

UNIVERSIDAD MIGUEL HERNÁNDEZ DE ELCHE

ESCUELA POLITÉCNICA SUPERIOR DE ELCHE

GRADO EN INGENIERÍA INFORMÁTICA EN
TECNOLOGÍAS DE LA INFORMACIÓN



"Attention is all you need". Arquitectura
Transformers: descripción y aplicaciones

TRABAJO FIN DE GRADO

Junio – 2023

AUTOR: Alexis Fabian Nasimba Tipan

DIRECTOR/ES: Antonio Peñalver Benavent



“De todos los medios que conducen a la fortuna, los más seguros son la perseverancia y el trabajo.”

Marie Roch Louis Reybaud.

RESUMEN

El procesado del lenguaje natural, más conocido por sus siglas en inglés NLP (Natural language processing), ha ido evolucionando constantemente a lo largo de los años, llegando a estar presente en herramientas que el usuario común usa a diario, como es el traductor de Google. Esta rama del famoso **Machine Learning** ha tenido una aceptación muy grande entre la comunidad científica y entre las empresas, lo que está permitiendo un desarrollo vertiginoso.

Algunas de las aplicaciones más comunes de estos algoritmos de NLP, están en la **clasificación de textos, traductores de idioma o la generación de texto**. Debido a su gran versatilidad ya se están utilizando para la resolución de problemas del mundo real.

En esta búsqueda de las soluciones más eficientes a los problemas de un mundo cada más digitalizado, se han realizado avances en las investigaciones de nuevos algoritmos para la comprensión y generación de texto, como son los **Transformers**, la red neuronal con mayor acogida en este ámbito hasta el momento, debido a su gran potencial demostrado en modelos de lenguaje grandes como **GPT- 4** o **LaMDA**.

El objetivo de este proyecto es llevar a cabo un estudio profundo de la red neural conocida como **Transformer**, empezando por sus inicios, las redes neuronales que le preceden, su estructura y funcionamiento, su aplicación práctica en modelos actuales y finalmente resolveremos un problema mediante la elaboración de la red neuronal, entrenamiento y pruebas, pudiendo así realizar un análisis completo de los resultados obtenidos.

AGRADECIMIENTOS

Durante este proyecto de investigación, he tenido tiempo para reflexionar acerca de a qué me gustaría dedicar mi carrera profesional en el futuro. Para mí, nunca ha sido fácil decidir qué camino tomar para alcanzar mis objetivos a nivel profesional. Las experiencias en diferentes trabajos me han ayudado a llegar a este punto con ideas más claras. A pesar de que este proyecto no ha sido nada sencillo y me ha costado más de lo que esperaba, puedo afirmar con gran orgullo que, gracias a las noches sin dormir y los días sin descanso, ahora tengo más claro el camino que deseo seguir para continuar mi desarrollo profesional. Por todo ello, agradezco el esfuerzo invertido para lograr lo que a veces creemos que nunca llegará.

Por otra parte, quiero expresar mi agradecimiento a mi familia: a mi madre por ser el motor de mi vida y siempre confiar en mí, a mi padre por enseñarme el valor del trabajo y a nunca rendirme, a mi hermana mayor por guiarme cuando más lo necesito, a mi hermana pequeña por ser luz en mi vida y a mis mascotas simplemente por existir. Gracias también a mis compañeros de trabajo, quienes han comprendido que no es fácil compaginar estudios y trabajo, y me han facilitado poder terminar mis estudios. Agradezco a todos mis profesores y en especial a mi tutor, Antonio Peñalver, por descubrirme este mundo que ahora me parece tan fascinante, por su trato cercano y familiar, y por toda la ayuda que me ha brindado en este proyecto que simboliza el final de una etapa muy importante en mi vida.

Sé que me faltan muchos agradecimientos, pero ellos ya lo saben. Gracias a todos los que han confiado en mí.

ÍNDICE

CAPÍTULO 1: INTRODUCCIÓN.....	13
1.1 ¿Qué es la Inteligencia Artificial?	14
1.1.1 Tipos de IA.....	14
1.1.2 Aprendizaje Automático o Machine Learning.....	14
1.1.3 Aprendizaje Profundo o Deep Learning	15
1.2 ¿Qué son las Redes Neuronales?	16
1.3 Red Neuronal Biológica	17
1.3.1 Funcionamiento del cerebro.....	17
1.3.2 Estructura de la Neurona Biológica	17
1.4 Red Neuronal Artificial.....	19
1.4.1 Historia de las RNA	21
1.4.2 Conceptos básicos y funciones de una RNA	22
1.4.3 Neurona Artificial Genérica	23
1.4.4 Arquitectura de una Red Neuronal Artificial	25
1.4.5 Función de Pérdida	26
1.4.6 Función de Activación	28
1.4.7 Algoritmo Backpropagation	33
1.4.8 Método de Descenso del Gradiente	34
1.4.9 Optimizadores	37
1.5 Redes Neuronales Precedentes.....	39
1.5.1 Redes Neuronales Recurrentes (RNN).....	39
1.5.2 Long-Short Term Memory (LSTM).....	47
1.5.3 Problema principal: falta de memoria	53
CAPÍTULO 2: ESTADO DE LA CUESTIÓN.....	54

2.1 Análisis del PLN	55
2.2 Utilidad actual del PLN	56
2.3 Modelos más innovadores y conocidos.....	56
2.4 Modelos de generación de lenguaje	63
2.4.1 ChatGPT	63
2.4.2 Bing Chat.....	65
2.4.3 BARD	66
CAPÍTULO 3: OBJETIVOS	68
3.1 Objetivos Generales	69
3.2 Objetivos Técnicos.....	69
3.3 Objetivos Didácticos	70
3.4 Objetivos Personales.....	70
CAPÍTULO 4: HIPÓTESIS DEL TRABAJO.....	71
4.1 Herramientas de desarrollo y entorno de trabajo.....	72
4.1.1 Google Colab.....	72
4.2 Lenguaje de programación y librerías	76
4.2.1 Python.....	76
4.2.2 Bibliotecas.....	79
CAPÍTULO 5: METODOLOGÍA Y RESULTADOS.....	84
5.1 Gestión y coordinación del proyecto.....	85
5.1.1 Desglose de tareas.....	86
5.1.2 Diagrama de Gantt	87
5.1 Preparación de datos	88
5.1.1 Limpieza del texto	88
5.1.2 Técnicas de PLN en la etapa de preprocesamiento	90
5.1.3 Representación numérica.....	95

5.2 Redes Transformers	103
5.2.1 Breve historia.....	103
5.2.2 Precursores y fundamentos de las redes Transformers	104
5.2.3 Arquitectura de la Red Neuronal Transformer	108
5.3 Desarrollos y desafíos	118
5.3.1 Desarrollo 1: traducción automática	119
5.3.2 Desarrollo 2: análisis de sentimientos.....	147
5.3.3 Desarrollo 3: Modelo Q&A para Hacienda.....	157
CAPÍTULO 7: CONCLUSIONES.....	162
7.1 Conclusión.....	163
7.2 Líneas Futuras.....	164
BIBLIOGRAFÍA	166



ÍNDICE DE ILUSTRACIONES

Ilustración 1. Esquema de los diferentes escenarios en la IA	16
Ilustración 2. Estructura de una neurona biológica	18
Ilustración 3. Estructura jerárquica de un sistema basado en RNA.	23
Ilustración 4. Estructura de una red neuronal artificial.....	24
Ilustración 5. Estructura de capas de una red neuronal.	26
Ilustración 6. Estructura neurona artificial con sesgo	29
Ilustración 7. Función escalonada	30
Ilustración 8. Función sigmoïdal	31
Ilustración 9. Función tangente hiperbólica.....	31
Ilustración 10. Función ReLU	31
Ilustración 11. Función Leaky ReLU	32
Ilustración 12. Función Swish	32
Ilustración 13. Función Mish	33
Ilustración 14. Gráfico de la función de pérdida con una curva sencilla.....	35
Ilustración 15. Gráfico de una función de pérdida cualquiera	36
Ilustración 16. Neurona recurrente simple.....	40
Ilustración 17. Falta de memoria de una RNN	42
Ilustración 18. Relación entre palabras distanciadas.....	43
Ilustración 19. Despliegue de una RNN	43
Ilustración 20. Pendientes de una función cualquiera	46
Ilustración 21. Desvanecimiento del gradiente	46
Ilustración 22. Estructura de una RNN.....	48
Ilustración 23. Estructura de una LSTM.....	48
Ilustración 24. Celda LSTM.....	49
Ilustración 25. Acceso a celda LSTM	50
Ilustración 26. Forget gate.....	50
Ilustración 27. Procesamiento de datos en Input gate.....	51
Ilustración 28. Actualización de celda LSTM mediante el input gate	52
Ilustración 29. Output gate	52

Ilustración 30. ChatGPT	65
Ilustración 31. Bing Chat.....	66
Ilustración 32. Bard.....	67
Ilustración 33. Canalización de PLN.....	69
Ilustración 34. Logo de Google Colab	72
Ilustración 35. Interfaz de Google Colab.....	73
Ilustración 36. Cuadernos de Google Colab en Drive	74
Ilustración 37. Entorno de ejecución.....	75
Ilustración 38. Logo de Python	77
Ilustración 39. Versión de Python.....	77
Ilustración 40. Gráfico de uso de Python en la industria	78
Ilustración 41. Comunidad de Python en España	79
Ilustración 42. Visualización de datos con Pandas	80
Ilustración 43. Gráfico dinámico en 3D con Matplotlib.....	81
Ilustración 44. Características de Numpy	82
Ilustración 45. Visualización de un tensor	83
Ilustración 46. Funcionalidades Scikit-Learn.....	83
Ilustración 47. Desglose de tareas.....	87
Ilustración 48. Diagrama de Gantt	88
Ilustración 49. Proceso del pre-procesamiento.....	95
Ilustración 50. Representación label encoding	95
Ilustración 51. Palabra en codificación one-hot	96
Ilustración 52. Codificación one-hot de 4 colores.....	96
Ilustración 53. Espacio tridimensional	98
Ilustración 54. Arquitectura de embedding.....	99
Ilustración 55. Embedding	100
Ilustración 56. Relación vectorial.....	100
Ilustración 57. Modelos de entrenamiento de Word2Vec	101
Ilustración 58. Representación tridimensional de embeddings.....	102
Ilustración 59. Estructura de palabras relacionadas	102
Ilustración 60. Modelo de secuencia a secuencia	106
Ilustración 61. Modelo de secuencia a secuencia con atención.....	106

Ilustración 62. Proceso de atención.....	107
Ilustración 63. Ultimo decodificador	108
Ilustración 64. Arquitectura de un Transformer.....	109
Ilustración 65. Bloque codificador	111
Ilustración 66. Multi-Head Attention Part	112
Ilustración 67. Vector de atención.....	113
Ilustración 68. Múltiples vectores de atención	113
Ilustración 69. Feed Forward Network	114
Ilustración 70. Bloque decodificador.....	115
Ilustración 71. Masked Multi-Head Attention.....	116
Ilustración 72. Bloque de atención codificador-decodificador	117
Ilustración 73. Bloque de salida del decodificador	118
Ilustración 74. Comparación de la longitud de las frases.....	121
Ilustración 75. Distribución de N-gramas más Comunes en Inglés y Español	122
Ilustración 76. Proceso de Normalización de los Datos	123
Ilustración 77. Proceso de División de Datos para Vectorización	124
Ilustración 78. Creación y Entrenamiento de la Capa de Vectorización.....	124
Ilustración 79. Incorporación de la Capa de Vectorización al Modelo	125
Ilustración 80. Transformación de Frases a Formato Numérico.....	125
Ilustración 81. Datos para Entrenamiento y Validación.....	125
Ilustración 82. Distribución de Longitud de Frases en Términos de Tokens ...	126
Ilustración 83. Generación de la Matriz de Codificación Posicional.....	128
Ilustración 84. Implementación de la Capa de Embedding Posicional	129
Ilustración 85. Implementación de la Atención de Múltiples Cabezas	130
Ilustración 86. Implementación de la Atención Cruzada	130
Ilustración 87. Auto-Atención en la Secuencia de Entrada.....	131
Ilustración 88. Atención Cruzada entre Secuencias de Entrada y Contexto ...	132
Ilustración 89. Red Neuronal de Avance Punto a Punto	133
Ilustración 90. Visualización de la Red de Avance Punto a Punto	134
Ilustración 91. Unidad de Codificación en el Modelo Transformer	135
Ilustración 92. Flujo de Datos a través de la Unidad de Codificación	136
Ilustración 93. Implementación de la Unidad de Decodificación.....	137

Ilustración 94. Visualización del Decodificación en el Transformer.....	137
Ilustración 95. Arquitectura y Proceso del Modelo Transformer	139
Ilustración 96. Proceso de Transformación en un Modelo Transformer.....	140
Ilustración 97. Ajuste en el optimizador	141
Ilustración 98. Programación de la Tasa de Aprendizaje.....	142
Ilustración 99. Funciones para el Cálculo de la Pérdida y Precisión.....	142
Ilustración 100. Configuración del Modelo Transformer.....	144
Ilustración 101. Evaluación de Resultados del Transformer	144
Ilustración 102. Evolución de la Pérdida Durante el Entrenamiento	145
Ilustración 103. Evolución de la Precisión Durante el Entrenamiento.....	146
Ilustración 104. Resultado traducción automática.....	147
Ilustración 105. Distribución de calificaciones.....	149
Ilustración 106. Longitud de reseñas.....	150
Ilustración 107. Calificaciones positivas y negativas	151
Ilustración 108. Función de codificador.....	152
Ilustración 109. Función para los embedding de posición	153
Ilustración 110. Resultados análisis de sentimientos	157
Ilustración 111. Prueba con reseña negativa.....	157
Ilustración 112. Dataset Q&A.....	159
Ilustración 113. modelo T5.....	160
Ilustración 114. Resultado modelo de Q&A.....	161

ÍNDICE DE ECUACIONES

Ecuación 1. Función de activación de una neurona artificial.....	24
Ecuación 2. Simplificación de la Función de Activación.....	25
Ecuación 3. Error cuadrático medio.....	27
Ecuación 4. Error absoluto medio	27
Ecuación 5. Error de sesgo promedio	28
Ecuación 6. Bisagra Pérdida.....	28
Ecuación 7. Pérdida de entropía cruzada.....	28
Ecuación 8. Función de activación en una neurona artificial convencional	41
Ecuación 9. Cálculo de la activación en una red neuronal recurrente.....	41
Ecuación 10. Generación de la salida en una red neuronal recurrente.....	41
Ecuación 11. Formula Accuracy.....	155
Ecuación 12. Formula precision	155
Ecuación 13. Formula Recall.....	155
Ecuación 14. Formula F1	155



CAPÍTULO 1: INTRODUCCIÓN

1.1 ¿Qué es la Inteligencia Artificial?

El término de **inteligencia artificial**, al que se hará referencia a partir de ahora por sus siglas IA, corresponde a una rama de la informática que se fundamenta en el desarrollo de programas con la capacidad de resolver problemas de forma inteligente. Los campos potenciales de aplicación y los usos de la IA se diversifican cada vez más, incluyendo la **comprensión del lenguaje natural, reconocimiento visual, robótica, sistemas autónomos**, etc [1].

La rápida evolución de la IA en los últimos tiempos se ha hecho posible gracias a la aparición del **Cloud Computing y Big Data**, lo cual ha permitido la computación de bajo coste y el acceso a una gran cantidad de datos.

1.1.1 Tipos de IA

La **IA**, tal y como se conoce en la actualidad, puede clasificarse generalmente como **Inteligencia Artificial débil**. Se trata de algoritmos que son capaces de imitar comportamientos humanos de manera bastante realista, pero carecen de auténtica consciencia o entendimiento, distando de la "**Inteligencia Artificial fuerte**" o "**general**" que John McCarthy definió en 1956, en la que imaginaba una máquina con capacidades equiparables a las del cerebro humano [2].

La capacidad de aprendizaje de la IA se fundamenta principalmente en un subconjunto de esta, conocido como **Aprendizaje Automático** o **Machine Learning** (ML, por sus siglas en inglés). Además, se encuentra el **Aprendizaje Profundo** o **Deep Learning** (DL), que es una especialización dentro del Machine Learning. Estos conceptos se definen a continuación:

1.1.2 Aprendizaje Automático o Machine Learning

El **Aprendizaje Automático** o **Machine Learning** (ML) es un subcampo de la IA que se encarga del desarrollo de algoritmos capaces de aprender y mejorar su funcionamiento basándose en su experiencia previa. Las técnicas empleadas en

estos algoritmos permiten la identificación de patrones complejos en grandes volúmenes de datos, permitiéndoles generar sus propias reglas para la detección de patrones similares en nuevos conjuntos de datos. Este aprendizaje se basa en el análisis de muestras de datos reales que reflejen el proceso que se desea modelar. Una vez el modelo ha sido entrenado, será capaz de extraer conclusiones y realizar tareas para las cuales no fue específicamente programado, partiendo únicamente de los datos observados. Existen diferentes formas de realizar este aprendizaje, dando lugar a distintos tipos de aprendizaje automático. De forma general, estos tipos pueden ser clasificados según distintos criterios [3]:

1. Aprendizaje supervisado

Se entrena al sistema con datos etiquetados, es decir, se conoce el resultado deseado y el algoritmo aprende a predecir estos resultados en un nuevo conjunto de datos.

