posible construir expresiones matemáticas de gran complejidad con una alta calidad visual asegurada. A continuación se presenta la tabla 2.1 con ejemplos para cada uno de los ambientes mencionados y su correspondiente renderización:

Ambiente	Código ejemplo	Visualización

`equation`	`\begin{equation}`
	`  A = \sum_{i=1}^n f(x_i)\Delta x`
	`\end{equation}`
		$A = \sum_{i=1}^{n} f(x_i)\Delta x$

`cases`	`\begin{cases}`
	`  x + y = \frac{1}{2}\\`
	`  3x - y = 4`
	`\end{cases}`
		$\begin{cases} x + y = \frac{1}{2} \\ 3x - y = 4 \end{cases}$

`pmatrix`	`\begin{pmatrix}`
	`  a_{11} & a_{12} & a_{13}\\`
	`  a_{21} & a_{22} & a_{23}\\`
	`  a_{31} & a_{32} & a_{33}`
	`\end{pmatrix}`
		$\begin{pmatrix} a_{11} & a_{12} & a_{13} \\ a_{21} & a_{22} & a_{23} \\ a_{31} & a_{32} & a_{33} \end{pmatrix}$

Tabla 2.1: *Ejemplos de uso de ambientes LaTeX para la definición de expresiones matemáticas.*

Cabe resaltar que LaTeX presenta dos modos de escritura, un modo de texto normal y otro especialmente definido para la especificación de fórmulas matemáticas. Es posible escribir en éste último modo "matemática" al encerrar el contenido requerido entre los símbolos "\[" y "\]" o al añadir un par de signos pesos "$" al inicio y fin del bloque.

Sumado a su gran utilidad en el ambiente académico como las universidades y los laboratorios de investigación de las facultades, LaTeX ha logrado una expansión significativa a un comunidad de más de 17 millones de usuarios[1] alrededor del mundo. Ésto puede atribuirse a la introducción de un editor online basado en una aplicación web llamada Overleaf que facilita a los usuarios de este sistema un entorno preparado con un amplio conjunto de paquetes de comandos pre instalados para la creación eficiente de documentos.

En definitiva, la facilidad de uso LaTeX y la calidad de los documentos producidos con él, han sido la motivación principal en la elección de este lenguaje como el formato base sobre el que se enfocará el presente trabajo de grado. Más específicamente, en la sección 4.1.1 del capítulo 4 se describirá una versión levemente modificada del LaTeX apodada *pseudoLaTeX*, que fue definida particularmente en esta

[1] https://www.overleaf.com/about

tesina para el proceso de entrenamiento e inferencia del modelo neuronal implementado y que aportó una mayor variabilidad de valores numéricos en las expresiones algebraicas generadas.

2.4 Estado del arte

A pesar de contar con un gran volumen de publicaciones relacionadas a modelos de aprendizaje profundo se identificó que relativamente poco porcentaje de éstos resuelven concretamente problemas de generación de secuencias en el marco de la ciencia matemática. Al indagar sobre el uso de modelos de redes neuronales aplicados en este área particular, gran parte de los artículos hallados trataban especialmente la resolución de problemas o ecuaciones, como en los estudios de (Lample & Charton, 2019) y (Charton et al., 2021). En ellos se hacía hincapié en el uso de modelos con la arquitectura Transformer para tratar tareas complejas de matemática como la resolución de integrales y ecuaciones diferenciales mediante la manipulación simbólica de sus elementos. Estos modelos demostraron un desempeño incluso superior al de los sistemas de álgebra computacional como Matlab y Mathematica, probando la gran capacidad de aprendizaje de las redes neuronales con arquitectura Transformer.

Por otro lado, un aspecto íntimamente relacionado a las expresiones matemáticas es su representación equivalente como árboles. Ésta fue utilizada en líneas de investigación como las de (Lample & Charton, 2019) y (Wang et al., 2021) como una forma de reescribir las expresiones de forma más precisa, sin ambigüedades y para reflejar su estructura jerárquica determinada por la precedencia de operadores. Los resultados de estos artículos mostraron que el uso de las representaciones de árboles ayudaron a obtener resultados satisfactorios en el rendimiento de los modelos Transformer entrenados.

Otros casos de soluciones inteligentes considerados como parte del estado del arte son los sistemas comerciales Symbolab y WolframAlpha orientados a la resolución interactiva de problemas matemáticos. Sin embargo, a pesar de mostrar resultados muy precisos y mencionar el uso de la inteligencia artificial como parte de los productos que ofrecen, las implementaciones y estructura de los modelos que utilizan se mantienen como código propietario, imposibilitando el análisis de estas soluciones.

Finalmente, resulta primordial destacar en este apartado a GPT-3 (Brown et al., 2020) como un modelo grande de lenguaje (LLM) de propósito general que ha producido un gran impacto en el área de investigación del procesamiento de lenguaje natural y ha desencadenado un aumento explosivo del interés por la confección de nuevos modelos de lenguaje (Fan et al., 2023). Particularmente, el lanzamiento del chatbot interactivo ChatGPT[2], que hacía uso de GPT-3, ha causado grandes repercusiones en el ámbito educativo debido a su creciente integración al proceso de enseñanza y aprendizaje. La particular facilidad de su uso y gran precisión en la generación de respuestas ha conducido, en ocasiones, a alumnos y docentes a utilizarlo como medio de resolución de problemas de matemática o incluso para la formulación de éstos. Sin embargo, se ha demostrado (Frieder et al., 2023) que, si bien ChatGPT puede ser utilizado exitosamente como un asistente de búsqueda o base de conocimiento de conceptos matemáticos, aún presenta fallas en múltiples aspectos de esta área de la ciencia, como la resolución de problemas con cierto grado de complejidad, la formulación de

[2] https://chatgpt.com/

demostraciones o la manipulación simbólica de expresiones. Los autores de (Frieder et al., 2023) han dejado en evidencia las inconsistencias del desempeño de ChatGPT al momento de resolver tareas específicas de matemática, lo que conlleva a poner en duda el grado de confianza que se puede depositar en las respuestas generadas por este modelo:

> Contrary to the media sensation that (Chat)GPT has caused, (Chat)GPT is not yet ready to deliver high-quality proofs or calculations consistently. At the same time, the quality of the answers can be positively surprising. [...] (Chat)GPT falls short of achieving the same performance as models specifically trained for single tasks. These models, in contrast, lack the flexibility of (Chat)GPT, which is a universal tool suitable for any area of mathematics. [Contrariamente a la sensación mediática que ha causado (Chat)GPT, (Chat)GPT aún no está listo para proveer pruebas de alta calidad y cálculos de forma consistente. Al mismo tiempo, la calidad de las respuestas puede ser positivamente sorprendente. [...] (Chat)GPT no alcanza el mismo desempeño que otros modelos específicamente entrenados sobre una sola tarea. Estos modelos, en contraste, no poseen la flexibilidad de (Chat)GPT, el cual es una herramienta universal adecuada para cualquier área de la matemática]
> (Frieder et al., 2023, p.9)

A pesar de no hallar casos de estudio que resolvieran concretamente la tarea de generación de expresiones en formato LaTeX a partir de descripciones textuales, los artículos y modelos mencionados en este apartado han ayudado a definir los pilares fundamentales del desarrollo del presente trabajo. Éstos han permitido especificar un camino más acotado, centrando el análisis en los modelos basados en la arquitectura Transformer y la definición de distintas representaciones de los datos para su uso durante el proceso de entrenamiento.

CAPÍTULO 3

Procesamiento de Lenguaje Natural

Contando con la introducción teórica a las técnicas del aprendizaje profundo vistas en el capítulo anterior, ésta tercera sección del trabajo de grado tiene el objetivo de continuar la profundización de la composición y funcionamiento de los modelos neuronales, centrándose en un área particular de aplicación: el procesamiento de lenguaje natural o NLP.

Este conjunto de algoritmos ha surgido con el objetivo de obtener representaciones interpretables del lenguaje humano para las computadoras, con las cuales se busca automatizar las actividades cotidianas. A pesar de no ser un campo nuevo de investigación en las ciencias informáticas, el procesamiento de lenguaje natural ha ganado particular tracción en la última década, insertándose en la vida diaria de los usuarios de la Internet. La traducción automática de idiomas, la detección de correo no deseado por medio de la clasificación de texto, la búsquedas en los navegadores web o los sistemas de conversación como chatbots, son algunos pocos ejemplos de las tareas que se contemplan al hablar del procesamiento de lenguaje natural (Eisenstein, 2019).

En los últimos años, los grandes avances obtenidos en el contexto de NLP utilizando modelos basados en las técnicas *"black-box"* del aprendizaje profundo, han resultado en un incremento acelerado en la popularidad del área. Con ello y la introducción en 2017 de la novedosa arquitectura Transformers (Vaswani et al., 2017) se ha conducido a una explosión (Fan et al., 2023) en el interés general por obtener nuevos avances en el área, viéndose reflejado en el gran número de artículos científicos publicados y en las implementaciones de nuevos modelos de lenguaje (LLMs) de propósito general.

> Large language models (LLMs) are a class of language models that have demonstrated outstanding performance across a range of natural language processing (NLP) tasks and have become a highly sought-after research area, because of their ability to generate human-like language and their potential to revolutionize science and technology. [Los grandes modelos de lenguaje (LLMs) son una clase de modelos de lenguaje que han demostrado un desempeño sobresaliente a lo largo de un rango de tareas de procesamiento de lenguaje natural (NLP) y se han convertido en un área de investigación muy buscada, debido a su habilidad para generar lenguaje similar al humano y su potencial de revolucionar la ciencia y la tecnología] (Fan et al., 2023, p. 1)

En términos de la organización del presente capítulo, se comenzará en la sección 3.1 una introducción a las técnicas y conceptos utilizados para el procesamiento de lenguaje natural que sirven de base para la fundamentación de los modelos neuronales sobre los que se hará foco en esta tesina. Posteriormente, en la sección 3.2 se detallarán las estructuras y métodos necesarios para lograr una correcta representación de las secuencias de palabras que conforman las oraciones del lenguaje natural, como lo son la tokenización de símbolos y la construcción de *embeddings* para su posterior utilización como entradas en las redes neuronales. Luego, a partir de la sección 3.3, se conducirá la explicación hacia la novedosa arquitectura Transformer y su uso en los grandes modelos neuronales de lenguaje que se describirán en 3.4.

Continuando en la línea del procesamiento de lenguaje natural utilizando modelos neuronales, la sección 3.5 presentará la técnica de entrenamiento *fine-tuning*, a la cual se le puede atribuir la rápida mejora en el rendimiento de los modelos entrenados sobre conjuntos pequeños de datos con un bajo número de iteraciones. Por otra parte, en pos de mostrar una visión general del reconocimiento de entidades nombradas, tarea sobre la que se hará énfasis a partir del siguiente capítulo 4, la sección 3.6 introducirá los detalles de su uso y la notación considerada en este trabajo. Finalmente, en el apartado 3.7 se expondrá el *aprendizaje multitarea*, la última de las técnicas implementadas en el desarrollo del modelo de esta tesina.

3.1 Modelos de lenguaje

Los modelos de lenguaje permiten la construcción de una distribución de probabilidades donde, dada una secuencia de palabras, como una oración en español, se asigna una probabilidad a cada elemento de la secuencia dado un contexto previo. Es posible, a su vez, utilizarlos para determinar la probabilidad de la ocurrencia de una oración completa.

Al cumplir con la tarea de asignación de probabilidades, los modelos de lenguaje son esenciales para resolver problemas complejos de procesamiento de lenguaje natural como la traducción automática (Vaswani et al., 2017), identificación de entidades nombradas (Devlin et al., 2018), interacción conversacional (Ouyang et al., 2022) o generación de texto artístico (Dal Bianco, 2021), entre muchos otros.

La presente sección introducirá conceptos primordiales de dos clases de modelos de lenguajes, aquellos basados en n-gramas y otros más complejos basados en redes neuronales artificiales. Así mismo, se expondrán las limitaciones respectivas de cada una de estas clases de modelos que propiciaron el surgimiento de las arquitecturas de redes neuronales Transformers que, actualmente, representan el estado del arte en el contexto del procesamiento de lenguaje natural (Devlin et al., 2018) (Radford et al., 2018).

3.1.1 Modelos basados en n-gramas

El modelo más simple de asignación de probabilidades a secuencias de palabras son aquellos basados en *n-gramas*, los cuales representan, sencillamente, una secuencia de *n* palabras. Para cada palabra *w* de la secuencia, el modelo define una probabilidad de ocurrencia, dado un contexto o historia previa *h*, es decir $P(w|h)$ (Jurafsky & Martin, 2024). Por ejemplo, suponiendo que la oración "me gusta el café" representa el contexto *h*, se desea especificar la probabilidad de que la palabra "negro" sea la siguiente en la oración:

$$P(negro\,|\,\text{me gusta el caf})$$

En términos generales dada una secuencia de palabras $w = w_1, \ldots, w_n$ que determina el contenido de una oración, se define la probabilidad de la ocurrencia de dicha oración de la forma

$$P(w) = P(w_1, \ldots, w_n) = P(w_1) \times P(w_2|w_1) \times P(w_3|w_2, w1) \times \cdots \times P(w_n|w_{n-1}, \ldots, w_1)$$

donde cada factor del producto representa la probabilidad de una palabra dadas todas sus predecesoras. Esta descomposición en un producto de factores es posible debido a la regla de la cadena de probabilidades. El problema principal de esta situación es que la regla de la cadena no ayuda a computar la probabilidad de la ocurrencia de una palabra dada una secuencia de otras previas $P(w_i|w_{i-1}, \ldots, w_1)$.

Un acercamiento ambicioso para obtener el valor de dicha probabilidad es el de calcular una estimación de la frecuencia relativa de aparición de la palabra w_i luego de la subsecuencia $w_{i-1}, \cdots, w_1$, con respecto a todos los casos en los que aparece la secuencia $w_{i-1}, \ldots, w_1$ dentro de un conjunto de documentos suficientemente grande (como la web). Esto es:

$$P(w_i|w_{i-1}, \ldots, w_1) = \frac{C(w_1, w_2, \ldots, w_i)}{C(w_1, \ldots, w_{i-1})} \tag{3.1}$$

Este método presenta dos problemas relevantes: si se busca obtener un resultado preciso de dicha probabilidad se debería considerar un conjunto prácticamente infinito de documentos, donde es posible que algunas oraciones tengan probabilidad 0 en el caso de que no tengan ninguna ocurrencia. Esto, a su vez, se debe a un segundo problema que es la propia naturaleza creativa del lenguaje que se construye y varía continuamente de forma que resulta impráctico implementar este método para oraciones extremadamente largas.

> While this method of estimating probabilities directly from counts works fine in many cases, it turns out that even the web isn't big enough to give us good estimates in most cases. This is because language is creative; new sentences are created all the time, and we won't always be able to count entire sentences. [A pesar de que este método de estimar probabilidades directamente desde el conteo funcione bien en muchos casos, resulta que incluso la web no es lo suficientemente grande para proveer buenas estimaciones en la mayoría de los casos.

> Esto es porque el lenguaje es creativo; nuevas oraciones son creadas todo el tiempo, y no siempre podremos contar oraciones enteras.] (Jurafsky & Martin, 2024)

La intuición detrás del modelo de lenguaje basado en n-gramas (MLN) es, en lugar de calcular la probabilidad exacta de una palabra dada toda su historia o contexto, se aproxima dicho contexto teniendo solo en cuenta algunas de las últimas palabras de la oración. Un modelo n-grama supone que la probabilidad de una palabra de ocurrir luego de una secuencia previa, es aproximadamente igual a la probabilidad de su ocurrencia condicionada a solo las $n - 1$ palabras anteriores:

$$P(w_m|w_{m-1}, \ldots, w_1) \approx P(w_m|w_{m-1}, \ldots, w_{m-n+1}) \tag{3.2}$$

Esto implica que la probabilidad de la oración completa w está dada por:

$$P(w) = P(w_1, \ldots, w_m) \approx \prod_{i=1}^{m} P(w_i|w_{i-1}, \ldots, w_{i-n+1})$$

El caso más sencillo es el del modelo bigrama que considera una sola palabra como parte del contexto previo. Si se cuenta con la oración “Me gusta el café”, su probabilidad de ocurrencia definida por este modelo es aproximada de la siguiente forma

$$P(\text{Me gusta el cafe}) = P(\text{Me}|\langle/s\rangle) \times P(\text{gusta}|\text{Me}) \times P(\text{el}|\text{gusta}) \times P(\text{cafe}|\text{el}) \times P(\langle/s\rangle|\text{cafe})$$

Los símbolos $\langle s \rangle$ y $\langle/s\rangle$ representan cadenas de apertura y cierre de la oración, respectivamente, con el fin de que el modelo posea contexto tanto para la primera y última palabra.

Luego, cada valor de probabilidad de la fórmula 3.2 se calcula con la frecuencia relativa estimada de la ecuación 3.1. Cabe notar que dicha división provee un resultado normalizado entre 0 y 1 por lo que es posible interpretarla como un valor de probabilidad.

Tomando sólo las últimas $n - 1$ palabras de la oración como contexto, el modelo de lenguaje basado en n-gramas permite reducir la cantidad de información necesaria a almacenar. Si V representa el conjunto de vocabulario considerado, el modelo debe almacenar la probabilidad de V^n eventos, que es exponencial en el orden del n-grama, en lugar de V^m, que es exponencial en la longitud de la oración.

El mayor desafío a la hora de implementar un modelo de n-gramas, es la correcta elección del hiperparámetro n. En el caso de que n sea muy pequeño, es posible que no se considere un tamaño de contexto lo suficientemente grande como para obtener probabilidades precisas, fallando ampliamente en la estimación de la ocurrencia de las palabras. En cambio, si n fuera demasiado grande, la cantidad de información necesaria crecería excesivamente y se aumentarían exponencialmente los casos de probabilidad 0 ya que no todas las palabras se ubican antes o después de todas las otras.

3.1.2 Modelos neuronales de lenguaje

El uso de las arquitecturas de redes neuronales artificiales en el contexto del procesamiento de lenguaje natural comenzó a crecer particularmente con el advenimiento de las redes neuronales recurrentes (LeCun et al., 1989). Los modelos de lenguaje basados en n-gramas (MLNs) han sido suplantados progresivamente por modelos de redes neuronales artificiales que difieren ampliamente de sus predecesores en el sentido que no realizan suposiciones sobre contextos limitados, sino que son capaces de procesar un contexto de longitud arbitraria, manteniéndose computacionalmente viables (Eisenstein, 2019). Los modelos de lenguaje neuronales tienen algunas ventajas por sobre los MLNs, como la capacidad de generalización sobre contextos mucho más extensos, proveen una forma de aprender y embeber el significado y contexto de cada palabra en vectores que permiten obtener predicciones de palabras más precisas. Por otro lado, es necesario resaltar que la complejidad de estos modelos de aprendizaje profundo disminuye la interpretabilidad del comportamiento del modelo y aumenta el cómputo y energía necesario para su construcción (Jurafsky & Martin, 2024).

En la presente sección se mencionará, primeramente, la aplicación de las redes neuronales de propagación hacia adelante o feedforward como un acercamiento al modelado de lenguaje utilizando técnicas del aprendizaje profundo. Luego se dará una muy breve introducción al uso de las redes neuronales recurrentes como modelos de lenguaje neuronales y su capacidad de procesamiento secuencial de información. A pesar de no ser representativos del estado del arte en el contexto del procesamiento del lenguaje, se utilizará esta sección y estos modelos iniciales como punto de partida en la explicación de conceptos fundamentales del uso de redes neuronales artificiales en NLP. Las ideas aquí presentadas serán posteriormente profundizadas en la sección 3.3 Transformers que representan la arquitectura subyacente en los algoritmos de NLP de hoy en día.

Modelado con redes neuronales feedforward

Como se ha definido al inicio de esta sección 3.1, el modelado de lenguaje consiste en la predicción de palabras dado una secuencia previa de ellas. Las redes neuronales de propagación hacia adelante o feedforward pueden ser utilizadas para resolver esta tarea, consiguiendo así modelos de lenguaje *neuronales*. Dado un tiempo t y una representación de palabras previas $w_{t-1}, w_{t-2}, \ldots, w_1$ el modelo feedforward aproxima la probabilidad de la ocurrencia de la palabra siguiente w_t de forma similar a los MLNs, considerando solo $n-1$ palabras como contexto previo:

$$P(w_t|w_{t-1}, \ldots, w_1) \approx P(w_t|w_{t-1}, \ldots, w_{t-n+1})$$

En lugar de utilizar a las propias palabras como entidades, las redes neuronales necesitan que sus entradas sean exclusivamente numéricas. Es por ello que cada palabra recibe una representación dada por un vector de números denominado *embedding*, el cual contiene información embebida de la semántica o significado de la palabra e incluso puede determinar el grado de similitud que tiene con las

palabras de su contexto. Con este tipo de representación los modelos neuronales de lenguaje son capaces de generalizar mejor sobre datos que nunca fueron introducidos (Jurafsky & Martin, 2024).

Dado que el concepto de embeddings será profundizado en la sección 3.2 la interpretación de dichos vectores durante esta sección se acotará simplemente como representaciones numéricas de las palabras en el contexto de redes neuronales artificiales. Cada una de estas representaciones vectoriales formarán columnas de una matriz de embeddings llamada E utilizada durante la propagación de la información a través de la red.

El objetivo de este modelo de lenguaje neuronal es obtener una distribución de probabilidades sobre todo un vocabulario V de palabras posibles, donde aquella a la cual se le asigna la mayor probabilidad representa la siguiente palabra a generar. Luego, dada una ventana de m palabras previas en una oración que sirven como entrada en la red neuronal, las palabras ingresan a la red con una representación vectorial unívoca x_t denominada vector *one-hot*. Para la i-ésima palabra del vocabulario, este vector con $|V|$ componentes está formado por todos ceros, exceptuando la posición i donde tiene un 1.

Los vectores one-hot son multiplicados por la matriz $E \in \mathbb{R}^{d \times |V|}$ (donde d es la dimensión de cada vector embedding), obteniéndose para cada palabra que ingresa a la red el embedding que le corresponde y que se encuentra almacenado como columna de dicha matriz. La siguiente Figura 3.1 demuestra este proceso producido por el producto vector-matriz:

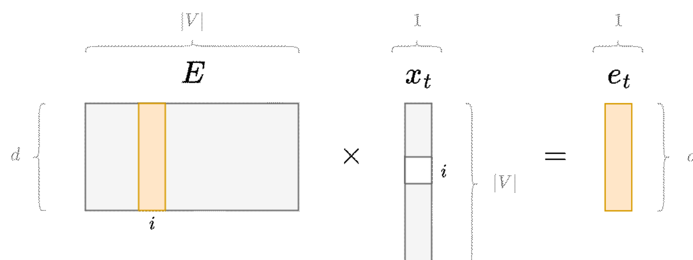

Figura 3.1: *Obtención del embedding de una palabra por medio del producto entre el vector one-hot y la matriz de embeddings.*

Luego, cada uno de los m embeddings resultantes son concatenados en un único vector de dimensión $m.d \times 1$ que sirve como entrada a la capa oculta donde se computan las primeras funciones de activación. El interior de la red neuronal actúa de forma idéntica a la presentada en el capítulo anterior, pudiendo variar en el número de capas ocultas, cantidad de neuronas por capa y las funciones de activación utilizadas. Finalmente, cabe destacar que la última capa de salida tiene $|V|$ neuronas, una para cada palabra del vocabulario. El resultado final es el obtenido por una función *softmax* que convierte los valores finales en una distribución de probabilidades, infiriendo para cada neurona i la probabilidad de que la i-ésima palabra del vocabulario sea la siguiente en la secuencia de m palabras previas (Jurafsky & Martin, 2024):

$$P(w_t = i | w_{t-1}, \dots, w_{t-m})$$

A la hora de entrenar esta red neuronal se utiliza el algoritmo de propagación hacia atrás *backpropagation* de la misma forma vista en el capítulo 2. Es posible disponer de una matriz de

embeddings previamente establecida o *preentrenada* cuyos valores se mantengan fijos (técnica llamada *transfer learning*) durante el entrenamiento, pero hay casos donde resulta beneficioso modificar los valores de E en el entrenamiento con el fin de aprender representaciones más significativas para la tarea particular a resolver. El cuerpo de documentos o ejemplos presentado a estos modelos de lenguaje neuronales representan tanto los datos de entrada como los valores de salida deseados, debido a que el modelo predice la siguiente palabra en una oración y la correctitud de dicha salida puede ser verificada en el propio ejemplo de entrada. Es por esta razón que este modo de entrenamiento recibe el nombre de *auto-supervisado*.

La Figura 3.2 muestra de forma gráfica un ejemplo genérico del funcionamiento de los modelos de lenguaje neuronales feedforward:

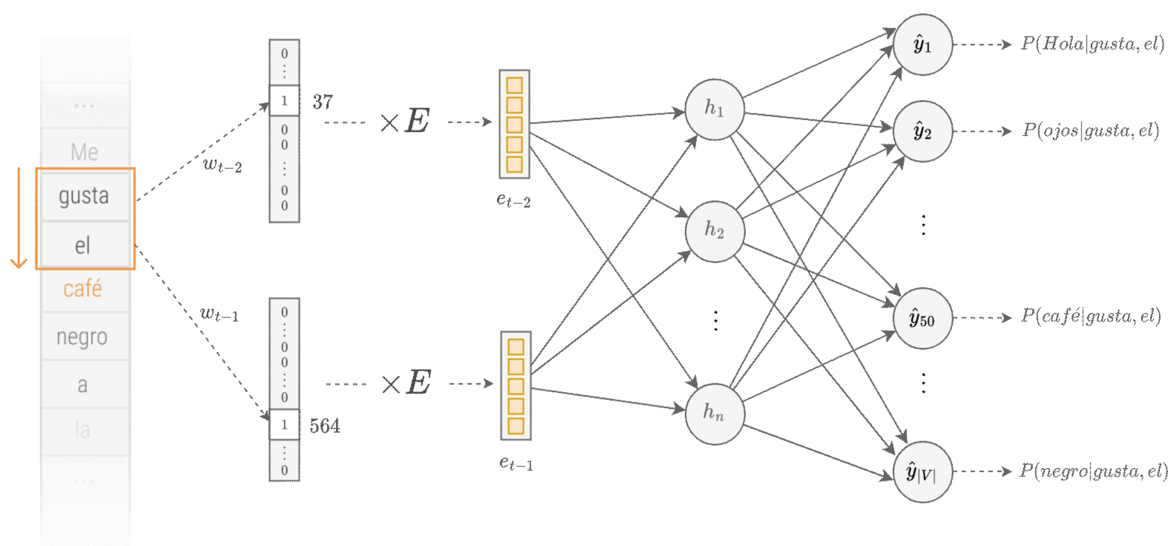

Figura 3.2: *Ejemplo de proceso de inferencia de un modelo de lenguaje neuronal con una ventana de contexto $m = 2$ y un vocabulario V*

Modelado con redes neuronales recurrentes

Una gran diferencia entre los modelos de lenguaje basados en redes neuronales recurrentes y feedforward (presentadas anteriormente), es que estos últimos toman una ventana fija de m palabras previas como contexto para predecir la siguiente en la secuencia, mientras que las RNN utilizan sus conexiones recurrentes en los estados ocultos para representar la información de todas las palabras previas a la que se intenta inferir, hasta la primera de la secuencia. Los modelos de lenguaje basados en RNN toman de entrada una palabra w_t a la vez y utilizan la información aprendida de la historia previa y acumulada en el estado oculto h_{t-1}, generando un nuevo estado h_t.

A la hora de entrenar los modelos de lenguaje basados en RNN (al igual que aquellos basados en redes feedforward), la función de error utilizada para determinar el desvió entre el valor inferido y el esperado es la Entropía cruzada multiclase la cual, recordando la definición presentada en la sección 2.2.1, calcula la diferencia entre dos distribuciones de probabilidades (la esperada y la inferida).

Con el fin de dirigir el curso de esta introducción de modelos de lenguajes hacia las aplicaciones concretas que serán vistas en los Capítulos 4 y 5, se utilizará esta última porción del apartado para presentar dos usos más de las redes neuronales recurrentes para tareas generales de procesamiento de lenguaje natural que surgen de los modelos de lenguaje neuronales previamente introducidos:

- ***Etiquetado de secuencias***: Las redes neuronales recurrentes pueden ser utilizadas para tareas de etiquetado de partes de oraciones o palabras orientado a distintos objetivos, por ejemplo la identificación de entidades nombradas (profundizada en la sección 3.6) o el etiquetado gramatical de oraciones. En lugar de generar una distribución de probabilidades final para la ocurrencia de la siguiente palabra en una secuencia, estos modelos neuronales pueden producir una distribución para un conjunto de etiquetas y utilizar la entropía cruzada como función de error durante el entrenamiento. La figura 3.3 muestra un diagrama que describe de forma general el funcionamiento de este etiquetado usando RNNs:

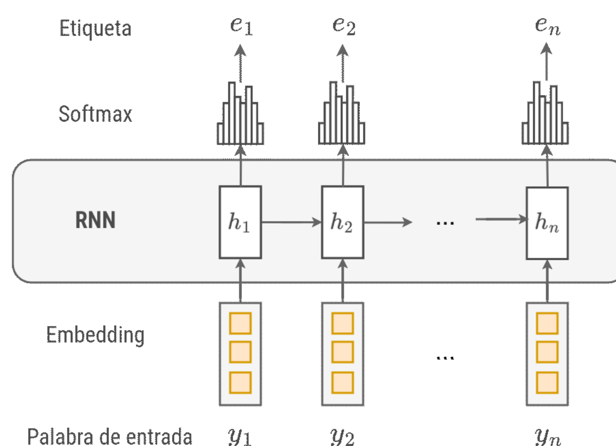

Figura 3.3: *Diagrama general del etiquetado de las palabras de una oración usando una red neuronal recurrente.*

- ***Generación de texto***: Los propios modelos de lenguaje basados en RNN también pueden ser utilizados para la generación de texto. Esta tarea de gran importancia, y donde los modelos de lenguajes neuronales tuvieron su mayor impacto, tiene el objetivo de producir texto que es condicionado por otro que se sirve de entrada al modelo. En particular, la generación de texto basada en modelos de lenguaje neuronales recibe el nombre de generación autorregresiva, debido a que cada palabra o símbolo generado en el momento t está condicionado por la palabra seleccionada por la red en el paso anterior $t-1$. La figura 3.4 muestra gráficamente este comportamiento de forma general:

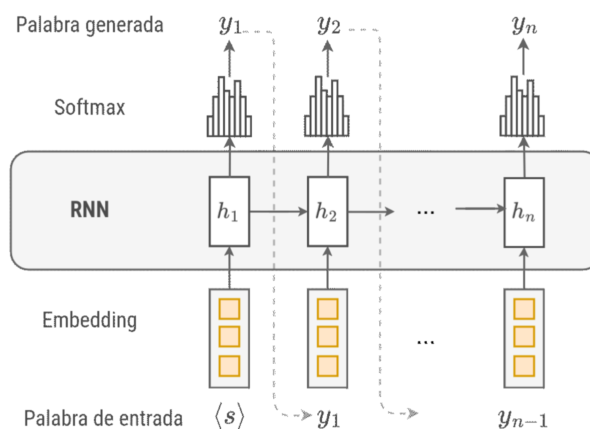

Figura 3.4: *Diagrama general de la generación autorregresiva de una oración utilizando una red neuronal recurrente.*

3.2 Preprocesamiento y semántica vectorial

Antes de poder realizar cualquier clase de procesamiento de lenguaje sobre un texto dado se debe realizar un proceso de *normalización*. Una tarea particular incluida en este proceso y que es relevante en el alcance del presente trabajo es la tarea de *tokenización*, la cual será presentada en el primer apartado 3.2.1 de esta sección. Posteriormente, el segundo apartado 3.2.2 describirá cómo los tokens resultantes de las técnicas de segmentación de texto durante la tokenización pueden transformarse a un formato numérico a partir de algoritmos de representación vectorial de palabras. De esta forma, al considerar modelos de lenguaje basados en aprendizaje profundo, es posible obtener un formato de estos tokens compatible con la arquitectura particular de las redes neuronales.