2. Aprendizaje no supervisado

El algoritmo no cuenta con información previa, el sistema observa las características de los datos y busca patrones por sí mismo.

3. Aprendizaje por refuerzo

El sistema aprende a partir de la experiencia, es decir, a través de un proceso de prueba y error, siendo recompensado cuando toma la decisión correcta, lo que le permite generar estrategias automáticas para optimizar un proceso.

1.1.3 Aprendizaje Profundo o Deep Learning

El **Aprendizaje Profundo** o **Deep Learning** (DL) es una especialización del aprendizaje automático que se inspira en el funcionamiento del cerebro humano. Su objetivo es que los algoritmos sean capaces de aprender por sí mismos a partir de un conjunto inicial de datos. Este enfoque utiliza **redes neuronales artificiales**, que imitan la estructura y funcionamiento de las neuronas en el cerebro humano, con el fin de lograr un aprendizaje similar al humano.

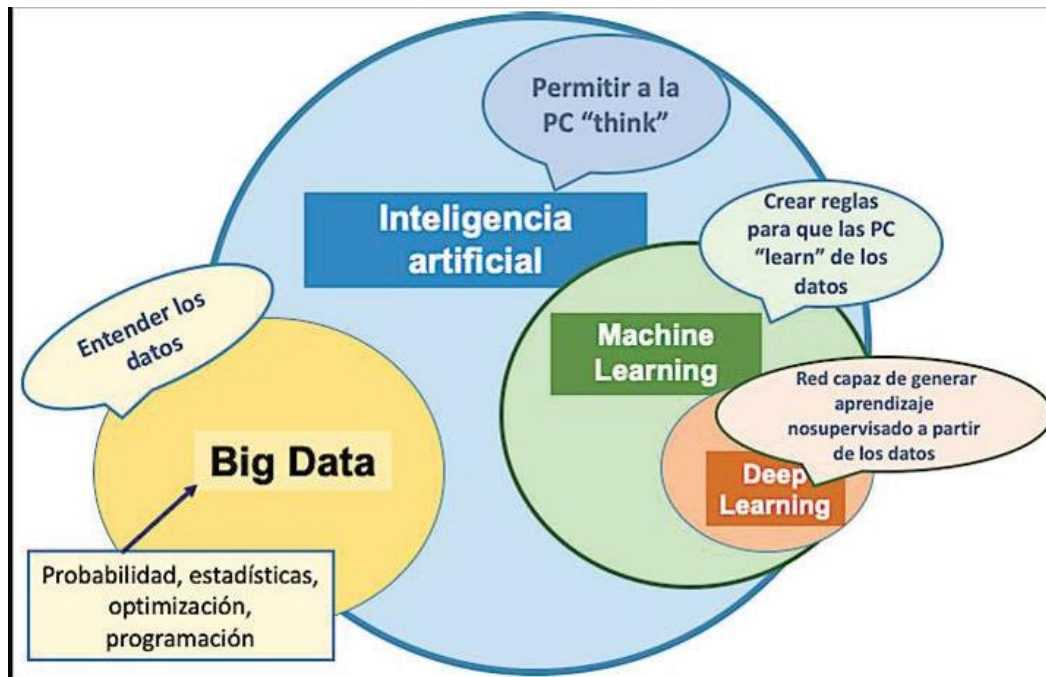


Ilustración 1. Esquema de los diferentes escenarios en la IA

1.2 ¿Qué son las Redes Neuronales?

Los **ordenadores** son lógicos, exactos y capaces de realizar tareas complejas, como el almacenamiento de grandes cantidades de información. Por otro lado, los seres humanos pueden manejar conceptos abstractos como la incertidumbre, la ambigüedad y la suposición, que resultan ser de una complejidad considerable para un ordenador.

Con la intención de combinar la gran capacidad de los ordenadores y la flexibilidad, creatividad y capacidad de aprendizaje humana, se ha buscado desarrollar un **sistema de inteligencia artificial** que imite las características de un sistema biológico humano. Este desarrollo ha representado un reto para la comunidad científica de múltiples disciplinas, ya que implica la posibilidad de crear sistemas capaces de emular el funcionamiento del cerebro humano y los procesos biológicos de las redes neuronales presentes en los organismos vivos. A lo largo de la historia, se han utilizado diferentes enfoques y tecnologías para intentar alcanzar este objetivo, siendo las **redes neuronales** uno de los últimos y más destacados.

Para lograr una mejor comprensión del funcionamiento de las **redes neuronales artificiales**, se explicará primero el funcionamiento de una red neuronal biológica.

1.3 Red Neuronal Biológica

1.3.1 Funcionamiento del cerebro

El **cerebro** es un componente indispensable en el ser humano, compuesto por millones de **neuronas interconectadas** entre sí, resultando fundamental para el aprendizaje y desarrollo humano. Se estima que un cerebro humano contiene alrededor de 100 mil millones de neuronas. Estas neuronas están meticulosamente organizadas y conectadas entre sí, con una única neurona recibiendo, aproximadamente, un millar de conexiones. Esta densidad de conexiones crea un sistema de una complejidad significativa que dificulta su estudio y comprensión.

Desde esta perspectiva, el cerebro puede ser considerado como una máquina altamente eficiente y sofisticada, refinada a lo largo de millones de años de **evolución biológica** [4].

1.3.2 Estructura de la Neurona Biológica

Independientemente de su morfología, la mayoría de las neuronas en el cerebro se conforman a un modelo general, tal como se describe en el "*principio de polarización dinámica*" de Cajal. Una neurona tiene zonas dedicadas a la recepción de mensajes, a la integración de estos, a la conducción y, finalmente, a la codificación y transmisión de la información a otra neurona. Estas funciones se asocian a las **dendritas**, al **soma neuronal**, al axón y a la terminal sináptica, respectivamente, que constituyen los principales compartimentos morfofuncionales.

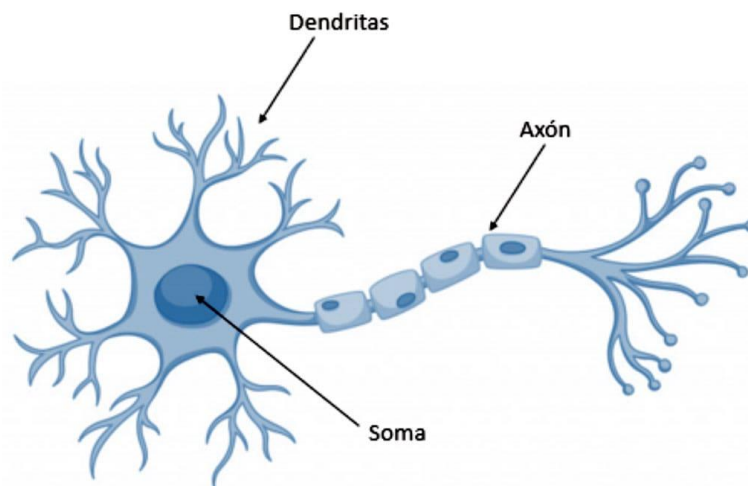


Ilustración 2. Estructura de una neurona biológica

- **Cuerpo celular o soma** (10 a 80 micras de longitud): Puede recibir información de otras neuronas a través de la sinapsis, la conexión entre ellas, actuando, así como un órgano de procesamiento.
- **Árbol dendrítico:** Este árbol se origina en el soma y está formado por dendritas, las cuales se especializan en la recepción de señales de otras células nerviosas a través de conexiones sinápticas, funcionando como un canal de entrada de información.
- **Axón:** Fibra tubular que se extiende desde el soma (longitud de 100 micras hasta un metro) y se ramifica en su extremo final para conectar con otras neuronas, facilitando la transmisión de información mediante impulsos a otras células nerviosas.

Por lo tanto, la función principal de la neurona, considerada de forma aislada del sistema nervioso, es recibir información, procesarla y transmitirla a la siguiente neurona. Según Charles Sherrington (fisiólogo contemporáneo de Ramón y Cajal que recibió el Premio Nobel de Fisiología y Medicina en 1932), la neurona es la **unidad de integración** y su función sintetiza el sistema nervioso en su totalidad [4].

Las funciones del sistema nervioso se deben a la actividad de grupos neuronales interconectados. Se estima que existen alrededor de 100 billones de conexiones

sinápticas, que son responsables de la compleja funcionalidad del cerebro humano.

1.4 Red Neuronal Artificial

Las **Redes Neuronales Artificiales** (RNA), también conocidas por sus siglas en inglés como Artificial Neural Systems (ANS), juegan un papel fundamental en el campo de la **Inteligencia Artificial** (IA). Su diseño se inspira en el funcionamiento del cerebro humano para procesar datos, aprender de los errores y mejorar de manera continua. Esta estrategia pertenece al **Aprendizaje Profundo** (Deep Learning), que utiliza nodos o neuronas interconectadas en una estructura de capas similar al cerebro humano. Por tanto, las neuronas artificiales intentan resolver problemas complejos aprendiendo a partir de los datos de entrada.

Las RNA se vuelven relevantes porque, con mínima intervención humana, pueden permitir que un ordenador tome decisiones inteligentes como:

- **Extraer conclusiones:** Las RNA pueden entender datos no estructurados y realizar observaciones generales sin entrenamiento explícito. Un ejemplo sería el reconocimiento de frases distintas con el mismo significado, basándose en su contexto.
- **Revelar patrones ocultos:** Las RNA pueden analizar datos sin procesar en profundidad y revelar nueva información para la que se ha entrenado previamente. Un ejemplo sería un patrón de reconocimiento de las compras de los consumidores, donde la red compara los patrones de los usuarios y sugiere nuevas compras con precisión.
- **Sistemas de aprendizaje autónomas:** Las RNA pueden aprender y mejorar en función del comportamiento del usuario. Por ejemplo, si se utiliza frecuentemente una palabra incorrecta en una red neuronal que se ha entrenado para corregir palabras en inglés, la red aprenderá y terminará por corregirla automáticamente.
- **Modelación de datos volátiles:** Algunos conjuntos de datos resultan complejos debido a sus variaciones, y estas redes pueden modelarlos.

Por ejemplo, pueden procesar datos complejos clave para resolver problemas biológicos difíciles como el plegamiento de proteínas o análisis del ADN.

Gracias a su sofisticación, muchos sectores aplican las RNA:

1. Visión artificial

Las redes neuronales pueden distinguir y reconocer imágenes de manera similar o incluso superior a los humanos.

1. Reconocimiento de voz

Pueden analizar el habla humana a pesar de las variaciones en patrones de habla, tono, idioma y acento.

2. Motores de recomendaciones

Pueden seguir la actividad y el comportamiento del usuario para generar recomendaciones personalizadas.

3. Procesamiento del lenguaje natural (PLN)

Ayudan al software a obtener información y significado a partir de documentos de texto.

En resumen, al crear una Red Neuronal Artificial, la intención es emular el sistema neuronal biológico, estableciendo una estructura jerárquica similar a la existente en el cerebro. El elemento principal en esta estructura es la **neurona artificial**, que se explicará más adelante [5].

1.4.1 Historia de las RNA

La historia de las **Redes Neuronales Artificiales** (RNA) tiene sus raíces en las primeras etapas del desarrollo computacional y científico. Aunque la revolución de la Inteligencia Artificial (IA) parece reciente, los primeros planteamientos e incursiones en este campo se han producido en este orden:

- **1936:** Alan Turing, conocido como el padre de la computación, comenzó a estudiar el cerebro humano como un medio para entender la computación. Sin embargo, los primeros teóricos que concibieron los fundamentos de la computación neuronal fueron Warren McCulloch, un neurofisiólogo, y Walter Pitts, un matemático. En 1943, publicaron una teoría sobre el funcionamiento de las neuronas, siendo los primeros en modelar una red neuronal simple mediante circuitos eléctricos.
- **1949:** Donald Hebb se convirtió en el primero en proporcionar una explicación detallada sobre los procesos de aprendizaje, creando una regla para entender cómo ocurre este proceso. Hoy en día, este principio se aplica en la mayoría de las funciones de aprendizaje utilizadas en una RNA.
- **1950:** A través de sus ensayos, Karl Lashley descubrió que la información no se almacenaba de manera centralizada en el cerebro, sino que se distribuía por toda la estructura cerebral. En 1956, se celebró el Congreso de Dartmouth, un evento que se considera el punto de partida de la IA moderna.
- **1957:** Frank Rosenblatt comenzó el desarrollo del Perceptrón, que se considera la primera red neuronal. Este modelo tenía la capacidad de generalizar, es decir, podía reconocer nuevos patrones similares a los que se le presentaron durante su entrenamiento. Sin embargo, tenía ciertas limitaciones, como la incapacidad de clasificar clases que no son linealmente separables.
- **1960:** Bernard Widrow y Marcian Hoff desarrollaron la primera RNA aplicada a un problema del mundo real: diseñaron filtros adaptativos para eliminar el eco en las líneas telefónicas. Este modelo fue conocido como Adaline (Adaptive Linear Elements).

- **1969:** Marvin Minsky y Seymour Papert demostraron matemáticamente que el Perceptrón no podía resolver problemas simples del mundo real, ya que no podía manejar funciones no lineales, que se utilizan extensamente en la computación y en problemas del mundo real. Este hallazgo marcó un declive en el estudio y desarrollo de las RNA.
- **1974:** Paul Werbos introdujo la idea de aprendizaje mediante retropropagación (backpropagation), un concepto que aclaró en 1985.
- **1977:** Stephen Grossberg, a través de la Teoría de la Resonancia Adaptativa, introdujo una arquitectura de red que simula otras habilidades cerebrales, como la memoria a corto y largo plazo.
- **1986:** David Rumelhart y James McClelland redescubrieron el algoritmo de aprendizaje por retropropagación (backpropagation) para el aprendizaje en las RNA.

Desde ese momento, la investigación y el desarrollo de las RNA han experimentado un crecimiento constante, con la publicación frecuente de trabajos sobre nuevas aplicaciones y desarrollos de las redes neuronales [6].

1.4.2 Conceptos básicos y funciones de una RNA

Para entender la estructura de una **Red Neuronal Artificial (RNA)**, es necesario destacar algunos conceptos clave del funcionamiento del cerebro humano que las RNA intentan reproducir de manera análoga. Estos conceptos son los siguientes:

1. Procesamiento paralelo

Esta funcionalidad, producto de los miles de millones de neuronas que intervienen en el cerebro humano, resulta esencial para realizar una gran cantidad de cálculos en el menor tiempo posible. Un buen ejemplo de esto es el proceso de visión, en el cual las neuronas están operando en paralelo sobre la imagen en su totalidad.

2. Memoria distribuida

A diferencia de los ordenadores, donde la información se almacena en posiciones de memoria definidas, en las redes neuronales biológicas, esta

información se distribuye a través de las sinapsis de la red. Esto permite una redundancia en el almacenamiento, lo cual previene la pérdida de información en caso de fallo en alguna sinapsis.

3. Adaptabilidad al entorno

A partir de la información de las sinapsis, las redes neuronales pueden aprender de la experiencia y generalizar conceptos a partir de casos particulares.

A partir de las tres propiedades mencionadas anteriormente, se puede concluir que es posible establecer una estructura jerárquica similar con las RNA. El elemento de partida sería la neurona artificial, organizada en capas, formando una red con sus interfaces de entrada y de salida. Como resultado final, estas neuronas constituirán un sistema global de procesamiento [7].

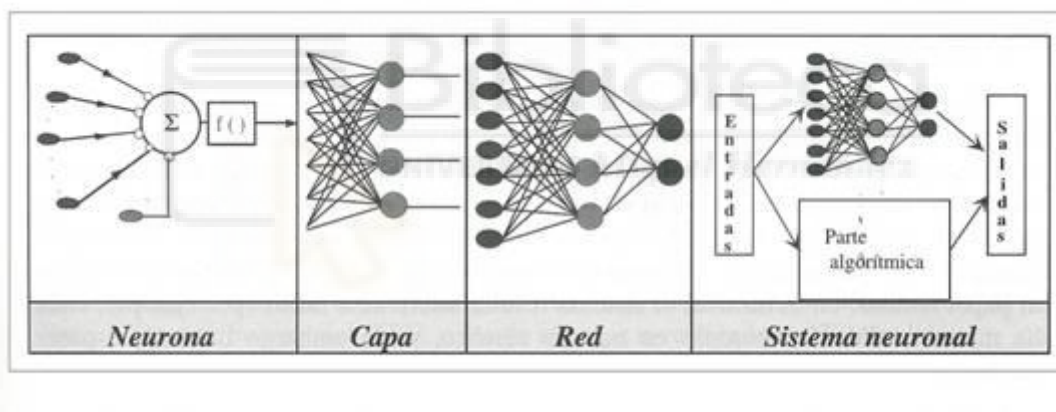


Ilustración 3. Estructura jerárquica de un sistema basado en RNA.