3.2.1 Tokenización

La *tokenización es* una tarea de normalización que consiste en separar un texto de lenguaje natural en unidades lingüísticas básicas denominadas *tokens*. Estas pequeñas porciones del texto son entonces consideradas como unidades indivisibles a la hora de realizar tareas de procesamiento sobre él. A pesar de aparentar ser un proceso sencillo, lograr una correcto particionamiento de un texto de forma de obtener tokens con información significativa no resulta ser una tarea con una solución directa (Jurafsky & Martin, 2024). Algunos casos usuales que presentan desafíos son, por ejemplo:

- *Signos de puntuación*: Si bien son buenas referencias para el particionamiento de tokens, existen casos en los que resulta importante mantenerlos dentro de estos tokens para mantener su significado. Ejemplos de estos son las expresiones numéricas como 42,3 o 1.200 que contienen los signos "coma" y "punto", direcciones de correo como "anemail@mail.com.ar" o incluso

expresiones de dinero como “$40.5”, para las cuales no deberían obtenerse dos tokens separados “$40” y “5” ya que se perdería parte de la información.

- *Expresiones de múltiples palabras*: Existen ciertas aplicaciones donde el proceso de tokenización debería mantener múltiples palabras como una única unidad, como por ejemplo “rock ’n’ roll” o “Nueva York”, cuyas palabras individualmente tienen significados muy diferentes a cuando se utilizan en conjunto. Por ejemplo, mientras la expresión “Nueva York” referencia a un estado de Estado Unidos, la palabra “Nueva” por sí sola es un adjetivo y “York” puede representar el nombre de la ciudad de Inglaterra. Estas situaciones están estrechamente relacionadas con el reconocimiento de entidades nombradas, una tarea particular de NLP que será descrita en la sección 3.6.

En forma resumida, se pueden categorizar los algoritmos de tokenización en dos grupos: tokenización *top-down*, donde se utilizan reglas determinísticas de particionamiento del texto, y la tokenización *bottom-up* en los cuales se utilizan medidas estadísticas de las secuencias de letras o símbolos en las oraciones de forma de obtener porciones de palabras o *subpalabras*. A continuación, se profundizará este acercamiento inicial al proceso de tokenización con un algoritmo de tipo bottom-up utilizado comúnmente en conjunto con los modelos grandes de lenguaje.

Algoritmos bottom-up de tokenización

Como se vió en el capítulo anterior, el entrenamiento de una red neuronal depende de un conjunto de entradas en formato numérico con las cuales se realizan correcciones iterativas por medio de los gradientes de una función de error. En el caso del procesamiento de lenguaje natural utilizando esta clase de modelos de aprendizaje automático, resulta indispensable transformar el texto de entrada a un conjunto de representaciones numéricas interpretables por la red. Los métodos con las que se pueden obtener estas representaciones se verán en la siguiente sección 3.2.2, pero se debe destacar que la correcta tokenización del texto fuente y la creación de un vocabulario abarcativo es una tarea primordial, ya que dichos métodos transforman a formato vectorial solamente los tokens que encuentran en el vocabulario confeccionado. En el caso que se hallen palabras que no estén incluidas en el vocabulario, no sería posible obtener representaciones numéricas de estas y, por ende, las entradas a la red serían incorrectas.

Para resolver esta problemática, los algoritmos bottom-up de tokenización, en lugar de definir tokens como palabras (separadas por reglas definidas o signos de puntuación) o caracteres, se utilizan los propios datos de entrenamiento para determinar automáticamente qué tokens deberían considerarse. A su vez, en lugar de obtener solamente palabras completas, los algoritmos de tokenización de esta clase suelen obtener tokens más pequeños que las palabras, denominados *subpalabras*. Una clase particular de las subpalabras son los *morfemas*, las unidades lingüísticas mínimas con significado de un idioma, por ejemplo, la palabra “atípico” contiene dos morfemas “a-” y “típico”. De esta forma, cuando se encuentran

palabras que no fueron asignadas a un token en particular, éstas pueden representarse usualmente como una secuencia de subpalabras (Jurafsky & Martin, 2024).

Como se verá en el algoritmo BPE que se encuentra a continuación, los esquemas de tokenización poseen dos partes: un *token learner* que se encarga de particionar los textos de entrenamiento en tokens con los cuales confecciona un vocabulario, y un *token segmenter* que toma oraciones de prueba y las segmenta utilizando los tokens que se encuentran en el vocabulario creado.

BPE: Byte-Pair Encoding

El token learner del algoritmo BPE comienza creando un vocabulario con todos los caracteres individuales, representando todas las palabras como secuencias de caracteres, entre ellos uno adicional de fin-de-palabra. Luego, analiza el conjunto de entrenamiento y por cada par de símbolos (‘A’, ‘B’) selecciona aquel con mayor frecuencia y lo añade como un nuevo símbolo unido ‘AB’ y reemplaza en el conjunto de entrada todas las ocurrencias contiguas de los símbolos ‘A’ y ‘B’ con el nuevo símbolo añadido. De esta forma, el algoritmo continúa contando las ocurrencias de símbolos contiguos y aquellos con mayor frecuencia son unidos y añadidos como un nuevo símbolo al vocabulario.

Éste proceso de unión se repite k veces, donde k es un parámetro del algoritmo. Finalmente, el tamaño del vocabulario final es igual al tamaño del vocabulario inicial de caracteres, sumado a los k nuevos tokens añadidos en las uniones.

> The main difference to other compression algorithms [...] is that our symbol sequences are still interpretable as subword units, and that the network can generalize to translate and produce new words (unseen at training time) on the basis of these subword units. [La diferencia principal con otros algoritmos de compresión [...] es que nuestras secuencias de símbolos aún son interpretables como unidades de subpalabras, y que la red puede generalizar al traducir y producir nuevas palabras (no vistas durante el entrenamiento) basándose en estas unidades de subpalabras] (Sennrich et al., 2016, p.4)

El siguiente Algoritmo 3 muestra de forma sintetizada el funcionamiento de este método de tokenización:

Algoritmo 3: Tokenización Byte-Pair Encoding

```
Inicializar el vocabulario V con todos los caracteres en el corpus C.
Inicializar valor de la cantidad de uniones k.
for i = 1 to k
   tL , tR ← El par de tokens más frecuentes contiguos en C
   tNEW ← concatenación de tL, tR
   V ← V + tNEW    # Añadir el nuevo token al vocabulario
   Reemplazar todas las ocurrencias contiguas de tL y tR en C con tNEW
retornar V
```

3.2.2 Aprendizaje de características - Embeddings

Al inicio de la presentación del aprendizaje automático en el Capítulo 2 se mencionó la importancia de la técnica de *feature learning* que consiste en hallar formas precisas de representar la información de entrada en modo correcto, incluyendo los aspectos o características (*features*) relevantes para la resolución de una tarea particular de machine learning. En el caso del procesamiento del lenguaje natural, el desafío se centra en hallar formas de representación del significado de las palabras o símbolos de las oraciones, mapeándolos a algún medio que contenga su significado. En especial, se debe destacar la dificultad de representar el significado de palabras que nunca se han visto con anterioridad.

La *hipótesis distribucional* es la teoría que hace posible el aprendizaje de la representación del significado de palabras (*representation learning*), incluso para aquellas nunca vistas, a partir del entrenamiento auto-supervisado con datos no etiquetados.

> Words that occur in similar contexts tend to have similar meanings. This link between similarity in how words are distributed and similarity in what they mean is called the distributional hypothesis. [Las palabras que aparecen en contextos similares tienden a tener significados similares. La relación entre la similitud en cómo las palabras están distribuidas y la similitud en su significado es llamada la hipótesis distribucional.] (Jurafsky & Martin, 2024, p.105)

El presente apartado tratará las ideas detrás de la semántica vectorial y cómo resuelve el aprendizaje de representaciones de significados de palabras con el uso de vectores d-dimensionales apodados *embeddings*. El nombre se debe al hecho de que cada palabra es "embebida" o inserta en un espacio vectorial continuo, donde aquellas con semántica similar se ubican más próximas entre sí. Estos vectores, al contener valores numéricos, sirven como entradas de gran contenido semántico a los modelos neuronales de aprendizaje profundo a la hora de resolver tareas de procesamiento de lenguaje natural.

Previo a la presentación de los métodos de construcción de los embeddings, debe ser puesta en relieve una métrica de similitud entre vectores que es utilizada en aplicaciones concretas de tareas de NLP: la similitud coseno (Jurafsky & Martin, 2024). Dados dos vectores v y w de dimensión d, se define el producto interno o producto punto entre ellos como la combinación lineal de sus componentes, en términos simbólicos:

$$v \cdot w = \sum_{i=1}^{d} v_i w_i = v_1 w_1 + v_2 w_2 + \cdots + v_d w_d$$

Es posible destacar que cuanto más similares sean los vectores, es decir, cuando posean valores grandes en las mismas componentes, el producto interno entre ellos será mayor. Esto es una posible métrica de similitud, pero debe notarse que beneficia a aquellos vectores que tengan una mayor norma o magnitud. Es por ello que, en ciertos casos, se normaliza el producto punto de los vectores, dividiéndolo por el producto de sus respectivas normas, obteniendo así el coseno del ángulo que se forma entre ellos en el espacio vectorial d-dimensional:

$$cos(\theta) = \frac{v \cdot w}{|v||w|} = \frac{\sum_{i=1}^{d} v_i w_i}{\sqrt{\sum_{i=1}^{d} v_i^2}\sqrt{\sum_{i=1}^{d} w_i^2}}$$

Representaciones vectoriales estáticas

La idea detrás de la semántica vectorial es construir representaciones de palabras como puntos en un espacio semántico multidimensional que se derivan de las distribuciones de las palabras en distintos contextos. El modelo ampliamente utilizado *word2vec* (Mikolov et al., 2013a) (Mikolov et al., 2013b) hace posible la construcción de los vectores *embeddings* de forma eficiente, representando las palabras de forma compacta debido a la pequeña dimensión de estos vectores. En lugar de utilizar vectores de V dimensiones, donde V es el vocabulario completo a considerar (cuyo cardinal puede ser muy superior a 50.000), *word2vec* hace uso de vectores con una cantidad de componentes considerablemente más pequeña, entre 50 y 1000, consiguiendo representaciones *densas*, donde la mayoría de las componentes de los vectores son valores reales no nulos. (Jurafsky & Martin, 2024)

Utilizar vectores densos de dimensiones relativamente pequeñas tiene un gran impacto en los casos empíricos de resolución de tareas de procesamiento de lenguaje natural, capturando de forma más efectiva la sinonimia de las palabras. El manejo de un menor número de parámetros hace posible también que los modelos de redes neuronales puedan generalizar mejor ante la aparición de palabras nunca vistas pero similares en términos de semántica vectorial a otras introducidas con anterioridad.

Word2vec es un método que computa embeddings estáticos, es decir, produce representaciones vectoriales para cada palabra de un vocabulario dado pero que no cambian para cada contexto distinto donde ocurren. A continuación se introducirá una forma particular de implementación del método word2vec, conocido como *skip-gram* y particularmente con una técnica de aproximación de probabilidades llamada *muestreo negativo*. A esta implementación se la conoce como *skip-gram with negative sampling* o, con su acrónimo, SGNS.

Intuitivamente, el método word2vec, entrena una clasificador de regresión logística (más simple que una red neuronal multicapa) que resuelva la siguiente tarea de clasificación binaria: ¿Es probable que la palabra objetivo w ocurra cerca de una palabra v? En verdad, el objetivo final de utilizar este clasificador no es para resolver la tarea de predicción de palabras en un contexto, sino de aprender los pesos durante el entrenamiento del clasificador para utilizarlos como los embeddings finales de las palabras. Luego, al ser posible introducir documentos o ejemplos completos como datos de entrenamiento para aprendizaje autosupervisado, resulta innecesario realizar el etiquetado manual de los datos de entrada.

El algoritmo comienza asignando valores aleatorios a los embeddings de cada una de las n palabras del vocabulario dos matrices de distintas, W y C, cuyas dimensiones son $|V| \times d$. El clasificador modifica iterativamente los valores de las componentes de estas matrices con el fin de construir los embeddings finales para las palabras del corpus de entrenamiento. Para cada palabra objetivo (target) w, se selecciona una ventana de L palabras de contexto anteriores y posteriores a w. A estas palabras que conforman el contexto cercano de la palabra objetivo se las denomina ejemplos *positivos*. El SGNS también considera k

ejemplos negativos para cada uno de los positivos seleccionados, es decir, palabras aleatorias que no forman parte del contexto para lograr entrenar al clasificador. Cada palabra posee dos representaciones vectoriales distintas, una en W cuando se toma como palabra objetivo y otra en C cuando se la utiliza como parte del contexto. Luego se modifican iterativamente los embeddings de cada palabra w con el fin de obtener un vector que sea similar a aquellos de las palabras del contexto de w y distinto a las palabras que no ocurren en su entorno.

De esta forma, el objetivo del algoritmo de aprendizaje SGNS consiste en:

- Maximizar la similitud entre los pares de palabras objetivo y de contexto (w, c_{pos}) de los ejemplos positivos
- Minimizar la similitud entre los pares de palabras objetivo y ejemplos negativos aleatorios (w, c_{neg})

Para definir la función de error que se debe minimizar es necesario expresar la forma en la que se computan los valores de las probabilidades de que una palabra c sea verdaderamente parte del contexto de la palabra objetivo w, es decir, $P(+|w,c)$. Si se recuerda la hipótesis distribucional que afirma que una palabra es similar a aquellas de su entorno, es posible calcular dicha probabilidad al tomar al producto interno entre vectores como métrica de similitud. Dado que este producto toma cualquier valor de la recta de los reales, es necesario llevar el resultado a una probabilidad, con lo cual se utiliza la función *logística* que tiene la forma:

$$\sigma(x) = \frac{1}{1+e^{-x}} \qquad \sigma : \mathbb{R} \longrightarrow [0,1]$$

De esta manera, se puede obtener $P(+|w,c)$ utilizando la siguiente ecuación:

$$P(+|w,c) = \sigma(w \cdot c) = \frac{1}{1+e^{-(w \cdot c)}}$$

Luego, se puede definir la probabilidad del evento negativo como su complemento:

$$P(-|w,c) = 1 - P(+|w,c) = \sigma(-w \cdot c) = \frac{1}{1+e^{(w \cdot c)}}$$

Finalmente, el skip-gram entrena al clasificador que asigna una probabilidad de que una palabra objetivo w ocurra dentro de una ventana de contexto $c_1, \ldots, c_L$ basado en la similitud existente entre w y el contexto. El skip-gram realiza la suposición de que todas las palabras del contexto son independientes por lo que se obtiene que la probabilidad de w de aparecer en el contexto se define por medio del siguiente producto:

$$P(+|w, c_1, \ldots, c_L) = \prod_{i=1}^{L} P(+|w, c_i) = \prod_{i=1}^{L} \sigma(w \cdot c_i)$$

Esta probabilidad es utilizada en la función de entropía cruzada la cual se busca minimizar para obtener los embeddings finales.

Representaciones vectoriales dinámicas

Un aspecto que no contemplan las representaciones vectoriales estáticas del apartado anterior es la potencial variación del significado de una misma palabra dependiendo del contexto en el que se utilice, lo cual se puede atribuir a la propia naturaleza de los idiomas y la polisemia existente en ellos (cuando una palabra o signo lingüístico tiene varias acepciones). Por ejemplo, la palabra "planta" en las siguientes oraciones podría interpretarse de formas muy distintas:

- La *planta* roja del jardín se ve hermosa. (Organismo vegetal)
- Luego de jugar al tenis me duele la *planta* del pie. (Parte del cuerpo)
- El departamento A1 se encuentra en *planta* baja. (Nivel de un edificio)
- La nueva *planta* de producción no es eficiente. (Edificación de una de fábrica)

Resulta evidente cómo "planta" recibe significados muy variados dependiendo de las palabras por las que se encuentra rodeada. Lograr representaciones que reflejen estos cambios puede resultar primordial a la hora de buscar soluciones precisas al resolver ciertas tareas de procesamiento de lenguaje natural.

A modo de introducción conceptual resulta importante destacar la posibilidad de obtener representaciones vectoriales dinámicas de los tokens de un texto. A diferencia de aquellos embeddings estáticos presentados en el apartado anterior, donde se posee un único vector por cada palabra del vocabulario, los *embeddings contextuales* pueden representar los significados de las palabras según el contexto en el que se encuentren, para utilizarlo en tareas que lo requieran.

> Where static embeddings represent the meaning of word types (vocabulary entries), contextual embeddings represent the meaning of word instances: instances of a particular word type in a particular context. [Mientras los embeddings estáticos representan el significado de tipos de palabras (elementos de un vocabulario), los embeddings contextuales representan el significado de instancias de palabras: instancias de un tipo de palabra particular en un contexto en particular.] (Jurafsky & Martin, 2024, p.250)

En términos de aprendizaje profundo, se puede pensar a las secuencias de salidas de los modelos de lenguaje como confecciones de embeddings contextuales para cada token de la entrada. Formalmente, dada una secuencia de tokens de entrada $x_1, \ldots, x_n$, se pueden utilizar los vectores de la salida z_i como vector contextual del token x_i que contiene información del significado de éste en el contexto particular de la secuencia de entrada.

En la sección 3.4.1 se presentará el modelo transformer BERT, cuya arquitectura utiliza este concepto de representaciones dinámicas de los tokens que se sirven de entrada.

3.3 Transformers

A partir de su surgimiento en el año 2017 de la mano de investigadores de Google, los modelos de aprendizaje profundo basados en la arquitectura Transformer probaron ser una evolución eficiente de los modelos predecesores como las redes recurrentes y convolucionales. Si bien no buscan suplantar el rol de estos otros modelos, los Transformers mostraron resultados muy satisfactorios ante la resolución de tareas de procesamiento de secuencias, en particular de texto natural.

> The ability to capture long-range dependencies and parallelizable architecture of the model made it successful in various NLP applications [...]. Since the development of the Transformer model, researchers have built on top of the Transformer, developing more advanced language models. [La capacidad de capturar dependencias en largas distancias y la arquitectura paralelizable del modelo, lo han convertido en un éxito en múltiples aplicaciones de NLP [...]. Desde el desarrollo del modelo Transformer, los investigadores han construido por encima del Transformer, desarrollando modelos de lenguajes más avanzados] (Fan et al., 2023, p. 3)

Los modelos de lenguaje grandes que son ampliamente conocidos al día de hoy, como GPT (Radford et al., 2018), BERT (Devlin et al., 2018) y T5 (Raffel et al., 2023), utilizan a los transformers (con sus respectivas modificaciones) como estructura subyacente para la resolución eficiente de tareas como traducción automática, etiquetado de secuencias, detección de spam, análisis de sentimiento, creación automática de resúmenes, sistemas conversacionales, entre muchas otras.

Las siguientes dos subsecciones 3.3.1 y 3.3.2 tratarán los conceptos del modelo encoder-decoder y el mecanismo de atención desde una perspectiva introductoria, basándose en la arquitectura de redes neuronales recurrentes. A pesar de no ser representativos del estado del arte, se verán como un primer acercamiento a los fundamentos de la arquitectura más compleja y actualizada de los transformers. Por último, la subsección 3.3.3 presentará la arquitectura Transformer y sus componentes, denotando las variaciones que utilizan del modelo encoder-decoder y el mecanismo de atención para aumentar el rendimiento de entrenamiento e inferencia.

3.3.1 Modelo Encoder-Decoder

Al inicio del presente capítulo se ha presentado el funcionamiento de los modelos de lenguajes neuronales y se ha destacado el uso de las RNNs en tareas de procesamiento de lenguaje natural como la generación de texto y el etiquetado de secuencias.

En el año 2015 se introdujo un nuevo modelo basado en redes neuronales recurrentes con una aplicación a la traducción automática neuronal (neural machine translation), apodado *Encoder-Decoder* (Cho et al., 2014). Este modelo, en general, es usado para la generación de secuencias de símbolos de entrada con una longitud variable condicionada por otra secuencia de otra longitud diferente. Es por esta

misma razón que pueden ser aplicados muy satisfactoriamente a tareas como la traducción automática, donde no siempre las oraciones a traducir tienen la misma cantidad de palabras en un idioma que en otro. En resumen, los autores Cho et al., 2014 definen al encoder-decoder como:

> [...] a novel neural network architecture that learns to *encode* a variable-length sequence into a fixed-length vector representation and to *decode* a given fixed-length vector representation back into a variable-length sequence. [[...] una novedosa arquitectura de red neuronal que aprende a *codificar* una secuencia de longitud variable en una representación vectorial de longitud fija y a *decodificar* una representación vectorial de longitud fija dada de vuelta en una secuencia de longitud variable] (Cho et al., 2014, p.2)

La estructura básica de esta red neuronal encoder-decoder, a veces denominada modelo secuencia-a-secuencia (seq2seq), está formada por tres componentes principales:

- ***Encoder (codificador)***: es una red neuronal que recibe una secuencia de símbolos $x_1, \ldots, x_n$ de manera sucesiva, modificando los estado internos de la red para finalmente formar un vector c llamado contexto que representa un resumen de la secuencia de entrada. Esta porción del modelo puede implementarse utilizando RNN, redes convolucionales o transformers (como se verá más adelante).

- ***Vector de contexto***: es el vector c obtenido como resultado final de aplicar una función sobre las representaciones contextuales intermedias durante el proceso de codificación de la secuencia.

- **Decoder (decodificador)**: es otra red neuronal que acepta el vector de contexto c codificado previamente y genera una secuencia de longitud variable de estados ocultos $h_1, \ldots, h_n$a partir de los cuales aprende a predecir la secuencia correspondiente de salida $y_1, \cdots, y_n$. Así como los encoders, los decoders pueden implementarse por medio de cualquier arquitectura neuronal de procesamiento secuencial.

De forma similar a la generación autorregresiva de los modelos neuronales de lenguaje, el encoder-decoder genera el siguiente símbolo o palabra de la secuencia al construir una distribución de probabilidades y seleccionar aquella que sea más probable de ocurrir luego del contexto provisto. En particular, el modelo encoder-decoder para la traducción toma como entrada a la secuencia x y luego, en cada paso de la generación, considera también como entrada a la palabra o símbolo previo generado.

Dada una secuencia de entrada x y una de salida , son concatenadas con un símbolo separador, por ejemplo $\langle s \rangle$ y se sirven como entrada y al encoder-decoder para su entrenamiento. Primeramente la secuencia x atraviesa la red del encoder, obteniéndose el estado oculto final denominado contexto $h_n^e = c$ que se interpreta como una representación contextualizada de la secuencia de entrada. Este vector es luego pasado al decoder para inicializar el primer estado oculto de su red y, a su vez, se sirve como parámetro en cada estado oculto de la etapa de decodificación.

El decoder, por su parte, actúa similar a modelos neuronales de lenguaje previamente introducidos, generando símbolos o palabras de forma autorregresiva de a uno a la vez, sirviéndose como entrada al estado oculto del paso previo (en caso del primer estado oculto es $h_0^d = c$), el contexto c obtenido del encoder y el elemento anterior en la secuencia (durante la inferencia, el elemento es el generado anteriormente ya que no habría un texto de referencia fuera del entrenamiento). De esa forma, el estado oculto en el paso de tiempo t se define como:

$$h_t^d = (\hat{y}_{t-1}, h_{t-1}^d, c)$$

Finalmente, los valores de salida y se obtienen al calcular una distribución de probabilidades con una función softmax sobre todos los valores del vocabulario V. El resultado final inferido es aquel que obtenga la mayor probabilidad de entre todas las posibles salidas:

$$\hat{y}_t = argmax_{w \in V} P(w|y_1, \ldots, y_{t-1}, x)$$

La siguiente Figura 3.5. muestra gráficamente el comportamiento y estructura de un modelo encoder-decoder:

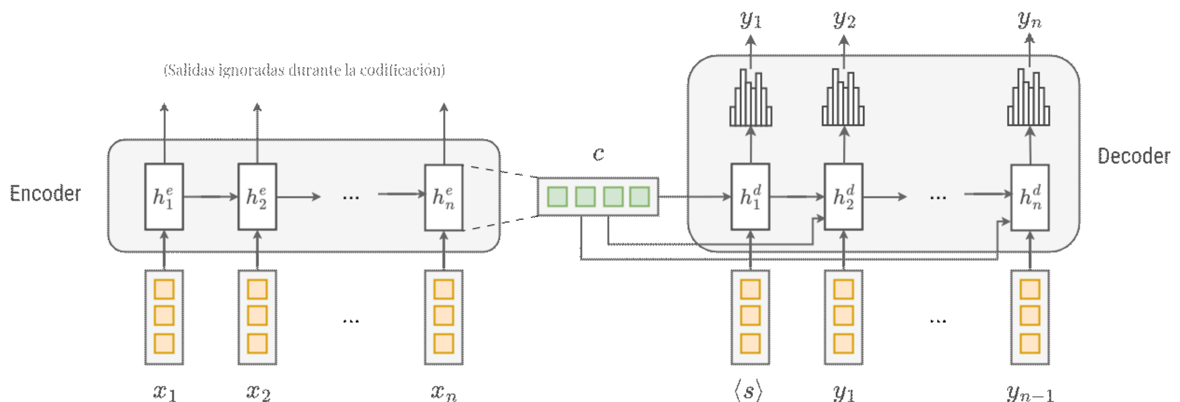

Figura 3.5: *Modelo Encoder-Decoder basado en red neuronal recurrente con estados ocultos.*

3.3.2 Mecanismo de atención

Dado que el vector de contexto funciona como el punto intermedio de conexión entre el módulo de codificador y decodificador del encoder-decoder, debe representar toda la información de la secuencia de entrada para una generación satisfactoria de la secuencia de salida. Esto convierte al contexto c en un "cuello de botella" donde se concentra toda la información posible de la entrada y es la única referencia de ella en el proceso de decodificación. A su vez, en secuencias cuya longitud es considerablemente grande, el vector de contexto no logra representar efectivamente toda la información que se sirvió en la secuencia de entrada. (Jurafsky & Martin, 2024)

Por esta razón, en el artículo de Bahdanau et al., 2014, se introdujo el denominado *mecanismo de atención* que es utilizado actualmente, con algunas modificaciones, en los modelos de transformers que se

verán en secciones posteriores. El mecanismo de atención reemplaza el único vector de contexto *c* por un conjunto de vectores $c_1, \ldots, c_n$, uno para cada elemento de salida del decodificador. Estos vectores son calculados por medio de una suma pesada entre los estados ocultos h_j^e de cada paso *t* en la secuencia de entrada y un *peso de atención* α_{ij}. Este peso puede interpretarse como un puntaje de cuánta atención debe ponerse en el elemento *j* de la entrada para generar el elemento *i* de la salida.

> The decoder decides parts of the source sentence to pay attention to. By letting the decoder have an attention mechanism, we relieve the encoder from the burden of having to encode all information in the source sentence into a fixed-length vector. [El decodificador elige las partes de la oración de entrada a las cuales prestarles atención. Permitiendo que el decodificador tenga un mecanismo de atención, se alivia al codificador la carga de tener que codificar toda la información de la oración de entrada en un vector de longitud fija.] (Bahdanau et al., 2014, p.4)

Para calcular cada vector de contexto c_i el decodificador necesita determinar cuánto enfoque debe poner en cada uno de los estados del encoder, definiendo cuán relevante es cada uno de estos al estado oculto previo del decoder h_{i-1}^d. Para ello se calculan los pesos de atención α_{ij} mediante una función de similitud entre el estado del decoder y cada uno de los estados de encoder. Los puntajes de atención pueden computarse utilizando el producto punto entre los vectores o incluso con una función más sofisticada (llamada, por ejemplo, *puntaje*) basada en una red neuronal denominada *modelo de alineamiento* (Bahdanau et al., 2014) que aprende una matriz W_s de pesos durante el proceso de entrenamiento del propio modelo encoder-decoder.

Finalmente, los puntajes de atención α_{ij} son normalizados con una función softmax y el vector de contexto c_i se calcula de la forma:

$$c_i = \sum_j \alpha_{ij} h_j^e$$

donde

$$\alpha_{ij} = softmax(puntaje(h_{i-1}^d, h_j^e))$$

Una forma intuitiva de comprender el comportamiento de este mecanismo de atención, y que será utilizada al presentar los transformers, es pensarlo como ejecutar una consulta a una memoria de pares clave-valor (Eisenstein, 2019). El estado oculto del decoder toma la forma de la consulta, ya que es aquello por lo cual se desea obtener los valores de atención. Luego, los pesos de atención son obtenidos por medio de una función *puntaje* como una medida de compatibilidad entre la consulta y los estados ocultos en la “memoria” del encoder que determinan las claves y los valores. Por último se obtiene el vector de contexto al multiplicar los pesos de atención con cada uno de los valores (los estados ocultos del encoder) y sumarlos. La Figura 3.6 muestra gráficamente el mecanismo para generar cada contexto c_i desde esta perspectiva de consulta, clave y valor.

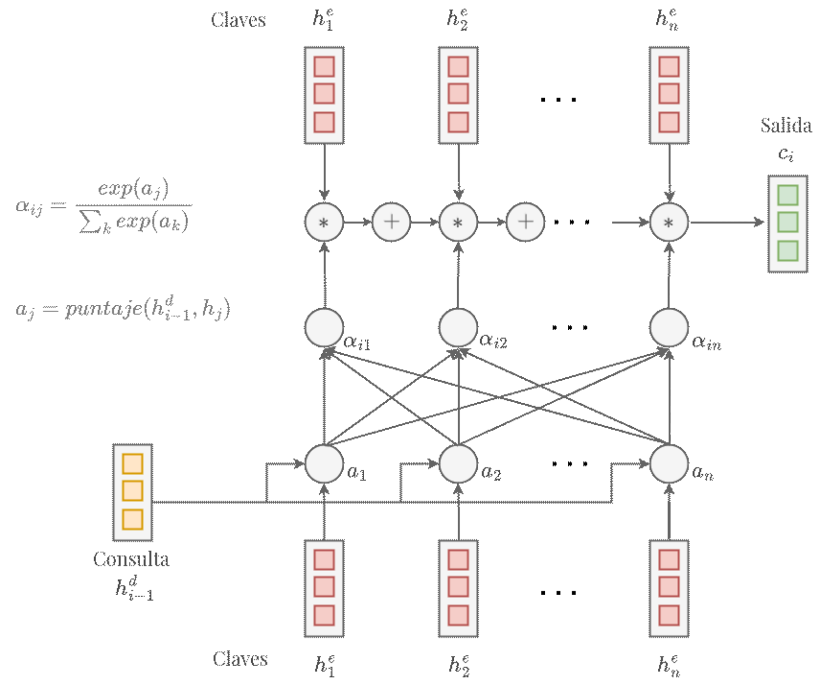

Figura 3.6: *Funcionamiento del mecanismo de atención desde la perspectiva de vectores consulta, clave y valor*

3.3.3 Arquitectura Transformer

A pesar de incluir un mecanismo de atención que solucione la pérdida de información al comprimir toda la secuencia de entrada en un único vector de contexto de longitud fija, las redes neuronales recurrentes poseen problemas críticos durante la etapa de entrenamiento e inferencia. Entre estos problemas se pueden resaltar primeramente el gradiente explosivo y desvaneciente ocasionado por la recursividad de los estados ocultos, y segundo, por la dificultad de paralelización de la arquitectura debido a la dependencia de información entre los distintos pasos en el tiempo de la secuencia.