1.4.3 Neurona Artificial Genérica

La neurona más comúnmente encontrada en las **Redes Neuronales Artificiales** (RNA) fue definida por **McCulloch y Pitts** en 1943, exhibiendo una estructura similar a una neurona biológica. Los componentes de esta estructura incluyen:

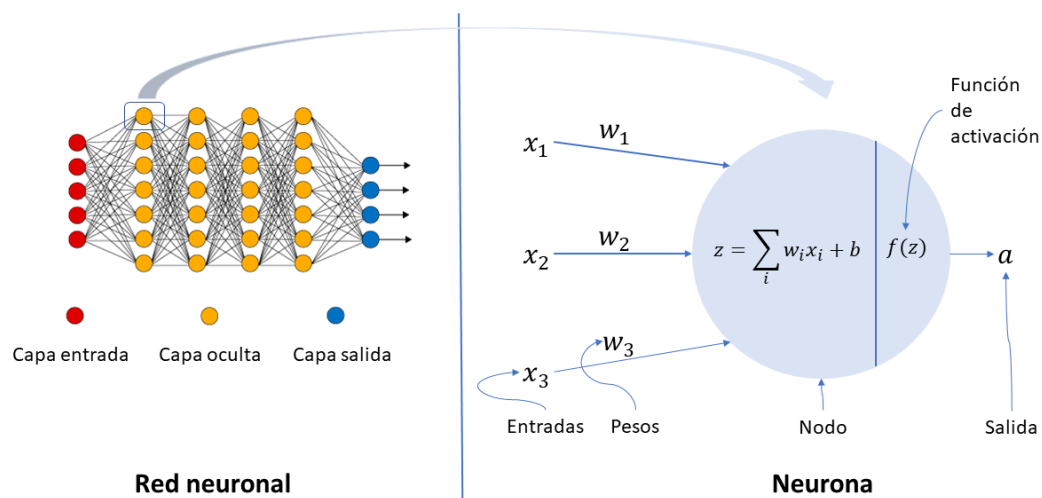


Ilustración 4. Estructura de una red neuronal artificial

- **Entrada:** Esta es representada por un vector x , que se corresponde con el vector de entradas. Estas entradas, que reciben los datos de otras neuronas, se corresponden con las dendritas en una neurona biológica
- **Pesos:** Son representados por un vector de pesos w , que se asemejan a las conexiones sinápticas en la neurona biológica. En una neurona artificial, a las entradas que provienen de otras neuronas se les asigna un peso, o factor de importancia. Este peso se modifica durante el entrenamiento de la red.
- **Suma ponderada:** Se realiza una suma ponderada con estos vectores de entrada (x) y pesos sinápticos (w), se realiza una suma ponderada para obtener el valor del potencial sináptico.
- **Función de activación:** El valor obtenido en la suma ponderada se filtra a través de esta función para tener una única salida (a). Dependiendo del uso que se le dé a la neurona, se utilizará una función de activación u otra.
- **Salida:** La salida es representada con la siguiente expresión, donde a es la salida que se corresponde con una función no lineal:

$$a = f(z) = f\left(\sum_i w_i x_i\right) = f(w^T x)$$

Ecuación 1. Función de activación de una neurona artificial

Simplificándola, obtendríamos la siguiente:

$$a = f(wx)$$

Ecuación 2. Simplificación de la Función de Activación

Esta representación simplificada de las funciones matemáticas utilizadas en la estructura de una neurona artificial proporciona una visión general y accesible. Sin embargo, es importante mencionar que las matemáticas subyacentes en la neurona artificial pueden ser bastante complejas y extensas, abarcando varias ramas de las matemáticas y la informática. Para los propósitos de este documento, la atención se centrará en una comprensión general y aplicada de las neuronas artificiales [8].

1.4.4 Arquitectura de una Red Neuronal Artificial

Se entiende por **Red Neuronal Artificial** (RNA) un modelo computacional que es capaz de efectuar procesos de aprendizaje, tanto supervisados, como la clasificación y la regresión, como no supervisados, como el agrupamiento de datos y clustering. Está compuesto este modelo por un conjunto de unidades neuronales simples, cada una dotada de una **función de activación** específica. La activación de esta función es necesaria antes de que la señal pueda propagarse al resto de las unidades neuronales.

Se encuentra cada unidad neuronal interconectada con las demás a través de enlaces, a los cuales se asocian ciertos pesos. Estos pesos tienen la capacidad de incrementar o disminuir el valor de cada neurona en la red, afectando así la salida de la misma.

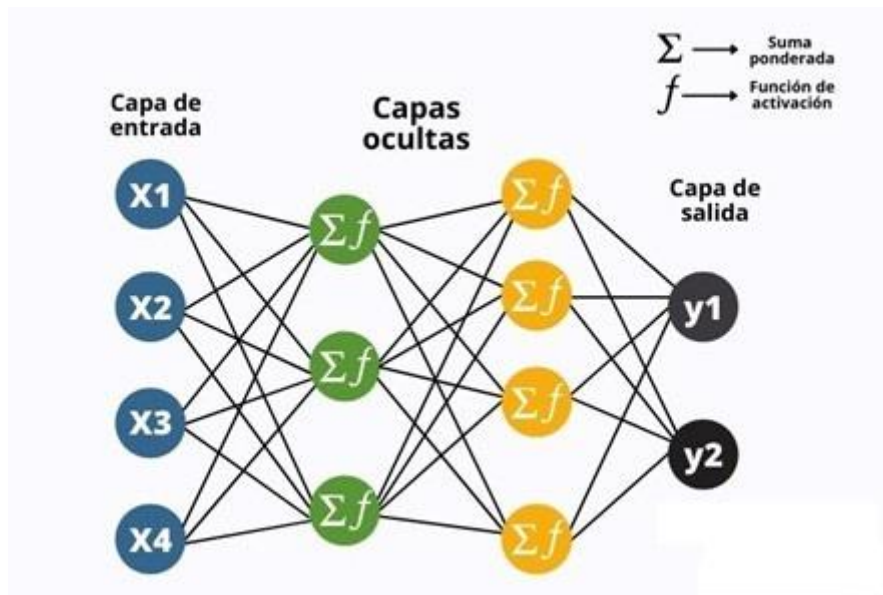


Ilustración 5. Estructura de capas de una red neuronal.

Las unidades neuronales se organizan en unidades estructurales denominadas capas. Estas capas son conjuntos de neuronas que se encargan de procesar la información que se desea tratar. En el proceso, se distinguen capas ocultas, que se encuentran en el medio, ya que su manipulación de la información no es directamente observable. Por otro lado, las capas de entrada y salida de datos son fácilmente identificables durante el entrenamiento y uso de la red [9].

1.4.5 Función de Pérdida

El objetivo principal de las **redes neuronales artificiales** es desarrollar la capacidad de aprender de manera autónoma. Durante el proceso de entrenamiento, se utiliza un método conocido como **propagación hacia atrás** (*backward*) para introducir el conocimiento previo del resultado deseado en la red. A través de este método, se realizan ajustes en las funciones de activación y los pesos neuronales de acuerdo con los resultados conocidos.

En el centro de este proceso se encuentra la **función de pérdida**, que evalúa la discrepancia entre el valor de salida de la red y el valor real de los datos de entrenamiento. De manera sencilla, la función de pérdida indica la precisión de la red neuronal en la realización de predicciones para una entrada específica.

La estrategia para minimizar el valor de la función de pérdida implica la continua modificación de los pesos asociados a los enlaces entre las neuronas, así como el ajuste del sesgo en cada neurona [10].

La elección de la función de pérdida varía según el tipo de problema abordado, ya sea de **regresión o clasificación**:

Para problemas de regresión:

- **Error cuadrático medio (Mean Square Error) / Pérdida cuadrática / Pérdida L2:** Esta función calcula el promedio de la diferencia al cuadrado entre las predicciones y las observaciones reales, sin tener en cuenta la dirección. Se penaliza de manera significativa las predicciones erróneas, aprovechando propiedades matemáticas que simplifican el cálculo de los gradientes.

$$MSE = \frac{\sum_{i=1}^n (y_i - \hat{y}_i)^2}{n}$$

Ecuación 3. Error cuadrático medio

- **Error absoluto medio (Mean Absolute Error) / Pérdida L1:** Similar a la pérdida L2, esta función calcula el promedio de la suma de las diferencias absolutas entre las predicciones y las observaciones reales. Aunque es robusta frente a valores atípicos, su cálculo de gradientes requiere herramientas más complejas.

$$MAE = \frac{\sum_{i=1}^n |y_i - \hat{y}_i|}{n}$$

Ecuación 4. Error absoluto medio

- **Error de sesgo promedio (Mean Bias Error):** Menos común en Machine Learning, esta función es similar a la pérdida L2, pero sin tomar valores absolutos. De este modo se podría ayudar a determinar si el modelo tiene un sesgo positivo o negativo, aunque los errores positivos y negativos puedan anularse mutuamente.

$$MBE = \frac{\sum_{i=1}^n (y_i - \hat{y}_i)}{n}$$

Ecuación 5. Error de sesgo promedio

Para problemas de clasificación:

- **Bisagra Pérdida (Hinge Loss) / Multi clase SVM Loss:** La pérdida de bisagra exige que el puntaje de la categoría correcta sea mayor que la suma de los puntajes de todas las categorías incorrectas por algún margen de seguridad. Se utiliza comúnmente en máquinas de vectores de soporte, es una función convexa que simplifica el trabajo con los optimizadores convexos habituales.

$$SVMLoss = \sum_{i=1} \max(0, s_j - s_j + 1)$$

Ecuación 6. Bisagra Pérdida

- **Pérdida de entropía cruzada (Cross Entropy Loss) / Probabilidad de registro negativo:** Comúnmente se utilizada para problemas de clasificación, la pérdida de entropía cruzada aumenta a medida que la probabilidad prevista diverge de la etiqueta real. Se penaliza fuertemente las predicciones que sean seguras pero erróneas [11].

$$CrossEntropyLoss = (y_i \log(\hat{y}_i) + (1 - y_i) \log(1 - \hat{y}_i))$$

Ecuación 7. Pérdida de entropía cruzada

De este modo, se logrará que el modelo evolucione en función de los datos de entrada, permitiéndole realizar ajustes y adaptarse de la mejor manera posible al conjunto de datos de entrenamiento.

1.4.6 Función de Activación

Para comprender completamente la **función de activación**, es esencial tener conocimiento del procesamiento de la información dentro de una red neuronal. En primer lugar, se reciben **valores de entrada** en la red que se multiplican por sus respectivos pesos, los cuales provienen de otras neuronas y representan su

importancia relativa. A estos valores se les agrega un término independiente conocido como sesgo, que es una variable de entrada adicional asociada a la constante 1. El sesgo puede ajustarse mediante la modificación de su peso correspondiente, lo que permite un **entrenamiento más preciso** de la red. Si la salida de un nodo o neurona supera el sesgo especificado, el nodo se activa, lo que permite la transmisión de datos a la siguiente capa. En caso contrario, la información no se transfiere.

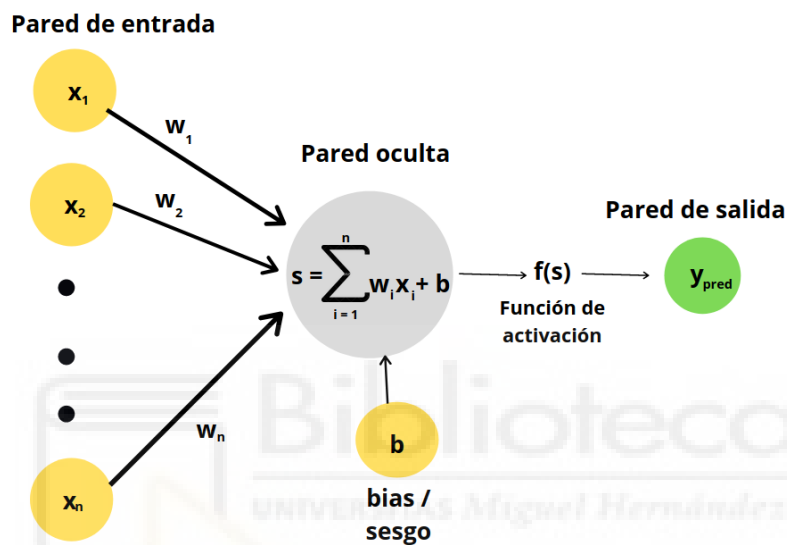


Ilustración 6. Estructura neurona artificial con sesgo

Al sumar estas multiplicaciones, se obtiene una **función lineal**, una situación que se intenta evitar. Esto se debe a que la suma de múltiples líneas rectas es equivalente a una única línea recta, es decir, se estaría realizando una única operación de regresión lineal. Este escenario plantea una limitación en la funcionalidad de la red neuronal, ya que por más capas que se apliquen a nuestros datos, el resultado final sería subóptimo.

Aquí es donde entra en juego la **función de activación**, proporcionando la no linealidad a la salida de cada neurona o nodo. Esto permite que la red neuronal pueda aproximarse a cualquier función, lo que la convierte en una pieza fundamental en el desempeño y la versatilidad de las redes neuronales.

La versatilidad de una red neuronal para realizar procesos de regresión y clasificación complejos proviene de la combinación adecuada de múltiples neuronas. Esta capacidad se atribuye en gran medida a la función de activación, una función no lineal que permite a la red neuronal aprender **funciones de transferencia no lineales**. Esta característica es crucial para abordar problemas de mayor complejidad y se utiliza en todos los tipos de redes neuronales. Las funciones de activación más comunes incluyen:

- **Función escalonada:** Se genera una salida binaria de valor real (0 o 1), dependiendo de la entrada.

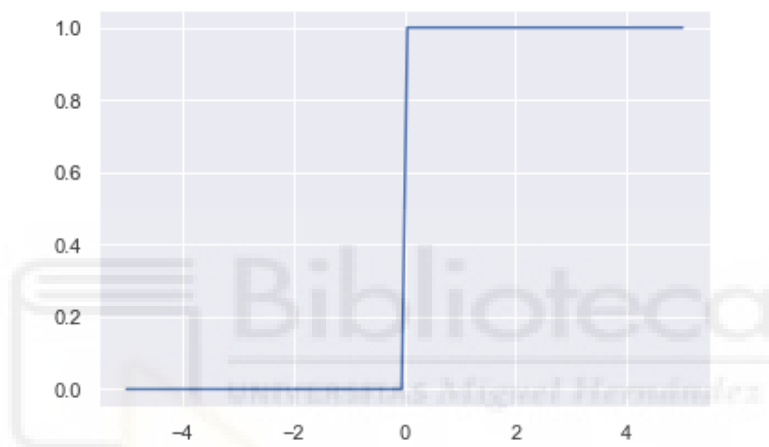


Ilustración 7. Función escalonada

- **Función sigmoide (Sigmoid):** Útil por su curva característica, permite representar probabilidades en el rango de 0 a 1. Sin embargo, su uso se encuentra limitado a la clasificación binaria debido a un problema de saturación.

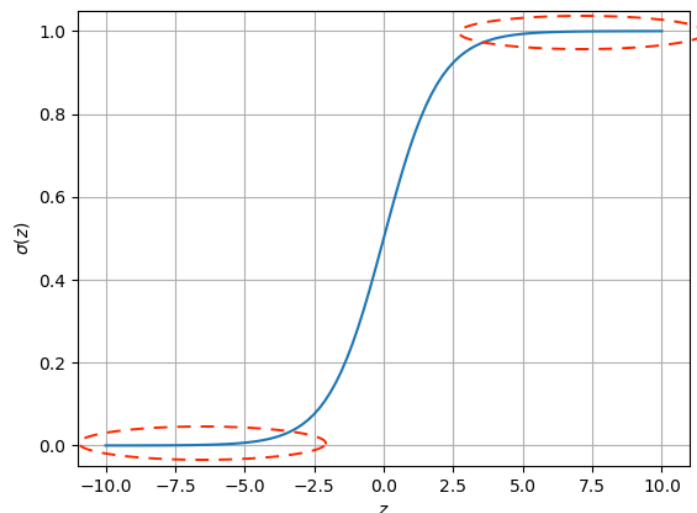


Ilustración 8. Función sigmoideal

- **Función tangente hiperbólica (tanh):** Se exhibe un comportamiento similar a la función sigmoide. Aunque también se sufre del problema de saturación, el cual ofrece la ventaja de tener una salida simétrica, facilitando de este modo el proceso de entrenamiento.

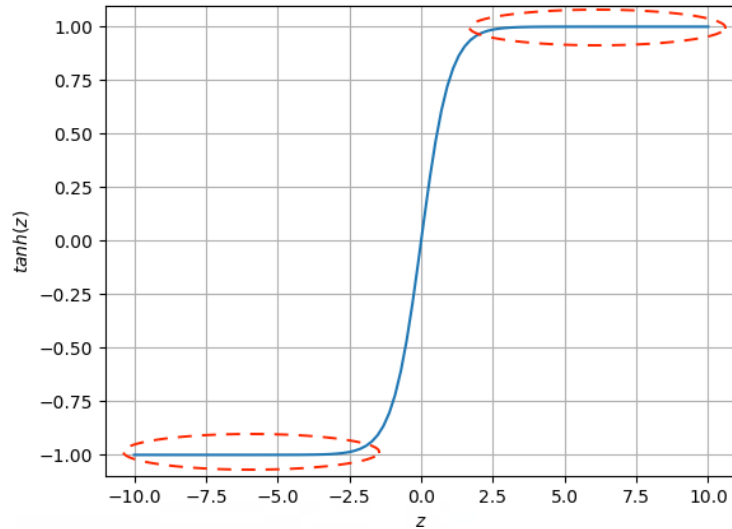


Ilustración 9. Función tangente hiperbólica

- **Función ReLU (Rectified Linear Unit):** se genera una salida igual a 0 cuando la entrada es negativa y una salida igual a la entrada cuando ésta se encuentra positiva. Se ha convertido en la más utilizada en los modelos de Deep Learning en los últimos años, gracias a su no saturación y a su fácil implementación computacional.