En 2017 tuvo lugar la publicación del artículo "Attention is All you Need" (Vaswani et al., 2017) que presentó una novedosa arquitectura de redes neuronales denominada *Transformer* basada en el modelo encoder-decoder previamente descripto. El objetivo de esta nueva estructura fue solucionar las limitaciones computacionales de los modelos basados en RNN, permitiendo paralelizar el proceso de cómputo del entrenamiento e inferencia y capturar con precisión la información de secuencias de longitud variable y extensa. En el artículo los autores presentaron la siguiente imagen 3.7 que describe la disposición de los distintos componentes y capas principales de la arquitectura, los cuales serán profundizados en las posteriores subsecciones de este apartado.

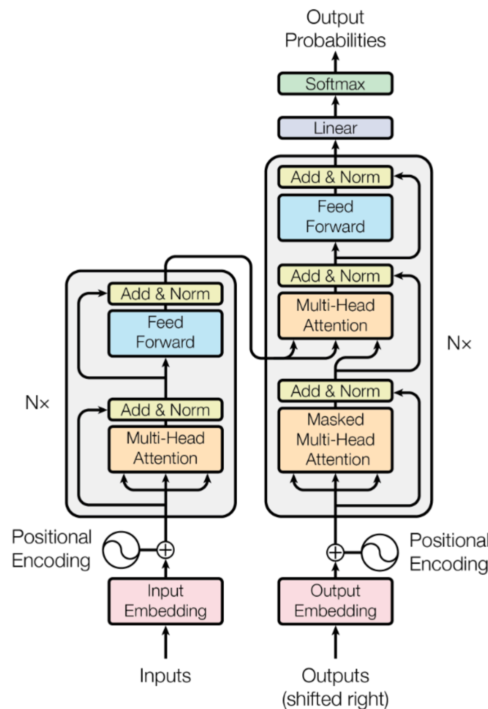

Figura 3.7: Representación gráfica de la arquitectura neuronal Transformer presentada en el artículo "Attention Is All You Need" (p. 3), por Vaswani et al., 2017.

En términos resumidos, un Transformer utiliza "pilas" de bloques fundamentales que contienen capas de sub-redes neuronales. Combinando secuencialmente estos bloques se constituye el encoder (codificador) y decoder (decodificador) del transformer que procesan la secuencia de entrada para obtener otra de salida, generando representaciones vectoriales intermedias con alto contenido semántico y contextual.

A diferencia de las redes neuronales recurrentes, los transformers no utilizan conexiones recurrentes con el fin de procesar información secuencial, lo que permite aumentar su rendimiento y escalarlos mediante paralelización. Este apartado proveerá una introducción a los conceptos fundamentales que forman parte de la arquitectura de transformers, entre ellos las representaciones vectoriales de las palabras o tokens de las secuencias, los bloques de encoders y decoders y los novedosos mecanismos de self-attention y multihead-attention característicos de esta arquitectura.

Representaciones vectoriales - Embeddings y Positional Encodings

Todo elemento o token de una secuencia que se sirva de entrada al Transformer, al ser éste una red neuronal, deben ser representados con valores numéricos. En particular este tipo de arquitectura considera dos representaciones vectoriales distintas para cada elemento de la secuencia:

- ***Embedding del token***: A cada token de la secuencia le corresponde una representación vectorial embedding, cuya definición fue introducida en la sección anterior 3.2.1. El objetivo de esta representación es embeber la semántica de cada elemento en un vector con valores numéricos

que pueden obtenerse por medio de algoritmos como el mencionado *word2vec*. Todos los embeddings del vocabulario son almacenado en una matriz E de dimensiones $|V| \times d$ donde cada fila i representa el embedding del i-ésimo token del vocabulario. Al ingresar los elementos de la secuencia de entrada de longitud máxima N, son multiplicados los vectores one-hot de cada uno de ellos con la matriz E para obtener otra con $W \in \mathbb{R}^{N \times d}$ solo los emb eddings de los tokens del contexto particular presentado.

- ***Codificación posicional***: Dado que todos los vectores embeddings de los tokens de la secuencia ingresan a la red transformer en forma simultánea, es necesario indicar para cada uno de ellos su posición dentro de la secuencia. Es por ello que se introduce un nuevo vector para cada elemento, que contiene información sobre la posición donde se encuentra, denominada *postional encoding* (*codificación posicional*). Si bien existen múltiples formas de definirlos, en el artículo original (Vaswani et al., 2017) sus componentes se especifican de la siguiente forma:

$$P_{(pos,2i)} = sen\left(\frac{pos}{10000^{\frac{2i}{d_{model}}}}\right)$$

$$P_{(pos,2i+1)} = cos\left(\frac{pos}{10000^{\frac{2i}{d_{model}}}}\right)$$

  Cada dimensión $i \in [0, d_{model}/2]$ del vector representante del token en la posición pos es el resultado de una función trigonométrica seno o coseno. Esta forma de computar las codificaciones posicionales permite obtener vectores que:

  - Sean únicos para cada posición en la secuencia.
  - La distancia entre los vectores posicionales correspondientes a elementos consecutivos es consistente a través de toda la secuencia de longitud variable.
  - Son independientes de la longitud de la secuencia.
  - Son calculados de forma determinística.

Todas las codificaciones posicionales son almacenadas en una matriz P de las mismas dimensiones que W Luego, para producir la entrada final al modelo ambas matrices son sumadas para obtener una única matriz $X = W + P$ que contiene en cada fila la representación vectorial de los elementos de la secuencia introducida al Transformer.

Capas de atención

Tanto los bloques de encoders como decoders hacen uso de un novedoso mecanismo de atención apodado *self-attention* (o inter-atención) y que funciona como bloque fundamental en la composición de las capas de atención de la arquitectura Transformer. En este apartado se presentará primeramente el funcionamiento interno de este mecanismo eficiente, para luego definir cómo se utiliza dentro de las distintas capas de atención que conforman la arquitectura.

Self-Attention

Los transformers tienen la particularidad de estar formados por múltiples bloques que construyen capas sucesivas, formadas por encoders y decoders que buscan enriquecer las representaciones de los significados de las palabras a medida que atraviesan la red. En cada capa se intenta construir una representación contextualizada de cada palabra o token w_i tomando en cuenta las representaciones formadas en la capa anterior y que se encuentran en la vecindad de w_i.

En lugar de tomar representaciones vectoriales estáticas para cada token como pueden ser inicialmente los embeddings de entrada, los transformers cuentan con un mecanismo que permite construir representaciones contextualizadas de cada token tomando en cuenta el contexto particular de éste en una secuencia dada. Este mecanismo se denomina *self-attention* (como será mencionado a través del presente trabajo) y consiste intuitivamente en determinar cuánto enfoque se debe poner en las representaciones de los tokens de la capa anterior $i - 1$ y cómo combinarlas para obtener la representación contextualizada a de la actual capa i. Cabe resaltar que el contexto a considerar en ciertos casos debe ser extenso, ya que algunas secuencias contienen elementos relacionados pero en posiciones distantes dentro de ellas. Por ejemplo, las estructuras que definen el lenguaje natural como el español hacen posible que palabras relacionadas que coreferencian al mismo concepto se encuentren lejanas dentro de una misma oración.

> The core intuition of attention is the idea of comparing an item of interest to a collection of other items in a way that reveals their relevance in the current context. [La intuición central de la atención es la idea de comparar un elemento de interés con una colección de otros elementos de forma de develar su relevancia en el contexto actual.] (Jurafsky & Martin, 2024, .217)

En el artículo de Vaswani et al., 2017, se muestran imágenes similares a la Figura 3.8 que describen el efecto de implementar un mecanismo de atención self-attention en la construcción de representaciones vectoriales dentro de los transformers. Cuando la secuencia atraviesa el proceso de self-attention se definen para cada token la atención que debe prestar a cada uno de los elementos de su vecindad con el fin de construir una representación semántica válida.

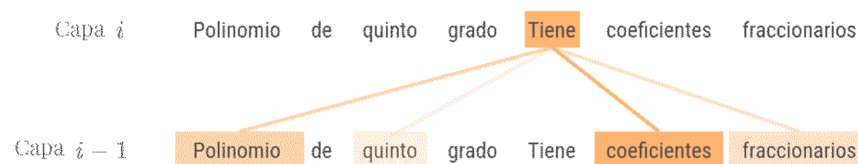

Figura 3.8: *Ejemplo ilustrativo del funcionamiento del mecanismo de self-attention para un elemento de una secuencia de palabras en español. Considerando a la palabra objetivo ``Tiene" se indica el enfoque que se toma sobre otras cuatros palabras de la secuencia en la anterior capa. La intensidad del color indica el nivel de atención.*

Self-attention toma las bases del mecanismo de atención introducido en la subsección anterior 3.3.2, pero difiere en dos grandes aspectos: primeramente no toma los estados ocultos del decoder y encoder para determinar cuán relevantes son estos últimos al primero, sino que considera las propias representaciones vectoriales previas de los tokens en las sucesivas capas del transformer. Por otro lado, permite crear representaciones más sofisticadas de los tokens al utilizar tres matrices diferentes por capa que capturan representaciones de un mismo elemento en tres posibles roles que pueden tomar: consulta (query), clave (key) o valor (value). (Jurafsky & Martin, 2024)

Dado un vector de entrada que representa una palabra o token, el mecanismo de self-attention considera las siguientes tres proyecciones de este vector según el rol que cumpla:

- Consulta (Query): $q_i = x_i W^Q, \quad W^Q \in \mathbb{R}^{d \times d_k}$
- Clave (Key): $k_i = x_i W^K, \quad W^K \in \mathbb{R}^{d \times d_k}$
- Valor (Value): $v_i = x_i W^V, \quad W^V \in \mathbb{R}^{d \times d_v}$

donde d es la dimensión del vector x_i , d_k la dimensión de los vectores consultas y claves y d_v la dimensión de los vectores valor.

Luego, dado un vector foco x_i y su respectiva proyección como consulta q_i, se computa un puntaje de cuánta atención mantener sobre los contextos previos x_j. El mecanismo de self-attention utiliza el producto punto entre los vectores q_i y k_j como medida de *puntaje* de atención pero, debido a que éste puede tomar valores arbitrariamente grandes, el resultado del producto es dividido por un factor relacionado al tamaño de los embeddings. Usualmente se utiliza como factor a la raíz cuadrada de la dimensión de los vectores de las claves o consultas d_k , obteniendo:

$$puntaje(x_i, x_j) = \frac{q_i \cdot k_j}{\sqrt{d_k}}$$

De forma similar que en el mecanismo de atención de la sección 3.3.2, se computan los pesos de atención en función de dichos puntajes aplicando una función softmax:

$$\alpha_{ij} = softmax(puntaje(x_i, x_j))$$

Finalmente, el vector de salida de este mecanismo self-attention a_i se obtiene al realizar la suma pesada entre los pesos de atención α_{ij} y los vectores de valor v_j:

$$a_i = \sum_j \alpha_{ij} v_j$$

Una particularidad de este mecanismo de atención utilizado por los transformers es su capacidad de ser paralelizado. Al compactar todos los embeddings de la secuencia de entrada en una única matriz *X* es posible multiplicarla con cada matriz de proyección W^Q, W^K y W^V para obtener todos los vectores de consulta, claves y valores respectivamente en tres matrices:

$$Q = XW^Q, \qquad K = XW^K, \qquad V = XW^V$$

De esta forma es posible reducir todo el proceso de self-attention a un solo cómputo entre estas matrices altamente paralelizable:

$$A = SelfAttention(Q, K, V) = softmax\left(\frac{QK^\top}{\sqrt{d_k}}\right)V$$

Multihead-Attention

Conociendo el concepto y funcionamiento del mecanismo de self-attention característico de la arquitectura de los transformers, debe destacarse que, en lugar de computar una sola función de self-attention por *capa de atención*, estos modelos realizan h proyecciones distintas de los vectores de entrada a vectores de consulta, clave y valor, utilizando $3 \times h$ matrices de proyección distintas. Es decir, los transformers aplican múltiple veces el mecanismo de atención al mismo conjunto de entrada $X \in \mathbb{R}^{N \times d}$, pero utilizando matrices de W_i^Q, W_i^K W_i^V $i = 1, \ldots, h$ proyección diferentes y donde .

Este novedoso mecanismo denominado *multihead-attention* o *atención de cabezas múltiples* permite a estos modelos de procesamiento secuencial obtener representaciones de las entradas que consideran múltiples relaciones de cada elemento con respecto a todo el resto. Por ejemplo, una palabra puede estar simultáneamente relacionada sintáctica y semánticamente con otra. Dado que resulta poco efectivo utilizar solo un modelo de self-attention para representar todas las relaciones existentes entre los elementos, los transformers utilizan el mecanismo de multihead-attention para resolver este problema.

Dentro de una misma capa de multihead-attention, se encuentran h capas de self-attention que se ejecutan en paralelo, obteniendo distintas salidas que son concatenadas en una única matriz de dimensión $N \times h.d_V$, y proyectadas a una salida final A de la dimensión esperada $N \times d$ por medio del producto con una matriz $W^O \in \mathbb{R}^{h.d_v \times d}$.

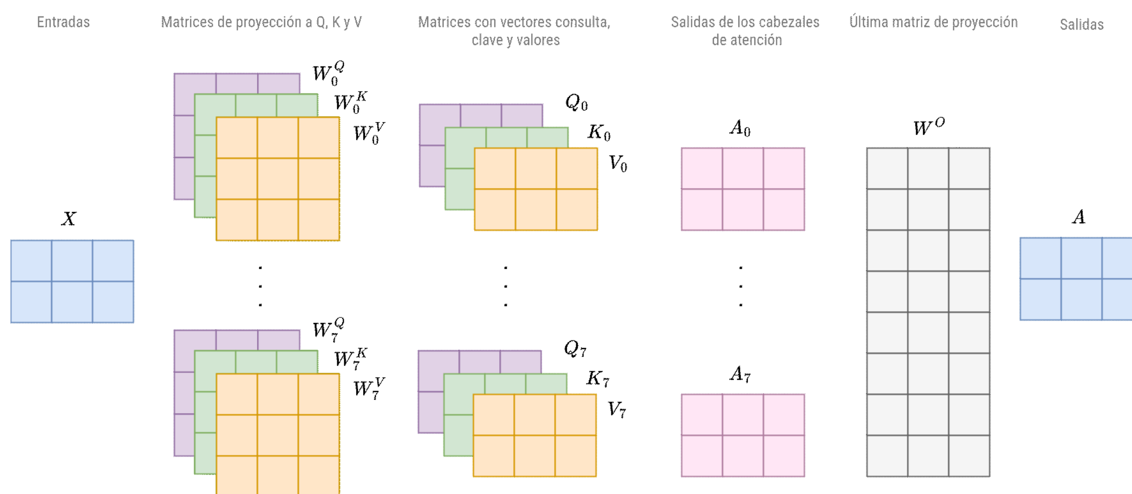

Figura 3.9: *Diagrama de matrices incluidas en el mecanismo multihead attention*

Multihead-Attention Enmascarado

Durante el entrenamiento de los transformers, se tiene acceso a todas las salidas verdaderas que deben ser generadas por los bloques del decoder, pero en múltiples tareas de procesamiento de lenguaje natural

como el modelado de lenguaje o traducción automática no resulta beneficioso que se le provea acceso a palabras futuras en la generación. Es por ello que en la primera capa de atención multihead, para la generación de un elemento objetivo w_i, solo se toman en cuenta los $i-1$ tokens previos a éste. Para lograr este efecto, se enmascara cada matriz $QK^\top$ de cada cabeza de atención al adicionarle una matriz máscara M que contiene $-\infty$ en aquellas posiciones de tokens futuros y 0 en las correspondientes a los tokens pasados (\cite{kamath-2022}). Esta leve modificación resulta en la siguiente forma de computar la atención multihead:

$$multiheadAttention(Q, K, V) = softmax\left(\frac{QK^\top + M}{\sqrt{d_k}}\right)V$$

Atención Encoder-Decoder

Si bien los bloques de encoders y decoders son similares en su composición, estos últimos poseen una capa adicional apodada capa de atención encoder-decoder que opera como una capa de multihead attention pero donde la atención se calcula sobre la salida del último bloque del encoder. Esto es necesario debido a que en cada paso de la secuencia de salida el decoder debe aprenderse las relaciones de atención que tiene el token de salida con todos los elementos de la entrada. Por ello los vectores de consultas son propios del decoder pero las llaves y valores de todos los elementos de la secuencia de entrada provienen del último bloque encoder.

Bloques del Transformer

Presuponiendo un entendimiento inicial de la capa de multihead-attention, el mecanismo de atención self-attention y la forma de las representaciones vectoriales de entrada de los transformers, es posible presentar la forma general de los bloques del encoder y decoder que conforman la estructura de esta arquitectura.

Todos los bloques de cada pila de encoder y decoder contienen dos capas base o sub-capas como las apodan lo creadores Vaswani et al., 2017:

- ***Capa de mecanismo multihead self-attention:*** Las representaciones vectoriales de cada token (o las salidas de un bloque anterior), ingresan a cada bloque atravesando una primera capa de atención de cabezas múltiples. En el caso de de los bloques del decoder, esta capa es enmascarada para evitar que en una posición se tome en cuenta elementos subsecuentes para la generación durante el entrenamiento.

- ***Capa feedforward***: Cada posición de la secuencia de salida de la capa de atención multihead atraviesa una red feedforward totalmente conectada con una sola capa oculta. Las matrices de pesos son compartidas entre todos los vectores de entrada en una misma capa pero varían en los distintos bloques.

A su vez, a de cada una de las sub-capas anteriores las "rodea" una *conexión residual* que pasa la información de una capa inferior a otra superior sin atravesar la capa intermedia. La adición de esta información con la salida de las sub-capas se sirve de entrada a una capa de normalización. En términos generales, la salida final de las sub-capas tiene la forma:

$$LayerNorm(x + SubLayer(x))$$

donde *SubLayer* representa alguna de las funciones de las sub-capas. La Figura 3.10 ilustra la organización de estas capas en los bloques que conforman la pila de encoder y decoder y sus diferencias.

Las capas de normalización intermedias facilitan el entrenamiento realizado por medio de métodos que utilizan descenso del gradiente, manteniendo los valores de los vectores de las salidas de cada capa en un rango aceptable. La normalización utilizada es la estandarización normal, pero aplicada a las componentes de un vector para las cuales se computa la media μ y la desviación estándar σ. La entrada de esta capa es un vector x de dimensión d el cual es reescalado utilizando la siguiente ecuación (estandarización):

$$\hat{x} = \frac{x - \mu}{\sigma}$$

Esto disminuye su norma pero mantiene su dimensión d (Jurafsky & Martin, 2024).

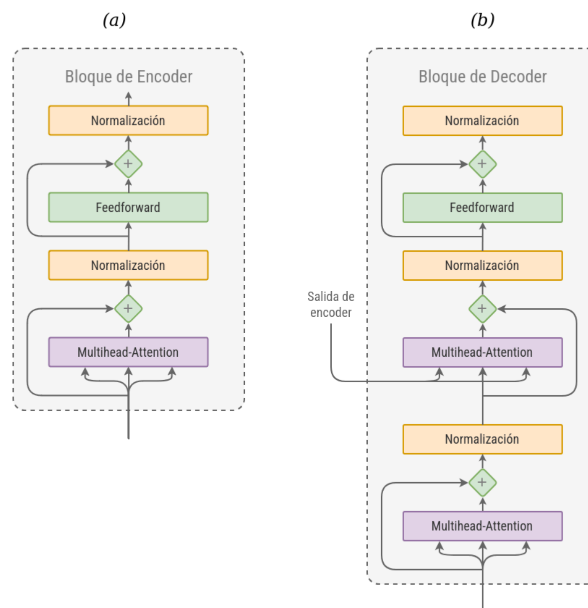

Figura 3.10: *Estructura en capas de cada bloque que conforma el encoder (a) o decoder (b) del Transformer.*

Si bien el transformer original presentado por (Vaswani et al., 2017) poseía un encoder con una pila de 6 bloques idénticos, al igual que el decoder, modelos más recientes utilizan muchas más capas de bloques:

> Transformers for large language models stack many of these blocks, from 12 layers (used for the T5 or GPT-3-small language models) to 96 layers (used for GPT-3 large), to even more for more recent models. [Los Transformers para modelos grandes de lenguaje apilan muchos de estos bloques, desde 12 (usados por los modelos de lenguaje T5 o GPT-3-small)

hasta 96 capas (usadas para GPT-3 large), o incluso más para modelos más recientes.] (Jurafsky & Martin, 2024, p.224)

En la sección que sigue a continuación se detallarán tres modelos de lenguaje que utilizan como base a la arquitectura Transformer y realizan algunas modificaciones en la estructura de las pilas de encoder y decoder.

3.4 Grandes modelos de lenguaje (LLMs)

Tomando la arquitectura base de los Transformers es posible especializar su comportamiento al añadir un *cabezal* dedicado a una tarea de procesamiento particular, incluyendo tareas de NLP. Esta nueva porción de conexiones neuronales que se inserta al final del Transformer puede conseguir el mismo funcionamiento de los modelos de lenguaje presentados en 3.1 al inicio del presente capítulo, al tomar la salida de la última capa para el último token N y utilizarlo para predecir la siguiente palabra en la posición N+1.

Al construir un modelo de lenguaje en base a la arquitectura Transformer, no solamente es posible predecir la ocurrencia de palabras como en los casos de los n-gramas o las redes feedforward, sino que el modelo es capaz de procesar una cantidad de entradas mucho mayor que sus predecesores (2048 a 4096 tokens en modelos grandes) y, utilizando el mecanismo de atención multicabezal, posibilita la obtención de información de una ventana de contexto más amplia. Ésto, sumado al alto grado de paralelismo que los caracterizan, convierten a esta arquitectura en una herramienta muy poderosa en la generación condicional, es decir, la tarea de producir texto de salida condicionado por otro de entrada.

La presente sección comenzará la introducción teórica del uso de los denominados grandes modelos de lenguaje (LLMs) con el apartado 3.4.1, donde se expondrá la definición de dos conceptos que tuvieron grandes efectos en el área del procesamiento de lenguaje natural: el *transfer-learning* y el *fine-tuning*. Posteriormente se presentarán tres casos de LLMs que han producido un impacto considerable en el campo de NLP y que, si bien comparten el hecho de basarse en la arquitectura Transformer, cada uno posee una disposición de sus componentes muy distinta. En la sección 3.4.2 se presentará a BERT, el LLM basado en un esquema de *solo-encoder* cuyo éxito "has led to its widespread adoption and other pre-trained language models" [ha conducido a su amplia adopción y de otros modelos de lenguaje pre-entrenados] (Fan et al., 2023, p.3). Por otra parte, en 3.4.3 se verá una descripción sintetizada de la familia de modelos GPT, los cuales siguen un esquema arquitectónico de *solo-decoder*. Finalmente, la sección 3.4.4 expondrá el modelo T5 introducido en 2023, sobre el cual se hará especial foco debido a su aplicación en la implementación del modelo final de la presente tesina. Se verá que la arquitectura de este último modelo respeta la composición original del Transformer presentado por Vaswani et al., 2017, es decir, un esquema *encoder-decoder*.

3.4.1 Transfer learning y Fine-tuning

La creciente cantidad de aplicaciones del aprendizaje automático fue acompañada por una dificultad común a todas ellas, la recolección de datos de entrenamiento. El aprendizaje semi-supervisado en ciertas tareas del procesamiento del lenguaje representó una solución parcial a esta problemática, ya que depende de un volumen pequeño de datos etiquetados para luego mejorar la exactitud sobre datos sin etiquetar. A pesar de ello existen casos en los que incluso resulta muy complejo reunir datos no etiquetados (Zhuang et al., 2020).

El aprendizaje por transferencia, o mejor conocido como *transfer learning*, es una método utilizado en el aprendizaje automático que consiste en tomar lo aprendido durante el desarrollo de una tarea y transferir dicho conocimiento a otro modelo para resolver una tarea similar o de un campo relacionado. Considerando una analogía, una persona puede transferir lo aprendido en una disciplina a otra similar siempre que cuente con el conocimiento necesario para generalizar su experiencia, por ejemplo aprender el idioma Italiano luego de saber Español, o aprender a tocar un segundo instrumento. Mantener un grado de similitud entre la tarea original y aquella a la que se transfiere el conocimiento puede conseguir un aumento en la performance de aprendizaje y una disminución en la cantidad de información necesaria para aprender correctamente la segunda tarea.

Un caso particular de transfer learning ampliamente utilizado en el campo del aprendizaje profundo es el denominado *fine-tuning*. Este es un proceso que toma una red neuronal previamente entrenada sobre un gran conjunto de datos, en su mayoría sin etiquetar, para luego aplicar otra fase de entrenamiento sobre datos específicos de una tarea de NLP. De forma intuitiva, la primera etapa de pre entrenamiento permite al modelo adquirir el conocimiento del lenguaje de forma general, lo cual hace posible acelerar el proceso de entrenamiento sobre la tarea específica y "afinar" *(fine tune)* el valor de los parámetros internos de la red neuronal.

3.4.2 Modelo BERT

Si bien los bloques de encoders originales vistos en la sección 3.3 permiten el procesamiento de los tokens de entrada de izquierda a derecha, utilizando mecanismos de self-attention causal, no resulta suficiente en tareas particulares como clasificación y etiquetado de secuencias, donde la información del contexto para cada token debería obtenerse de ambos lados, tanto a izquierda como derecha de éste.

El foco de los *encoder bidireccionales* se encuentra en obtener representaciones contextualizadas de los tokens de entrada, de la misma forma que se ha mencionado previamente en el apartado Representaciones vectoriales dinámicas de la sección 3.2.2. Los bloques de codificadores bidireccionales utilizan el mecanismo de self-attention sobre los embeddings $x_1, \ldots, x_n$ de una secuencia de entrada, para obtener otros vectores contextualizados $y_1, \ldots, y_n$ obtenidos al utilizar la información de toda la secuencia de entrada en dada y_i. La Figura 3.11 muestra este comportamiento bidireccional de forma gráfica, comparándolo al enfoque de atención causal original.

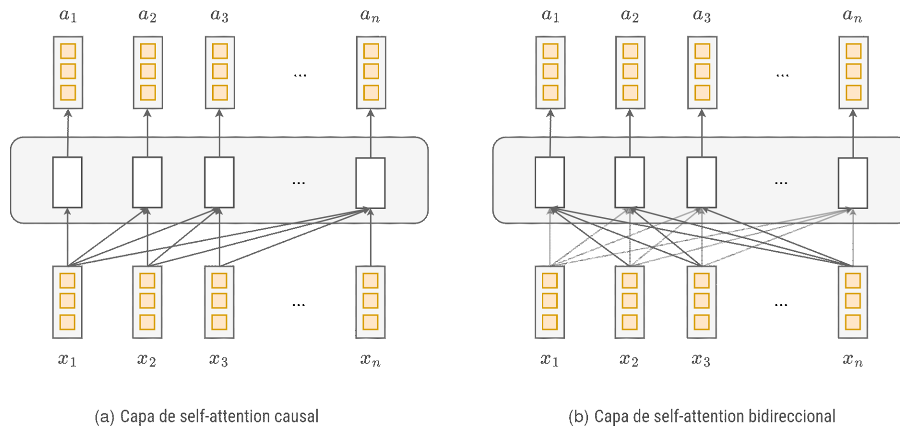

***Figura 3.11**: Diagrama del comportamiento del mecanismo de atención bidireccional.*

Para lograr estas representaciones con contexto bidireccional basta con la eliminación del enmascaramiento de los tokens futuros en la matriz de atención. Más allá de este pequeño cambio, los modelos con encoders bidireccionales poseen los mismos elementos que la arquitectura original de transformer, donde los textos de entrada son particionados utilizando tokenización en subpalabras y son combinados con las codificaciones posicionales que determinan la ubicación de estos en la secuencia.

Particularmente, este apartado destaca un modelo que simplemente posee una pila de bloques que forman un encoder bidireccional (es *encoder-only*) denominado BERT, o extensivamente Bidirectional Encoder Representations from Transformer (Devlin et al., 2018). Su introducción ha causado importantes efectos en la comunidad científica y de desarrolladores de modelos de aprendizaje profundo de NLP:

> The advent of Bidirectional Encoder Representations from Transformer (BERT) is considered the onset of a revolution in the field of Natural Language Processing (NLP). [El advenimiento de Bidirectional Encoder Representations from Transformer (BERT) es considerado el inicio de una revolución en el campo del Procesamiento de Lenguaje Natural (NLP)] (Kamath et al., 2022, p.43)

Concretamente la arquitectura original de BERT estaba constituída por capas ocultas con 768 neuronas cada una y 12 bloques de encoder con 12 cabezas de atención en cada capa de self-attention. Ésto resultaba en un modelo de alrededor de 100 millones de parámetros (pesos) actualizables durante el entrenamiento.

Para entrenar a BERT se utilizó la técnica de *modelado de lenguaje enmascarado* o *MLM*, donde se selecciona un subconjunto de palabras de cada secuencia de entrada, para las cuales se realiza una de las siguientes operaciones:

- Se reemplaza la palabra por un token máscara [MASK].
- Se reemplaza la palabra con otro token elegido desde el vocabulario al azar.
- Se mantiene la palabra sin cambios.

Luego, utilizando los bloques de encoders bidireccionales, la tarea de BERT era minimizar la función de error de Entropía Cruzada al predecir las palabras enmascaradas utilizando solamente el subconjunto de palabras seleccionado como fuente de aprendizaje. Debe resaltarse que la aplicación de esta técnica de MLM resultó necesaria debido a que la predicción de "la siguiente palabra" ya no tenía sentido al considerar todo el contexto, tanto pasado como futuro, durante el mecanismo de atención bidireccional.

3.4.3 Modelos GPT

A diferencia de BERT, la familia de los modelos de lenguaje GPT (Generative Pretrained Transformer) introducidos a lo largo de los años sigue una arquitectura similar al transformer pero solo considerando bloques de decoder, por lo que reciben el nombre de modelos *only-decoder*.

Desde la introducción del primer modelo GPT (Radford et al., 2018a) y particularmente de GPT-2 (Radford et al., 2018b) en 2018, el éxito y reconocimiento de esta familia ha crecido exponencialmente (Fan et al., 2023), así como también el número de parámetros incluidos en su arquitectura. El acercamiento de éstos modelos fue pre entrenar una red neuronal basada en un conjunto de bloques de decoder de transformer sobre un conjunto de datos sin etiquetar, es decir, utilizando aprendizaje no supervisado. En definitiva, Radford et al., 2018 demostraron que realizar esta clase de pre entrenamiento generativo sobre un gran volúmen de datos con texto sin etiquetar y a continuación afinar el modelo sobre una tarea específica resulta en un mejor desempeño que símplemente entrenar desde un inicio sobre dicha tarea específica.