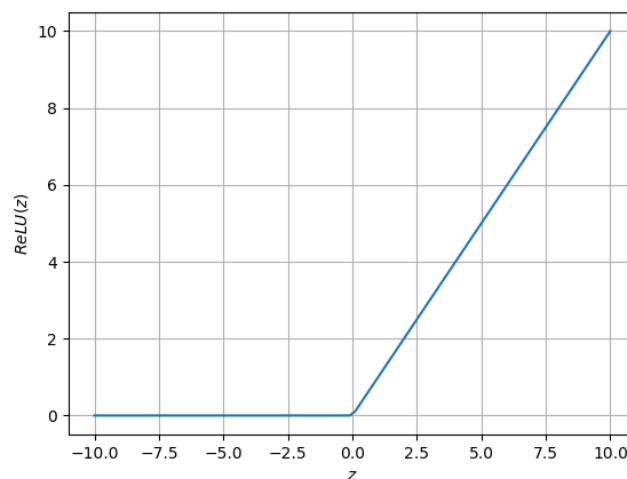


Ilustración 10. Función ReLU

- **Función Leaky ReLU (Leaky Rectified Linear Unit):** Como variante de ReLU, la Leaky ReLU aborda el problema de las activaciones negativas

desechadas al permitirse una pequeña activación negativa. Esto se logra aplicando un pequeño coeficiente a las entradas negativas en lugar de asignarles cero, evitando así la "muerte" de las neuronas y mejorando la convergencia del modelo. Además, se mantiene todas las ventajas de eficiencia computacional de la ReLU original [12].

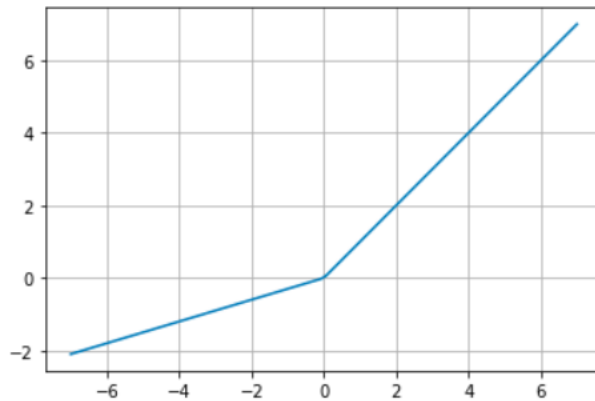


Ilustración 11. Función Leaky ReLU

- **Función Swish:** introducida en 2017, ha superado a la función ReLU en redes neuronales convolucionales más profundas. A diferencia de ReLU, la función Swish ofrece una respuesta suave y no lineal tanto para las entradas negativas como para las positivas, similar a la función sigmoide. Aunque proporciona una respuesta leve a entradas negativas, lo que facilita la retropropagación, su costo computacional es mayor que el de ReLU [13].

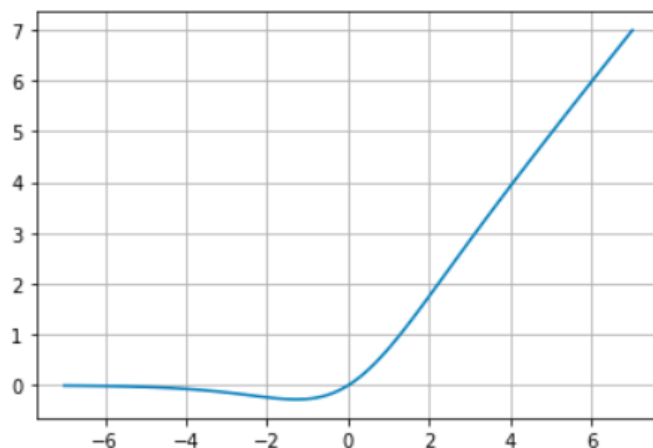


Ilustración 12. Función Swish

- **Función Mish:** Introducida en 2020, la función de activación Mish es una función auto-regulada no-monotónica que ofrece una innovación en la regularización de la optimización de las redes neuronales. Al igual que la función Swish, Mish tiene una característica única en su derivada, lo que facilita la comprensión del comportamiento durante la retropropagación. Sin embargo, su implementación puede ser más costosa en términos computacionales [14].

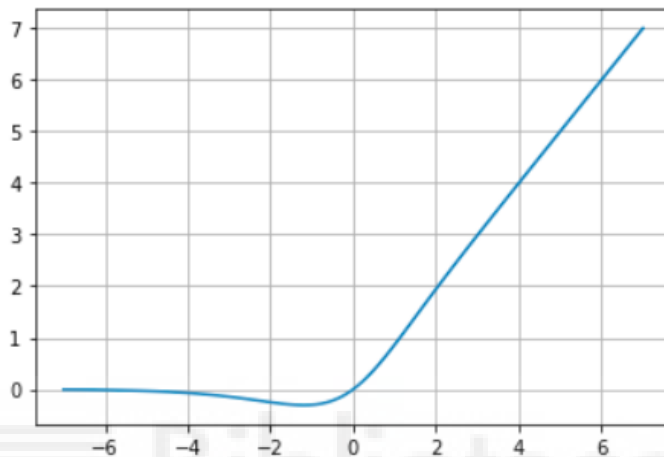


Ilustración 13. Función Mish

Por lo tanto, se obtiene una concatenación de modelos lineales que, una vez modificados por la función de activación, pueden procesar eficientemente la información de entrada [15].

1.4.7 Algoritmo Backpropagation

El **aprendizaje** en una red neuronal implica ajustar cada peso para minimizar la discrepancia entre el valor computado por la red y el valor objetivo, que se obtiene del conjunto de entrenamiento o **training set**. Este conjunto se introduce en la red neuronal, que a partir de él hace predicciones.

Históricamente, el aprendizaje en redes neuronales se llevaba a cabo "hacia adelante", desde la introducción del **Perceptrón**, observando cómo los cambios en la primera capa afectaban a las subsiguientes. No obstante, este enfoque resultaba problemático en **una red neuronal densa**, con una gran cantidad de

conexiones entre las neuronas, ya que requería rastrear cada ruta desde cada neurona hasta la salida final.

La solución a este problema se encontró en el algoritmo de **Backpropagation**, propuesto por Rumelhart, Hinton y Williams en 1986. Este algoritmo revolucionó el campo del Machine Learning y marcó el fin de la era conocida como "*El invierno de la Inteligencia Artificial*", que duró más de 15 años.

El enfoque de Backpropagation implica calcular el error "hacia atrás" en lugar de "hacia adelante". Tras calcular una respuesta, el algoritmo retrocede y **ajusta los pesos de cada conexión y sesgo**. Cada ciclo de corrección del error propagado hacia atrás para reducir la pérdida se conoce como **época**.

En términos claros, la **Backpropagation** determina los mejores ajustes de los pesos y los sesgos para obtener un resultado más preciso o "**minimizar la pérdida**". Este proceso es intensivo en recursos, pero con la llegada de hardware computacionalmente poderoso, se ha vuelto práctico [10].

Este proceso se realiza capa por capa, simplificando el trabajo, ya que en cada paso sólo se modifican los pesos de una sola capa y no todos los pesos de cada ruta posible.

El algoritmo de Backpropagation se basa en el cálculo de las **derivadas parciales** de la función de coste con respecto a los pesos y sesgos. Este proceso permite conocer la influencia de cada neurona en la salida a través del **vector gradiente**, que se utiliza en el método de **descenso del gradiente** para el aprendizaje de la red. [9]

Es importante mencionar que este algoritmo requiere una comprensión profunda de las matemáticas subyacentes, incluyendo derivadas parciales y cálculo del gradiente, para su total comprensión

1.4.8 Método de Descenso del Gradiente

El ajuste de las ponderaciones (o **pesos**) en cada época de entrenamiento de la red neuronal se lleva a cabo utilizando la **función de pérdida**. Esta función condensa la precisión del modelo en un único número, que representa cuán distantes están las predicciones de la red neuronal de los resultados esperados según el **conjunto de entrenamiento**. El objetivo del entrenamiento es encontrar la combinación de ponderaciones que minimicen el valor de esta función de pérdida.

La función de pérdida se representa como una curva con sus propios gradientes, los cuales están representados por un **vector de gradiente**. Este vector orienta los ajustes de las ponderaciones. La pendiente de la curva de la función de pérdida nos indica hacia dónde debemos mover los pesos para alcanzar el mínimo de la función de pérdida, que representa el estado en el que nuestra red hace las predicciones más precisas.

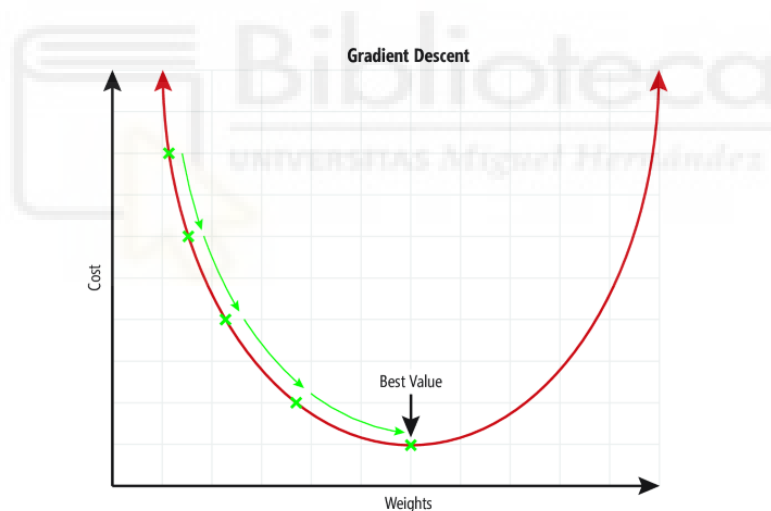


Ilustración 14. Gráfico de la función de pérdida con una curva sencilla

Cuando la pendiente es negativa, se incrementan las ponderaciones; cuando es positiva, se reducen. La cantidad que se añade o se resta a las ponderaciones se conoce como tasa de aprendizaje. Un equilibrio adecuado en la **tasa de aprendizaje** es crucial: una tasa demasiado alta puede ocasionar que el algoritmo oscile y no encuentre el mínimo, mientras que una tasa demasiado baja puede ralentizar en exceso el proceso de aprendizaje.

Este procedimiento de hallar el punto mínimo de una función se denomina **método del descenso del gradiente**. Utiliza el cálculo de la derivada de la función de pérdida para determinar hacia dónde "*descender*".

Es importante recordar que, en la práctica, los gráficos de las funciones de pérdida suelen ser intrincados, con numerosos picos y valles, debido al alto número de variables y la complejidad inherente de la red neuronal. Nuestro objetivo es encontrar el punto más bajo entre todos los puntos bajos —**el mínimo global**— evitando confundirlo con otros puntos bajos locales [10].

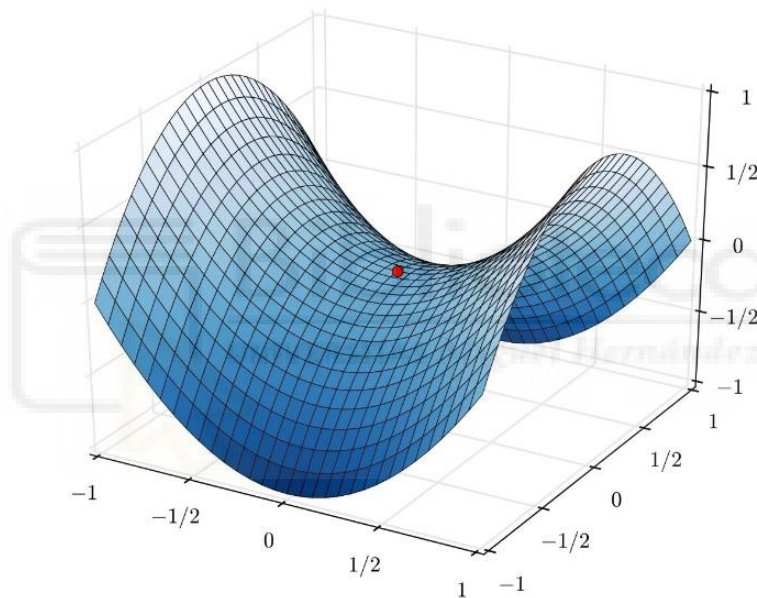


Ilustración 15. Gráfico de una función de pérdida cualquiera

Una estrategia común en estos casos es escoger un punto inicial al azar en la curva y proceder desde allí con el proceso de descenso del gradiente. Una vez que se ha obtenido el vector gradiente, se resta al error tantas veces como sea necesario, hasta que se minimice el error o hasta que se superen un número predefinido de iteraciones durante el entrenamiento. En cada neurona, se ajustan los pesos y el sesgo y restando el valor correspondiente del vector gradiente para minimizar el error.

En conclusión, mediante el proceso de **descenso del gradiente**, combinado con la **propagación hacia atrás**, se busca minimizar el error de salida de cada

neurona al ajustar los valores de entrada basándonos en el vector gradiente obtenido [9].

1.4.9 Optimizadores

Los optimizadores desempeñan un papel crucial en el aprendizaje profundo y aprendizaje automático al trabajar para minimizar la **función de pérdida**, es decir, la discrepancia entre el resultado esperado y el real. Este proceso de optimización incorpora múltiples iteraciones con distintos pesos para hallar el coste mínimo, un principio fundamental en el **descenso del gradiente**. En este caso, el gradiente es un indicador de la dirección de incremento, y en nuestro objetivo de encontrar el punto mínimo, se actualizan parámetros en la dirección opuesta al gradiente.

En secciones previas, hemos establecido que **Backpropagation** se apoya en el **Descenso del Gradiente** para realizar ajustes en los pesos de la red. Este concepto básico, el Descenso del Gradiente (GD), es un optimizador ampliamente utilizado y la piedra angular de algunos otros. Aquí, se realizará un breve repaso de otros optimizadores basados en él.

El Descenso del Gradiente en Lote (Batch Gradient Descent, BGD) es una variante de GD en la que, en cada iteración, se considera el conjunto completo de datos de aprendizaje para calcular el valor del gradiente. A pesar de ser un algoritmo fácil de entender, su convergencia puede ser lenta.

El Descenso del Gradiente Estocástico (Stochastic Gradient Descent, SGD) es lo opuesto: en lugar de utilizar todo el conjunto de datos para calcular el gradiente y actualizar los pesos como lo hace BGD, en este caso, en cada iteración, utilizamos un solo punto de datos, lo que resulta en una convergencia mucho más rápida. Sin embargo, puede presentar ciertas inestabilidades y fluctuaciones.

El Descenso del Gradiente en Mini-lotes (Mini-batch Gradient Descent) es un término medio entre los dos anteriores. En este caso, en cada iteración, se selecciona un conjunto relativamente pequeño de puntos de datos.

Momentum: Este optimizador acelera el descenso de gradiente, especialmente en superficies que tienen una curvatura más pronunciada en una dirección que en otra. Al tener en cuenta el gradiente del paso actual y de los pasos anteriores, se puede avanzar más rápidamente hacia la convergencia, lo que resulta en una convergencia más rápida en superficies curvas.

Nesterov accelerated gradient (NAG): Esta técnica de optimización realiza un cálculo del gradiente no respecto al paso actual, sino al futuro. Esto ha permitido optimizar de manera más eficiente el descenso, y usualmente, NAG muestra un rendimiento ligeramente mejor que el Impulso estándar.

Adagrad (Adaptive Gradient Algorithm): Este optimizador adapta la tasa de aprendizaje a los parámetros, realizando actualizaciones más grandes para parámetros infrecuentes y actualizaciones más pequeñas para los frecuentes. Esta propiedad resulta muy útil cuando se trabaja con datos escasos.

RMSProp: Este optimizador soluciona el problema de Adagrad de la disminución radical de las tasas de aprendizaje mediante el uso de un promedio móvil del gradiente cuadrado. Esto ha permitido que la tasa de aprendizaje se ajuste automáticamente y elija una tasa de aprendizaje diferente para cada parámetro.

Adam (Adaptive Moment Estimation): Este optimizador une las fortalezas de Adagrad y RMSProp, calculando una tasa de aprendizaje adaptativa individual para cada parámetro. A diferencia de Adagrad, Adam implementa el promedio móvil exponencial de los gradientes para escalar, el cual es uno de los optimizadores más populares hoy en día.