Haciendo foco especialmente en el primer modelo de la familia, GPT fue entrenado en un esquema de dos etapas: Una primera fase de entrenamiento no supervisado sobre un gran corpus de documentos, consiguiendo un modelo agnóstico con respecto a las tareas de NLP. Posteriormente, para evaluar la mejoría aportada por la primera fase, se aplicó un entrenamiento de afinación o fine-tuning sobre cuatro tareas particulares de procesamiento de lenguaje, con su correspondiente objetivo supervisado y datos etiquetados.

En términos de los componentes arquitectónicos que utiliza la familia GPT, si bien todos se basan en una pila de solamente decoders de Transformer, a continuación se listan algunos de los aspectos que fueron cambiando a través de cada nueva publicación (Kamath et al., 2022):

- **GPT (Radford et al., 2018a)**: Basado en Transformer con 12 bloques decoder, una dimensión de 768 estados en las sub-capas y 12 capas de self-attention en los bloques. Las capas feedforward posicionales eran de dimensión 3072.

- **GPT-2 (Radford et al., 2018b)**: Si bien era muy similar al original GPT, éste cambiaba la disposición de las capas de normalización dentro de los bloques decoder. Se presentaron 4 variantes de GPT-2 con distintos tamaños, donde el más grande tenía 48 bloques en el decoder y un total de alrededor 1.5 mil millones de parámetros entrenables.

- **GPT-3 (Brown et al., 2020)**: Con algunos cambios en las capas de self-attention con respecto a GPT-2, al alternar patrones de atención densos y dispersos (donde se seleccionan sólo un subconjunto de pares clave-valor), la versión más grande de este modelo incluía una pila de 96 bloques decoder, 12288 dimensiones para las capas ocultas y alrededor de 175 mil millones de parámetros entrenables. Esta última versión "GPT-3 175B" tenía más de 1000 veces más parámetros que su versión reducida "GPT-3 small", el cual apenas superaba los 100 millones.

3.4.4 Modelo T5

En un contexto donde el transfer learning y fine-tuning se convirtieron en métodos ampliamente utilizados, se introdujo en 2023 el modelo T5 o Text-to-Text Transfer Transformer (Raffel et al., 2023).

El objetivo de éste nuevo modelo basado en la arquitectura Transformer fue la unificación de la técnica del transfer learning para un estudio sistemático de su impacto al luego entrenar al modelo en tareas específicas de NLP. Para lograr esto, los desarrolladores de T5 se basaron en la idea de representar a todos los problemas de procesamiento de lenguaje como un problema texto-a-texto, es decir, tomar un texto de entrada y generar otro texto de salida.

> Crucially, the text-to-text framework allows us to directly apply the same model, objective, training procedure, and decoding process to every task we consider. [...] This framework provides a consistent training objective both for pre-training and fine-tuning. To specify which task the model should perform, we add a task-specific (text) prefix to the original input sequence before feeding it to the model. [Crucialmente, el framework texto-a-texto nos permite aplicar directamente el mismo modelo, objetivo, procedimiento de entrenamiento, y proceso de decodificación a cada tarea que consideramos. [...] Este framework provee un objetivo de entrenamiento consistente para ambos pre-entrenamiento y fine-tuning. Para especificar qué tarea debe realizar el modelo, añadimos un prefijo específico para la tarea a la secuencia original de entrada antes de servirlo al modelo.] (Raffel et al., 2023, p.2,8)

Por ejemplo, en el contexto del presente trabajo y como se verá en el capítulo 4, para indicar la tarea de generación de una lista de símbolos, el reconocimiento de entidades nombradas, o la generación de código, se añadieron "generar lista:", "reconocer entidades nombradas:" y "generar pseudolatex:", respectivamente, como prefijos a las secuencias de entrada. La Figura 3.12 extraída del artículo de Raffel et al., 2023, muestra cómo el modelo simplemente recibe texto con una porción que le indica la tarea a realizar con él y genera la solución al problema solicitado con formato texto.

En lo referido a la arquitectura de T5, como se mencionó, éste utiliza la estructura del Transformer visto en la sección 3.3 y, a diferencia de la familia GPT y BERT, la composición de este modelo de lenguaje no se aleja mucho de la forma original propuesta por Vaswani et al., 2017. Si bien en el contexto en el cual surgió T5 se utilizaban mayormente modelos confeccionados con solo una pila de bloques (encoder-only o decoder-only), sus creadores concluyeron que utilizar el esquema encoder-decoder

original condujo a mejores resultados sobre el framework texto-a-texto de su estudio. A pesar de requerir el doble de parámetros en la red completa por tener tanto una pila de encoders como de decoders, las optimizaciones que aplicaron, como la técnica de compartir parámetros a través de las capas, lograron amortiguar la caída de rendimiento.

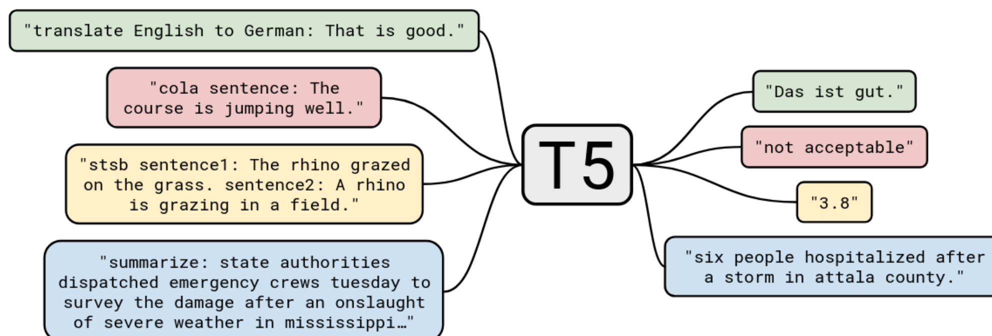

***Figura 3.12**: Diagrama del modelo T5 que muestra las entradas que recibe y salidas que produce en formato de texto. Extraído de "Exploring the Limits of Transfer Learning with a Unified Text-to-Text Transformer."* (Raffel et al., 2023, p.3)

Los bloques de encoder y decoder del modelo base de T5 siguen una configuración semejante a la de BERT. Tanto el encoder como decoder están conformados por 12 bloques, los cuales poseen capas de self-attention, de normalización y capas feedforward de dimensión 3072. Las capas de atención se conforman por 12 cabezas cada una, con matrices de claves y valores de dimensión 64. Finalmente, todas las sub-capas y embeddings tienen una dimensión de 768, lo que resulta en un modelo T5 Base de alrededor de 220 millones de parámetros, prácticamente duplicando aquellos de BERT.

De forma similar como en la familia GPT, los desarrolladores de T5 crearon otros 4 modelos de distintos tamaños, aparte del Base (Raffel et al., 2023):

- **T5 Small**: Éste presenta una reducción considerable de la cantidad de parámetros con respecto a T5 Base, ya que cada pila de encoder y decoder poseen solo 6 bloques. A su vez, cada capa de self-attention posee 8 cabezas de atención y la dimensión del resto de sub-capas es de 512. Esta variante contiene alrededor de 60 millones de parámetros totales.

- **T5 Large**: Con 770 millones de parámetros, esta variante añade más bloques en el encoder y decoder, pasando de 12 a 24 bloques y aumenta las distintas dimensiones de los elementos del modelo. Los embeddings pasan a ser de 1024 dimensiones y las capas feedforward dentro de cada bloque son de 4096 dimensiones.

- **T5 3B y 11B**: Ambas variantes incluyen 24 bloques en las pilas de sus encoders y decoders, al igual que T5 Large. En el caso de la versión "3B", se utilizan 32 cabezas de atención en las capas de self-attention y 16.384 dimensiones en las capas feedforward de los bloques, resultando en alrededor de 2.8 mil millones de parámetros. T5 11B incrementa considerablemente estas dos

cantidades, utilizando 126 cabezas de atención y capas feedforward de 65.536 dimensiones, obteniendo aproximadamente 11 mil millones de parámetros.

Finalmente, resulta relevante destacar que todas las variantes de T5 fueron entrenadas sobre un conjunto de datos que los autores apodaron C4, "Colossal Clean Crawled Corpus", que contenía texto curado sin etiquetar en idiomas Inglés (su mayoría), Francés, Rumano y Alemán.

3.5 Reconocimiento de entidades nombradas

El *etiquetado de entidades nombradas (NER)* dentro de un texto es una tarea útil a resolver durante las primeras etapas de un gran número de tareas de procesamiento de lenguaje natural, como análisis de sentimiento, extracción de relaciones o clasificación de textos.

Una *entidad nombrada* puede definirse como todo conjunto de 1 o más palabras que hace referencia a un objeto dentro del texto, como sustantivos propios, valores de dinero, fechas, etc. Éstas entidades luego pueden ser categorizadas según el tipo de objeto al que representan, por ejemplo, ubicaciones, organizaciones, fechas, dinero, personas. La tarea mencionada de reconocimiento de entidades nombradas consiste en identificar porciones del texto que representan entidades y etiquetarlas con una abreviación del tipo de entidad (Jurafsky & Martin, 2024). La oración a continuación muestra un ejemplo de este procedimiento con tres tipos de entidades: Persona (PER), Ecuación (EQU), Fecha o tiempo (TIME)

> *La segunda ley de **[PER Newton]**, publicada en **[TIME 1687]**, establece que la fuerza aplicada a un objeto es igual al producto de su masa por la aceleración, **[EQU F = ma]**.*

El mayor desafío en la resolución de NER, y por el cual debe entrenarse un modelo que lo resuelva, es la correcta agrupación de los tokens que conforman una misma entidad, lo que puede depender incluso del contexto donde se encuentre. Por ejemplo, en la siguientes dos oraciones, según cómo se agrupen las palabras, se podrían utilizar dos tipos de entidades diferentes, persona (PER) o ubicación (LOC):

*"**[PER Dardo Rocha]** fue el fundador de la ciudad de La Plata"*

*"La **[LOC plaza Dardo Rocha]** es una de las más grandes en La Plata"*

En términos prácticos, para confeccionar un conjunto de datos con el cual poder entrenar a un modelo de aprendizaje automático en esta tarea, se suele realizar el etiquetado de las secuencias de texto utilizando una notación estándar como *BIO* (Jurafsky & Martin, 2024). Ésta consiste en asignar a cada token o palabra del texto una etiqueta que indica si forma parte del inicio (B) de una entidad nombrada, si se encuentra dentro (I) de ella, o si no pertenece a ninguna entidad, es decir, se encuentra fuera (O). Luego, para etiquetar a un token que pertenece a una entidad, ya sea al inicio, medio o final de ella, se concatena la letra B o I a una abreviatura del tipo de la entidad, unidos por un guión . Por ejemplo, si se

desea indicar que una palabra constituye el inicio de la entidad nombrada "Clase de expresión" con abreviatura "EXTYPE", se lo expresaría de la forma:

palabra [B-EXTYPE]

Ésta conversión se realiza para cada palabra del texto procesado, indicando si pertenece o no a una entidad. En caso de hacerlo se indica en qué parte de la entidad se encuentra el token (B o I) y de qué tipo es. De esta forma, utilizar la notación BIO permite tratar al problema de NER como un procesamiento de secuencia-a-secuencia, donde todos los tokens reciben una etiqueta.

3.6 Aprendizaje multitarea

Debido a su relevancia en las implementaciones presentadas en los posteriores capítulos 4 y 5, ésta sección expone la técnica de *aprendizaje multitarea*, o más conocido como *multitask learning (*MTL*).* Este otro acercamiento al aprendizaje por transferencia consiste en el entrenamiento de los modelos de aprendizaje automático sobre múltiples tareas de forma simultánea, logrando un mayor poder de generalización y aumentando la eficiencia con respecto a los datos requeridos y la velocidad de aprendizaje.

En el contexto de las redes neuronales, y particularmente en el procesamiento de lenguaje natural, el MTL ha ganado tracción debido a los casos satisfactorios de aplicación sobre modelos de lenguaje, donde se obtuvieron mejoras de desempeño al entrenarlos para la resolución de múltiples tareas relacionadas y utilizar sus semejanzas y diferencias en la actualización de parámetros. En general, para implementar el aprendizaje multitarea en modelos neuronales se comparten ciertas porciones de la red para la resolución de las diferentes tareas, incluyendo matrices de pesos y capas ocultas, mientras que se añaden múltiples capas de salida específicas para las tareas consideradas, donde es posible utilizar diferentes funciones de error para un correcto ajuste de los parámetros.

Uno de los requisitos del MTL es la similitud o relación entre las distintas tareas y los datos utilizados durante el entrenamiento simultáneo. Ésto promueve a que durante el aprendizaje de las representaciones vectoriales compartidas de los datos de entrada y de los pesos internos de la red se logren construir atributos más complejos y generales que provean información de la naturaleza de cada tarea. Otro aspecto ventajoso de MTL, aparejado a la compartición de parámetros de la red, es la mejora potencial en la velocidad de aprendizaje y en la disminución de los datos necesarios para la convergencia durante el proceso de entrenamiento, lo que puede atribuirse al hecho de que cada tarea aprendida suma una porción de conocimiento que puede ayudar a acelerar el aprendizaje de otra de las tareas.

> First, MTL improves data efficiency for each sub-task. Different tasks provide different aspects of information, enriching the expression ability of the hidden representation to the input text. [...]. This encourages the multi-task model to produce more generalizable representations in shared layers. Thus, the model is prevented from overfitting to a single task and gains stronger generalization ability. [Primero, MTL mejora la eficiencia con

respecto a los datos para cada sub-tarea. Diferentes tareas proveen diferentes aspectos de información, enriqueciendo la capacidad de expresión de la representación oculta del texto de entrada. [...] Ésto promueve al modelo a producir representaciones más generalizables en las capas compartidas. De este modo, se previene que el modelo se sobreajuste a una sola tarea y gane una mejor capacidad de generalización.] (Zhang et al., 2023, p.3)

Un caso de uso reciente de esta técnica fue durante el análisis del pre-entrenamiento del modelo T5 introducido en la sección 3.4.4. Los creadores de T5 describieron en su artículo de presentación (Raffel et al., 2023) las pruebas realizadas al comparar el desempeño obtenido del modelo cuando aplicaban, por una parte, pre entrenamiento no supervisado y luego fine-tuning, contra el pre-entrenamiento con multitask learning y una posterior aplicación de fine-tuning. En su análisis, los autores destacaron el desafío que impone el uso de MLT al momento de seleccionar la proporción correcta de datos destinados al aprendizaje de cada tarea, a lo que refieren como "an extremely important factor in multi-task learning" [un factor extremadamente importante en el aprendizaje multitarea] (Raffel et al., 2023, p.31). Al aplicar MTL se agrava la problemática de conseguir que el entrenamiento del modelo neuronal le posibilite resolver las tareas correctamente sin alcanzar niveles altos de sobreajuste a los datos. Ésto se debe a que existen múltiples factores atados a la incorporación de cada tarea durante el aprendizaje, como el tamaño del conjunto de datos para dicha tarea, la "dificultad" de su aprendizaje (es decir, cuánta información necesita el modelo para efectuar correctamente la tarea), los mecanismos regularización, el porcentaje de dropout, etc.

T5, al interpretar todas las problemáticas de NLP como una tarea de conversión texto-a-texto, tiene la particularidad de poder aplicar multitask learning sin la necesidad de modificar la arquitectura subyacente del modelo para ajustarlo a las distintas tareas consideradas:

> We also note that in our unified text-to-text framework, "multi-task learning" simply corresponds to mixing data sets together. [...] In contrast, most applications of multi-task learning to NLP add task-specific classification networks or use different loss functions for each task. [También destacamos que en nuestro framework texto-a-texto, "multi-task learning" simplemente corresponde a mezclar conjuntos de datos. [...] En contraste, la mayoría de las aplicaciones del aprendizaje multi-tarea en NLP añaden redes de clasificación específicas para las tareas o utilizan diferentes funciones de error para cada tarea.] (Raffel et al., 2023, p.31)

Considerando el formato de las entradas del modelo T5, es posible obtener un caso simple de aplicación del aprendizaje multitarea, donde basta simplemente con fijar prefijos al inicio de las secuencias de entrada que denoten las tareas a resolver y proveer un conjunto de datos que expongan ejemplos etiquetados relevantes. Los capítulos 4 y 5 detallarán un caso similar a éste que fue desarrollado para la implementación y análisis del modelo final del presente trabajo.

3.7 Métricas de evaluación de modelos

A diferencia de las métricas típicamente utilizadas para los modelos de aprendizaje automático, aquellas que buscan determinar el desempeño de un modelo de lenguaje no siempre pueden ser calculadas de forma numérica. En ciertos casos resulta un tanto subjetivo definir un puntaje para especificar qué tan bien el modelo ha generado el siguiente símbolo u oración, razón por la cual en algunas ocasiones aún se mantiene la evaluación por parte de expertos.

Debido al alto costo y esfuerzo requerido en la evaluación manual de los modelos, se han confeccionado métricas de evaluación automática que buscan puntuar la calidad de los resultados obtenidos por los modelos de lenguaje. Entre las más populares en el ámbito de la generación de texto se pueden mencionar las métricas de Perplejidad, BLUE y ROUGE.

La perplejidad representa el estándar de métrica intrínseca para la evaluación de modelos de lenguaje, tanto para aquellos basados en n-gramas o más complejos como los modelos neuronales. Se define como una métrica intrínseca ya que permite evaluar a los modelos independientemente del contexto particular donde se los apliquen, sin la necesidad de efectuar testeos costosos punto-a-punto de su funcionamiento en aplicación (Jurafsky & Martin, 2024). La perplejidad está definida como el inverso de la probabilidad de una secuencia o de un conjunto de prueba W normalizado por el número N de tokens involucrados, obteniendo de esa forma la siguiente función que la describe:

$$perplejidad(W) = \sqrt[N]{\frac{1}{P(w_1 w_2 \ldots w_N)}}$$

Notar que cuanto mayor sea el valor de probabilidad para el conjunto W, menor será el valor final de la perplejidad, indicando que el modelo está "más seguro" o "menos sorprendido" de la presencia de los tokens en ese orden determinado. De esta forma, cuanto menor sea la perplejidad del modelo sobre los datos, mejor será su desempeño. Igualmente debe destacarse que, a pesar de poder evaluar el modelo de lenguaje de forma aislada y dar indicaciones rápidas de mejoras, no garantiza que su comportamiento sea bueno en la resolución de tareas particulares de NLP como la traducción.

Por otra parte, la métrica BLUE (Papineni et al., 2002) o *Bilingual Evaluation Understudy*, ha surgido inicialmente como forma de evaluación de modelos dedicados a la traducción automática. Dado que un mismo texto puede ser traducido de múltiples formas, esta métrica compara, normalmente a nivel de frase, la secuencia de palabras generada por el modelo contra un conjunto de traducciones de referencia. De esta forma, BLUE define a la similitud o "cercanía de traducción" entre los resultados generados y los textos de referencia como el criterio de evaluación de desempeño. Particularmente utiliza en su cálculo una medida de precisión denominada *Precisión N-grama Modificada* en la cual no se beneficia a las palabras comunes que tienden a ser "sobre generadas", como lo hace la medida de precisión usual. BLUE obtiene valores entre 0 y 1 para cada oración traducida, donde solo aquellas traducciones totalmente exactas a las de referencia pueden obtener el máximo valor 1. A su vez, la métrica tiende a asignar valores mayores para los casos de oraciones con más traducciones de referencia,

haciendo necesario mantener un número similar de ejemplos de referencia para cada caso de prueba evaluado.

Finalmente, es posible destacar en esta sección otro conjunto de métricas un tanto similar a BLUE, apodado ROUGE o *Recall-Oriented Understudy for Gisting Evaluation* (Lin, 2004), que incorpora múltiples sub-métricas como:

- ROUGE-N: Estadísticas de co-ocurrencia de n-gramas
- ROUGE-L: Subsecuencia común más larga
- ROUGE-W: Subsecuencia común más larga ponderada
- ROUGE-S: Estadísticas de co-ocurrencia de skip-bigrams

Éste conjunto de métricas surgieron en 2004 para resolver la evaluación automática de modelos orientados a la sumarización o resumen de textos. De forma similar a BLUE permiten evaluar de forma automática la calidad de los textos generados en comparación con otros de referencia creados por humanos. En términos más formales, las métricas ROUGE cuentan la cantidad de unidades de n-gramas, secuencias de palabras o pares de palabras de los resúmenes generados que se “superponen” con los textos ideales de referencia.

Si bien la técnica ROUGE-N es la más comparable a la métrica BLUE, esta última basa su cálculo en la medida de precisión, mientras que ROUGE-N utiliza el *recall.* Es decir, en lugar de centrarse en la proporción de casos positivos correctos *generados* por el modelo (precision), utiliza la proporción de los casos positivos *reales* que el modelo ha podido capturar (recall).

A modo de conclusión se debe destacar que las métricas presentadas en esta sección representan solamente un subconjunto acotado del universo de todas las formas de evaluación confeccionadas para medir el desempeño de los modelos de aprendizaje profundo. Sin embargo, la breve introducción al funcionamiento de BLUE y ROUGE se consideró relevante debido a las implementaciones enmarcadas en el contexto del presente trabajo. En el capítulo 6 se verá especialmente una discusión del uso de estas métricas sobre el caso particular de generación de expresiones algebraicas y las razones por las que se optó realizar una evaluación manual del modelo desarrollado.

CAPÍTULO 4

Datos de entrenamiento

Un pilar indispensable en la construcción de un nuevo modelo de redes neuronales basado en aprendizaje supervisado es el conjunto de datos a partir del cual se realiza la ingesta durante el entrenamiento. Éste debe poseer un volúmen significativo de ejemplos que provea casos de gran variedad de la tarea a resolver, junto con los resultados esperados a producir por el modelo.

Con el fin de construir el conjunto de datos del presente trabajo de grado se comenzó con el análisis del problema a resolver mediante la esquematización de múltiples ejemplos, con lo cual se comprendió que la naturaleza de las entradas y salidas esperadas del modelo presentaban patrones fundamentales para formular una solución efectiva. Este descubrimiento permitió definir el esquema base sobre el que se construyó el conjunto de datos de entrenamiento, validación y testeo del nuevo modelo de aprendizaje profundo.

Al buscar una simplificación de la tarea, se identificaron tres componentes básicos involucrados: (1) Las descripciones de entrada, (2) el modelo de redes neuronales y (3) las expresiones de salida. La siguiente Figura 4.1 muestra dichos componentes en un diagrama simplificado de la tarea de generación.

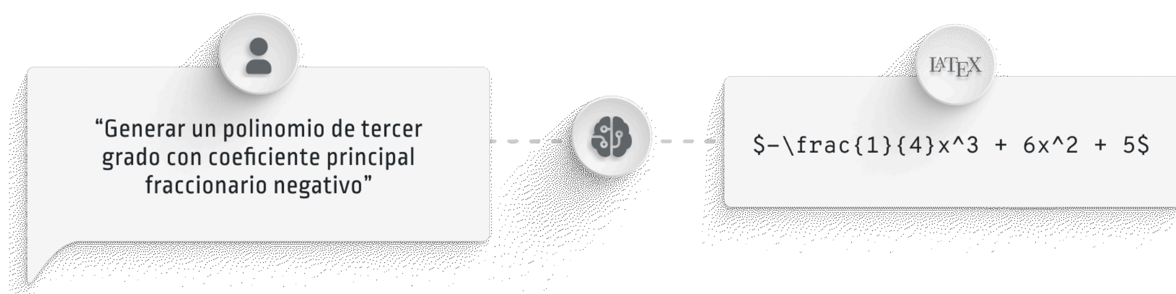

Figura 4.1: *Esquematización simplificada de la tarea de generación de expresiones algebraicas.*

Durante todo el presente capítulo se realizará un enfoque particular sobre los extremos del proceso de generación, es decir, las descripciones de entrada y las expresiones de salida. Ambas componentes conformaron los elementos de datos a recolectar y sobre los que debieron tomarse las primeras decisiones de diseño. Este capítulo comenzará presentando las ideas que guiaron dicha recolección de datos junto al modelo físico implementado para su almacenamiento. A su vez, en la sección 4.2 se mostrarán los detalles del desarrollo del subproyecto LEDProject que hizo posible la recolección eficiente de ejemplos variados de descripciones en español de las expresiones. Finalmente, en el apartado 4.3 se verá una

descripción del flujo de actividades de preprocesamiento de los datos recolectados, que fueron implementadas con el fin de obtener un conjunto de datos libre de errores y con los formatos esperados durante el entrenamiento.

4.1 Diseño conceptual de los datos

Desde la perspectiva del problema a resolver resultó fundamental la toma de ciertas decisiones con respecto a la estructura de los datos. Con el objetivo de analizar el problema e identificar posibles patrones a simple vista, se esquematizaron ejemplos de la generación de código LaTeX a partir de la entrada de textos descriptivos como los siguientes:

`Entrada`: *“Generar un polinomio de tercer grado con coeficiente principal fraccionario negativo”*

`Salida`: “`-\frac{1}{3}x^3 + 10x^2 - 4`”

`Entrada`: *“Sistema homogéneo de 2 ecuaciones con 2 variables”*

`Salida`: “`\begin{cases} 3x + y = 0 \\ 2x = 0 \end{cases}`”

`Entrada`: “Polinomio de 4to grado y 3 términos negativos”

`Salida`: “`-x^4 -5x^2 - 10`”

A raíz de lo observado en ejemplos similares a los presentados, se identificó un conjunto de patrones comunes a todos los casos, que caracterizaban tanto las descripciones de entrada como las expresiones de salida. En los apartados que se describen a continuación se presentarán estas características inherentes de los datos utilizados, y que se transformaron en piezas claves del diseño del conjunto de datos de entrenamiento.

4.1.1 PseudoLaTeX

El primer aspecto identificado en todos los casos de análisis considerados fue la presencia de valores numéricos específicos en las expresiones de salida como los enteros `5` y `-10` o fraccionarios como `-\frac{1}{3}`, los cuales podrían causar problemas de variabilidad en los resultados generados por el modelo. Ésto se tomó como un aspecto crítico debido a la baja capacidad de extrapolación de representaciones numéricas demostrada por los modelos basados en redes neuronales, los cuales tienden a obtener resultados satisfactorios sólo para aquellos valores sobre los cuales fueron entrenados (Trask et al., 2018).

A raíz de ello, se planteó un nuevo formato para las expresiones algebraicas representadas como código LaTeX, al cual se apodó *pseudoLaTeX*. Ésta variante del formato de las secuencias a generar fue orientado exclusivamente a la representación de fórmulas matemáticas, dejando por fuera de su definición al modo de escritura de texto normal de LaTeX (visto en 2.3). Luego, dado que todas las

cadenas de símbolos de pseudoLaTeX representaban elementos matemáticos, se descartó la necesidad de indicar explícitamente el modo de escritura "matemática" con los símbolos de pesos "$" alrededor de las fórmulas, lo cual era interpretado por defecto.

Aparte de las diferencias gramaticales con LaTeX, pseudoLaTeX tenía el objetivo de lograr expresiones con una mayor variabilidad al incorporar los denominados *operandos variables*. En términos simples, éstos operandos representaban símbolos especiales de dos caracteres dentro de las expresiones algebraicas y actuaban como *placeholders* o marcadores que luego podía ser reemplazados correspondientemente con valores numéricos aleatorios. A continuación se presenta la lista de los operandos variables utilizados en este trabajo:

- `VR:` Valor variable del conjunto numérico de los reales.
- `VD:` Valor variable del conjunto numérico de los racionales en formato decimal.
- `VI:` Valor variable del conjunto numérico de los irracionales.
- `VC:` Valor variable del conjunto numérico de los complejos.
- `VN:` Valor variable del conjunto numérico de los enteros.

Luego, tomando en consideración estos nuevos elementos se planteó la definición del formato pseudoLaTeX como salida de las expresiones del modelo final, en lugar de utilizar LaTeX con valores numéricos predefinidos.

Cabe resaltar que no todos los números de las expresiones generadas en pseudoLaTeX debían ser marcados por medio de los operandos variables, sino que podían adquirir valores específicos. Ésto se debió a dos razones:

- Las descripciones de entrada al proceso de generación podían definir valores específicos para ciertos elementos de una expresión, como por ejemplo, en el extracto de oración "coeficiente principal igual a 5", el valor 5 representa un operando dentro de la expresión que no debería ser variable.

- Los elementos que conformaban la estructura base de las expresiones algebraicas, como los exponentes de las indeterminadas de un polinomio, en pocas ocasiones podían variar. Por ejemplo al definir un "polinomio de 4to grado", el exponente 4 de la indeterminada del término principal debía estar presente y éste valor numérico no debía variar.

4.1.2 Entidades Nombradas

Tomando casos como los mencionados al inicio de esta sección, resultó directo identificar una característica común a todas las descripciones en español de entrada. Si se observa con cierto detenimiento, es posible verificar que los múltiples textos descriptivos contienen palabras que refieren a los mismos elementos semánticos u objetos. Dos ejemplos evidentes son, por ejemplo, las palabras que referencian a la clase de expresión a generar como "polinomio" o "sistema" y, por otra parte, las porciones de oración que mencionan atributos y valores particulares de dichas expresiones, como "4to

grado". Luego, al continuar el análisis sobre las descripciones en español se logró extraer un grupo de *entidades* comunes a la mayoría de ellas, que podía ser extrapolado a un gran número de casos diferentes.

De esta forma, la extracción de entidades nombradas, tomó un papel principal como una posible mejora en el entrenamiento del modelo de generación. Tanto la estructura sintáctica de las oraciones como el contenido semántico de cada una de las entidades que las conforman se convirtieron en una potencial fuente de información que podía ser utilizada a la hora de generar las expresiones algebráicas. Es así como surgieron los primeros interrogantes que, a futuro, constituyeron parte de la hipótesis central del trabajo que será introducida en el capítulo 5:

¿El modelo de aprendizaje profundo es capaz de utilizar la estructura gramatical y semántica del texto ingresado para lograr una generación de expresiones correcta?

¿Entrenar al modelo en la tarea de extracción de entidades nombradas aumenta la efectividad y exactitud en los resultados de la generación?

Si bien se verá en más detalle en el siguiente capítulo, resulta relevante mencionar que el descubrimiento de este primer patrón en las descripciones representó el disparador de la idea de incorporar el *aprendizaje multitarea* (visto en Sección 3.6) en el proceso de entrenamiento del modelo final. Tomando en todo momento a la generación de pseudoLaTeX como tarea principal, se propuso añadir al reconocimiento de entidades nombradas (Sección 3.5) como una segunda tarea auxiliar que podría aportar información diferente al proceso de entrenamiento e inferencia, brindado una capacidad potencialmente mayor de generalización. Luego, dada esta incorporación, resultó necesaria la definición del grupo de etiquetas que permitieron anotar las entidades contenidas en las descripciones en español. A continuación se detallará el diccionario de entidades utilizadas en las anotaciones de una parte del conjunto de datos de entrenamiento:

Diccionario de entidades nombradas

Las entidades listadas a continuación fueron las únicas utilizadas durante el etiquetado de las descripciones en español que conformaron el conjunto de datos final de entrenamiento:

- `EXTYPE`: (Expression Type) Etiqueta para la clase de expresión algebraica. Ejemplos: "Polinomio", "Sistema de ecuaciones".

- `EXCHAR`: (Expression Characteristic) Etiqueta para una característica de la expresión en su totalidad escrito en forma de adjetivo. Ejemplos: "completo", "desordenado".

- `ATRIB`: (Attribute) Etiqueta para un atributo de una expresión escrita en forma de sustantivo. Ejemplos: "grado", "orden".

- `ATVAL`: (Attribute Value) Etiqueta para el valor de un atributo de una expresión. Puede ser alfanumérico. Ejemplos: "1er", "ascendente".

- `ELEM`: (Element) Etiqueta para un elemento de una expresión algebraica. Ejemplos: "variable", "coeficiente principal".

- `ELCHAR`: (Element Characteristic) Etiqueta para una característica de un elemento de una expresión algebraica. Ejemplos: "negativo", "nulo".

- `ELVAL`: (Element Value) Etiqueta para el valor de un elemento de una expresión algebraica. Puede ser alfanumérico. Ejemplos: "10", "uno".

- `ELCOUNT`: (Element Count) Etiqueta para la cantidad de instancias de un elemento dentro de una expresión algebraica. Puede ser alfanumérico. Ejemplos: "2", "tres".

- `VAR`: (Variable) Etiqueta para el nombre de una variable dentro de una expresión algebraica. Ejemplos: "y", "x".

4.1.3 Árboles de Operadores

De forma similar al análisis previo de las descripciones, se realizó otro enfoque sobre las salidas del proceso de generación, es decir, sobre el formato de las expresiones algebraicas generadas y los patrones ocultos en ellas. Al mismo tiempo que se buscaba entrenar al modelo con expresiones en formato pseudoLaTeX, también resultaba evidente que toda fórmula matemática que se planteara (polinomios y sistemas de ecuaciones en este caso) poseía una estructura jerarquizada e incluso un tanto repetitiva. Por ejemplo, la expresión algebraica

$$-\frac{1}{4}x^3 + 6x^2 + 5$$

podría descomponerse en diferentes niveles según el alcance de cada operador. Ésto lleva a pensar en una posible organización de las expresiones en formato de árboles n-arios, donde los operadores son elementos que conforman los nodos internos y raíz del árbol, mientras que los operandos y variables se encuentran en sus hojas.

A continuación, la Figura 4.2 demuestra de forma gráfica la estructura jerarquizada del polinomio de tercer grado mencionado previamente. A su vez, del lado izquierdo de la imagen es posible observar tres representaciones distintas de la misma expresión, donde la última, la forma de PseudoLaTeX, representa un ejemplo de las salidas esperadas del modelo construido en este trabajo. Notar que se marcan dos símbolos en color verde que representan los *operando variables* que diferencian al formato del PseudoLaTeX del LaTeX original. En particular, el ejemplo demuestra cómo los valores fraccionarios pueden representarse de forma genérica con el operando variable *VF* y los valores naturales con el operando *VN*. Finalmente, se debe notar que el valor "5" no fue reemplazado por un operando variable,

haciendo referencia a los casos mencionados en el apartado 4.1.1, en los que se requiere mantener fijos ciertos valores numéricos.

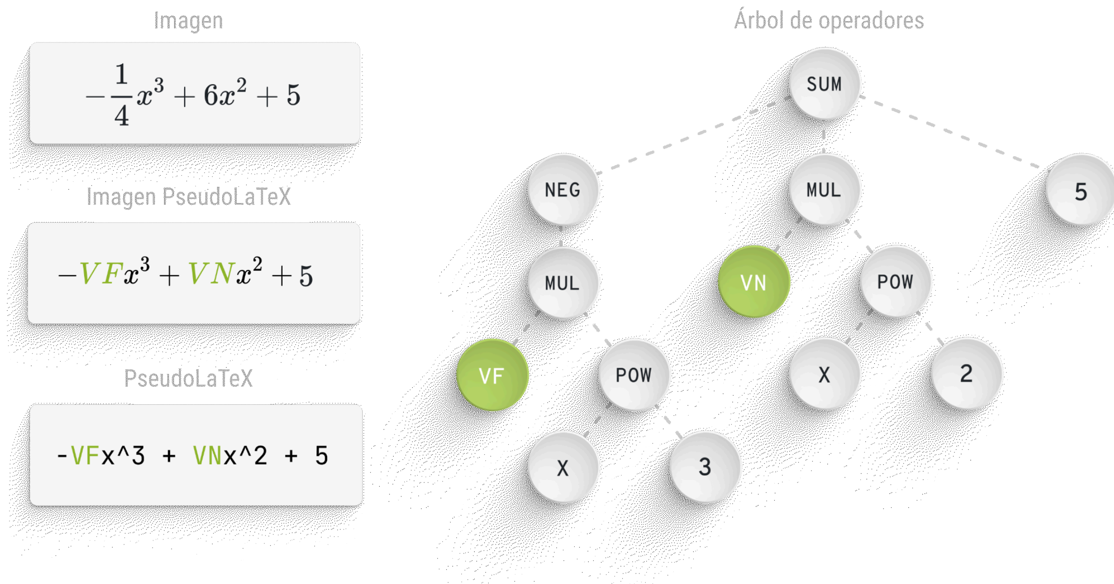

Figura 4.2: *Representaciones de una expresión algebráica en dos imágenes y en formato de Árbol de operadores y PseudoLaTeX*

La iniciativa de representar a las expresiones algebraicas como árboles de operadores fue impulsada por una emergente línea de investigación que busca obtener mejoras en el procesamiento del lenguaje matemático (MLP) mediante la representación de las fórmulas matemáticas en forma de árboles. Particularmente se tomó como referencia al trabajo de (Wang et al., 2021) para motivar la representación de los polinomios y sistemas de ecuaciones como árboles de operadores. En dicho artículo se hace énfasis en la coexistencia del lenguaje matemático y el natural, como lo es el idioma español:

> With its unique set of symbols and language structure, mathematical language complements natural language in concisely and precisely communicating essential scientific knowledge. [Con su conjunto de símbolos y estructura de lenguaje únicos, el lenguaje matemático complementa al lenguaje natural al comunicar conocimiento científico esencial de forma concisa y precisa.] (Wang et al., 2021, p.1)

A su vez, deja en relieve la importancia de hacer uso de la estructura jerárquica inherente en el lenguaje matemático para lograr resultados satisfactorios:

> Indeed, mathematical formulae are inherently hierarchical and tree structures are appropriate for organizing the math symbols in a formula. Compared to representing a formula simply as a sequence of math symbols, the symbolic tree representation has the advantage to encode both the semantics and the inherent hierarchical structure of a formula. Similar to works in natural language processing (NLP) that leverage inherent

> language structure. [En efecto, las fórmulas matemáticas son inherentemente jerárquicas y las estructuras de árbol son apropiadas para organizar los símbolos matemáticos en una fórmula. Comparado a representar una fórmula simplemente como una secuencia de símbolos, la representación del árbol simbólico tiene la ventaja de codificar tanto la semántica como la estructura jerárquica inherente de una fórmula. Similar a trabajos en procesamiento de lenguaje natural (NLP) que aprovechan la estructura inherente del lenguaje.] (Wang et al., 2021, p.1)

Tomando como base al trabajo desarrollado en dicho artículo y los resultados satisfactorios obtenidos, se planteó otro interrogante con respecto al aprendizaje multitarea a validar en el funcionamiento del nuevo modelo implementado:

¿Entrenar al modelo en la tarea de generación de expresiones algebraicas en forma de árboles de operadores aumenta la efectividad y exactitud en el desempeño de la generación de PseudoLaTeX?

Luego, con el objetivo de formar representaciones de árbol que se ajustaran al subconjunto acotado de polinomios y sistemas de ecuaciones, se construyó un vocabulario de símbolos de operadores y operandos que codificaran la información necesaria de las fórmulas a generar. La gramática utilizada para la especificación de los árboles consistió en la colocación secuencial de instancias de los símbolos mencionados, con el fin de lograr una notación semejante a la *polaca*, o también conocida como *notación prefija*, la cual tiene la particularidad de ubicar a los símbolos de operadores matemáticos a la izquierda de sus operandos. A continuación se presentará un diccionario que lista el conjunto de los símbolos utilizados en las representaciones de árboles de operadores junto con la explicación del uso y su expresión gráfica.

Diccionario de operadores y operandos

1. Operadores N-arios:

Estos operadores pueden abarcar un número indeterminado de operandos y deben tener como último elemento al marcador de fin “E”

- SUM: Indica la suma de todos los términos que incluye. En el caso de ser un solo operando, el signo “+” es insertado a su izquierda. A su vez, si uno de los términos de la suma es un elemento negativo (le precede el operador NEG) el símbolo “+” es reemplazado por el “-” delante del operando correspondiente:

$$[\ \mathtt{SUM}\ T_1\ \ T_2 \cdots T_n\ \mathtt{E}\ ] \quad \rightarrow \quad T_1 + T_2 + \ \cdots \ + T_n$$

$$[\ \mathtt{SUM}\ T_1\ \ \mathtt{NEG}\ \ T_2 \cdots T_n\ \mathtt{E}\ ] \quad \rightarrow \quad T_1 - T_2 + \ \cdots \ + T_n$$

$$[\ \mathtt{SUM}\ T_1\ \ \mathtt{E}\ ] \quad \rightarrow \quad + T_1$$

- MUL: Indica el producto de todos los términos que abarca. La operación se traduce a la concatenación de los términos unidos por el símbolo ".". En caso de ser 1 o 2 los términos el símbolo es omitido. Por último, si entre la lista de términos se encuentran otros operadores n-arios, el resultado final de cada uno de ellos será encerrado entre un par de paréntesis:

$$[\ \mathtt{MUL}\ \ T_1\ \ T_2 \cdots T_n\ \mathtt{E}\ ] \quad \rightarrow \quad T_1.T_2.\ \cdots\ .T_n$$

$$[\ \mathtt{MUL}\ \ T_1\ \ T_2\ \ \mathtt{E}\ ] \quad \rightarrow \quad T_1 T_2$$

$$[\ \mathtt{MUL}\ \ T_1\ \ \mathtt{E}\ ] \quad \rightarrow \quad T_1$$

$$[\ \mathtt{MUL}\ \ T_1\ \ \mathtt{SUM}\ \ T_2\ \ T_3\ \ \mathtt{E}\ \ \mathtt{E}\ ] \quad \rightarrow \quad T_1 . (T_2 + T_3)$$

- CASES: Se traduce a un conjunto de casos que son agrupados por medio de una llave a la izquierda. Este operando puede ser utilizado, por ejemplo, para definir un sistema de ecuaciones o una función a trozos. Cada uno de sus términos representa un caso distinto y son ordenados en filas:

$$[\ \mathtt{CASES}\ \ C_1 \cdots C_n\ \mathtt{E}\ ] \quad \rightarrow \quad \begin{cases} C_1 \\ \vdots \\ C_n \end{cases}$$

2. Operadores Binarios:

Los operadores binarios no dependen de un símbolo de finalización para determinar los operandos que abarca la operación, ya que siempre consideran solo dos operandos.

- EQ: Representa una ecuación y se traduce a la igualación de dos expresiones:

$$[\ \mathtt{EQ}\ \ EXPR_1\ \ EXPR_2\ ] \quad \rightarrow \quad EXPR_1 = EXPR_2$$

- POW: Indica el operador de exponenciación o superíndice. El primer operando o expresión es interpretado como la base y el segundo como el exponente. En el caso donde la base sea una expresión (abarca otros operadores), se incluyen dos paréntesis que la encierran:

$$[\ \mathtt{POW}\ \ BASE\ \ EXP\ ] \quad \rightarrow \quad BASE^{EXP}$$

$$[\ \mathtt{POW}\ \ EXPRESSION\ \ EXP\ ] \quad \rightarrow \quad (EXPRESSION)^{EXP}$$

- SUB: Permite definir un subíndice de un operando o expresión. El primer término es interpretado como la base y el segundo como el subíndice. En el caso donde la base sea una expresión (abarca otros operadores), se incluyen dos paréntesis que la encierran:

$$[\ \mathtt{SUB}\ \ BASE\ \ IDX\ ] \quad \rightarrow \quad BASE_{IDX}$$

$$[\ \mathtt{SUB}\ \ EXPRESSION\ \ IDX\ ] \quad \rightarrow \quad (EXPRESSION)_{IDX}$$

- `ROOT`: Representa la raíz n-ésima de un operando. El primer término que involucra se interpreta como el radicando y el segundo como el índice de la raíz:

$$[\ \texttt{ROOT}\ \ OPERAND\ \ IDX\ ] \quad \rightarrow \quad \sqrt[IDX]{OPERAND}$$

- `FRAC`: Representa la fracción de dos expresiones, ya sean operandos individuales o expresiones más complejas. El primer elemento es interpretado como el numerador de la fracción y el segundo como el denominador.

$$[\ \texttt{FRAC}\ \ EXPR_1\ \ EXPR_2\ ] \quad \rightarrow \quad \frac{EXPR_1}{EXPR_2}$$

3. Operadores Unarios:

Los operadores unarios no necesitan de un símbolo de finalización ya que siempre preceden al único operando sobre el cual se aplican.

- `NEG`: Utilizado para denotar que el elemento que le sigue es un valor numérico negativo. Se representa con el símbolo "-" y se inserta previo al operando.

$$[\ \texttt{NEG}\ \ OPERAND\ ] \quad \rightarrow \quad -\ OPERAND$$

4. Operandos:

Los operandos presentados en esta sección incluyen símbolos o expresiones de dos clases:

- Los *operandos variables* definidos en el formato pseudoLaTeX visto en 4.1.1 y que representan parámetros con valores genéricos que cumplen con ciertas propiedades. Tienen el comportamiento de "variables" que serán reemplazadas posteriormente por valores particulares que cumplen con las propiedades que estos operandos especifican. Los mismos operandos definidos en 4.1.1 fueron incorporados como parte de este diccionario.

- Aquellos que no varían y representan algún símbolo matemático, texto o indicador de fin de secuencia:
  - `E`: Símbolo de terminación de una secuencia de términos abarcados por un operador n-ario.
  - `DOTS`: Símbolo que representa puntos suspensivos.
  - `$...$`: Cualquier texto que se encuentre entre dos signos `$` será presentado como una cadena de caracteres concatenados.

4.2 Recolección de datos - LEDProject

Una vez establecidas las características esenciales de los datos que conformaron el conjunto de entrenamiento, hallar un grupo significativo de ejemplos se convirtió en el siguiente objetivo central. Debido a la naturaleza del problema a resolver se identificaron dos limitaciones críticas iniciales en la recolección de los datos:

- Hallar bases de datos previamente definidas que se ajustaran al dominio de datos particular planteado para el presente trabajo resultó ser una tarea compleja de cumplir. El formato de árboles de operadores, el idioma español para las descripciones y el formato personalizado de pseudolatex significaron las mayores limitaciones en la búsqueda de conjuntos de datos previamente creados por la comunidad de desarrolladores.

- Los datos debían reflejar el grado de variabilidad existente en el habla de las personas. De esta forma sería posible conseguir un modelo que lograra generalizar su comportamiento de forma efectiva, generando satisfactoriamente expresiones que cumplieran con las características pautadas por el usuario y evitando posibles sesgos durante el entrenamiento.

Con la intención de evadir dichas dificultades surgió el subproyecto *LEDProject*[3] o, extensivamente, *LaTeX Expression Descriptor Project*, que consistió en el desarrollo de una simple página web que permitió a alumnos de la universidad contribuir con descripciones propias para un conjunto dado de expresiones. La implementación de esta página web fue principalmente motivada por el uso por parte de los alumnos y profesores de las materias de Matemática B, C y D de la Facultad de Ingeniería de la Universidad Nacional de La Plata, quienes colaboraron por un período de tres meses con un amplio conjunto de más de setecientas descripciones de expresiones algebraicas en idioma español.

En términos resumidos, LEDProject consistió de una página web que inicialmente presentaba al usuario una sección tutorial que le indicaba la actividad a realizar. Luego, una vez superado el instructivo, se le proveía un subconjunto de cinco expresiones, entre las cuales podían incluirse polinomios y sistemas de ecuaciones, que debía describir con sus palabras utilizando el idioma español. Al finalizar las cinco descripciones, el usuario era incentivado a comenzar nuevamente el desafío para continuar colaborando con el crecimiento de la base de datos. La figura 4.3 muestra estáticamente las 3 secciones que conformaban LEDProject.

Para lograr la recolección efectiva de los datos requeridos resultó indispensable la implementación de un conjunto de algoritmos que generaran expresiones de ejemplo en el formato correspondiente de árboles de operadores y pseudoLaTeX y, a su vez, decodificaran dichas estructuras para poder administrarle al usuario de la página web una imagen correcta de la expresión a describir. En las secciones que continúan se presentarán los detalles de implementación de los algoritmos mencionados y de los componentes que constituyeron a LEDProject.

[3] Link a la página web: https://latexpressions-thesis-project.onrender.com

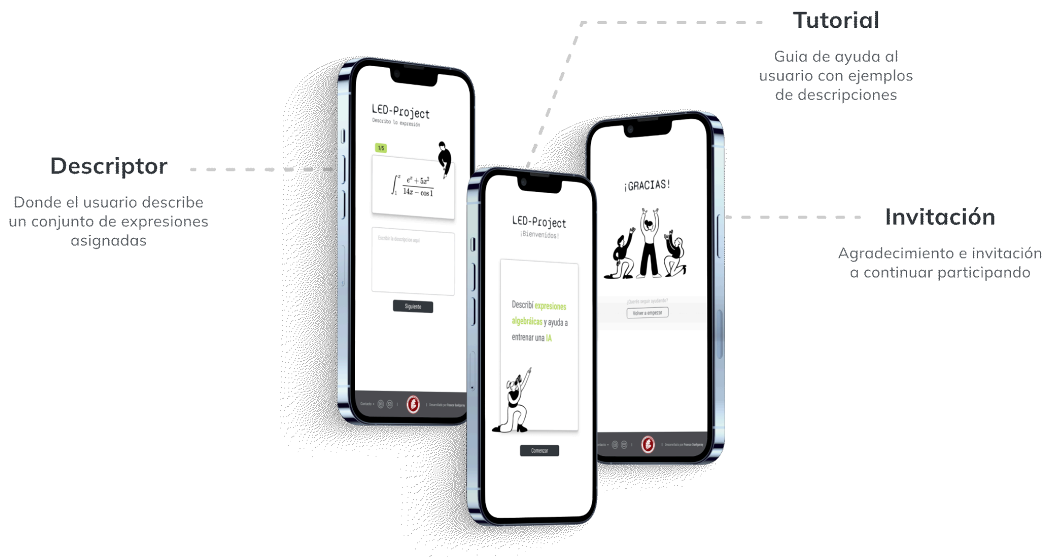

Figura 4.3: Las tres secciones de la página web LEDProject

4.2.1 Generación de expresiones

Con el fin de proveer expresiones a los usuarios de LEDProject, resultó necesario producir un conjunto considerable de ellas en el formato definido de árboles de operadores y pseudoLaTeX. Durante la recolección de estas fórmulas de ejemplo se priorizó la variabilidad en la estructura y características de las expresiones recolectadas, de forma que el posterior entrenamiento del modelo incluyera el mayor número posible de casos diferentes.

Generador de árboles de operadores

Debido al volumen de expresiones que debían proveerse a los usuarios de LEDProject, se optó por agilizar el proceso de generación de los árboles de operadores mediante la implementación de un algoritmo que los produjera de forma automatizada. Por otra parte, para obtener expresiones que aportaran variabilidad en los datos de entrenamiento, se identificaron conjuntos de atributos generales de las expresiones que definían características claves de éstas durante la descripción en lenguaje natural, como el grado máximo de los polinomios, la homogeneidad de un sistema de ecuaciones o el número de variables que incluían.

De esta forma, con el objetivo de guiar la generación de las expresiones algebraicas de forma estructurada, se crearon *archivos descriptores* que dividían el conjunto total de expresiones a generar en grupos con características en común. Cada uno de dichos archivos, apodados "*group descriptor*" (*descriptor de grupo*), especificaba los valores de los parámetros variables de cada grupo de expresiones, indicando:

- Número de grupo (`group number`)
- Clase de expresiones (`expression type`)
- Cantidad de expresiones pertenecientes al grupo (`number of expressions`)
- Lista de conjuntos numéricos de donde generar los coeficientes (`coefficients`)
- Parámetros y valores particulares para la clase de expresiones del grupo (`parameters`)

La Tabla 4.1 al final de este apartado "Generador de árboles de operadores" contiene el detalle de los archivos descriptores de cada uno de los 12 grupos de expresiones que fueron almacenados en la base de datos.

Si bien los group descriptors especificaban las características en común que tenían todas las expresiones pertenecientes a un mismo grupo, ésto lo indicaban en formato legible denominado *markdown*, pensado principalmente como guía textual de la descripción de cada grupo. En cambio, para proveer al algoritmo de generación los parámetros necesarios para formar las expresiones de cada grupo se construyó un archivo de configuración en formato JSON (Javascript Object Notation) del cual se leían los atributos de cada grupo. Este archivo llamado `config.json` contaba con un conjunto de parámetros editables que indicaban de forma precisa la estructura de las expresiones que debía generar el algoritmo.

A continuación se presenta la especificación del archivo de configuración `config.json` junto con una explicación de cada uno de los campos:

```
{
   "expression_type": "POLYNOMIAL"|"SYSTEM_OF_EQ",
   "num_expressions": int,
   "coefficients": [
           "real": boolean,
          "complex": boolean,
          "decimal": boolean,
          "irrational": boolean,
          "rational": boolean ],
   "polynomials": {                            //utilizado si expression_type = polynomial
         "variables": str[],
         "degree": int|null,
         "max_degree": int,
         "is_complete": boolean,
         "order": "asc"|"desc"|null,
         "sign": "positive"|"negative"|"mixed"
    },
    "systems-of-eq": {                //utilizado si expression_type = system-of-eq
         "variables": str[],
         "num_unknowns": int,
         "num_equations": int,
         "is_complete": boolean,
         "homogeneous": boolean,
         "sign": "positive"|"negative"|"mixed",
```

```
      "both_sides": boolean,
      "is_ordered": boolean,
    }
}
```

- `expression_type:` La clase de expresiones a generar. Puede ser polinomios en caso de seleccionar el valor `POLYNOMIAL` o sistemas de ecuaciones cuando se define el valor `SYSTEM_OF_EQ`.
- `num_expressions:` La cantidad de expresiones a generar utilizando el archivo de configuración.
- `coefficients:` Lista de los posibles conjuntos numéricos de los cuales se deben seleccionar los coeficientes de las expresiones. Para cada conjunto numérico se especifica un valor booleano True o False para indicar si el conjunto debe ser considerado o no, respectivamente.
- `polynomials:` Parámetros de configuración que se aplican cuando el tipo de expresiones a generar es polinomios, es decir, cuando `expression_type = 'POLYNOMIALS'`
  - `polynomials.variables:` La lista de las variables de las cuales seleccionar para generar las indeterminadas de los polinomios. Cada variable es definida como una cadena de caracteres o string.
  - `polynomials.degree:` El grado específico de los polinomios a generar. Puede no definirse indicándose con valor `None`.
  - `polynomials.max_degree:` El grado máximo de los polinomios a generar en caso de no ser definido el atributo `polynomials.degree.`
  - `polynomials.is_complete:` Determina si los polinomios a generar deben incluir los términos con todas las potencias desde 0 hasta el grado.
  - `polynomials.order:` Determina el orden de los términos dentro del polinomio. Si se selecciona el valor `asc`, los términos se ordenan de forma ascendente según la potencia de la indeterminada. Si se especifica el valor `desc` se ordenan de forma descendente y en caso de ser `None`, ningún orden particular es aplicado.
  - `polynomials.sign:` Indica si los términos de los polinomios son positivos, negativos o mixtos con los valores `positive, negative` o `mixed` respectivamente
- `systems-of-eq:` Parámetros de configuración que se aplican cuando el tipo de expresiones a generar es sistemas de ecuaciones, es decir, cuando `expression_type = 'SYSTEM_OF_EQ'`
  - `systems-of-eq.variables:` La lista de las variables de las cuales seleccionar para generar las ecuaciones. Cada variable es definida como una cadena de caracteres o string.

- `systems-of-eq.num_unknowns:` La cantidad de variables a considerar en cada ecuación.
- `systems-of-eq.num_equations:` La cantidad de ecuaciones a generar para cada sistema.
- `systems-of-eq.is_complete:` Determina si las ecuaciones del sistema deben contener todas las variables o no.
- `systems-of-eq.is_ordered:` Determina si las variables de las ecuaciones del sistema se encuentran todas en el mismo orden.
- `systems-of-eq.homogenous:` Determina si el sistema es homogéneo.
- `systems-of-eq.sign:` Indica si los términos de las ecuaciones son positivos, negativos o mixtos.
- `systems-of-eq.both_sides:` Determina si los términos de las ecuaciones lineales deben estar distribuidos a ambos lados de la igualdad o todos sobre el miembro izquierdo de ella.

Actualmente, el algoritmo de generación de árboles de operadores que utiliza los archivos mencionados se encuentra en un repositorio de GitHub[4] de acceso público. Las funciones de generación permitieron producir 560 árboles diferentes que fueron posteriormente almacenados en la base de datos para su posterior parseo a pseudoLaTeX y uso dentro de la página de LEDProject.

Los siguientes son cuatro ejemplos de los árboles generados en notación prefija por el algoritmo mencionado anteriormente:

1. `CASES EQ SUM MUL VN x E MUL VF y E E VN EQ SUM MUL VN x E E VF E`
2. `SUM NEG VD NEG MUL VD x E NEG MUL VD POW x 2 E E`
3. `SUM VC MUL VC x E NEG MUL VC POW x 2 E MUL VC POW x 3 E E`
4. `SUM VD MUL VD x E MUL VD POW x 2 E MUL VD POW x 3 E MUL VD POW x 4 E E`

Tabla 4.1: *Especificación de los valores de los atributos para los 12 grupos descriptores definidos para el algoritmo de generación de árboles de operadores.*

Grupo	Atributo	Valor
Grupo 0	Tipo de expresión	“POLYNOMIALS”
	Cantidad	90
	Coeficientes	[“real”]
	Parámetros	- max-degree: 10

[4] Link a repositorio en GitHub: LINK

		- order: [null, 'asc', 'desc'] - sign: ['mixed', 'negative', 'positive'] - is_complete: true
Grupo 1	Tipo de expresión	"POLYNOMIALS"
	Cantidad	45
	Coeficientes	["complex"]
	Parámetros	- max-degree: 8 - order: [null, 'asc', 'desc'] - sign: ['mixed', 'negative', 'positive'] - is_complete: true
Grupo 2	Tipo de expresión	"POLYNOMIALS"
	Cantidad	45
	Coeficientes	["natural"]
	Parámetros	- max-degree: 8 - order: [null, 'asc', 'desc'] - sign: ['mixed', 'negative', 'positive'] - is_complete: true
Grupo 3	Tipo de expresión	"POLYNOMIALS"
	Cantidad	45
	Coeficientes	["fractional"]
	Parámetros	- max-degree: 8 - order: [null, 'asc', 'desc'] - sign: ['mixed', 'negative', 'positive'] - is_complete: true
Grupo 4	Tipo de expresión	"POLYNOMIALS"
	Cantidad	45
	Coeficientes	["decimal"]
	Parámetros	- max-degree: 8 - order: [null, 'asc', 'desc'] - sign: ['mixed', 'negative', 'positive'] - is_complete: true
Grupo 5	Tipo de expresión	"POLYNOMIALS"
	Cantidad	60
	Coeficientes	["fractional","natural"]
	Parámetros	- max-degree: 15 - order: [null, 'asc', 'desc'] - sign: ['mixed', 'negative', 'positive'] - is_complete: false
Grupo 6	Tipo de expresión	"POLYNOMIALS"
	Cantidad	40
	Coefficients	["fractional","natural", "complex"]
	Parámetros	- variables: ["y"] - max-degree: 15 - order: [null] - sign: ['mixed'] - is_complete: false
Grupo 7	Tipo de expresión	"POLYNOMIALS"
	Cantidad	40
	Coeficientes	["fractional","natural", "complex"]

	Parámetros	- variables: [“z”] - max-degree: 15 - order: [null] - sign: [’mixed’] - is_complete: false
Grupo 8	Tipo de expresión	“SYSTEM_OF_EQ”
	Cantidad	30
	Coeficientes	[“fractional”,”natural”]
	Parámetros	- variables: ["x","y"] - is_complete: [true, false] - sign: [’mixed’, ’negative’, ’positive’] - is_ordered: true - homogeneous: false - both_sides: false - num_equations: 2
Grupo 9	Tipo de expresión	“SYSTEM_OF_EQ”
	Cantidad	30
	Coeficientes	[“fractional”,”natural”]
	Parámetros	- variables: ["x","y"] - is_complete: true - sign: [’mixed’, ’negative’, ’positive’] - is_ordered: [true, false] - homogeneous: true - both_sides: false - num_equations: 2
Grupo 10	Tipo de expresión	“SYSTEM_OF_EQ”
	Cantidad	50
	Coeficientes	[“fractional”,”natural”]
	Parámetros	- variables: ["x","y",”z”] - is_complete: [true, false] - sign: [’mixed’] - is_ordered: [true, false] - homogeneous: [true, false] - both_sides: [true, false] - num_equations: [2, 3]
Grupo 11	Tipo de expresión	“SYSTEM_OF_EQ”
	Cantidad	40
	Coeficientes	[“fractional”,”natural”]
	Parámetros	- variables: ["a","b",”c”,”d”] - is_complete: [true, false] - sign: [’mixed’] - is_ordered: [true, false] - homogeneous: [true, false] - both_sides: [true, false] - num_equations: [2, 3, 4]

Generador de pseudoLaTeX

A pesar de formar un conjunto de expresiones algebraicas en forma de árbol, las cuales constituyeron parte del conjunto de datos de entrenamiento, como se verá en 4.2.2, resultó necesario atravesar otro paso intermedio donde aquellos árboles generados fueron transformados al formato personalizado de pseudoLaTeX. Ésto se realizó por dos razones: por un lado, para obtener casos de entrenamiento para el conjunto de datos de generación de pseudoLaTeX y, por otro lado, para la correcta renderización de las expresiones en las pantallas de los usuarios de LEDProject.

Esta conversión requirió el procesamiento y decodificación de las listas de símbolos de operadores y operandos de los árboles por medio de un algoritmo de parseo construido a medida para el contexto en cuestión. El parseador constituyó una porción de código encargada de analizar cada elemento de la secuencia de entrada representada por una lista de símbolos o tokens, y determinar si dicho elemento era un operador, un operando o un elemento de fin de operación. El algoritmo de parseo fue construido a partir de un conjunto de cuatro funciones que procesaban fragmentos de la secuencia tomando para cada operador hallado los correspondientes operandos involucrados. Luego, cada subconjunto de elementos seleccionados era enviado como entrada a una función de traducción que estructuraba los operandos en el orden necesario según la operación seleccionada, insertando símbolos adicionales entre ellos, como símbolos de suma, resta, fracción o exponenciación.

Es posible resumir el funcionamiento del proceso recursivo de parseo como una selección creciente de subconjuntos de elementos, traduciendo en primer lugar aquellas operaciones que estaban contenidas dentro de otras para luego utilizar el resultado de la traducción como un nuevo operando de una operación superior. Este efecto "burbuja" puede observarse en la Figura 4.4 que demuestra gráficamente un ejemplo del proceso de conversión de una lista de tokens a una expresión en pseudoLaTeX.