Todos estos optimizadores tienen un objetivo común: minimizar la **función de pérdida**. Su propósito es guiar el proceso de aprendizaje hacia un modelo que pueda realizar predicciones más precisas. [16]

1.5 Redes Neuronales Precedentes

En este apartado, se explora la operación y desarrollo de ciertas redes neuronales, que, debido a su estructura y capacidades de aprendizaje, se han implementado ampliamente en el procesamiento del lenguaje natural.

1.5.1 Redes Neuronales Recurrentes (RNN)

Las Redes Neuronales Recurrentes (RNN por sus siglas en inglés), son una clase de arquitectura de aprendizaje profundo, cuyos fundamentos se originan en los trabajos de David Rumelhart en 1986. Las RNN han adquirido reconocimiento por su aptitud para procesar y extraer información de secuencias de datos. Esta particularidad ha permitido su aplicación en diversas áreas, como el análisis de video, la generación de subtítulos para imágenes, el análisis de música y el procesamiento de lenguaje natural[17].

Una característica crucial de las RNN es su habilidad para compartir los **pesos** de los parámetros en la red. En comparación con una red neuronal multicapa tradicional, las RNN demuestran mayor efectividad en el procesamiento de lenguaje natural (PLN). Una red tradicional podría generar interpretaciones lingüísticas basadas en parámetros únicos fijados para cada posición o palabra en un enunciado, lo cual puede ser insuficiente. Por el contrario, las RNN, al compartir pesos entre datos secuencialmente espaciados, han permitido un análisis más contextual y holístico. En este sentido, una RNN procesa la información palabra por palabra, enlazando la salida de una palabra a la entrada de la siguiente, logrando así una comprensión a corto plazo del contexto de la frase.

Para comprender a fondo el funcionamiento de estas redes, es imprescindible familiarizarse con los siguientes conceptos clave:

Neurona recurrente

Hasta ahora, se ha discutido sobre redes neuronales tradicionales en las cuales la función de activación se aplica en una única dirección: hacia adelante, desde

la capa de entrada hasta la capa de salida. Esta estructura carece de la capacidad de recordar valores anteriores. Sin embargo, las **Redes Neuronales Recurrentes (RNN)** han incorporado un aspecto novedoso: presentan conexiones que apuntan hacia atrás, creando una especie de retroalimentación entre las neuronas dentro de las mismas capas

Para simplificar la explicación, se va a considerar una RNN compuesta por una sola neurona. Esta neurona recibe una entrada, produce una salida, e indica mediante una flecha horizontal la dependencia entre la activación actual y la que se generó en un instante anterior. Esto se ilustra a continuación.

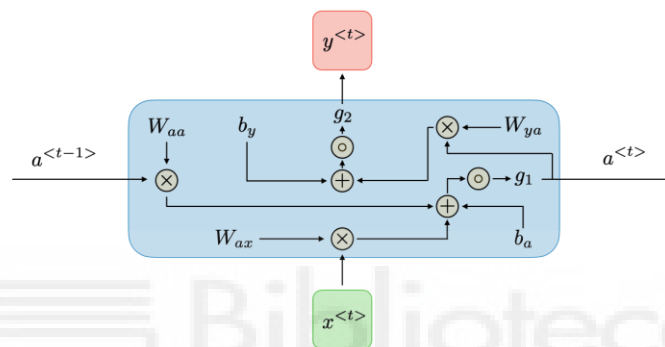


Ilustración 16. Neurona recurrente simple

Para un entendimiento más nítido, se enfoca en la estructura de la RNN, en particular en sus entradas y salidas, omitiendo el proceso matemático que ocurre en su interior. Estos componentes se definen como:

- x^t : es el dato actual entrada.
- a^{t-1} : activación generada en un instante de tiempo anterior
- y^t : resultado o predicción actual
- a^t : activación actual. Esta activación es conocida como *hidden state* o estado oculto.

Sean precisamente estas últimas dos activaciones, las que conforman la "memoria" de la red, las cuales permiten conservar y compartir información entre diferentes instantes de tiempo, habilitando a la RNN para predecir el siguiente carácter en una secuencia.

Funcionamiento matemático de la neurona recurrente

Referenciando la Figura 25, se puede entender el funcionamiento de la neurona recurrente que sigue los principios básicos de una neurona artificial convencional. En ella, a partir de una entrada se genera una salida que es el resultado de aplicar una transformación y una función de activación no lineal (f) al dato de entrada. Los coeficientes W y b se obtienen durante la fase de entrenamiento de la red.

$$y = f(wx + b)$$

Ecuación 8. Función de activación en una neurona artificial convencional

El procedimiento en las RNN es análogo, incorporando el concepto de recurrencia. Para comprender este principio, se especificarán los cálculos asociados a los siguientes procesos:

- **Función de activación:** La operación es similar, transformando los datos de entrada, que consistirán en la activación anterior y la entrada actual, y aplicando una función de activación no lineal para obtener el resultado.

$$a_t = f(W_{aa}a_{t-1} + W_{ax}x_t + b_a)$$

Ecuación 9. Cálculo de la activación en una red neuronal recurrente

- **La salida:** Esta se obtiene utilizando la activación del instante anterior y aplicando las mismas operaciones de transformación y función de activación.

$$y_t = f(W_{ya}a_t + b_y)$$

Ecuación 10. Generación de la salida en una red neuronal recurrente

Como se puede observar, los cálculos realizados son idénticos, pero incorporando la relación entre los datos. Esta dependencia se manifiesta en la salida, la cual depende no solo de la activación actual (a_t), sino también de la entrada actual (x_t) y del valor previo de la activación (a_{t-1}). De esta forma, se implementa la memoria de la RNN [18].

Celda de memoria

Como se analizó en el segmento anterior, la salida de una neurona recurrente en un instante específico es una función de las entradas de instantes anteriores, lo que implica que la neurona recurrente posee una forma de memoria. Este componente el cual permite conservar un estado a lo largo del tiempo es denominado '**celda de memoria**' o '**memory cell**'.

Este principio de memoria es crucial para la utilidad de las RNN en problemas de aprendizaje automático que involucran datos secuenciales. Esta característica les ha permitido recordar información relevante de la entrada previa, mejorando la precisión en la predicción del siguiente elemento y permitiendo conservar el contexto de la información.

Se expondrá el principal desafío asociado con la memoria de una RNN mediante el siguiente ejemplo. Se supone que existe una red neuronal recurrente a la que se ha proporcionado la palabra '*información*' como entrada. La red procesará la entrada carácter por carácter y al llegar a la letra '*c*', debido a la conexión entre las salidas y las entradas de la red, el peso de la primera salida se irá reduciendo, disminuyendo así su importancia. En otras palabras, al llegar a este punto, la red no recordará la primera letra '*i*'. Este fenómeno resulta en un problema considerable de memoria a corto plazo que impide establecer conexiones entre frases completas.

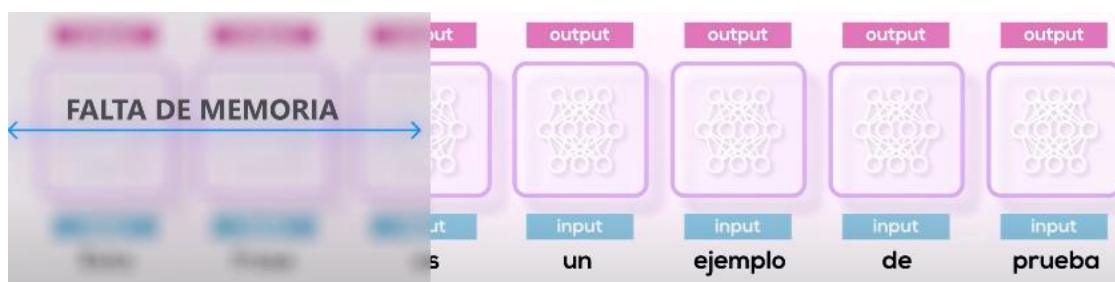


Ilustración 17. Falta de memoria de una RNN



Ilustración 18. Relación entre palabras distanciadadas

Backpropagation a través del tiempo

Una vez comprendida la estructura fundamental de una **red neuronal recurrente** (RNN), es importante explicar cómo dicha red aprende mediante la **retropropagación a través del tiempo**.

Ha de recordarse brevemente el proceso en las **redes neuronales tradicionales**. Este se basa en obtener un resultado al aplicar el modelo y verificar si este resultado es correcto o incorrecto, para obtener así la **función de pérdida** o error en la predicción. Posteriormente se aplica el algoritmo de **retropropagación** (Backpropagation), el cual busca ajustar los pesos del modelo durante su entrenamiento.

En las **RNN**, este proceso se conoce como **Retropropagación a Través del Tiempo** (Backpropagation Through Time, BPTT). En la siguiente imagen se ilustra el concepto de "desenrollar" una RNN, para comprender cómo se aplica este algoritmo incluyendo la dimensión tiempo.

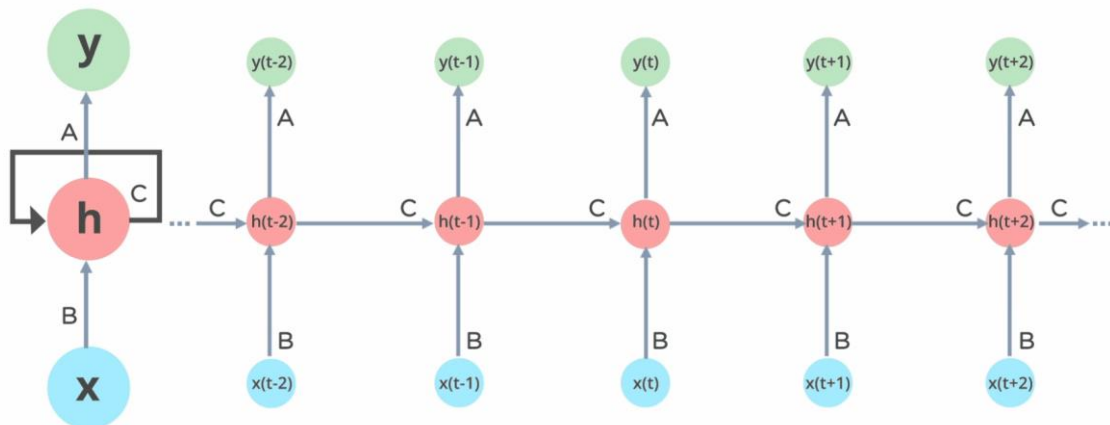


Ilustración 19. Despliegue de una RNN

Al observar el caso simple de una sola neurona, se aprecia la recurrencia con una flecha sobre sí misma, indicando que las RNN son redes con bucles que permiten que la **información persista**. De este modo, se puede considerar una RNN como múltiples copias de la misma red, cada una de las cuales pasa un mensaje a su sucesor.

Si se desarrolla el ciclo, como se muestra en la imagen de la versión desplegada, se observa que no quedan bucles y que la información se pasa de un instante de tiempo a otro. Por lo tanto, una vez que se comprende esta representación de una RNN, se analiza por qué se puede considerar como una secuencia de redes neuronales a las que se puede aplicar el algoritmo de retropropagación.

Al realizar el proceso de **BPTT**, se debe incluir la conceptualización de desenrollar, ya que el cálculo de los **gradientes** de la **función de pérdida** depende no solo de los valores de un determinado instante de tiempo, sino también del instante anterior. Entonces, en BPTT, el error se propaga hacia atrás desde el último hasta el primer instante de tiempo, mientras se van desenrollando todos los instantes de tiempo de la red. Esto permite calcular el error para cada instante de tiempo y actualizar los **pesos** en consecuencia. Sin embargo, este despliegue en el tiempo puede ser de gran tamaño, lo que hace que la aplicación de BPTT sea **computacionalmente costosa**.

Gradiente explosivo y desvanecido

Este último punto de las **Redes Neuronales Recurrentes (RNN)** nos lleva a los principales problemas que enfrentan estas redes, aunque también se aplican a las redes tradicionales. Estos problemas son conocidos como **desvanecimiento del gradiente y gradiente explosivo**, o en inglés, "*vanishing gradients*" y "*exploding gradients*".

Para entrenar una red neuronal, se deben actualizar los parámetros de tal manera que la función de pérdida alcance un mínimo. Los gradientes calculados en la retropropagación nos ayudan a ajustar los parámetros de la red en la dirección y cantidad correctas.

Un **gradiente** es una derivada parcial con respecto a sus entradas, que mide cuánto cambia la salida de una función al ajustar las entradas. En otras palabras, el gradiente indica el cambio a realizar en todos los pesos con respecto al cambio en el error.

Durante este entrenamiento, se pueden producir dos sucesos:

- **Desvanecimiento del gradiente:** Si los gradientes se vuelven demasiado pequeños, los parámetros se actualizarán de forma muy lenta, llegando a detener el aprendizaje o requiriendo demasiado tiempo para este proceso. Esto genera el problema del desvanecimiento del gradiente. Este problema fue de gran importancia en la década de 1990 y se resolvió aplicando "*gate units*" o puertas, concepto que se analizara en el siguiente apartado.
- **Gradiente explosivo:** De manera análoga, si el gradiente asociado a un parámetro se vuelve extremadamente grande, es decir, el algoritmo asigna una importancia excesiva a los pesos, la actualización de los parámetros será también muy grande, generando problemas en el entrenamiento y provocando una inestabilidad en los gradientes que terminará evitando que el algoritmo converja. Esto genera el problema del gradiente explosivo, que se puede resolver truncando o reduciendo los gradientes.

Se puede visualizar estos problemas mediante la representación gráfica de una función, que es una forma de ver el gradiente. El gradiente se puede interpretar como la pendiente de una función en un punto: cuanto más alto es el gradiente, más pronunciada es la pendiente y más rápido será el proceso de aprendizaje del modelo. Sin embargo, si la pendiente es cero, el modelo deberá detener su proceso de aprendizaje.

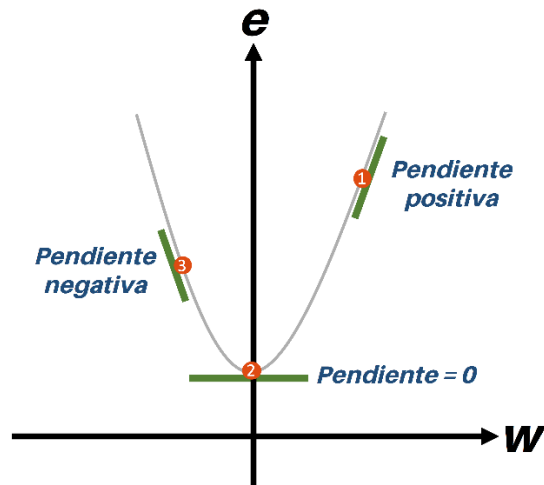


Ilustración 20. Pendientes de una función cualquiera

Para ilustrar, se muestra el desvanecimiento del gradiente y se observa la **función de activación sigmoide**, que limita la salida de la neurona entre 0 y 1. Por lo tanto, un gran cambio en la entrada de la función sigmoide causa un cambio muy pequeño en la salida y, por ende, su derivada será también pequeña.

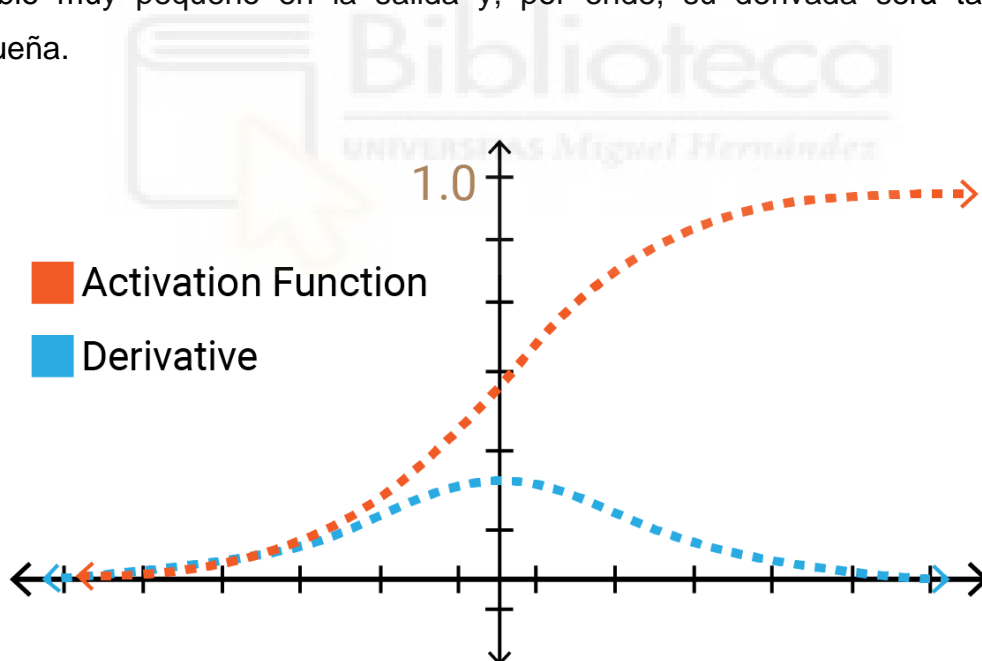


Ilustración 21. Desvanecimiento del gradiente

Las Redes Neuronales Recurrentes (RNN) son efectivas en el procesamiento de datos secuenciales gracias a su "memoria" y su habilidad para compartir pesos. No obstante, enfrentan desafíos como los problemas de gradientes que desvanecen y explotan, lo que dificulta su aprendizaje. Para abordar estas

limitaciones, se crearon variantes de RNN, entre las que destaca la Long-Short Term Memory (LSTM).