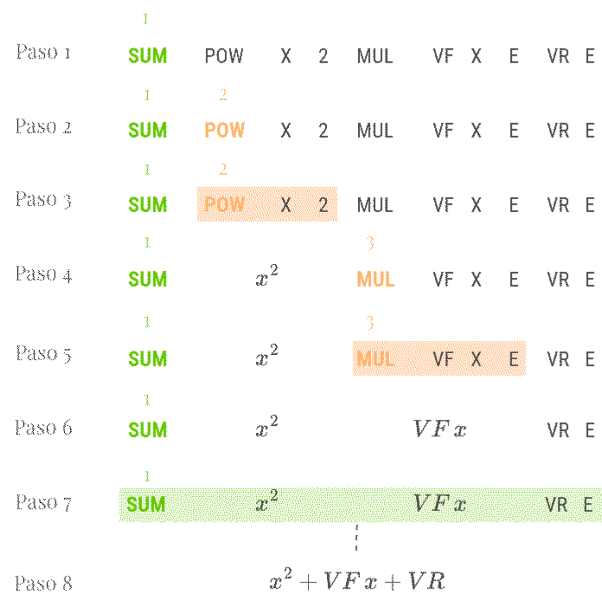

Figura 4.4: *Ejemplo del algoritmo de parseo de árboles de operadores. El cambio de color de un símbolo (pasos 1, 2 y 4) representa la acción de selección de un operador. Por otra parte, el cambio de color del fondo (pasos 3, 5 y 7) indica la acción de selección de los operandos involucrados en la operación previamente seleccionada.*

Resulta evidente de la Figura 4.4 anterior que la salida del algoritmo de parseo y traducción no representa una expresión algebraica en formato LaTeX con números específicos, sino que la presencia de los *operandos variables* VF y VR denotan que la expresión resultante está, efectivamente, en formato pseudoLaTeX.

Finalmente, para obtener expresiones algebraicas válidas que pudieran enviarse a los usuario de LEDProject, es decir, con valores numéricos en lugar de operandos variables, se utilizó una función auxiliar que tomaba como entrada un extracto de pseudoLaTeX para luego identificar los operandos variables y reemplazarlos con valores numéricos aleatorios que respetaran los conjuntos numéricos especificados para cada tipo de operando.

A continuación se muestran para los ejemplos de árboles vistos en el apartado anterior los distintos formatos que atravesaron (en orden) antes de ser expuestos al usuario en LEDProject:

1. **Árbol**: `CASES EQ SUM MUL VN x E MUL VF y E E VN EQ SUM MUL VN x E E VF E`

   pseudoLaTeX: `\begin{cases} VN x + VF y = VN \\ VN x = VF \end{cases}`

   LaTeX: `$\begin{cases} 5x + \frac{1}{4}y = 10 \\ 4x = \frac{1}{2} \end{cases}$`

2. **Árbol**: `SUM NEG VD NEG MUL VD x E NEG MUL VD POW x 2 E E`

   pseudoLaTeX: `- VD - VD x - VD x^2`

   LaTeX: `$ -1.5 - 2.4x - 12.01x^2 $`

3. **Árbol**: `SUM VC MUL VC x E NEG MUL VC POW x 2 E MUL VC POW x 3 E E`

   pseudoLaTeX: `VC + VC x - VC x^2 + VC x^3`

   LaTeX: `$ 4i + (1+2i)x - (22+i)x^2 + (3+2i)x^3 $`

4. **Árbol**: `SUM VN MUL VN x E MUL VF POW x 2 E MUL VD POW x 3 E MUL VN POW x 4 E E`

   pseudoLaTeX: `VN + VN x + VF x^2 + VD x^3 + VN x^4`

   LaTeX: `$ 3 + 10x + \frac{1}{4}x^2 + 2.5x^3 + 30x^4 $`

4.2.2 Modelo físico de la base de datos

Una vez especificados los distintos formatos de las entradas y salidas del modelo, fue posible pensar en la implementación de un modelo físico que representara correctamente al conjunto de datos a recolectar. Para ello se utilizó un sistema de gestión de bases de datos denominado PostgreSQL que permitió la creación y administración de una base de datos relacional con dos tablas. Una de ellas fue utilizada para modelar las expresiones, para las cuales se almacenaron los tres formatos mencionados en los apartados anteriores: el formato de árbol de operadores, formato pseudoLaTeX y, por último, el

código LaTeX. A su vez, esta tabla registraba para cada expresión el número del grupo descriptor al que pertenecía y la cantidad de veces que fue entregada a un usuario dentro de la página web para su descripción.

Luego, debido a que cada una de las expresiones podían ser descritas de múltiples formas por distintos usuarios, se especificó otra tabla separada de descripciones, cuyos registros almacenaban las cadenas de texto de las descripciones en español y una clave foránea que referenciaba a la expresión correspondiente. De esta forma se consiguió un modelo de datos normalizado sin redundancias.

La figura 4.5 muestra el diagrama Entidad-Relación que modela las tablas implementadas de la base de datos.

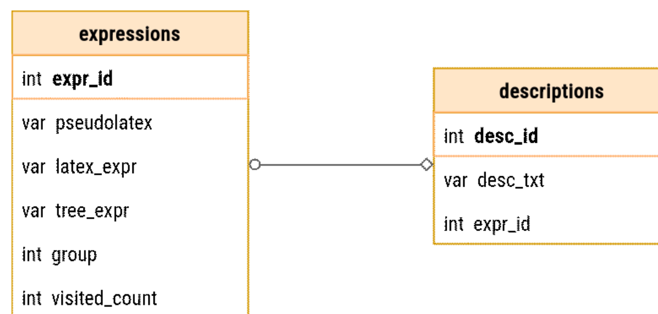

Figura 4.5: *Diagrama Entidad-Relación de la base de datos utilizada para LEDProject.*

Cabe resaltar que este modelo físico de la base de datos se utilizó para almacenar los datos servidos y recolectados en LEDProject, incluyendo tanto las expresiones generadas como las descripciones provistas por los alumnos de la universidad.

Esta base de datos, si bien sirvió como punto de partida para la creación del conjunto de datos de entrenamiento del modelo, requirió de un posterior procesamiento de curado, en pos de eliminar los errores introducidos durante la etapa de recolección. El detalle de dicho preprocesamiento será introducido en la sección 4.3, donde se presentarán las distintas etapas de curado que condujeron a la construcción del conjunto final de datos servidos al modelo.

4.2.3 Tecnologías de desarrollo de LEDProject

En este apartado se describirá la arquitectura subyacente del proyecto que permitió la construcción de LEDProject. Ésta consistió en cuatro componentes principales:

Interfaz de usuario (Frontend)

Constituyó el componente visual que permitió la interacción directa entre el usuario y la lógica de aplicación. Representó todo el conjunto de componentes gráficos como imágenes, botones y campos de texto. El diseño de la interfaz de usuario, también denominada UI, fue particularmente adaptado a dispositivos móviles ya que se consideró a las aulas de la universidad como espacio principal donde se haría uso de la página, siendo altamente probable que los alumnos tuvieran consigo un teléfono celular.

A la hora de diseñar la página web se utilizó la herramienta de diseño *Figma*, procurando mantener una UI limpia y sencilla, evitando distracciones para el usuario y enfocando la atención a la tarea principal del proyecto: recolectar descripciones para una base de datos. Ejemplos del diseño pueden observarse en los “mockups” de la Figura 4.3 presentada al inicio de la sección 4.2.

Las acciones realizadas por cada usuario sobre los campos y botones de la página fueron captadas por funciones apodadas “event listeners” (oyentes de evento) que ejecutaban un conjunto de sentencias al ocurrir uno de los eventos que esperaban, como por ejemplo, el pulsado de un botón. Dichas funciones procesaban la información requerida con el fin de realizar cambios en la interfaz o enviar a la lógica de aplicación mensajes en protocolo HTTP con datos a almacenar o solicitar.

Lógica de aplicación (Backend)

Si bien la interfaz de usuario representó el punto de interacción entre el usuario y la página web, la lógica de aplicación se definió por medio de un conjunto de funciones que cumplían tres clases de propósito:

- **Recibir información del front-end:** Aquella información que debía ser almacenada en una base de datos primeramente debía ser recibida y procesada con el fin de insertarse correctamente como registro en las tablas de la base de datos.

- **Enviar información hacia el front-end:** La información que esperaba el usuario debía seleccionarse y extraerse de la base de datos, empaquetarse correctamente siguiendo formatos específicos y enviarse para su posterior exposición en los componentes de la UI. Un ejemplo de esta información requerida son las expresiones en formato LaTeX que se renderizaban en un componente particular del descriptor de expresiones.

- **Búsqueda, inserción y modificación de información en la base de datos**: El backend representaba el puente que permitía la comunicación de la información que fluía desde la interfaz del usuario con las tablas de la base de datos que la almacenaban, haciendo posible la búsqueda, inserción y modificación de los registros requeridos.

El conjunto de funciones de recepción y envío de información conformaron lo que se denomina una API REST. En términos simples, representaba una interfaz entre la lógica de fondo de la aplicación y los componentes de UI con los que interactuaba el usuario. La comunicación entre estas dos secciones de la página se estableció mediante el protocolo de red de capa de aplicación HTTP, utilizando mensajes GET y POST entre los distintos dominios del frontend y backend.

Base de datos

Con el fin de almacenar la información requerida por los usuarios y las descripciones que introducían desde la interfaz gráfica, fue necesaria la implementación del modelo físico de la base de datos

presentado en el apartado 4.2.2. La lógica de aplicación realizaba consultas y actualizaciones sobre los registros almacenados dentro de las tablas de esta base de datos relacional.

Servicio de Alojamiento Web (Hosting)

Tanto el frontend, backend y el DBMS PostgreSQL con la base de datos de la página se encontraban alojados en un mismo sitio de alojamiento de servicios web, denominado comúnmente como *hosting*, permitiendo el acceso a la página por parte de los usuarios en cualquier instante de tiempo y ubicación que desearan.

La siguiente Figura 4.6 muestra gráficamente la arquitectura general de LEDProject, indicando los componentes previamente descritos y la orientación de sus comunicaciones:

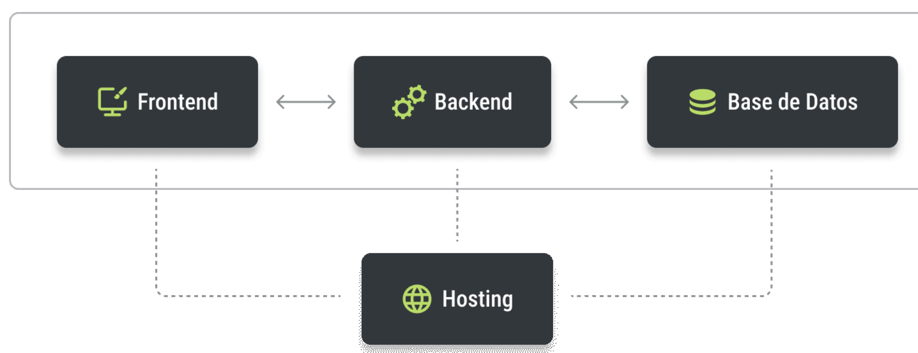

Figura 4.6: *Esquema general de los componentes de la arquitectura de LEDProject.*

Ahondando más en detalle, a continuación se especificarán las tecnologías particulares con las que se implementaron cada uno de los componentes de la arquitectura. Cabe destacar que las tecnologías fueron seleccionadas priorizando la familiaridad con la herramienta y la facilidad de uso a la hora de realizar las implementaciones en código, sin tomar decisiones particulares respecto a su nivel de performance durante la ejecución.

Para comprender correctamente el uso de cada tecnología seleccionada se recomienda observar la Figura 4.7 que demuestra la disposición a través de la arquitectura del proyecto de cada una de las herramientas que se describirán a continuación.

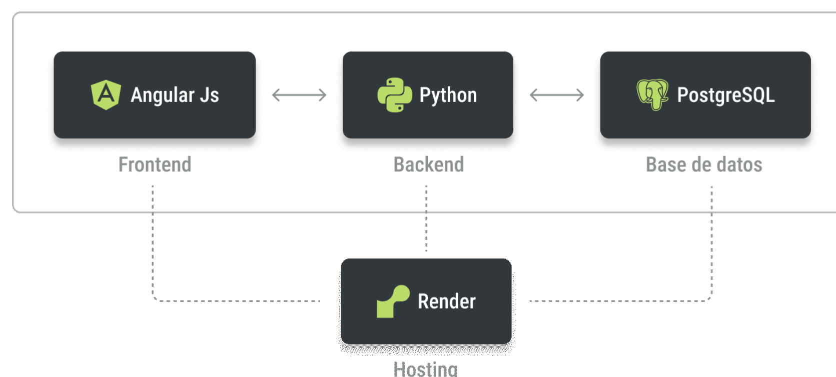

Figura 4.7: *Esquema de las tecnologías específicas utilizadas en los componentes de la arquitectura de LEDProject.*

Angular JS

Categorizado como uno de los principales frameworks de desarrollo de interfaces de usuario, esta herramienta basada en TypeScript y mantenida por Google hace posible la implementación de componentes visuales modularizados que se muestran al usuario proveyendo información y modos de interacción con la aplicación. El componente de frontend fue completamente implementado utilizando esta herramienta de desarrollo que permitió la definición de mensajes HTTP a los controladores de la API especificada en el backend. Por otra parte, utilizando componentes particulares que contiene Angular JS por defecto, se construyó el denominado mecanismo de "routing" que permitó la navegación entre las distintas secciones de la página web.

Vale realizar una mención a la librería *KaTeX* que fue utilizada dentro del código TypeScript del framework para renderizar las cadenas de caracteres que representaban expresiones en formato LaTeX dentro de la sección del descriptor de expresiones.

Python

De forma similar a como se desarrolló la interfaz gráfica, se utilizó un framework de *Python* llamado *Flask* para implementar parte de la lógica de la aplicación, particularmente para la construcción de la API REST que recibía y enviaba la información desde y hacia el front-end. A pesar de existir múltiples herramientas de desarrollo similares, se seleccionó el framework de back-end Flask por la familiaridad preexistente con la tecnología y la sencillez en su uso. Cada una de las funciones (controladores) definidos para la API en el lenguaje de programación Python tenían asociadas un dominio o URL y eran ejecutadas correspondientemente al recibirse mensajes HTTP con los verbos GET y POST sobre dichos dominios.

Por otro lado, cabe destacar que se utilizó una librería de Python llamada *psycopg2* que contiene un conjunto de objetos y funciones que hizo posible la comunicación entre el back-end y el servidor de la base de datos PostgreSQL. Este paquete de código se utilizó para definir y establecer conexiones con el servidor de base de datos y realizar consultas en el lenguaje estructurado SQL.

PostgreSQL

Como fue mencionado en el apartado 4.2.2, la implementación del modelo físico de la base de datos donde se almacenaron las expresiones algebraicas y sus respectivas descripciones se realizó mediante el sistema de gestión de base de datos (DBMS) *PostgreSQL*, a través del cual se administró la creación de las tablas, la inserción de las expresiones y descripciones, y sus correspondientes actualizaciones. A su vez, este sistema fue capaz de gestionar las consultas realizadas por los usuarios finales para la asignación de los subconjuntos de expresiones algebraicas. Finalmente permitió la exportación de toda la información recolectada en formato CSV para su posterior procesamiento.

Render

La plataforma de alojamiento de servicios en la web *Render* permitió globalizar el acceso a la página web, extendiéndolo hacia cualquier usuario, independientemente del horario y lugar donde se situara. Debido a la imposibilidad de mantener un servidor ejecutando continuamente y sin interrupciones en la máquina donde fue desarrollado el proyecto, resultó necesario utilizar el servicio de hosting Render. Éste hizo posible ubicar cada componente de la aplicación en los servidores de esta plataforma, la cual brindó a cada servicio creado un dominio distinto a través de los cuales se comunicaron fluidamente el frontend, backend y el servidor de la base de datos.

4.2.4 Resultados de la recolección

Luego de finalizado el período definido de tres meses de recolección de descripciones, se afirmó el logro del objetivo dispuesto para LEDProject. Con la ayuda de esta herramienta se consiguió satisfactoriamente la recolección de más de setecientas descripciones que expresaban en formato de texto las características principales de las expresiones algebraicas provistas a los usuarios.

A pesar de definir un alcance acotado de su uso en el presente trabajo, LEDProject fue pensado desde el inicio como una herramienta web con la posibilidad de ser ampliamente extendida para abarcar toda clase de expresiones de cualquier área de la matemática, como el álgebra, cálculo, geometría analítica, entre muchas otras. Simplemente bastaría con añadir las nuevas expresiones en formato LaTeX a la base de datos para que sean distribuidas a los usuarios que utilicen la página web y sean descriptas correspondientemente. Resulta correcto afirmar entonces que LEDProject no solamente fue ideado con una utilidad inmediata sino con una potencialidad de extensión a futuro, siendo posible recolectar más información para entrenar nuevos modelos de aprendizaje profundo más sofisticados.

4.3 Preprocesamiento de datos

En términos concretos, el preprocesamiento de los datos comenzó obteniendo la combinación (INNER JOIN) de las tablas de descripciones y expresiones algebraicas de la base de datos de LEDProject, de forma de construir registros completos que contuvieran todos los formatos de la fórmula (árbol, pseudoLaTeX y LaTeX) junto con la descripción asociada. Luego, este conjunto de datos resultante fue exportado en formato CSV desde el DBMS y posteriormente importado al servicio de Google Sheets provisto por Google como un SaaS gratuito, sobre el cual se estructuraron los datos en forma tabular.

Los siguientes apartados describirán las etapas de preprocesamiento donde se corrigieron, actualizaron, etiquetaron y aumentaron los datos importados en Google Sheets, para finalmente formar el conjunto de datos curados de entrenamiento del modelo.

4.3.1 Curado de datos recolectados

A pesar de buscar un gran número neto de descripciones a través de la colaboración de los alumnos de la universidad, se contempló desde un inicio la posibilidad de hallar casos que no reflejaran correctamente las características de las expresiones algebraicas presentadas. Casos como descripciones incongruentes o errores de tipeo y ortografía fueron la razón por la que se implementó una primera etapa de limpieza de datos. Entre los errores cometidos con mayor frecuencia se destacan los dos siguientes:

- Discrepancias conceptuales entre la expresión algebraica y su respectiva descripción: Esta clase de errores fueron descartados debido a que proveían descripciones incorrectas que hubieran impactado negativamente en el proceso de entrenamiento.

- Errores de ortografía e introducción de valores no permitidos como los saltos de línea: Estos errores fueron corregidos manualmente al momento de ser identificados.

De forma ilustrativa, los siguientes dos ejemplos demuestran los casos anteriormente mencionados de errores frecuentes:

Descripción	"Polinomio de grado 9 \n con coeficientes enteros."
Expresión (LaTeX)	"4x^9 + 19x^8 - 20x^4 + 3 + 12x^2"
Tipo de error	Caracter inválido (salto de linea \n)
Solución	Eliminación manual del caractér inválido

Descripción	"Ecuación de numeros complejos"
Expresión (LaTeX)	"(19+24i).x - (12+20i)"
Tipo de error	Conceptual en la descripción
Solución	Eliminación completa de la descripción

Para identificar los errores se recorrió la totalidad del conjunto de datos, validando de forma manual la correctitud de las descripciones provistas para cada expresión algebraica y aplicando los criterios de corrección correspondientes para cada clase de error.

4.3.2 Afinación de árboles de operadores

Paralelamente al proceso de corrección de errores, se identificaron casos donde las descripciones especificaban características o elementos puntuales que no se veían reflejados sobre los árboles de operadores a los que referían. Esto se debió a que los usuarios de LEDProject recibían fórmulas con valores específicos como la siguiente:

$$-5x^5 + 31x - 23$$

mientras que los árboles a partir de los cuales se derivaron simplemente contenían operandos variables, como por ejemplo, el operando "VN" para el caso anterior:

```
SUM NEG MUL VN POW x 5 E MUL VN x E NEG VN E
```

De esta forma, al describir la expresión anterior como:

"Polinomio de grado 5 incompleto con coeficiente principal -5"

el usuario podría especificar el valor particular del coeficiente principal del polinomio (-5), anulando la posibilidad que éste fuera un valor variable, lo cual no es correctamente reflejado al utilizar el operando "VN" en el formato de árbol.

Ante esta discrepancia entre las descripciones provistas en LEDProject y los árboles de operadores sobre los que debía entrenar el modelo, se modificaron manualmente aquellos árboles que requerían un cierto grado de afinación, cambiando operandos variables por números específicos. Este proceso tomó lugar al mismo tiempo que la corrección de errores y, si bien estos casos no fueron considerados como tales, se realizaron las actualizaciones necesarias para conseguir un conjunto de datos de entrenamiento lo más fiel y preciso posible.

4.3.3 Etiquetado de descripciones

Posteriormente, una vez reunidas todas las descripciones correctas a considerar, es decir, luego de la corrección de errores, se procedió al etiquetado manual de cada descripción para la especificación de las entidades nombradas contenidas en ellas, siguiendo el esquema o notación BIO (vista en 3.5).

Tomando en consideración al diccionario de entidades introducido en la sección 4.1.2, se optó por utilizar una herramienta que agilizara el mecanismo de identificación manual de dichas entidades y su exportación a un formato útil para su posterior procesamiento. De esta manera se seleccionó la plataforma de código abierto *Label Studio*, que permite el etiquetado de diversos tipos de datos, como imágenes, videos, audio, series temporales y, particularmente, texto. A su vez, vale destacar que la plataforma cuenta con una plantilla predefinida dedicada al reconocimiento de entidades nombradas que respeta la notación BIO, razones por la cual la elección resultó ser una tarea simple.

Por otro lado, la Figura 4.9 detalla dos ejemplos que ilustran el producto final del etiquetado que se realizó sobre cada una de las descripciones provistas

Descripción	"Polinomio de grado 4, completo"
Anotación	"Polinomio [B-EXTYPE] de [O] grado [B-ATRIB] 4 [B-ATVAL] , [O] completo [B-EXCHAR]"

Descripción	"Sistema de ecuaciones con 1 incógnita"
Anotación	"Sistema [B-EXTYPE] de [I-EXTYPE] ecuaciones [I-EXTYPE] con [O] 1 [B-ELCOUNT] incógnita [B-ELEM]

Figura 4.9: *Ejemplos de etiquetado de descripciones de expresiones algebraicas para especificación de entidades nombradas.*

El proceso completo consistió en una primera exportación del conjunto de descripciones y sus respectivos identificadores desde Google Sheets a un formato CSV. Luego, al importar dicho archivo dentro de la plantilla de etiquetado de Label Studio se generó una tarea para cada descripción cargada que permitió realizar la identificación de las entidades en el texto. Al finalizar la iteración sobre todo el conjunto de datos, se exportaron los resultados en dos formatos, CONLL-U y JSON, que fueron finalmente procesados por un pequeño algoritmo[5] desarrollado en Python particularmente para la lectura de estos archivos y la reconstrucción de las descripciones etiquetadas en un formato legible y correcto para su ingesta durante el entrenamiento del modelo.

4.3.4 Aumento de volúmen de datos

Debido a la necesidad de obtener un conjunto de datos de entrenamiento lo más grande posible, la última etapa del preprocesamiento de datos consistió en aplicar sucesivamente técnicas de aumento del volúmen de datos mediante leves modificaciones de las formas originales de éstos.

En el caso del presente trabajo se aplicaron tres clases de aumento sobre el texto de las descripciones en español:

- **Conversión de números a palabras**: Aquellas descripciones que contenían números del 1 al 10 fueron replicadas convirtiendo aquellos tokens que representaban números por su versión en palabras. Aplicando esta técnica se logró obtener más de trescientas expresiones diferentes a las ya existentes, ampliando el conjunto original de datos.

- **Intercambio de palabras por abreviaciones**: En el caso de las descripciones de polinomios que contenían menciones al atributo "grado", los tokens que indicaban el valor en palabras de éste fueron modificados por abreviaciones, como por ejemplo “primer grado” fue transformado a “1er grado" y “segundo grado” a “2do grado”.

- **Adición de prefijos**: El formato esperado de las entradas del modelo preentrenado T5S utilizado en el trabajo (derivado de T5 visto en 3.4.4), y que será descrito en el siguiente capítulo, requirió

[5] Link a archivo .py con el algoritmo en repositorio GitHub: LINK

añadir al incio de cada descripción una frase que detallara la tarea a cumplir durante el entrenamiento. En este caso se añadieron tres prefijos distintos: "generar pseudolatex:" para la tarea de generación de pseudoLaTeX, "generar lista:" para indicar la generación de árboles de operadores (lista de símbolos), y "reconocer entidades nombradas:" para el reconocimiento de entidades nombradas. Ésto resultó en la triplicación del conjunto de ejemplos, donde todas las descripciones de cada réplica tenían como prefijo una de las anteriores frases de tarea.

Finalmente, partiendo de poco más de setecientos casos en el conjunto original de datos, al aplicar este proceso de aumento de volumen se obtuvo una cantidad final de alrededor de 3700 ejemplos, logrando quintuplicar el número original de datos.

CAPÍTULO 5

Desarrollo del modelo

Habiendo establecido en el capítulo previo los elementos fundamentales para la resolución potencial de la tarea de generación de expresiones en pseudoLaTeX, entre ellos las anotaciones de entidades en las descripciones en lenguaje natural y los árboles de operadores para la representación de fórmulas algebráicas, en el presente capítulo se describirá la implementación central de este trabajo de grado: el desarrollo del modelo de aprendizaje profundo.

En la primera sección 5.1 se presentará la hipótesis central que definió el flujo completo del proceso de implementación del modelo final y las etapas intermedias para lograr un correcto estudio y validación de su comportamiento. En el apartado 5.2 se detallarán los usos particulares de los datos reunidos con el subproyecto LEDProject visto en el capítulo anterior y cómo éstos fueron distribuidos en las respectivas etapas de desarrollo de forma de lograr resultados que permitieran un análisis completo y gradual del comportamiento del modelo. Por último, la sección 5.3 ahondará en los detalles de cómo se seleccionaron los hiperparámetros de entrenamiento y la arquitectura física de la computadora sobre la que se realizaron las ejecuciones de los bucles de aprendizaje e inferencia.

5.1 Hipótesis y modelos

A lo largo del capítulo anterior se describieron los aspectos analizados sobre la naturaleza del problema a resolver: la estructura jerárquica inherente de las expresiones algebraicas y las entidades incluidas dentro de las descripciones en español junto a su sintaxis. Como fue mencionado en los apartados 4.1.2 y 4.1.3 del capítulo previo, para representar dichos elementos se tomó por un lado la implementación de los árboles de operadores y por el otro el reconocimiento de entidades nombradas, siendo ambos elementos disparadores de la hipótesis central que acompañó el desarrollo del modelo del presente trabajo:

> *¿Tomando al reconocimiento de entidades nombradas para representar la gramática de las oraciones del lenguaje natural y a los árboles de operadores para embeber la estructura jerárquica de las expresiones algebráicas, incorporarlos dentro del proceso de entrenamiento del modelo como tareas auxiliares disminuiría el error cometido durante la generación de fórmulas algebraicas en pseudolatex?*

De forma de obtener un aporte de análisis significativo y una validación completa de la hipótesis se definió un flujo de trabajo seccionado en etapas que permitió lograr una visión gradual de cuánto sumaba cada elemento al rendimiento del modelo. Se plantearon cuatro fases que incluyeron el entrenamiento independiente de cuatro redes neuronales sobre diferentes conjuntos de datos, las cuales representaron evoluciones de un mismo modelo final. En las secciones que se encuentran a continuación se describirán para cada uno de ellos los puntos principales considerados durante su implementación y la porción de hipótesis que buscó validar.

5.1.1 Modelo inicial

La primera etapa de análisis de rendimiento del modelo final consistió en el fine-tuning de un modelo pre-entrenado sobre un conjunto de datos que solamente consideraba ejemplos de generación de expresiones algebráicas en pseudoLaTeX. De esta forma, con el objetivo de definir un punto base de comparación, se partió de un modelo que simplemente fuera entrenado sobre una única tarea donde dicha red neuronal pre entrenada tuviera escaso "conocimiento" del formato de las expresiones a generar y no incluyera la información adicional sobre los elementos gramaticales de las descripciones ni la estructura jerárquica de las fórmulas esperadas.

Las siguientes tuplas (entrada y salida), son algunos ejemplos de los datos servidos para el entrenamiento de este modelo:

`Entrada:` *"generar pseudolatex: Polinomio en variable "y" de décimo grado, incompleto, con 6 términos"*

`Salida: VNy^{3} - VC + VC.y^{6} - VNy^{10} - VNy^{5} - VNy`

`Entrada:` *"generar pseudolatex: Polinomio de séptimo grado incompleto en indeterminada z"*

`Salida: VFz - VNz^{5} + VF - VNz^{6} - VNz^{2} + VNz^{7}`

`Entrada:` *"generar pseudolatex: Sistema de ecuaciones compatible indeterminado, dos ecuaciones y 3 incógnitas"*

`Salida: \begin{cases} VN.z + VN.y + VF.x = VF \\VN.y + VF.z - VN.x = VN \end{cases}`

Tomándolo como punto de partida, el desarrollo de este modelo y la obtención de resultados de su comportamiento sobre datos de prueba permitió resolver el siguiente interrogante:

¿Para definir un punto de comparación, cuánto error comete el modelo al ser entrenado sólo con ejemplos de generación de expresiones algebraicas en formato pseudoLaTeX?

5.1.2 Modelos intermedios

Prosiguiendo con una siguiente etapa evolutiva del modelo final, se implementó un segundo modelo entrenado en simultáneo para la resolución de dos tareas: la generación de expresiones algebraicas en

pseudoLaTeX a partir de descripciones en español, al igual que el modelo previamente mencionado, y el reconocimiento de entidades nombradas incluidas dentro de dichas descripciones.

Como se detallará en la siguiente sección 5.2, el modelo pre-entrenado T5S sobre el cual se realizó el fine-tuning para construir los modelos de cada etapa, posee una arquitectura basada en el modelo T5. Así como se destacó en la sección 3.6, éste LLM trata a todo problema de NLP como una tarea de texto-a-texto, lo cual permite el soporte directo para la aplicación del aprendizaje multitarea. Luego, tomando provecho de esta característica, en este caso, se buscó que el primer modelo intermedio aprendiera a realizar tanto la tarea principal de generación de pseudoLaTeX como la tarea auxiliar de reconocimiento de entidades nombradas, en pos de responder a la siguiente porción de hipótesis:

> *¿Tomando como punto de referencia al modelo inicial, el error cometido al generar las expresiones algebraicas en formato pseudoLaTeX disminuye al incorporar el reconocimiento de entidades nombradas como tarea auxiliar durante el proceso de aprendizaje?*

Se debe destacar que para el entrenamiento de este primer modelo intermedio, el conjunto de datos fue constituido por casos de entrada-salida para la resolución de ambas tareas en forma proporcional. Los siguientes son tres ejemplos de las tuplas de datos incluidas en dicho conjunto:

`Entrada:` *"reconocer entidades nombradas: Polinomio de 3er grado sin término cuadrático"*
`Salida:` *"Polinomio [B-EXTYPE] de [O] 3er [B-ATVAL] grado [B-ATRIB] sin [O] término [B-ELEM] cuadrático [I-ELEM]"*

`Entrada:` *"generar pseudolatex: Polinomio de séptimo grado incompleto en indeterminada z"*
`Salida: VFz - VNz^{5} + VF - VNz^{6} - VNz^{2} + VNz^{7}`

`Entrada:` *"reconocer entidades nombradas: sistema de ecuaciones de 3 ecuaciones"*
`Salida:` *"sistema [B-EXTYPE] de [I-EXTYPE] ecuaciones [I-EXTYPE] de [O] 3 [B-ELCOUNT] ecuaciones [B-ELEM]"*

Notar que el conjunto de datos contiene casos de entrada para las dos tareas, que se indican correspondientemente con los prefijos "generar pseudolatex" o "reconocer entidades nombradas", como fué previamente descrito en 4.3.4. Finalmente, al recordar lo mencionado en 4.3.3, es posible identificar que las salidas de aquellos casos destinados al reconocimiento de entidades nombradas respetan la notación BIO.