1.5.2 Long-Short Term Memory (LSTM)

Dado el problema de desvanecimiento del gradiente, las redes neuronales recurrentes (RNN) se ven limitadas en su capacidad para captar dependencias necesarias entre palabras distantes en un texto, lo que implica que no pueden comprender completamente su contexto.

Un ejemplo que considerar sería la frase: "*Crecí en Francia... hablo francés fluido*". En este escenario, predecir la palabra "*Francia*" no sería sencillo para una RNN. Lógicamente, la próxima palabra más probable sería un idioma, pero sería necesario del contexto "*Francia*", que se ubica al inicio del texto. Cuando se genera una brecha considerable entre la información relevante y el punto donde se necesita, las RNN se tornan incapaces de aprender a conectar dicha información.

Arquitectura

La solución a este problema reside en las redes **LSTM (Long Short-Term Memory)**, que son una evolución de las RNN, diseñadas para el aprendizaje a largo plazo. Fueron introducidas por Hochreiter & Schmidhuber en 1997, y posteriormente fueron refinadas y popularizadas por numerosos investigadores debido a su efectividad en una amplia gama de problemas[19].

Estas redes LSTM están específicamente diseñadas para abordar el problema de la **dependencia a largo plazo**, es decir, su comportamiento predeterminado consiste en recordar información durante largos periodos de tiempo. Su objetivo es mitigar los problemas de desvanecimiento y explosión del gradiente que pueden surgir en las RNN.

Como vimos anteriormente, las redes neuronales recurrentes tienen una estructura en forma de cadena de módulos o celdas repetidas de la red neuronal. A continuación, a través de las ilustraciones, se examinará la diferencia entre la

estructura de una RNN y una LSTM. La principal diferencia reside en la estructura del módulo y la incorporación de un nuevo elemento.

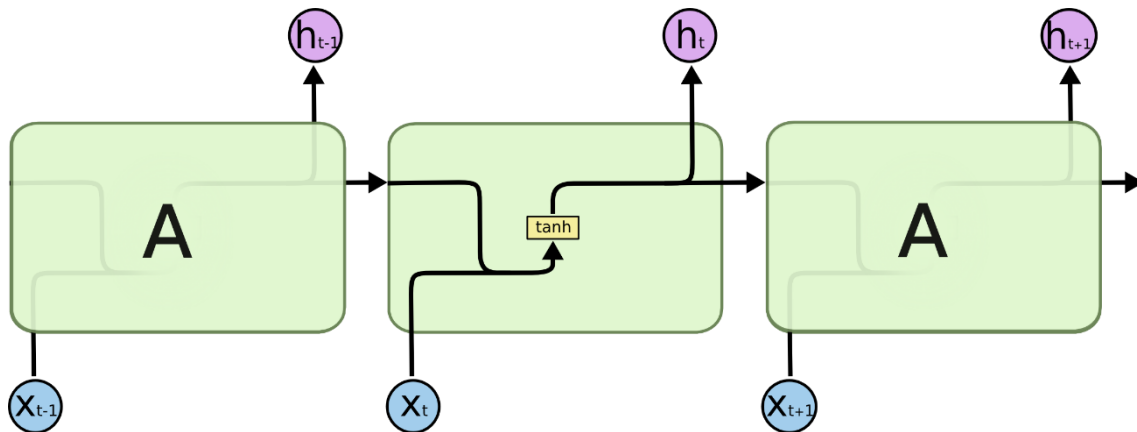


Ilustración 22. Estructura de una RNN

En la **arquitectura LSTM**, la estructura de la celda cambia. Ahora, en lugar de tener una sola capa, consta de cuatro capas que interactúan entre sí. Técnicamente, la arquitectura de esta red ahora comprende tres puertas o "gates", que se conectan directamente con el nuevo elemento, la **celda LSTM**. Esta celda tiene la capacidad de regular la información que entra o sale, y de recordar dicha información en intervalos de tiempo específicos.

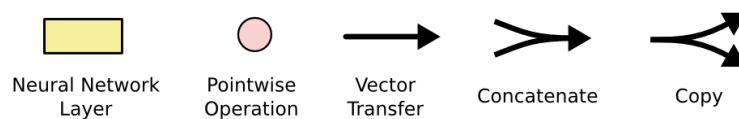
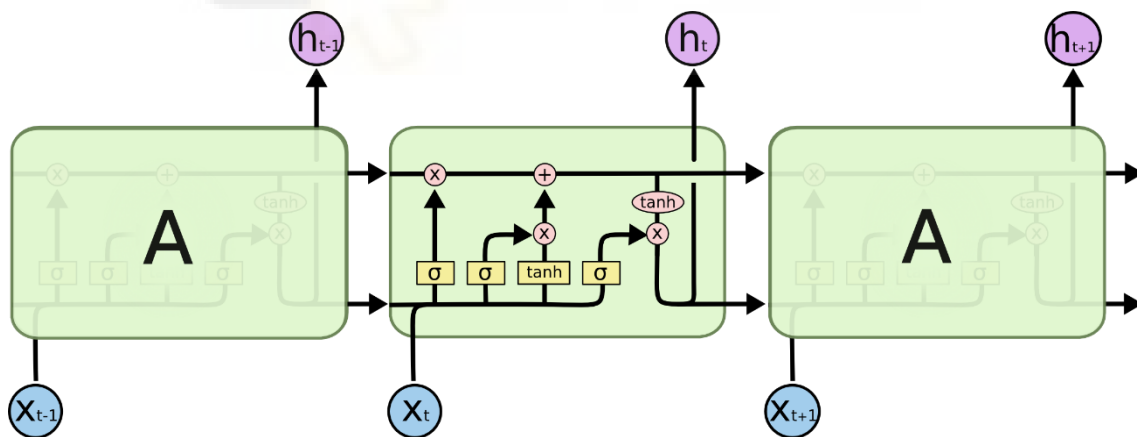


Ilustración 23. Estructura de una LSTM

Una vez que se ha comprendido cómo se ha modificado la RNN para convertirse en una LSTM, se puede profundizar en el desarrollo del procesamiento de las LSTM. A continuación, se detallarán las partes fundamentales mencionadas anteriormente:

Celda LSTM

Esta es la componente central de una red LSTM. Representada por la nueva línea horizontal en el diagrama, el estado de la celda opera como una cinta transportadora, desplazándose a lo largo de toda la cadena. Su función es proteger y controlar el estado de la celda.

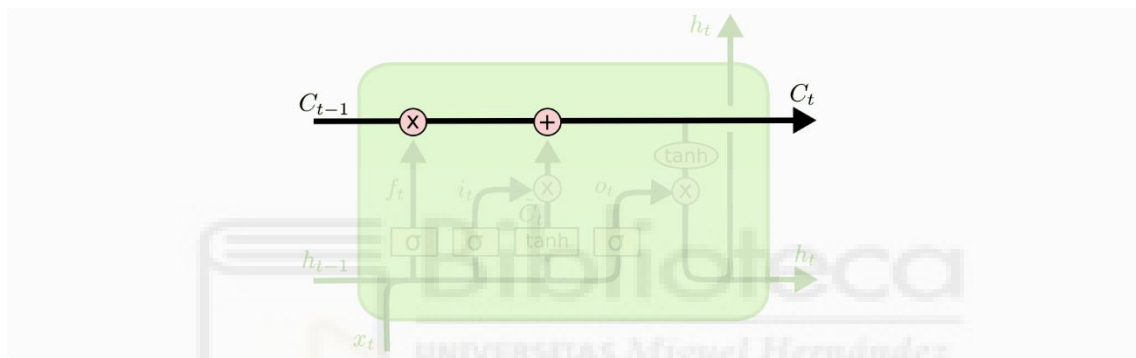


Ilustración 24. Celda LSTM

Por medio de las puertas específicas, la LSTM tiene la habilidad de agregar o eliminar información que no es relevante para la memoria de la red. Estas puertas son, esencialmente, mecanismos que regulan el flujo de información. Funcionan mediante una capa con la **función de activación sigmoidea**, la cual produce en su salida los valores 0 o 1. Esto se puede resumir así:

- **0:** Denota que la información no será transmitida al estado de la celda.
- **1:** Indica que la información será transmitida al estado de la celda.

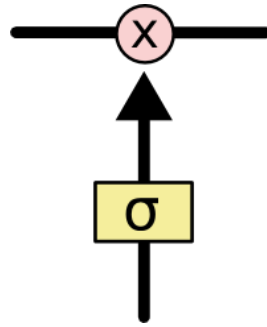


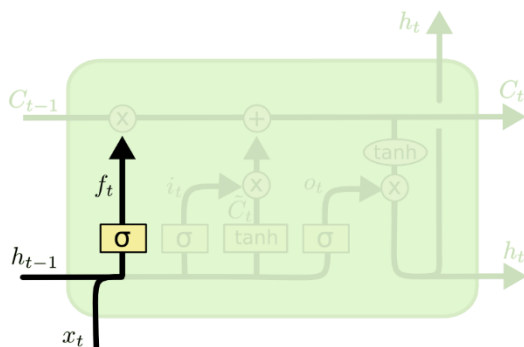
Ilustración 25. Acceso a celda LSTM

Forget gate o puerta de olvido

Esta puerta es el punto de partida en el recorrido de la LSTM. Su principal objetivo es determinar qué elementos deben ser eliminados de la memoria, es decir, qué información se descartará y, por consiguiente, no será transmitida al estado de la celda. Esta función permite que las celdas LSTM sean capaces de diferenciar entre lo que es relevante y lo que no lo es.

La decisión se toma a través de la función de activación sigmoidea, mencionada anteriormente, que proporciona los valores 0 o 1. El valor 0 indica que se debe olvidar toda la información anterior, mientras que el valor 1 sugiere que todo el estado de la memoria anterior debe ser preservado.

Una vez obtenido el valor de la puerta, este debe ser multiplicado por los valores del estado de la memoria de la celda anterior. De esta forma, se conserva la información más importante de los estados previos.



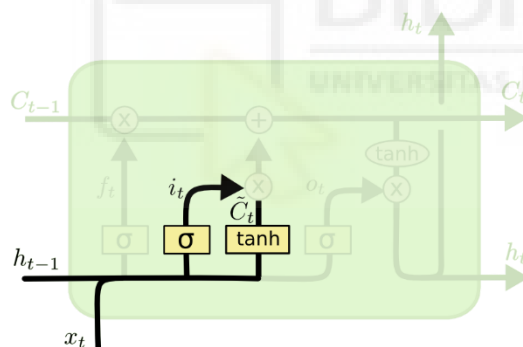
$$f_t = \sigma(W_f \cdot [h_{t-1}, x_t] + b_f)$$

Ilustración 26. Forget gate

Input gate o puerta de entrada

La función de esta puerta es decidir qué nueva **información** se añadirá al **estado de la celda**, identificando así las nuevas dependencias a retener. Este proceso se divide en tres partes:

1. En primer lugar, la información se somete a una **función de activación sigmoidea**, que decidirá qué valores serán actualizados. La salida de esta función nos proporcionará un 0, indicando que las entradas no son significativas para ser recordadas, y un 1, que señala que la información de entrada debe ser almacenada en la **memoria**. Esto se calcula como i_t
2. En esta fase, la **función de activación tangente hiperbólica** (tanh) desempeña un papel crucial. Esta función permite crear un vector de nuevos valores candidatos que podrían actualizar el estado de la celda. Su objetivo es detectar la información más relevante que la celda puede actualizar, con valores de salida que oscilan entre -1 y 1. Este cálculo se expresa en C_t

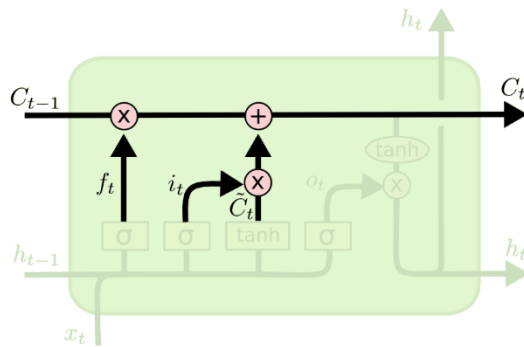


$$i_t = \sigma(W_i \cdot [h_{t-1}, x_t] + b_i)$$

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

Ilustración 27. Procesamiento de datos en Input gate

3. Por último, se debe actualizar el **estado de la celda anterior (C_{t-1}) al nuevo estado (C_t)**. Los valores de salida obtenidos previamente de las funciones sigmoidea y tangente hiperbólica se multiplican elemento por elemento. Esto indica la relevancia de los elementos obtenidos en la función tanh, dada la salida de la función sigmoidea. Finalmente, la salida generada se suma al vector de memoria, obtenido de la **forget gate**, permitiendo así actualizar la nueva información en el estado de la celda LSTM, representado como C_t



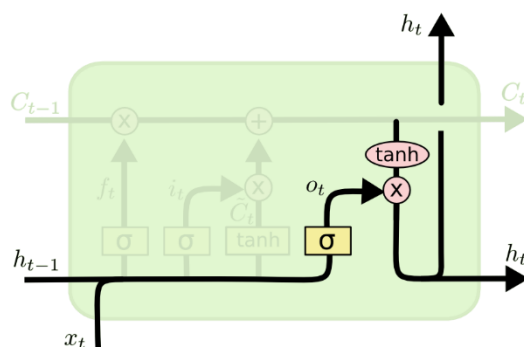
$$C_t = f_t * C_{t-1} + i_t * \tilde{C}_t$$

Ilustración 28. Actualización de celda LSTM mediante el input gate

Output gate o puerta de salida

En esta última puerta, se decide qué **salida** se generará. Para un mejor entendimiento, se dividirá en tres pasos:

1. La puerta de salida recibe el **input** de la celda. A estos valores se les aplica la **función de activación sigmoidea**, que indica qué partes del estado de la celda se van a generar. Esto se expresa como o_t
2. Una vez que el **estado de la celda** ha sido modificado mediante las puertas de olvido (forget) e entrada (input), para determinar qué información debe ser olvidada y cuál debe formar parte de la memoria, estos valores se someten a la **función de activación tangh**.
3. Finalmente, los valores de salida obtenidos en el paso anterior se multiplican elemento por elemento a los valores de salida de la función sigmoidea del punto 1. De este modo, solo se seleccionan las partes que se eligieron previamente y se genera la salida definitiva de esta celda LSTM. Cabe recordar que, debido a la recurrencia de estas redes, esta salida será el **estado oculto** para el siguiente instante de tiempo. Esto se representa como h_t



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

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

Ilustración 29. Output gate

Entrenamiento

El entrenamiento de este tipo de redes se realiza utilizando el mismo algoritmo, *backpropagation*, con la diferencia de que ahora se cuentan con tres puertas en cada celda LSTM. Por consiguiente, el número de parámetros a actualizar es mayor, pero el proceso es similar. Por lo tanto, no se repetirá la explicación en detalle. Es importante resaltar que gracias al uso de las puertas se solucionan los problemas de gradientes que se presentaban con las RNN, específicamente, el llamado **desvanecimiento del gradiente o gradiente explosivo** [20].

1.5.3 Problema principal: falta de memoria

Las Redes Neuronales Recurrentes (RNN) y la Memoria a Largo Plazo (LSTM), a pesar de haber revolucionado el manejo de secuencias temporales, se enfrenta a un desafío importante: la "falta de memoria". Este problema radica en su dificultad para procesar información relevante en secuencias largas, debido a su naturaleza secuencial. Aunque LSTM intentó abordar esto con su estructura de puertas, las secuencias de gran longitud seguían siendo un desafío.

Los **Transformers**, los cuales llegaron después resolviendo efectivamente este problema de memoria. Al cambiar el enfoque de procesamiento secuencial a paralelo, los Transformers pueden capturar dependencias a largo plazo sin necesidad de procesar secuencias paso a paso. Con la adición de la **atención multi-cabeza**, la red puede manejar simultáneamente diferentes características de la secuencia, superando las limitaciones de memoria de las RNN y LSTM. En resumen, los Transformers han proporcionado una solución más eficiente y efectiva para el manejo de secuencias largas de datos [21].

En este apartado, se han analizado las arquitecturas más importantes para el **procesamiento del lenguaje natural** y su evolución. Sin embargo, como se menciona anteriormente, el campo del PLN avanza rápidamente y estas redes también evolucionan, desarrollando métodos innovadores de manera constante. Este progreso continuo nos lleva a la red que se expondrá en detalle en el apartado de la metodología: los **Transformers**.

CAPÍTULO 2: ESTADO DE LA CUESTIÓN

2.1 Análisis del PLN

El **Procesamiento de Lenguaje Natural (PLN)**, también conocido en inglés como Natural Language Processing, es un desafío complejo debido a la ambigüedad y falta de precisión inherente al lenguaje humano durante la interpretación de textos. La comunicación humana depende en gran medida del contexto, como el tema y el medio de comunicación, y la interpretación varía en función de la experiencia, las expectativas y las creencias del receptor.

Los **modelos de PLN tradicionales** se basan en reglas explícitas que a menudo requieren la intervención de expertos para implementar estos conocimientos en forma de bases de datos y reglas. La interacción entre estas reglas puede volverse compleja cuando se intenta aplicar múltiples reglas a una situación específica, necesitando establecer protocolos de aplicación o un orden de prioridad.

Las **tecnologías actuales de PLN** han superado en muchos aspectos a los modelos explícitos tradicionales mencionados anteriormente. Sin embargo, hay casos en los que estos pueden ser los únicos disponibles, especialmente cuando no hay suficientes datos para entrenar un sistema, como en el caso de lenguajes menos comunes.

Los sistemas de mayor importancia y más extendidos actualmente son los **modelos de aprendizaje profundo** (Deep Learning). Estos modelos son implícitos, ya que no dependen de reglas programadas por expertos. Sin embargo, tienen la desventaja de ser poco interpretables, especialmente cuando se escalan a grandes dimensiones. Cuando un modelo de aprendizaje profundo genera una salida deseada, puede ser difícil entender cómo ha llegado a esa conclusión. Del mismo modo, cuando la salida es incorrecta, resulta desafiante identificar la causa del error. Estos sistemas se basan en redes neuronales que aprenden automáticamente, mediante entrenamiento con grandes cantidades de ejemplos etiquetados.

2.2 Utilidad actual del PLN

El **Procesamiento de Lenguaje Natural (PLN)** tiene una variedad de aplicaciones en el mundo real que son diversas y extensas. Las más comunes incluyen:

- **Resumen de textos:** Los algoritmos de PLN pueden identificar la idea principal en un texto dado e ignorar la información que no sea relevante. Esto ayuda en la rápida comprensión de documentos largos.
- **Chatbots:** Son capaces de mantener una conversación fluida con los usuarios y responder a sus preguntas de manera automática. Esta aplicación de PLN ha revolucionado la atención al cliente y la interacción del usuario.
- **Generación automática de palabras clave y generación de textos:** Los algoritmos de PLN pueden producir texto con sentido y lógica, y también generar palabras clave relevantes para el contenido.
- **Reconocimiento de entidades:** Los algoritmos de PLN pueden identificar y etiquetar entidades en un texto, como personas, entidades comerciales, países, ciudades y marcas.
- **Análisis de sentimientos:** Esta es una aplicación de PLN que comprende el contexto de un texto para determinar si tiene una connotación positiva o negativa. Se utiliza con frecuencia en la comprensión de tweets o reseñas, y es muy utilizado en redes sociales, política, opiniones de productos y motores de recomendación.
- **Traducción automática de idiomas:** El PLN permite la traducción automática entre idiomas, facilitando la comunicación entre personas que hablan diferentes lenguas.
- **Clasificación automática de textos:** A partir de textos completos, los algoritmos de PLN pueden realizar la clasificación para detectar temas recurrentes y crear categorías. Esta aplicación es útil para la organización de grandes volúmenes de información.

2.3 Modelos más innovadores y conocidos

El campo del **Procesamiento del Lenguaje Natural (PLN)** ha experimentado una revolución y un rápido desarrollo, dando lugar a modelos cada vez más grandes y potentes. Estas arquitecturas se han entrenado con grandes volúmenes de texto de manera no supervisada, permitiéndoles realizar tareas como completar oraciones "cortadas" o "incompletas". Los modelos más destacados incluyen:

ELMO:

ELMo (Embeddings from Language Models) representa un hito en el procesamiento del lenguaje natural (PLN). Este modelo, desarrollado por el Allen Institute for Artificial Intelligence, utiliza una arquitectura LSTM bidireccional para aprender embeddings de palabras profundamente contextualizadas. A diferencia de los embeddings estáticos tradicionales como Word2Vec o GloVe, ELMo ofrece embeddings dinámicos, adaptándose al contexto de la palabra y captando así significados sutiles y diversas connotaciones.

En su entrenamiento, ELMo se basa en el modelado de lenguaje, cuyo objetivo es predecir la siguiente palabra en una secuencia considerando las palabras anteriores. Esta estrategia permite al modelo aprender relaciones relevantes entre palabras y comprender las estructuras gramaticales y sintácticas del lenguaje [22].

A pesar de que otros modelos como ULMFiT y GPT han igualado o superado a ELMo en ciertas tareas, estos comparten con ELMo la utilización del modelado del lenguaje como tarea de aprendizaje no supervisado en la etapa previa al entrenamiento. De esta forma, ELMo se destaca como un precursor de muchas de las técnicas actuales de aprendizaje profundo en PLN.

BERT:

BERT (Representaciones de codificador bidireccional de transformadores) es un avance en la técnica de preentrenamiento para el procesamiento del lenguaje natural (NLP) desarrollada por Google. Este modelo es preentrenado en un gran

corpus de texto sin etiquetas y luego se ajusta para tareas específicas, mejorando la precisión en comparación con el entrenamiento desde cero.

Lo que distingue a BERT es su capacidad para aprender representaciones de lenguaje profundamente bidireccionales. A diferencia de modelos anteriores que solo tenían en cuenta el contexto unidireccional de las palabras, BERT considera el contexto antes y después de cada palabra, lo cual le permite comprender mejor el significado de las palabras en función del contexto. Esta bidireccionalidad se logra mediante el enmascaramiento de algunas palabras en la entrada y luego condicionando cada palabra para predecir las palabras enmascaradas.

BERT también aprende a modelar las relaciones entre oraciones al entrenarse en una tarea simple: determinar si una oración dada es la siguiente oración real en el corpus después de una oración inicial. Además, BERT ha demostrado resultados excepcionales en 11 tareas de NLP, incluyendo la competitiva tarea de respuesta a preguntas del conjunto de datos de Stanford (SQuAD v1.1), superando incluso el rendimiento humano. *Miguel Hernández*

La utilización de Cloud TPU ha sido esencial para lograr el rendimiento de BERT, proporcionando la capacidad de experimentar, depurar y ajustar modelos rápidamente. BERT se basa en la arquitectura del modelo Transformer, lo que ha facilitado su éxito. Por último, el modelo BERT está disponible para el público, permitiendo a los investigadores ajustarlo para diversas tareas de NLP en unas pocas horas o menos [23].

T5

El Transformador de Transferencia de Texto a Texto (T5) es un modelo de procesamiento de lenguaje natural desarrollado por Google que utiliza aprendizaje por transferencia. T5 se preentrena en un conjunto de datos de texto masivo sin etiquetar llamado Colossal Clean Crawled Corpus (C4) y luego se perfecciona con datos etiquetados para tareas específicas.

El enfoque innovador de T5 es su marco "texto a texto", donde todas las tareas de PNL son reformuladas para que tanto la entrada como la salida sean siempre cadenas de texto. Esto permite la utilización del mismo modelo y parámetros para una variedad de tareas de PNL, incluyendo traducción automática, resumen de documentos, respuesta a preguntas y tareas de clasificación.

T5 también puede trabajar en un entorno de "libro cerrado", es decir, se basa únicamente en el conocimiento incorporado durante su preentrenamiento para responder a las preguntas sin recurrir a información externa. En su configuración más grande, el modelo T5 posee 11 mil millones de parámetros y ha logrado resultados de vanguardia en una serie de benchmarks de PNL.

En resumen, T5, con sus 11 mil millones de parámetros, es un modelo de PNL altamente versátil y poderoso que ha establecido un nuevo estándar en el campo del procesamiento del lenguaje natural [24].

GPT-4

OpenAI ha presentado GPT-4, un avance significativo en el procesamiento del lenguaje natural y la generación de texto. Utilizando una arquitectura Transformer preentrenada, GPT-4 se destaca por su versatilidad, capacidad para manejar una gama más amplia de tareas y por su mejora en el rendimiento en pruebas académicas simuladas.

En comparación con su predecesor, GPT-4 puede manejar un contexto de entrada de hasta 32,000 palabras de texto, proporcionando un mayor rango de contexto. Sin embargo, como sus predecesores, GPT-4 puede carecer de precisión y cometer errores de razonamiento, y no tiene conocimiento de eventos ocurridos después de septiembre de 2021 [25].

El potencial de GPT-4 para diversas aplicaciones es considerable. Desde la traducción de idiomas hasta la respuesta a preguntas, pasando por la generación de imágenes y videos, el modelo podría ser de gran utilidad en una amplia gama de tareas de procesamiento de lenguaje natural.

Es importante destacar que GPT-4 no es simplemente una versión más grande de GPT-3. OpenAI ha reorientado su enfoque, pasando de la construcción de modelos más grandes a la maximización del rendimiento de modelos más pequeños, centrándose en otros aspectos como datos, algoritmos, parametrización y alineación para obtener beneficios más rápidamente [26].

LaMDA

Google ha desarrollado LaMDA, su último avance en investigación, para mejorar la comprensión y generación del lenguaje natural en conversaciones. Diferente a los chatbots convencionales, que siguen caminos predefinidos y limitados, LaMDA (abreviatura de "Language Model for Dialogue Applications") puede dialogar de manera fluida sobre una gran cantidad de temas, promoviendo formas más naturales de interactuar con la tecnología.

LaMDA, al igual que modelos de lenguaje recientes como BERT y GPT-3, está construido sobre la arquitectura Transformer, que Google Research inventó y liberó en código abierto en 2017. Sin embargo, a diferencia de la mayoría de los modelos de lenguaje, LaMDA fue entrenado en diálogo, permitiéndole captar las sutilezas que diferencian la conversación abierta de otras formas de lenguaje[27].

En términos de rendimiento, LaMDA se centra en la sensibilidad y especificidad de las respuestas. La sensibilidad implica que las respuestas tengan sentido en el contexto de la conversación, mientras que la especificidad se refiere a la relevancia directa de la respuesta con respecto al contexto de la conversación. Además, Google está explorando otras dimensiones como la "interesante", evaluando si las respuestas son perspicaces, inesperadas o ingeniosas[28].

PALM

Google Research ha desarrollado un nuevo modelo de lenguaje, Pathways Language Model (PaLM), con 540 mil millones de parámetros. Entrenado con el sistema Pathways, que permite la eficiencia y escalabilidad de la computación

distribuida, este modelo de lenguaje demuestra un rendimiento líder en el aprendizaje de pocos disparos en una multitud de tareas de generación y comprensión del lenguaje[29].

PaLM se ha entrenado a escala masiva utilizando una combinación de conjuntos de datos en inglés y multilingües de diversas fuentes, incluyendo documentos web, libros, Wikipedia, conversaciones y código de GitHub. Este modelo de lenguaje no sólo ha mejorado el rendimiento en tareas de procesamiento de lenguaje natural, sino que también ha mostrado capacidades impresionantes en tareas de razonamiento y generación de código.

Al analizar PaLM en términos de consideraciones éticas, los investigadores de Google destacan la importancia de evaluar los posibles riesgos asociados con los modelos de lenguaje a gran escala entrenados en textos de la web. Hacen hincapié en la necesidad de un análisis exhaustivo de los conjuntos de datos y los resultados del modelo para identificar y mitigar sesgos y riesgos potenciales.

Finalmente, PaLM representa un hito importante hacia la visión de Pathways de Google de desarrollar un sistema de inteligencia artificial único capaz de generalizar miles o millones de tareas, entender diferentes tipos de datos y hacerlo con una eficiencia notable. Esto sienta las bases para la creación de modelos aún más capaces que combinan capacidades de escalado con nuevas opciones arquitectónicas y esquemas de capacitación[30].

BLOOM (BigScience Large Open-science Open-access Multilingual Language Model)

BLOOM es un modelo de lenguaje multilingüe con 176 mil millones de parámetros, desarrollado por Hugging Face en colaboración con más de 1000 científicos. Se destaca por su capacidad para realizar tareas complejas como aritmética, traducción y programación con alta precisión, pudiendo generar respuestas adecuadas y coherentes en base a entradas específicas. A pesar de su tamaño (un punto de control ocupa 330 GB), puede ejecutarse en una

computadora personal con al menos 16 GB de RAM y espacio en disco suficiente.

El modelo utiliza una arquitectura Transformer compuesta por una capa de incrustaciones de entrada, 70 bloques Transformer y una capa de modelado de lenguaje de salida. BLOOM se entrena para predecir el siguiente token en una oración basándose en los tokens anteriores, lo que le permite captar ciertas habilidades de razonamiento y conectar conceptos de forma coherente. Para predecir el siguiente token en una oración, los tokens de entrada se pasan a través de los 70 bloques de BLOOM, pudiendo cargar en RAM un bloque a la vez para evitar el desbordamiento de la memoria[31].

BLOOM ha sido reconocido como un hito importante en el campo de la Inteligencia Artificial debido a su carácter multilingüe y de acceso abierto. Aunque la versión completa de BLOOM puede ser demasiado grande para ciertos usos o recursos computacionales, existen versiones más pequeñas disponibles en el repositorio de modelos de Hugging Face [32].

PaLM 2

Google ha presentado **PaLM 2**, la nueva generación de su modelo de lenguaje, caracterizado por sus capacidades mejoradas en **multilingüe, razonamiento y codificación** [33].

PaLM 2 se distingue por ser más competente en el procesamiento de texto multilingüe, abarcando más de **100 idiomas** y demostrando una mejor capacidad para comprender, generar y traducir texto en diversos lenguajes, incluso en cuestiones complejas como modismos, poemas y acertijos.

Además, el modelo incluye en su conjunto de datos **artículos científicos** y **páginas web** con expresiones matemáticas, lo cual ha potenciado sus habilidades en **lógica, razonamiento de sentido común y matemáticas**.

En términos de codificación, PaLM 2 ha sido pre-entrenado en numerosos conjuntos de datos de **código fuente** disponibles públicamente, lo cual lo hace eficiente en lenguajes de programación populares como **Python y JavaScript**, pero también capaz de generar código en lenguajes más especializados como **Prolog, Fortran y Verilog**.

PaLM 2, a pesar de ser más competente que los modelos anteriores, destaca por su **rapidez** y **eficiencia**. Se encuentra disponible en una variedad de tamaños, desde **Gecko**, que está diseñado para funcionar en dispositivos móviles, hasta **Unicornio**.

Paralelamente, la innovación continúa con el desarrollo de **Gemini**, un modelo en proceso de creación desde cero para ser **multimodal** y **altamente eficiente**, y con el objetivo de incorporar futuras innovaciones, como la **memoria** y la **planificación**. Aunque Gemini aún se encuentra en entrenamiento, muestra prometedores signos de capacidades multimodales sin precedentes[34].

Es importante destacar que estos modelos están basados en Transformers, una arquitectura que se encuentra en constante desarrollo y expansión.

2.4 Modelos de generación de lenguaje

Los modelos de lenguaje son sistemas de inteligencia artificial diseñados para entender, generar y manipular el lenguaje humano de forma coherente y comprensible. Estos son algunos de los modelos de lenguaje más avanzados hasta la fecha:

2.4.1 ChatGPT

ChatGPT, desarrollado por **OpenAI**, es una aplicación de inteligencia artificial que se basa en la arquitectura avanzada de GPT-4. Esta aplicación, especializada en la interacción en lenguaje natural, facilita una amplia gama de funciones. ChatGPT puede explicar conceptos, generar guiones para diversas

plataformas de medios sociales, y proporcionar contenido en una variedad de idiomas, incluyendo la realización de traducciones. Gracias a su sofisticado algoritmo, puede adaptarse a diferentes estilos de escritura, desde dialectos regionales hasta tonos humorísticos o profesionales[35].

ChatGPT, utilizando **GPT-4**, trasciende la mera generación de texto y ofrece características que expanden enormemente su utilidad. Puede generar contenidos diversos y creativos, como adivinanzas y chistes, y hasta cuestionarios educativos y fichas de especificaciones. Además, ahora puede interactuar con imágenes, permitiendo la identificación y comunicación a través de ellas. Esto hace de ChatGPT una herramienta educativa valiosa que facilita el aprendizaje a un ritmo acelerado.

Para experimentar el potencial de GPT-4 a través de ChatGPT, los usuarios pueden suscribirse a ChatGPT Plus, la versión premium de la aplicación. Mientras tanto, en la versión gratuita de ChatGPT, los usuarios interactúan con GPT-3.5, la versión anterior del modelo[36].

En conclusión, ChatGPT se ha convertido en una herramienta esencial en la era digital actual. Su capacidad para entender, generar y traducir textos en varios idiomas, así como su habilidad para interactuar con imágenes, la hace única y altamente valiosa. A medida que OpenAI continúa mejorando y ampliando sus capacidades, ChatGPT se consolida como un recurso imprescindible para diversas aplicaciones, desde la educación y el entretenimiento hasta la creación de contenidos y más allá.



Ilustración 30. ChatGPT

2.4.2 Bing Chat

Bing Chat es una función de inteligencia artificial incorporada en el motor de búsqueda Bing, impulsada por GPT-4 de OpenAI, el mismo núcleo tecnológico de ChatGPT. Aunque ambos comparten la misma base, Bing Chat se distingue por su integración con Bing, ofreciendo respuestas detalladas y validadas a las consultas de los usuarios.

Además de responder a preguntas, Bing Chat tiene capacidades creativas, permitiendo escribir poemas, historias, canciones y más. Su funcionalidad se extiende más allá del texto, pudiendo generar imágenes a partir de texto con Image Creator de Bing.