Posteriormente, y de forma análoga al anterior, se construyó un segundo modelo intermedio con la misma arquitectura pero con un conjunto diferente de datos de entrenamiento. Éste consistía, por una parte, en pares de entrada-salida como casos para la generación de pseudoLaTeX (de forma exacta al modelo inicial) y, por otro lado, se añadieron pares de descripciones y árboles de operadores, expresados como lo visto en 4.1.3. De esta forma, el segundo modelo intermedio fue orientado a un entrenamiento basado en mulitask learning con una tarea principal de generación de pseudoLaTeX y una tarea auxiliar

de generación de árboles de operadores en notación prefija. La razón para la construcción de este tercer modelo fue la búsqueda de la respuesta a una tercer parte de la hipótesis:

> *¿Tomando como punto de referencia al modelo inicial, el error cometido al generar las expresiones algebraicas en formato pseudoLaTeX disminuye al añadir la generación de árboles como segunda tarea durante el proceso de aprendizaje?*

De manera análoga al segundo modelo intermedio, éste contaba con un conjunto de datos formados por casos como los que se listan a continuación:

`Entrada:` *"generar lista: Polinomio de grado 6, coeficiente principal -75/154, incompleto"*
`Salida: SUM NEG MUL VN POW x 3 E NEG MUL FRAC 75 154 POW x 6 E E`

`Entrada:` *"generar pseudolatex: Polinomio de séptimo grado incompleto en indeterminada z"*
`Salida: VFz - VNz^{5} + VF - VNz^{6} - VNz^{2} + VNz^{7}`

`Entrada:` *"Sistema de ecuaciones con 2 ecuaciones y 2 incógnitas"*
`Salida: CASES EQ SUM NEG MUL VN x E NEG MUL VF y E E VN EQ SUM NEG MUL VF y E NEG MUL VN x E E VN E`

donde se combinan pares de entrada-salida para resolver ambas tareas de generación.

5.1.3 Modelo final

Finalmente se construyó el último modelo definitivo que agrupó el comportamiento de aquellos mencionados previamente. En este caso final, la red neuronal fue entrenada sobre tres tareas de forma simultánea para la resolución de:

- La generación de expresiones algebraicas en formato pseudoLaTeX.
- El reconocimiento de entidades nombradas en descripciones en español de fórmulas.
- La generación de árboles de operadores como listas de símbolos.

Es decir, durante el desarrollo de este último modelo se buscó reunir el comportamiento de los previos modelos intermedios e inicial, de forma de lograr una red neuronal con una mayor eficacia y exactitud al generar pseudoLaTeX.

Cabe destacar nuevamente que los modelos neuronales previamente introducidos en los apartados 5.1.1 y 5.1.2 fueron implementados con el mero objetivo de definir puntos de comparación que permitieran apreciar una mejoría gradual en el rendimiento de este modelo final. De dicha forma, fue posible reunir los resultados descritos en el siguiente capítulo 6 y dar una respuesta concreta a la hipótesis mencionada al inicio de la presente sección 5.1.

5.2 Implementación

Con el objetivo de desarrollar los modelos planteados y obtener resultados que permitan un análisis valioso se tomó como modelo base a una variante de T5 presentado en 3.4.4, denominado T5S, sobre el cual se aplicó la técnica de fine-tuning. El primer apartado 5.2.1 de esta sección detallará las razones de su elección y los beneficios que aportó al proceso de implementación de los correspondientes modelos neuronales. Por otra parte, en 5.2.2 se describirán las tecnologías de desarrollo utilizadas para llevar a cabo la implementación del código de confección y entrenamiento de las redes neuronales.

5.2.1 Modelo de lenguaje base

El hecho de considerar descripciones de expresiones algebraicas particularmente en idioma español no sólo limitó la recolección de ejemplos de entrenamiento (como se vió en 4.2), sino que también condicionó la elección del LLM sobre el cual realizar *fine-tuning*.

Dado que el modelo T5, descrito en la sección 3.4.4, contaba con una arquitectura basada en encoder-decoder de Transformer y trataba a toda tarea de NLP como un problema de procesamiento texto-a-texto, representó la primera opción seleccionada como modelo de lenguaje base para el desarrollo del análisis de esta tesina. Sin embargo, se identificó y se puso en relieve en 3.4.4 la particularidad del conjunto de datos sobre el que fue pre-entrenado, el cual solamente contemplaba ejemplos en idiomas inglés, francés, alemán y rumano. Al no considerar casos en español, tanto los parámetros internos de la red como las representaciones vectoriales de los tokens del vocabulario, no fueron entrenados para incorporar información relevante de secuencias de texto en éste idioma. Ésto podría haber provocado un entrenamiento ineficaz durante el fine-tuning debido a la necesidad de aprender desde el inicio un nuevo vocabulario.

A raíz de esta limitación se realizó una búsqueda de modelos de lenguaje con un esquema arquitectónico basado en encoder-decoder que hayan sido previamente entrenados sobre datos que contuvieran texto en idioma español. Fue así cómo se halló el artículo "Sequence-to-Sequence Spanish Pre-trained Language Models" (Araujo et al., 2024) en el cual los autores destacan la complejidad de encontrar modelos de lenguaje basados en esta arquitectura particular de Transformer y que consideren al idioma español como parte de su entrenamiento:

> While Spanish language models based on BERT and GPT have demonstrated proficiency in natural language understanding and generation, there remains a noticeable scarcity of encoder-decoder models explicitly designed for sequence-to-sequence tasks, which aim to map input sequences to generate output sequences conditionally. [Mientras que los modelos de lenguaje de español basados en BERT y GPT han demostrado competencia en el entendimiento y generación de lenguaje natural, continúa habiendo una notable escasez de modelos encoder-decoder explícitamente diseñados para tareas secuencia-a-secuencia, las

cuales buscan mapear secuencias de entrada para generar condicionalmente secuencias de salida] (Araujo et al., 2024, p.1)

El aspecto más relevante de este artículo para el presente trabajo fue la introducción de un nuevo modelo de lenguaje denominado T5S, basado en la arquitectura del modelo T5 Base. Éste incluye 12 bloques en cada pila de encoder y decoder, 12 cabezales de atención en cada capa de self-attention y capas ocultas de 768 dimensiones. Particularmente su vocabulario en español está formado por 32.000 tokens obtenidos con la librería *sentencepiece* que contiene los algoritmos de tokenización BPE y unigram model. Finalmente, cabe destacar que la función de error que utiliza este modelo durante el proceso de entrenamiento es la Entropía Cruzada Multiclase descrita en 2.2.1

Considerando que los resultados demostrados en el artículo destacaban al modelo T5S como uno de los dos más performantes en la resolución de múltiples tareas de NLP (de entre otros modelo desarrollados por los autores), se optó por seleccionar a éste LLM como el punto base sobre el cual se aplicó *fine-tuning* para obtener los demás modelos planteados en el análisis de este trabajo.

5.2.2 Herramientas y desarrollo

Con el fin de lograr una implementación efectiva y eficiente de los modelos de aprendizaje profundo planteados, se priorizó la utilización del lenguaje de programación Python debido a la simplicidad de su sintaxis y a la velocidad de producción de código. A su vez, se seleccionó un conjunto de herramientas ágiles de desarrollo que utilizaban como base dicho lenguaje de programación. A continuación se detallarán las características principales de dichas tecnologías y una descripción de las tareas realizadas con ellas.

Plataforma Huggingface

La plataforma de código abierto *Huggingface* actúa como un repositorio de modelos de aprendizaje automático y ciencia de datos, donde los desarrolladores pueden compartir sus propias implementaciones. A su vez, Huggingface provee una de las interfaces de programación de aplicaciones (API) más utilizadas en el desarrollo de modelos de aprendizaje profundo orientados al procesamiento de lenguaje natural, que permite descargar aquellos modelos pre-entrenados compartidos por otros desarrolladores y utilizarlos mediante métodos de definición y entrenamiento de redes neuronales.

En el caso del presente trabajo se utilizó la plataforma de Huggingface para obtener, en un primer lugar, el modelo pre-entrenado T5S descrito en 5.2.1 y el tokenizador asociado, y por otro, la instanciación de un objeto de la clase Trainer que facilitó la definición de los hiperparámetros de entrenamiento para cada modelo planteado. Particularmente, la abstracción de la definición de la arquitectura de las redes neuronales, junto con la encapsulación de la lógica de entrenamiento de los modelos, resultaron ser los beneficios primordiales que aportó esta herramienta en el proceso de desarrollo.

Jupyter Notebooks

También denominados simplemente notebooks, los *Jupyter Notebooks* constituyen un espacio interactivo de desarrollo de código Python donde se combina la programación del código fuente y la definición de texto de tipo Markdown que permite describir las implementaciones. En el contexto de este trabajo, estos archivos[6] con extensión .ipynb permitieron la definición interactiva de algoritmos para: la preparación de los datasets necesarios para cada uno de los modelos y la propia implementación y entrenamiento de estos.

Primeramente se definió un notebook denominado `dataset-prep.ipynb` para implementar el preprocesamiento del conjunto completo de ejemplos de entrenamiento. Este proceso comenzó con el particionamiento del conjunto original de ejemplos y el almacenamiento en archivos CSV de ejemplos de entrenamiento (train), validación (validation) y prueba (test) para los cuatro subconjuntos de datos destinados al desarrollo de cada uno de los modelos considerados. A continuación se listan estos cuatro *datasets* (conjuntos de datos) con los respectivos archivos que los componen:

- **Dataset 1 - PseudoLaTeX**: Éste subconjunto de datos contenía solamente pares de entrada-salida para la generación de expresiones algebraicas en formato pseudoLaTeX, como se mostraron en los ejemplos de la sección 5.1. Los archivos que incluían a este dataset fueron destinados al entrenamiento del modelo inicial:
  - `train_split.csv`: archivo con 1.143 casos para el entrenamiento del modelo
  - `validation_split.csv`: archivo con 142 casos para la validación del modelo
  - `train_split.csv`: archivo con 143 casos para la prueba del modelo
- **Dataset 2 - PseudoLaTeX + NER**: Éste subconjunto de datos fue destinado al entrenamiento del primer modelo intermedio planteado en 5.1.2, el cual contenía pares de entrada-salida tanto para la generación de pseudoLaTeX como para el reconocimiento de entidades nombradas. Los archivos de validación y testeo simplemente contaban con ejemplos de generación de pseudoLaTeX:
  - `train_split.csv`: archivo con 2.571 casos para el entrenamiento del modelo, donde 1.143 eran réplicas de aquellos contenidos en Dataset 1.
  - `validation_split.csv`: archivo con 142 casos para la validación del modelo
  - `train_split.csv`: archivo con 143 casos para la prueba del modelo
- **Dataset 3 - PseudoLaTeX + Árboles**: Éste subconjunto de datos fue orientado al entrenamiento del segundo modelo intermedio visto en 5.1.2, el cual incluía pares de entrada-salida tanto para la generación de pseudoLaTeX como de árboles de operadores. Los archivos de validación y testeo simplemente contaban con ejemplos de generación de pseudoLaTeX:

[6] Link a repositorio GitHub: LINK

- `train_split.csv`: archivo con 2.571 casos para el entrenamiento del modelo, donde 1.143 eran réplicas de aquellos contenidos en Dataset 1.
  - `validation_split.csv`: archivo con 142 casos para la validación del modelo
  - `train_split.csv`: archivo con 143 casos para la prueba del modelo
- **Dataset 4 - PseudoLaTeX + NER + Árboles**: El modelo final fue entrenado sobre este conjunto de datos, el cual correspondía al conjunto original obtenido como resultado del preprocesamiento visto en 4.3. Éste incluía un archivo de entrenamiento con ejemplos para todas las tareas (generación de pseudoLaTeX y árboles, y NER) y dos archivos con casos de validación y testeo idénticos al Dataset 1.
  - `train_split.csv`: archivo con 3.714 casos para el entrenamiento del modelo
  - `validation_split.csv`: archivo con 142 casos para la validación del modelo
  - `train_split.csv`: archivo con 143 casos para la prueba del modelo

Cabe notar que todos los archivos con ejemplos de validación y prueba eran réplicas, donde solo se contaban con casos de la tarea de generación de pseudoLaTeX que, como se ha mencionado, representaba la tarea principal del modelo final. La razón por la que se optó por mantener el mismo par de archivos en cada dataset contemplado fue para lograr una correcta comparación de los valores de error al validar y probar los modelos neuronales. Dado que el objetivo del análisis era identificar si la integración de cada tarea auxiliar aportaba mejoras en el nivel de error cometido en la generación de pseudoLaTeX, resultaba coherente que la comparación de los resultados obtenidos para cada modelo contemplaran solamente aquellos casos comunes a todos los modelos, es decir, de la tarea principal, dejando por fuera la validación y el testeo sobre las tareas auxiliares.

Posteriormente, se realizó la tokenización de cada secuencia de palabras o símbolos de los ejemplos, incluyendo las descripciones en lenguaje natural, las fórmulas en pseudolatex, las anotaciones con entidades nombradas y los árboles de operadores. Como se mencionó anteriormente, la configuración del tokenizador utilizado, junto con su vocabulario predefinido, fueron descargados del repositorio de Huggingface utilizando la API que ésta plataforma expone para el lenguaje Python.

Por otra parte, se definió un cuaderno o notebook apodado `model-building.ipynb` que incluía celdas interactivas para la construcción de cada uno de los cuatro modelos entrenados. La definición de los hiperparámetros de entrenamiento se realizó por medio de la instanciación de la clase `Trainer` provista por Huggingface, la cual, a su vez, define un método `train()` que permitió la ejecución de las iteraciones de entrenamiento y la generación de logs de las distintas clases de error cometidos (durante evaluación y entrenamiento).

Finalmente, la herramienta de Jupyter Notebooks se utilizó para la confección de un prototipo de sistema interactivo donde un usuario era capaz de introducir un texto descriptivo de una expresión algebraica y como resultado obtener un ejemplo en forma de código LaTeX. Ésto se logró al utilizar una función de generación que enviaba al modelo final la descripción indicada para ejecutar el proceso de inferencia y generar una secuencia de pseudoLaTeX. Luego se reemplazaban los operandos variables de

éste formato con valores numéricos que respetaban las características de cada clase de operando para obtener el código LaTeX final.

5.3 Entrenamiento

Como fue mencionado en la sección 5.2, los cuatro modelos entrenados tomaron como base al modelo pre-entrenado T5S, el cual poseía tanto los pesos internos a los bloques de encoders y decoder como de los embeddings del vocabulario en español con valores previamente definidos. Partiendo de dichos parámetros entrenados, se utilizó la técnica de *fine-tuning* para lograr el afinamiento de estos pesos para las tareas involucradas en cada caso. De este modo se logró obtener resultados satisfactorios con conjuntos de datos de volúmenes relativamente pequeños y pocas épocas de entrenamiento.

La presente sección busca detallar, en primer lugar, la configuración base seleccionada para los respectivos bucles de entrenamiento y, en un segundo lugar, la arquitectura de hardware sobre la cual dichos procesos de entrenamiento fueron ejecutados.

5.3.1 Hiperparámetros

De forma de lograr una correcta comparación entre los distintos modelos construidos, se decidió mantener el mismo conjunto de hiperparámetros de entrenamiento para todos ellos. A continuación se lista la configuración utilizada y una explicación del propósito de cada parámetro:

- **Épocas**: Cada época de entrenamiento consiste en la ejecución del proceso de propagación hacia adelante (forward pass) a través de la red y la actualización de los pesos internos de ésta para todos los ejemplos de entrenamiento del conjunto de datos. En el caso de los cuatro modelos implementados, al servirse de conjuntos pequeños de datos (menores a cinco mil ejemplos) se optó por utilizar una cantidad de 5 épocas, con el fin de lograr un entrenamiento relativamente efectivo. A su vez, seleccionar un número bajo de épocas permitió mantener el tiempo total de ejecución en rangos razonables para el hardware utilizado (se presentará en el apartado 5.3.2). Finalmente, como se verá en el capítulo 6, al seleccionar dicha cantidad de épocas fue posible apreciar la disminución gradual de los errores de validación, sin obtener un excesivo sobreajuste a los datos presentados.

- **Tasa de aprendizaje**: Este factor de actualización en los gradientes calculados durante el algoritmo de backpropagation (o una optimización de éste) fue definido con un valor inicial de 0.0003. Éste fue seleccionado debido a la recomendación de los desarrolladores del modelo T5 dentro de la documentación de la plataforma Huggingface y fue actualizado a medida que se avanzaba en las iteraciones de entrenamiento.

- **Tamaños de lote**: Debido al pequeño tamaño de memoria RAM y caché y bajo nivel de paralelización de hilos, se seleccionó una cantidad de 4 ejemplos por lote independientemente

del modelo entrenado. Las pruebas realizadas con lotes de 8 ejemplos probaron obtener tiempos de ejecución similares o incluso mayores que aquellos de 4, lo cual podría atribuirse a la saturación de memoria RAM y "Cache misses".

- **Frecuencia de logs**: Con el fin de obtener un número considerable de mediciones intermedias para los valores de error de validación y entrenamiento se definió una frecuencia de 1 log de resultados por cada cien pasos de entrenamiento.

5.3.2 Hardware utilizado y tiempos de ejecución

A lo largo del proceso de entrenamiento de los modelos se utilizaron dos computadoras con arquitecturas de hardware diferentes. El primer intento de ejecución se llevó a cabo utilizando una PC de escritorio con las siguientes especificaciones de hardware:

Procesador	Intel Core i7 - 4ta generación
Memoria RAM	8GB
GPU	Nvidia Geforce GTX 1050
Memoria VRAM	2GB
Memoria secundaria	256GB HDD

Las unidades de procesamiento gráfico (GPUs) permiten alcanzar un grado de paralelización de hilos de ejecución muy superior en relación a las arquitecturas que solo cuentan con núcleos de CPU. A pesar de contar con uno de estos componentes, el entrenamiento realizado sobre esta máquina sufrió un gran número de limitaciones, principalmente causadas por el tamaño de las memorias RAM. La insuficiencia de memoria resultó en ejecuciones fallidas o extremadamente lentas para tamaños pequeños de conjuntos de datos (< 1500) y pocas épocas de entrenamiento (3).

Debido a ello se optó por realizar el entrenamiento de los modelos sobre una computadora portátil con las siguientes características:

Procesador	Intel Core i5 - 10ma generación
Memoria RAM	16GB
GPU	N/A
Memoria VRAM	N/A
Memoria secundaria	1TB SSD

El incremento de la memoria RAM permitió resolver las fallas de ejecuciones por insuficiencia de memoria y, si bien esta laptop no contaba con una unidad de procesamiento gráfico, los 8 núcleos lógicos

de la CPU, sumado al uso de un dispositivo de almacenamiento de estado sólido, compensaron en parte la reducción de velocidad de entrenamiento.

Luego, utilizando para todos los modelos los hiperparámetros especificados en 5.3.1, se realizaron las respectivas ejecuciones de los bucles de entrenamiento, las cuales resultaron con los siguientes tiempos totales de ejecución:

- **Modelo inicial PseudoLaTeX**: 3 horas
- **Modelo intermedio PseudoLaTeX + NER**: 6 horas 50 minutos
- **Modelo intermedio PseudoLaTeX + Árboles**: 6 horas 40 minutos
- **Modelo final PseudoLaTeX + NER + Árboles**: 9 horas 15 minutos

En el siguiente capítulo se profundizará en los resultados obtenidos sobre los errores de prueba, validación y entrenamiento para cada uno de estos modelos.

CAPÍTULO 6

Resultados obtenidos

Una vez llevadas a cabo las sucesivas etapas de entrenamiento para los modelos establecidos y descritos en el capítulo anterior, se obtuvieron cuatro conjuntos de resultados que permitieron comparar sus respectivos rendimientos y observar el nivel de aporte de cada elemento estructural añadido (entidades y árboles) al comportamiento del modelo final.

La primer sección del presente capítulo abordará la descripción de las expectativas y análisis de los resultados obtenidos desde dos perspectivas: una comparación "intra modelo" donde se muestra la evolución individual de cada modelo al ser entrenado sobre su respectivo conjunto de datos, y una comparación "inter modelo" que marca las diferencias de rendimiento y mejoras en cada fase de construcción del modelo final, permitiendo formular una respuesta a la hipótesis presentada en el capítulo 5.

Finalmente, la última sección 6.2 detallará la aplicación y resultados de una métrica de rendimiento que fue construida para el contexto particular del modelo del presente trabajo y calculada de forma manual sobre un pequeño conjunto de ejemplos de testeo.

6.1 Comparación de modelos

Los hiperparámetros de entrenamiento establecidos para cada uno de los modelos, como se introdujo en el capítulo previo, se mantuvieron sin modificaciones para todos ellos, con el objetivo de lograr equidad en el análisis comparativo de esta sección. En particular, la frecuencia de impresión de logs fue seleccionada de modo que se obtuviera una cantidad significativa de resultados intermedios para construir los gráficos que se mostrarán en los siguientes apartados. Cabe recordar que, en lugar de obtener un único valor de error de entrenamiento y de validación al final del proceso, o para cada fin de época, se optó por ejecutar validaciones e imprimir los resultados de error cada 100 pasos del entrenamiento.

La función de error utilizada durante el fine-tuning de T5S para la obtención de cada modelo fue la misma función de entropía cruzada utilizada por este primero, la cual permitió realizar la actualización de los pesos de la red a partir del cálculo de sus gradientes y obtener los resultados que se presentarán a continuación.

6.1.1 Errores de entrenamiento VS errores de validación

Comenzando con un primer análisis comparativo “intra modelo” de los resultados de error de entrenamiento y validación, se definió un conjunto de expectativas correspondientes a cada modelo. Dado que se tomó un camino de evolución hacia el modelo final, apodado “PseudoLaTeX + NER + Trees” (modelo 4) en el contexto de los gráficos del presente capítulo, se esperó obtener un reflejo del comportamiento de mejora gradual en los valores de error para cada modelo entrenado en las distintas fases. El patrón base esperado para todos ellos consistió en obtener una disminución del error final de entrenamiento a medida que se añadían los componentes estructurales al entrenamiento, como las representaciones de entidades nombradas en el modelo “PseudoLaTeX + NER” (modelo 2) y los árboles de operadores en el tercer modelo “PseudoLaTeX + Trees” (modelo 3).

Cabe recordar que los cuatro modelos implementados fueron entrenados sobre los mismos casos de la tarea principal de generación de pseudoLaTeX. A su vez el mismo conjunto de datos de validación fue utilizado por todos los modelos para el cálculo de los errores de validación, en el cual se incluían 143 casos de generación de pseudoLaTeX, dejando fuera del cálculo a ejemplos de las tareas auxiliares. Este aspecto permitió definir una expectativa para los modelos 2, 3 y 4, para los cuales se esperaba una convergencia más temprana del error de entrenamiento debido a un mayor volúmen de datos, y, sobre todo, que la curva de error de validación se mantuviera con una pendiente similar a la de entrenamiento, lo que indicaría que la inclusión de las tareas auxiliares mejoraron la disminución del error al generar pseudoLaTeX.

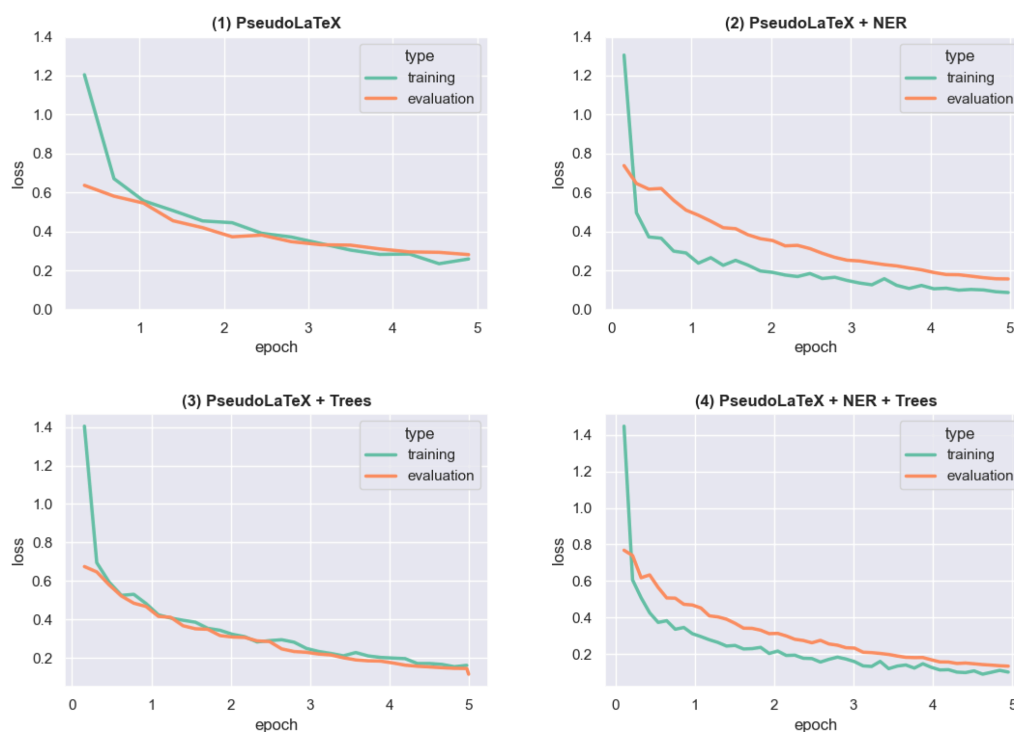

Figura 6.1: Gráficos de error entrenamiento-vs-error validación para los cuatro modelos implementados.

A partir del gráfico de la figura 6.1, se pudo identificar que el rendimiento del modelo “PseudoLaTeX”, o simplemente modelo 1, entrenado únicamente para resolver la tarea de generación de expresiones algebraicas en pseudoLaTeX a partir de una descripción en español, logró una disminución en el error de entrenamiento a lo largo de las 5 épocas, pero ésta mejora ocurrió de forma mucho más gradual que en el resto de modelos. Mientras que el modelo 1 logró valores de error cercanos y superiores a 0.4 para el final de la segunda época, el resto de modelos la superaron con errores menores, con valores cercanos a 0.2 para los modelos 2 y 4, y 0.3 para el tercer modelo.

Por otro lado, desde una perspectiva interna a cada caso, fue posible identificar la disminución en todas las curvas de errores de validación, las cuales comenzaron con valores significativamente menores a los errores de entrenamiento. Ésto podría atribuirse al hecho que el cómputo del error de validación se realizaba una vez superados los 100 pasos de entrenamiento, mientras que el error de entrenamiento era el resultado del promedio de aquellos valores obtenidos para esos 100 pasos.

Particularmente, se puede poner en relieve que los dos modelos que incluían NER entre sus tareas (modelos 2 y 4), lograron una mayor convergencia del error de entrenamiento, lo cual puede explicarse por contener más datos de entrenamiento y casos de NER para los cuales el modelo T5S base se esperaba que performara bien. A pesar de ello, estos dos modelos mostraron curvas de error de validación un tanto más alejadas de las de entrenamiento, lo que indicaría que la incorporación de NER como tarea auxiliar no afectó de forma tan evidente a la generación de pseudoLaTeX, especialmente en caso del modelo 2, donde la diferencia de los errores finales fue mayor.

Finalmente, cabe destacar que, mientras la curva de error de entrenamiento del primer modelo disminuía, el error de validación siguió una tendencia relativamente constante en comparación al resto de casos, lo cual indicaría un posible sobreajuste de parámetros a la tarea de generación. En cambio, al aplicar aprendizaje multitarea sobre las demás redes, la amplitud del error de validación, es decir, la diferencia entre el valor máximo y mínimo alcanzados, fue mayor.

6.1.2 Resultados de validación y prueba

Luego, con el objetivo de observar una comparación entre los modelos y obtener una respuesta para la hipótesis planteada, se recolectaron los valores de error de validación cada 100 pasos de entrenamiento, los cuales fueron calculados sobre el mismo conjunto de 143 ejemplos de validación para la tarea final del modelo: la generación de expresiones en pseudoLaTeX. De esta forma fue posible realizar una correcta comparación e identificar las mejoras producidas sobre la tarea de generación al añadir de forma incremental las demás tareas relacionadas a la construcción de la estructura gramatical de las descripciones y la estructura jerárquica de las expresiones algebraicas, estas eran, el reconocimiento de entidades nombradas (NER) y la generación de árboles de operadores, respectivamente.

La siguiente figura 6.2 muestra un gráfico donde es posible comparar las curvas de errores de validación obtenidas en cada proceso de entrenamiento de los modelos.

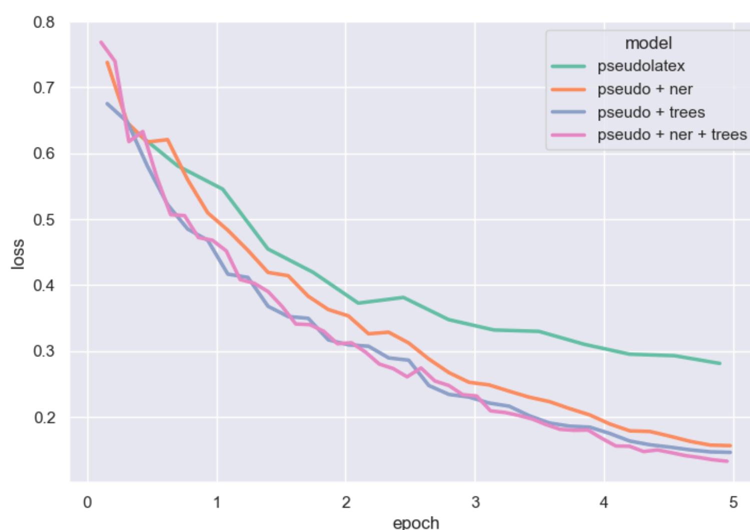

Figura 6.2: *Gráfico de comparación del error de validación obtenido para los cuatro modelos implementados.*

Al realizar un primer análisis de la figura, resultó evidente la reducción progresiva del error de validación cuando se comparan entre sí las curvas obtenidas para cada modelo. Una expectativa predefinida para estos resultados fue el hecho de obtener, al inicio del proceso de entrenamiento, errores de validación superiores para aquellos modelos que realizarán aprendizaje sobre un mayor número de tareas, lo que probó ser cierto, ya que el modelo final "pseudolatex + ner + trees" obtuvo el mayor error, similar a las otras dos redes entrenadas con una tarea adicional (NER o Generación de árboles). Esto se debió a que se contó con el modelo base T5S, el cual no fue pre entrenado sobre tareas como la generación de pseudolatex, ni de árboles de operadores y tampoco se entrenó para realizar el reconocimiento de las entidades nombradas particulares definidas dentro del contexto de este trabajo (vistas en 4.1.2). Es por esta razón que se esperaba que los modelos obtuvieran peores rendimientos iniciales cuando mayor fuera el número de tareas desconocidas presentadas en el comienzo del entrenamiento.

En contraposición, resulta directo identificar el aumento en la magnitud de la razón de cambio del error de validación con respecto al número de épocas, es decir, se logró una mayor reducción del error en un menor número de pasos de entrenamiento a medida que se añadían nuevas tareas en el aprendizaje, superando ampliamente el rendimiento del primer modelo entrenado sobre una única tarea de generación.

Finalmente, al observar los resultados de la última época es claro distinguir que se obtuvieron menores valores de error para aquellos modelos que incluían más elementos estructurales de las descripciones y expresiones algebráicas al proceso de entrenamiento. En definitiva, el gráfico 6.2 permitió verificar la correctitud de la hipótesis vista en el capítulo 5, donde el modelo final entrenado sobre un conjunto de datos orientado al aprendizaje multitarea (pseudo + ner + trees) logró el mejor rendimiento en comparación a todas las evoluciones previas.

En lo que respecta a los valores de error obtenidos para los conjuntos de datos de prueba, la siguiente tabla 6.1 muestra un resúmen de los resultados para cada modelo entrenado, sumados al T5S utilizado como punto de partida. Éste último fue considerado en este análisis con el fin de determinar la eficacia de la red neuronal previo al proceso de fine-tuning de sus parámetros para la construcción de cada modelo intermedio y final.

Modelo	Error de prueba medio
T5S Base	6.99299383
PseudoLaTeX	0.26089024
PseudoLaTeX + NER	0.13965684
PseudoLaTeX + Árboles	0.12346486
PseudoLaTeX + NER + Árboles	0.11347503

Tabla 6.1: *Errores de prueba finales para cada modelo considerado, incluyendo T5S Base*

La rápida disminución del error de prueba detallado en los resultados de la tabla demuestran claramente la amplia mejoría de las secuencias generadas por parte de cada modelo afinado con respecto al modelo base T5S. Adicionalmente, considerando el volúmen de los conjuntos de datos mencionados en 5.2.2, resulta válido afirmar que la aplicación de la técnica de fine-tuning al entrenamiento del modelo final obtuvo grandes mejoras con solo una pequeña cantidad de casos de ejemplo.

6.2 Observaciones de rendimiento

La sección previa demostró a través de resultados numéricos la mejora en el rendimiento de cada modelo entrenado, aparejada con la incorporación de las respectivas tareas de generación y reconocimiento de entidades nombradas. En relación a esto se debe destacar que, al momento de evaluar los modelos sobre los conjuntos de datos de prueba, se planteó el uso de métricas automáticas como BLUE o ROUGE introducidas en 3.7. Luego de un análisis del funcionamiento de cada una de ellas se llegó a la conclusión de que, en caso de ser utilizadas, no reflejarían con suficiente precisión el desempeño de los modelos en aspectos como la correctitud de la sintaxis o la semántica de las expresiones generadas. Dos de las causas por las que no fueron utilizadas en este caso son:

- La necesidad de contar con múltiples ejemplos de referencia para cada caso de generación. Si bien la mayoría de las descripciones podrían referenciar a múltiples formas de expresiones posibles, incluso infinitas, algunos textos descriptivos podrían determinar casos extremadamente particulares, reduciendo las posibilidades de generación de expresiones a solo una correcta.

- Dado que se busca analizar la correlación entre la semántica del texto descriptivo y la sintaxis del código pseudoLaTeX generado, ésto no se vería reflejado en los resultados numéricos de estas métricas ya que utilizarían medidas de "superposición" de los símbolos del código generado contra aquellos presentes en los casos de referencia.

Luego, con el fin de obtener una visión más clara del verdadero comportamiento de los modelos sobre descripciones de entrada en lenguaje natural, se optó por dejar de lado las abstracciones numéricas de las métricas automáticas. De esta forma, se propuso la evaluación *manual* de un puntaje de correctitud sobre las salidas generadas, donde cada una de ellas fue puntuada según una escala de seis códigos de error definidos para el contexto particular de esta tesina. Éstos códigos fueron especificados siguiendo un orden creciente de correctitud, para la cual se contemplaba tanto la precisión sintáctica como semántica de las expresiones de pseudoLaTeX generadas a partir del procesamiento de su descripción. A continuación se presenta la escala de códigos utilizados:

- Código *0* : La expresión no es sintácticamente correcta ni coherente.
- Código *1a*: La expresión es sintácticamente correcta pero no es coherente.
- Código *1b*: La expresión no es sintácticamente correcta pero sí contiene un grado de coherencia.
- Código *2a*: La expresión es sintácticamente correcta y coherente, pero no respeta la totalidad de características descriptas.
- Código *2b*: La expresión es sintácticamente correcta y coherente, pero está levemente incompleta.
- Código *3*: La expresión es sintácticamente correcta, coherente y completa.

Tomando esta escala de códigos de error se evaluó manualmente una misma muestra aleatoria de 30 descripciones sobre los resultados generados por cada modelo implementado. El gráfico de barras de la figura 6.3 detalla la frecuencia absoluta de los distintos códigos de error en el contexto de cada modelo.

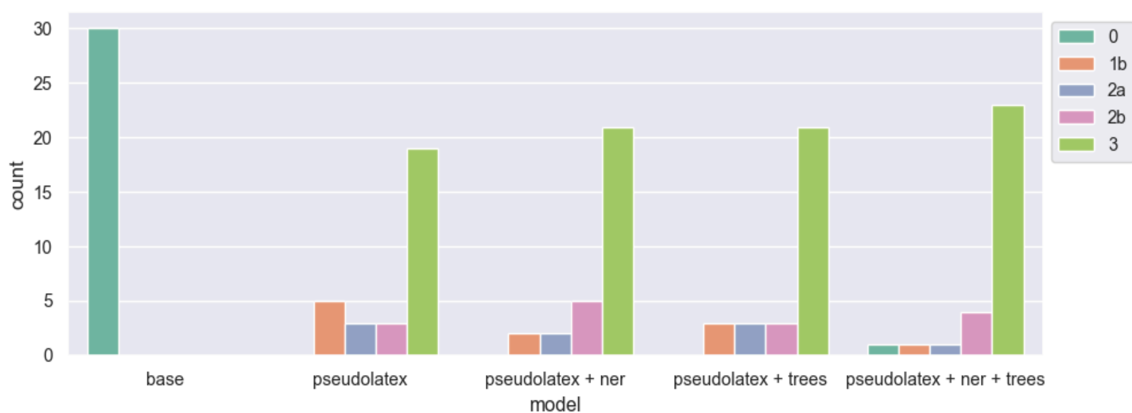

Figura 6.3: *Gráfico de barras con los distintos códigos de error para cada modelo implementado. Incluye el modelo T5 Base.*

Resulta evidente al observar la figura 6.3 que la totalidad de secuencias generadas por el modelo T5S (base) fueron puntuadas con el código 0. Ésto se alineó correctamente con los errores de prueba vistos en la sección anterior, donde la red neuronal base T5S sin fine-tuning presentó un gran nivel de error sobre el conjunto de datos de prueba y una amplia diferencia con el resto de modelos.

Por otra parte, el gráfico reiteró la validez de la hipótesis planteada, asegurando que el aprendizaje multitarea y las representaciones estructurales adicionales embebidas en el proceso de entrenamiento, incrementaron el rendimiento del modelo final.

Por último, resulta importante poner en relieve el gran número de casos con código 3 (casos correctos) desde la etapa más temprana de la evolución del modelo, donde solo se realizó el entrenamiento sobre la tarea de generación de pseudoLaTeX. Este rápido aprendizaje basado en fine-tuning se pone en contraposición con los resultados erróneos obtenidos para el modelo base T5S.

A modo de finalización del análisis de los resultados obtenidos sobre el desarrollo de este trabajo de grado, las tablas 6.2 y 6.3 muestran algunos de los ejemplos considerados en esta evaluación manual junto con las secuencias generadas por cada modelo y sus respectivos puntajes. Por un lado, los casos descritos en la Tabla 6.2 muestran el comportamiento de cada uno de los modelos implementados sobre la entrada de tres descripciones diferentes de expresiones algebraicas. La Tabla 6.3, por otro lado, detalla los resultados obtenidos por el sistema ChatGPT (con el uso de GPT-4) sobre las mismas tres descripciones, a modo de ejemplificar el comportamiento de esta importante herramienta al realizar la misma tarea de generación. Cabe destacar que este modelo fue probado de dos formas: una primera generación sin la especificación previa del formato pseudoLaTeX y otra posterior donde se detalló el formato particular esperado mediante un *prompt.* La Figura 6.4 muestra el prompt enviado a modo de configuración del formato de salida de ChatGPT.

De los resultados detallados en la Tabla 6.3 resulta evidente la gran capacidad de generación, versatilidad e interpretación del sistema ChatGPT, en particular, del modelo neuronal GPT-4. Resulta sorprendente la exactitud en los resultados generados en pseudoLaTeX al solo contar con un prompt que especifica el formato esperado y sin haber definido casos de ejemplo.

Ahora debes utilizar el siguiente formato denominado pseudoLaTeX como salida de generación, donde los valores numéricos de los operandos deben ser reemplazados correspondientemente por los siguientes operandos variables como placeholders:
"VN": para valores naturales,
"VR": para valores reales,
"VC": para valores complejos o imaginarios,
"VF": para valores fraccionarios,
"VD": para valores decimales
Tambien debe mantenerse el código resultante sin utilizar el modo "math" de LaTeX, es decir, presentar unicamente los resultados como strings y no aportar otra clase de comentarios extra.

A continuación te enviare uno a uno los casos a generar

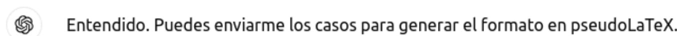

Figura 6.4: prompt enviado a ChatGPT a modo de configuración del formato de salida.

Tabla 6.2: Ejemplos de resultados generados por los distintos modelos implementados para ciertas descripciones. Se incluyen los puntajes con los códigos de la métrica manual utilizada.

Descripción		
"Polinomio de grado 7 completo con coeficientes enteros"		
Modelo	**Resultado**	**Puntaje**
Base T5S	s: de.o con.. con:. de.::. de grado 7 completo:.o:latex: completo. Polinomio de grado 7 completo...	0
PseudoLaTeX	`VN + VNx + VNx^{2} + VNx^{3} + VNx^{4} + VNx^{5} + VNx^{6}`	2b
PseudoLaTeX + NER	`- VNx^{7} - VNx^{6} + VNx^{5} + VNx^{4} + VNx^{3} + VNx^{2} + VNx - VN`	3
PseudoLaTeX + Árboles	`- VNx^{7} - VNx^{6} + VNx^{5} + VNx^{4} + VNx^{3} + VNx^{2} + VNx - VN`	3
Completo	`- VNx^{7} - VNx^{6} + VNx^{5} + VNx^{4} + VNx^{3} + VNx^{2} + VNx - VN`	3
Descripción		
"Sistema de tres ecuaciones. Una de ellas de tres variables y las dos restantes de dos variables o incógnitas."		
Modelo	**Resultado**	**Puntaje**
Base T5S	. Sistema de tres ecuaciones. Sistema de tres ecuaciones...s.. o incógnitass......s. Sistema. Sist ...	0
PseudoLaTeX	`\begin{cases} VN.z + VF.x = VF \\VN.y + VF.z - VF.x = VN \end{cases}`	1b
PseudoLaTeX + NER	`\begin{cases} - VF.y + VN.x = VF \\VN.y + VF.x = VF \end{cases}`	1b
PseudoLaTeX + Árboles	`\begin{cases} VF.x + VN.z = VN \\VN.y + VF.z = VF \end{cases}`	1b
Completo	`begin{cases} - VF.y + VN.z = VF \\- VF.x + VN.z = VN \\- VN.y - VF.z = VN \end{cases}`	2b
Descripción		
"Polinomio con coeficientes complejos y naturales, de noveno grado e incompleto. Tiene 5 términos"		
Modelo	**Resultado**	**Puntaje**
Base T5S	. tiene 5::..::...: en::. tiene 5. tiene 5,::, de:latex: :::: Polinomio con coeficientes coeficientes:: tiene ...	0
PseudoLaTeX	`- VC.z^{4} - VNz^{8} + VN`	1b
PseudoLaTeX + NER	`- VC.z^{4} - VNz^{8} + VC.z^{6} - VNz^{9} + VN`	3
PseudoLaTeX + Árboles	`- VC.z^{4} - VNz^{8} + VN`	1b
Completo	`- VC.z^{4} - VNz^{8} + VC.z^{6} - VNz^{9} + VN`	3

Tabla 6.3: Ejemplos de resultados generados por ChatGPT para ciertas descripciones de expresiones algebraicas. Se incluyen los puntajes con los códigos de la métrica manual utilizada.

Descripción (prompt)
"Generar Polinomio de grado 7 completo con coeficientes enteros"

<table>
<tr><th>Prompting</th><th>Resultado</th><th>Puntaje</th></tr>
<tr><td>Sin definición de pseudoLaTeX</td><td>Un polinomio de grado 7 completo con coeficientes enteros incluye todas las potencias de la variable hasta el grado 7. Un ejemplo de este tipo de polinomio sería:

\[
P(x) = 3x^7 - 5x^6 + 2x^5 - 4x^4 + x^3 - 6x^2 + 7x - 1
\]

Este polinomio tiene coeficientes enteros, y cada término está presente desde \(x^7\) hasta \(x^0\) (el término constante).</td><td>1b</td></tr>
<tr><td>Con definición de pseudoLaTeX</td><td>P(x) = VN x^7 + VN x^6 + VN x^5 + VN x^4 + VN x^3 + VN x^2 + VN x + VN</td><td>3</td></tr>
<tr><th colspan="3">Descripción (prompt)</th></tr>
<tr><td colspan="3">"Generar Sistema de tres ecuaciones. Una de ellas de tres variables y las dos restantes de dos variables o incógnitas."</td></tr>
<tr><th>Prompting</th><th>Resultado</th><th>Puntaje</th></tr>
<tr><td>Sin definición de pseudoLaTeX</td><td>Aquí tienes un sistema de tres ecuaciones en el que una es de tres variables y las dos restantes son de dos variables:

\[
\begin{aligned}
1) \quad 2x + 3y - z &= 7 \\
2) \quad 4x - y &= 5 \\
3) \quad 3y + 2z &= 8
\end{aligned}
\]

En este sistema:
- La primera ecuación involucra tres variables: \(x\), \(y\), y \(z\).
- Las dos siguientes ecuaciones solo involucran dos variables: \(x\) e \(y\) en la segunda, y \(y\) y \(z\) en la tercera.</td><td>1b</td></tr>
<tr><td>Con definición de pseudoLaTeX</td><td>1) VN x + VN y + VN z = VN

2) VN x + VN y = VN

3) VN y + VN z = VN</td><td>2b</td></tr>
<tr><th colspan="3">Descripción (prompt)</th></tr>
<tr><td colspan="3">"Generar Polinomio con coeficientes complejos y naturales, de noveno grado e incompleto. Tiene 5 términos"</td></tr>
<tr><th>Prompting</th><th>Resultado</th><th>Puntaje</th></tr>
<tr><td>Sin definición de pseudoLaTeX</td><td>Aquí tienes un ejemplo de un polinomio de noveno grado, con coeficientes complejos y naturales, que es incompleto (es decir, no tiene todos los términos de grado 9 a 0) y con 5 términos:

\[
P(x) = (2 + 3i)x^9 - 5x^7 + (4 - i)x^4 + 7x^2 - 6
\]

En este polinomio:
- Los coeficientes son números complejos (como \(2 + 3i\) y \(4 - i\)) y números naturales (como \(7\) y \(5\)).
- Es de noveno grado, y está incompleto porque faltan algunos términos, como \(x^8\), \(x^6\), \(x^5\), \(x^3\), y \(x\).</td><td>1b</td></tr>
<tr><td>Con definición de pseudoLaTeX</td><td>P(x) = VC x^9 + VN x^7 + VC x^4 + VN x^2 + VN</td><td>3</td></tr>
</table>

CAPÍTULO 7

Conclusiones y Trabajos futuros

Este último capítulo expondrá, en primer lugar, las conclusiones obtenidas de las actividades de desarrollo e investigación llevadas a cabo a lo largo de la presente tesina. Junto a ellas se describirá un pequeño resumen a modo de recapitulación de las etapas de trabajo y análisis contempladas a través de todo el período de esta tesina.

Por otro lado, se introducirá un listado de las posibles líneas futuras de trabajo que podrían extender la investigación e implementaciones aplicadas en el contexto de este trabajo de grado.

7.1 Conclusiones generales

La generación de expresiones algebraicas basadas en descripciones en lenguaje español condujo el enfoque del presente trabajo hacia el área particular del aprendizaje profundo, o deep learning, enmarcado en el contexto de la inteligencia artificial y el aprendizaje automático. La familiarización con el estado del arte sobre los conceptos y técnicas utilizadas en procesamiento de lenguaje natural (NER) resultaron ser primordiales para el desarrollo de este trabajo y fueron detallados a lo largo de toda su extensión.

A pesar de contar con un amplio número de modelos de lenguaje pre-entrenados basados en arquitecturas de redes neuronales, como los más difundidos GPT y BERT, éstos son en su mayoría modelos de propósito general. En cambio, este trabajo buscó indagar la posibilidad de aplicar los conceptos del procesamiento de lenguaje natural en el campo particular de la matemática. A raíz de la falta de mayor investigación de las aplicaciones de las redes neuronales en este área de la ciencia, se planteó el desarrollo de un nuevo modelo experto de generación de expresiones algebraicas (polinomios y sistemas de ecuaciones) en un formato cercano al LaTeX, apodado pseudoLaTeX, de forma de investigar sobre posibles mejoras para el proceso de entrenamiento de modelos de esta índole y construir una potencial herramienta para uso educativo.

Dado el alcance y objetivo de este trabajo, la importancia de desarrollar los conceptos de modelos para el procesamiento de lenguaje natural condujo a la profundización de la arquitectura de redes neuronales Transformer, que hoy en día representa el estado del arte en el contexto del procesamiento y generación de secuencias. A su vez, se describió la estructura y funcionamiento del novedoso mecanismo de self-attention que, gracias a su capacidad de paralelización, permitió a los Transformers superar

ampliamente el rendimiento de otras arquitecturas orientadas al procesamiento secuencia a secuencia, como las redes neuronales recurrentes.

Una vez introducidas las ideas y conceptos centrales necesarios para el desarrollo del modelo final, se detallaron los dos patrones estructurales hallados en sus datos de entrada y salida. Tanto las entidades embebidas en las descripciones en español de entrada como la estructura jerárquica de las expresiones algebraicas de salida fueron componentes disparadores para la definición de una hipótesis que acompañó la implementación del modelo. Este interrogante consistió en identificar el grado de mejoría aparejado con la utilización de un enfoque basado en aprendizaje multitarea para el entrenamiento simultáneo sobre la generación de pseudolatex, el reconocimiento de entidades nombradas y la construcción de árboles de operadores.

Posteriormente, con intenciones de hallar evidencias que verificaran la correctitud de la hipótesis, surgieron dos grandes limitaciones que obstaculizaron la implementación del modelo experto. Éstas fueron la construcción de una base de datos de un volúmen significativo para el entrenamiento de la red neuronal y, por otra parte, la representación de dichos datos para la correcta descripción de las tareas y el formato de sus entradas y salidas. Dado que no se hallaron bases de datos públicas que contuvieran los datos requeridos al momento del desarrollo de esta tesina, se confeccionó una solución basada en la implementación de una página web. Este subproyecto, apodado LaTeX Expression Descriptor Project o simplemente LEDProject, consistió en la construcción de una herramienta sencilla para el uso sobre dispositivos móviles orientada a un subconjunto de alumnos de la Facultad de Ingeniería de la Universidad Nacional de La Plata, donde se les presentaban grupos de expresiones algebraicas y un campo de texto donde podían describirlas.

Con el fin de generar las expresiones algebraicas a presentar se implementaron algoritmos de generación de árboles de operadores y de parseo de éstos para la confección de expresiones en pseudoLaTeX. De esta forma, al desarrollar y utilizar LEDProject fue posible resolver de forma eficiente la problemática de la recolección manual de ejemplos con los formatos requeridos, evitando a su vez la introducción de posibles sesgos al conjunto de datos de entrenamiento.

Luego de transcurrido un período de tres meses de recolección de datos, se alcanzó una cantidad satisfactoria de ejemplos con los que fue posible comenzar una etapa de preprocesamiento que incluyó el curado de las descripciones recolectadas y la aplicación de técnicas de aumento de volumen de datos. A su vez se utilizó una herramienta web de etiquetado de entidades nombradas sobre las descripciones, en pos de obtener eficientemente una copia del conjunto de descripciones con anotaciones en formato BIO.

Una vez obtenido el conjunto completo de datos de entrenamiento, con cerca de 4000 ejemplos, se seccionó la construcción del modelo final en cuatro fases que permitieron analizar diferentes etapas evolutivas de éste. En cada una de ellas se entrenó con la técnica de fine-tuning al modelo de lenguaje T5S, sobre un subconjunto distinto de los datos originales, con el fin de obtener diferentes modelos en cada etapa. El primero de los conjuntos de datos solo contenía ejemplos para la generación de expresiones en pseudoLaTeX, el segundo incluía además casos para el reconocimiento de entidades nombradas y, en forma análoga, el tercero abarcaba ejemplos para la generación de pseudoLaTeX y de

árboles de operadores. En una instancia final, el modelo experto de la última fase fue entrenado sobre el conjunto completo original de datos, donde se incluían las tres tareas contempladas en este trabajo.

Al realizar el entrenamiento y validación sobre cada modelo utilizando una baja cantidad de épocas y conjuntos de datos con volúmenes relativamente pequeños, se ha podido confirmar la efectividad de la técnica de fine-tuning y el enfoque multitarea utilizado para mejorar la convergencia del error cometido durante el entrenamiento e inferencia sobre casos nunca presentados. En conclusión, podría atribuirse esta mejora al hecho que el entrenamiento sobre la extracción de entidades nombradas de las descripciones y la construcción de árboles de operadores lograron capturar elementos sintácticos y semánticos de las estructuras de los lenguajes natural y matemático, para luego utilizarlos como información adicional en la generación de las secuencias de pseudoLaTeX.

Finalmente, el modelo implementado en la última etapa, al demostrar el mejor desempeño de entre los cuatro modelos desarrollados, fue utilizado en un notebook basado en el lenguaje de programación Python para confeccionar un prototipo de sistema web. En éste se permite la introducción de un texto descriptivo de una expresión algebraica y como resultado de la ejecución de una función de generación se obtiene un ejemplo de la expresión en forma de código LaTeX, al reemplazar los operandos variables de la secuencia de pseudoLaTeX producida por el modelo.

7.2 Líneas futuras de trabajo

Como posibles extensiones a lo desarrollado en la presente tesina de grado, se define el siguiente listado de posibles líneas de trabajo futuro:

- Experimentar con la aplicación de la técnica de fine-tuning sobre otros modelos de lenguaje preentrenados, incluso con arquitecturas transformers basadas en bloques de solo encoders como BERT o solo decoders como GPT. Con ello podrían identificarse las arquitecturas que mejor se ajustan al caso particular de la aplicación del procesamiento de lenguaje natural en el ámbito de la matemática.

- Realizar pruebas con otras funciones de error durante el entrenamiento del modelo, entre las cuales sería posible definir nuevas funciones personalizadas que varíen el grado de impacto que produciría cada tarea aprendida sobre el ajuste de los pesos internos de la red neuronal.

- Construir un benchmark utilizando métricas automatizadas que permita comparar el rendimiento obtenido por modelos preentrenados sin aplicar fine-tuning y aquellos donde sí se hace uso de esta técnica de afinamiento de parámetros.

- Extender el alcance del modelo desarrollado en esta tesina a otras clases de expresiones algebraicas, como las inecuaciones, o incluso a elementos de otras áreas de la matemática. Algunos ejemplos a integrar podrían ser sucesiones y series numéricas, integrales, ecuaciones

diferenciales y matrices, entre muchos otros. Ésto podría realizarse al expandir el conjunto de datos de entrenamiento utilizando la herramienta LEDProject creada en este trabajo.

- Confeccionar un modelo experto que sea entrenado desde sus inicios solamente sobre datos del área de la matemática, sin hacer uso de la técnica de fine-tuning ni de modelos de lenguaje preentrenados. Ésto podría obtener resultados satisfactorios una vez se reúna un conjunto de datos de mayor volúmen.

- Extender el prototipo web de interacción con el modelo utilizado en esta tesina para la confección de una herramienta de creación de trabajos prácticos o exámenes, orientada al uso por parte de los docentes de todos los niveles educativos. También podría utilizarse el modelo obtenido para la construcción de aplicaciones para el aprendizaje interactivo con los alumnos.

Bibliografía

Araujo, V., Trusca, M. M., Tufiño, R., & Moens, M. F. (2024, Mar 21). Sequence-to-Sequence Spanish Pre-trained Language Models. *arXiv preprint*. https://arxiv.org/abs/2309.11259

Bahdanau, D., Cho, K., & Bengio, Y. (2014). Neural machine translation by jointly learning to align and translate. *arXiv (Cornell University)*. 10.48550/arxiv.1409.0473

Brown, T. B., Mann, B., & Ryder, N. (2020). Language Models are Few-Shot Learners. https://arxiv.org/abs/2005.14165

Charton, F., Hayat, A., & Lample, G. (2021). Learning advanced mathematical computations from examples. https://arxiv.org/abs/2006.06462

Cho, K., Van Merrienboer, B., Gulcehre, C., Bahdanau, D., Bougares, F., Schwenk, H., & Bengio, Y. (2014). Learning Phrase Representations using RNN Encoder-Decoder for Statistical Machine Translation. *arXiv (Cornell University)*. 10.48550/arxiv.1406.1078

Dal Bianco, P. A. (2021). Generación de texto estructurado en español para batallas de freestyle utilizando modelos de Deep Learning. http://sedici.unlp.edu.ar/handle/10915/118994

Datta, D. (2017). *LaTeX in 24 Hours: A Practical Guide for Scientific Writing*. Springer International Publishing.

Devlin, J., Chang, M., Lee, K., & Toutanova, K. (2018). BERT: Pre-training of Deep Bidirectional Transformers for Language Understanding. https://arxiv.org/abs/1810.04805

Eisenstein, J. (2019). *Introduction to Natural Language Processing*. MIT Press.

Fan, L., Li, L., Ma, Z., Lee, S., Yu, H., & Hemphill, L. (2023). A Bibliometric Review of Large Language Models Research from 2017 to 2023. https://arxiv.org/abs/2304.02020

Ferre, N., Galli, A. C., & Guzmán Mattje, E. B. (2018). *Álgebra y Geometría. Una manera de pensar*. EDULP. doi.org/10.35537/10915/87638

Frieder, S., Pinchetti, L., & Chevalier, A. (2023, Jul 20). Mathematical Capabilities of ChatGPT. https://arxiv.org/abs/2301.13867

Glorot, X., Bordes, A., & Bengio, Y. (2011). Deep Sparse Rectifier Neural Networks. *Proceedings of the fourteenth international conference on artificial intelligence and statistics*, 315--323.

Goldberg, Y. (2017). *Neural Network Methods for Natural Language Processing* (G. Hirst, Ed.). Morgan & Claypool.

Goodfellow, I., Bengio, Y., & Courville, A. (2016). *Deep Learning*. MIT Press.

Hernández Orallo, J., & Ferri Ramírez, C. (2004). *Introducción a la minería de datos*. Pearson Educación.

Hinton, G. E., Srivastava, H. N., Krizhevsky, A., Sutskever, I., & Salakhutdinov, R. R. (2012). Improving neural networks by preventing co-adaptation of feature detectors. *arXiv (Cornell University)*. 10.48550/arxiv.1207.0580

Jurafsky, D., & Martin, J. (2024). *Speech and Language processing*. https://web.stanford.edu/~jurafsky/slp3/

Kamath, U., Graham, K., & Emara, W. (2022). *Transformers for Machine Learning: A Deep Dive*. CRC Press.

Lample, G., & Charton, F. (2019). Deep Learning for Symbolic Mathematics. https://arxiv.org/abs/1912.01412

Lamport, L. (1994). *LATEX: a document preparation system : user's guide and reference manual*. Addison-Wesley.

LeCun, Y., Boser, B., Denker, J. S., Henderson, D., Howard, R. E., Hubbard, W., & Jackel, L. D. (1989). Backpropagation applied to handwritten Zip code recognition. *Neural computation*, *1*, 541--551. 10.1162/neco.1989.1.4.541

LeCun, Y., Bottou, L., Bengio, Y., & Haffner, P. (1998). Gradient-based learning applied to document recognition. *Proceedings of the IEEE*, *86*(11), 2278--2324. 10.1109/5.726791

Lin, C.-Y. (2004). ROUGE: A Package for Automatic Evaluation of Summaries. https://aclanthology.org/W04-1013

Marsden, J., & Tromba, A. (2012). *Vector Calculus*. Macmillan Learning.

Mikolov, T., Chen, K., Corrado, G., & Dean, J. (2013). Efficient Estimation of Word Representations in Vector Space. https://arxiv.org/pdf/1301.3781

Mikolov, T., Sutskever, I., Chen, K., Corrado, G., & Dean, J. (2013). Distributed Representations of Words and Phrases and their Compositionality. https://arxiv.org/pdf/1310.4546

Mittelbach, F., & Rowley, C. (1999). The LaTeX 3 Project. https://www.latex-project.org/help/documentation/ltx3info.pdf

Nielsen, M. A. (2015). *Neural networks and deep learning*. http://neuralnetworksanddeeplearning.com/

Ouyang, L., Wu, J., Jiang, X., Almeida, D., Wainwright, C., Mishkin, P., & Zhang, C. (2022). Training language models to follow instructions with human feedback. https://arxiv.org/pdf/2203.02155

Papineni, K., Roukos, S., Ward, T., & Zhu, W. (2002). BLEU: a Method for Automatic Evaluation of Machine Translation. https://dl.acm.org/doi/pdf/10.3115/1073083.1073135

Radford, A., Narasimhan, K., Salimans, T., & Sutskever, I. (2018, Jun 11). Improving Language Understanding by Generative Pre-Training. https://cdn.openai.com/research-covers/language-unsupervised/language_understanding_paper.pdf

Radford, A., Wu, J., Child, R., Luan, D., Amodei, D., & Sutskever, I. (2018). Language Models are Unsupervised Multitask Learners.

Raffel, C., Shazeer, N., Roberts, A., Lee, K., Narang, S., Matena, M., Zhou, Y., Li, W., & Liu, P. J. (2023, Sep 19). Exploring the Limits of Transfer Learning with a Unified Text-to-Text Transformer. https://arxiv.org/pdf/1910.10683

Rumelhart, D. E., Hinton, G. E., & Williams, R. J. (1986). Learning representations by back-propagating errors. *Nature*, (323), 533--536. 10.1038/323533a0

Sennrich, R., Haddow, B., & Birch, A. (2016). Neural Machine Translation of Rare Words with Subword Units. https://arxiv.org/abs/1508.07909

Trask, A., Hill, F., Reed, S., Rae, J., & Dyer, C. (2018). Neural Arithmetic Logic Units. https://arxiv.org/pdf/1808.00508

Trask, A. W. (2019). *Grokking Deep Learning*. Manning.

Vaswani, A., Shazeer, N., Parmar, N., Uszkoreit, J., Jones, L., Gomez, A. N., Kaiser, L., & Polosukhin, I. (2017). Attention is all you need. *arXiv (Cornell University)*. 10.48550/arxiv.1706.03762

Wang, Z., Lan, A. S., & Baraniuk, R. G. (2021). Mathematical formula representation via tree embeddings. *iTextbooks@ AIED*, 121--133.

Zhang, Z., Yu, W., Yu, M., Guo, Z., & Jiang, M. (2023). A Survey of Multi-task Learning in Natural Language Processing: Regarding Task Relatedness and Training Methods. arXiv:2204.03508

Zhuang, F., Qi, Z., Duan, K., Xi, D., Zhu, Y., Xiong, H., & He, Q. (2020). A Comprehensive Survey on Transfer Learning. https://arxiv.org/pdf/1911.02685