En esencia, Bing Chat amplía las posibilidades de la interacción basada en IA, incorporándola de manera significativa en las plataformas de búsqueda en línea. Aunque comparte similitudes con ChatGPT, Bing Chat ofrece una experiencia única centrada en la verificación de información y la generación de contenido creativo, demostrando la versatilidad y el potencial de la tecnología GPT-4[37].

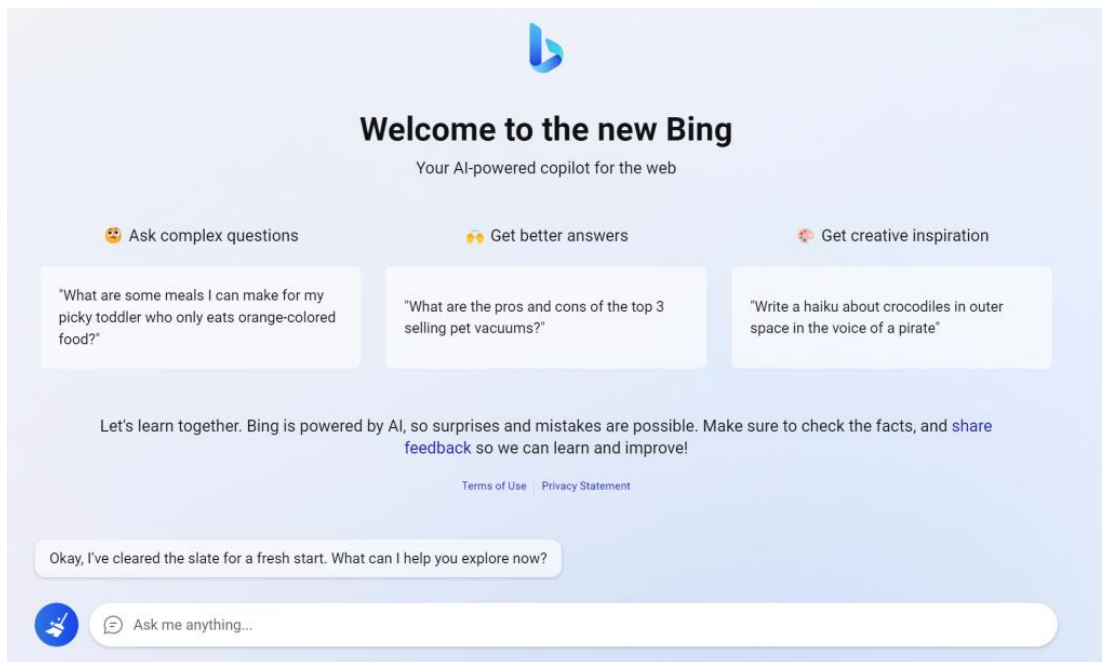


Ilustración 31. Bing Chat

2.4.3 BARD

Bard es un servicio de conversación asistido por inteligencia artificial que ha sido desarrollado por Google. Su funcionamiento se encuentra respaldado por los modelos de inteligencia artificial de Google conocidos como **LaMDA** y **PaLM 2**.

LaMDA es un modelo de lenguaje por inteligencia artificial diseñado para procesar y comprender el lenguaje humano de manera coherente y natural. Sin embargo, la novedad más reciente es la incorporación de **PaLM 2**, una versión mejorada y más potente del anterior modelo de lenguaje de Google[38].

PaLM 2 es la segunda generación del **Pathways Language Model de Google**. Fue presentado el 10 de mayo de 2023 durante el evento Google I/O y se encuentra destinado a ser utilizado en Google Bard. En términos de rendimiento, Google asegura que PaLM 2 tiene una capacidad tres veces superior a GPT-3, por lo que se espera que pueda competir directamente con **GPT-4 de OpenAI**.

Con PaLM 2, Bard se beneficia de un modelo de inteligencia artificial aún más avanzado y eficiente. Bard puede proporcionar respuestas de alta calidad basadas en la amplia información recopilada de la web. Además, se puede

utilizar para simplificar temas complejos y fomentar la creatividad, proporcionando explicaciones sobre una variedad de temas desde descubrimientos científicos hasta las mejores estrategias en deportes.

El objetivo de Google con Bard y la implementación de PaLM 2 es proporcionar una experiencia de conversación impulsada por IA de mayor calidad y rendimiento, capaz de competir con sistemas avanzados como ChatGPT y ChatGPT Plus de OpenAI [39].



Ilustración 32. Bard

CAPÍTULO 3: OBJETIVOS



El objetivo principal de mi trabajo de fin de grado se puede dividir en los siguientes apartados:

3.1 Objetivos Generales

Este estudio tiene como principal propósito realizar un **análisis detallado sobre la Inteligencia Artificial (IA)**, con un enfoque particular en las técnicas de **Machine Learning y Deep Learning** que han demostrado ser de gran relevancia en los últimos años. Estas técnicas actuarán como hilo conductor hacia nuestra exploración en el campo del **Procesamiento del Lenguaje Natural (PLN)**.

Dentro del PLN, se presenta una secuencia de procesamiento de texto, conocida como **Pipeline de PLN**. Este pipeline representa la serie de pasos involucrados en la construcción de cualquier modelo de PLN y será la guía estructural de nuestro estudio. Cabe mencionar que solo algunos de estos pasos serán analizados en profundidad.

El cúmulo de conocimiento previo constituirá la base para comprender con mayor claridad el objetivo central de este trabajo, el cual es el estudio de la Red Neuronal **Transformer**. Esta arquitectura, presentada en el influyente paper "**Attention is All You Need**" en 2017, ha experimentado significativas mejoras y ha sido de gran impacto en el campo del PLN[40].

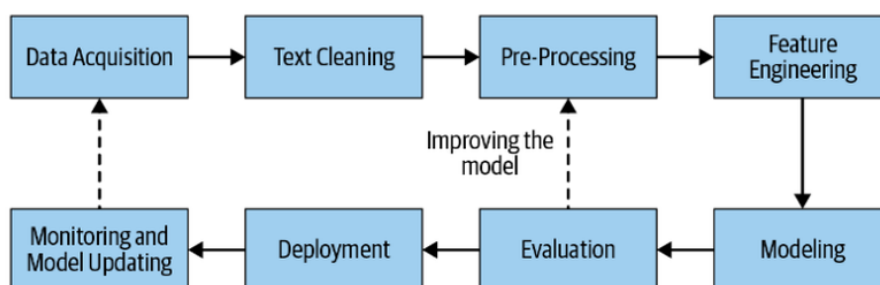


Ilustración 33. Canalización de PLN

3.2 Objetivos Técnicos

El foco principal de los objetivos técnicos es el desarrollo de modelos basados en diferentes tipos de **redes neuronales**. Este enfoque nos permitirá centrarnos

en la creación de un modelo básico, basado en la arquitectura Transformer, capaz de abordar algunos de los desafíos más comunes en PLN. Algunos de estos desafíos pueden incluir la **clasificación de texto, extracción de información, respuesta a preguntas, traducción automática, comprensión del lenguaje o generación de texto.**

3.3 Objetivos Didácticos

El desarrollo del proyecto se llevará a cabo utilizando la plataforma **Google Colab**, con la ayuda del lenguaje de programación **Python**. Se emplearán librerías populares como **Pandas, Numpy, Matplotlib, TensorFlow y su API Keras**, para poder desarrollar un modelo de Transformer al completo. También se analizarán herramientas que ayudan a simplificar el aprendizaje y la creación del modelo, como **HuggingFace**, que facilita el entrenamiento, la evaluación e implementación de modelos de PLN de vanguardia.

3.4 Objetivos Personales

Mis objetivos personales se centran en la inmersión en el mundo de las redes neuronales y, más concretamente, en las que se utilizan en el campo del PLN. Este estudio abordará desde los aspectos más teóricos, permitiéndome aprender su funcionamiento y desarrollo desde un punto de vista conceptual y matemático, hasta una parte más práctica, donde podré analizar los resultados y sacar conclusiones. Además, pretendo ampliar mi visión sobre el campo de la informática y sus múltiples enfoques, lo que me permitirá buscar una formación más especializada.

CAPÍTULO 4: HIPÓTESIS DEL TRABAJO

En este proyecto, exploramos cómo **Python** y sus **bibliotecas especializadas**, en conjunto con **Google Colab**, pueden utilizarse para implementar y entrenar eficazmente modelos basados en la arquitectura **Transformer**. La hipótesis se centra en destacar la relevancia de los Transformers en el ámbito de la **Inteligencia Artificial** y el **Procesamiento del Lenguaje Natural**, subrayando su aplicabilidad y eficacia.

4.1 Herramientas de desarrollo y entorno de trabajo

4.1.1 Google Colab

Google Colab es una herramienta de desarrollo en la nube creada por Google que ofrece acceso gratuito a **GPU** y **TPU** para el desarrollo de aplicaciones de IA y análisis de datos. Se puede considerar una versión avanzada de **Jupyter Notebook**, un bloc de notas online que permite la creación y ejecución de documentos programables.

A pesar de algunas limitaciones, Google Colab se ha convertido en una opción muy atractiva en el mundo de la investigación y el análisis de datos, gracias a sus ventajas y a su facilidad de uso. Permite un desarrollo rápido y colaborativo de proyectos de IA y análisis de datos en Python, convirtiéndose en una herramienta clave para investigadores y profesionales de estas áreas.[41]



Ilustración 34. Logo de Google Colab

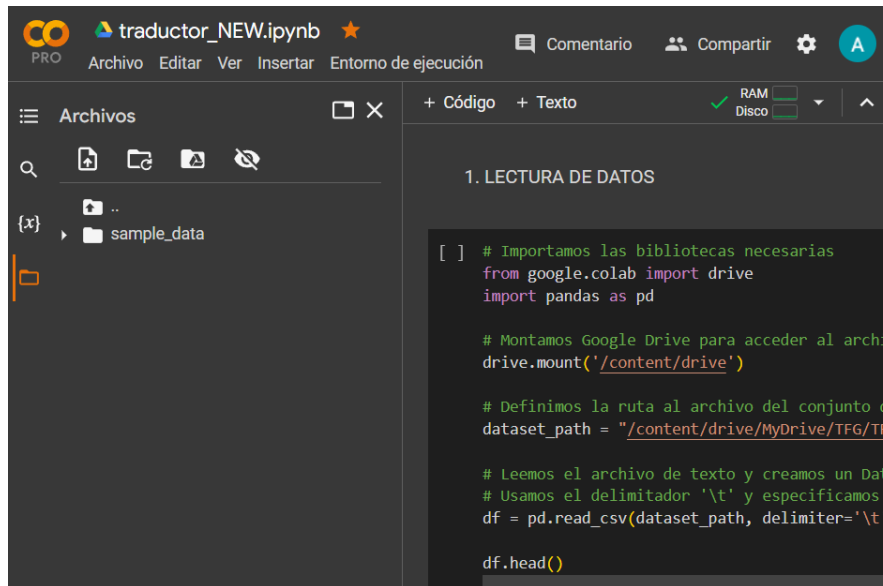


Ilustración 35. Interfaz de Google Colab

Características

Google Colab posee características que potencian la productividad y facilitan el trabajo en equipo, entre las que destacan:

- Ejecución de comandos de terminal desde el cuaderno.
- Importación de conjuntos de datos de fuentes externas como Kaggle.
- Posibilidad de guardar los documentos editados con Google Colab en Google Drive.
- Importación de cuadernos desde Google Drive.
- Uso gratuito de GPU y TPU en la nube.
- Integración con herramientas de análisis de datos e IA como PyTorch, Tensor Flow, Open CV.
- Importación y publicación directa en o desde GitHub.

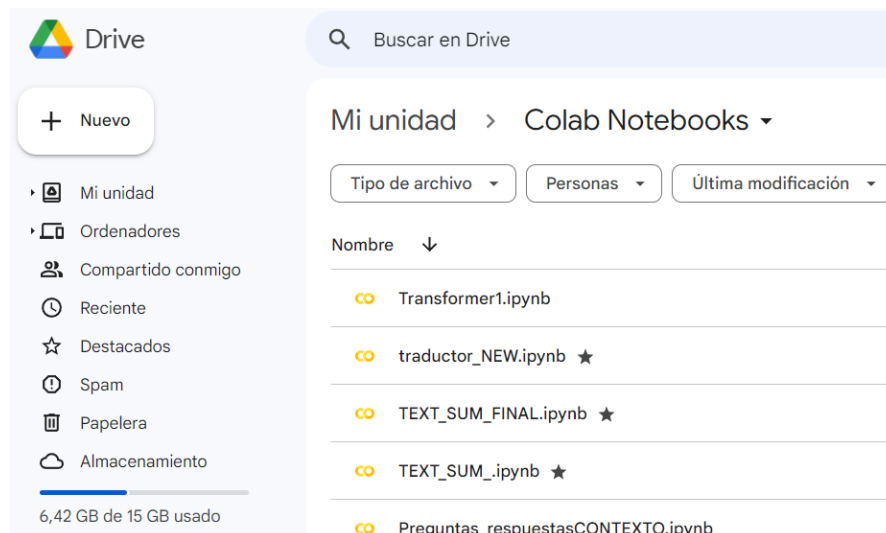


Ilustración 36. Cuadernos de Google Colab en Drive

Configuración específica utilizada

Para este proyecto, se aprovechó la versión Pro de Google Colab, una opción de suscripción premium que ofrece ventajas significativas sobre la versión estándar, incluyendo prioridad de acceso a recursos, tiempos de ejecución más largos y más memoria.

La configuración específica de **Google Colab Pro** utilizada fue esencial para implementar y entrenar eficientemente nuestros modelos basados en la arquitectura **Transformer**. Los detalles de la configuración son los siguientes:

- **Tipo de entorno de ejecución:** La compatibilidad de **Python 3** con una amplia gama de bibliotecas y módulos especializados es crucial para tareas de procesamiento y modelado de datos.
- **Acelerador por hardware:** GPU. Una GPU (Unidad de Procesamiento Gráfico) es esencial para el entrenamiento de modelos de aprendizaje automático, ya que pueden procesar operaciones paralelas y proporcionar una velocidad de cálculo significativamente más rápida en comparación con las CPUs tradicionales.
- **Tipo de GPU:** se eligió una GPU de tipo **T4**, conocida por su rendimiento y eficiencia en la aceleración de diversas cargas de trabajo en la nube. Este componente, que forma parte de la arquitectura **NVIDIA Turing™**, es especialmente útil en tareas de aprendizaje profundo, procesamiento de alto rendimiento, inferencia, análisis de datos y gráficos. Gracias a su

bajo consumo y su capacidad para proporcionar rendimiento revolucionario a escala, la GPU T4 resulta idónea para la implementación y entrenamiento de nuestros modelos basados en la arquitectura **Transformer**. Además, esta configuración cuenta con alta capacidad de **RAM**, ofreciendo los recursos necesarios para la ejecución eficiente de los procesos.[42]

- **Características del entorno de ejecución:** Alta capacidad de **RAM**. El manejo de grandes conjuntos de datos y el entrenamiento de modelos complejos requieren una gran cantidad de memoria RAM. En este proyecto, la alta capacidad de RAM permitió un manejo eficiente de los datos y garantizó el entrenamiento sin problemas de los modelos.

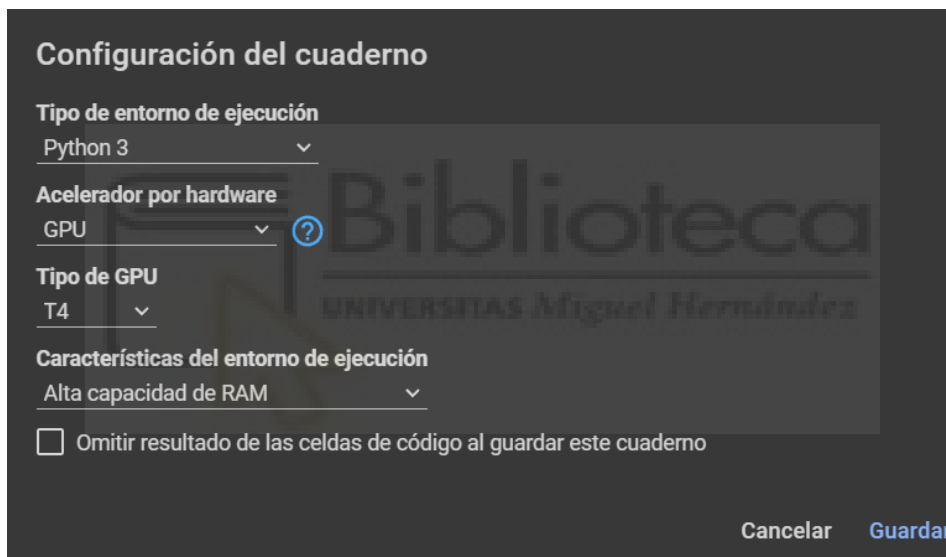


Ilustración 37. Entorno de ejecución

Utilización

Para comenzar a usar Google Colab, solo es necesario iniciar sesión con una cuenta de Google, seleccionar "Nuevo cuaderno" y empezar a programar. Google Colab permite el uso de Python como lenguaje de programación, el más utilizado en el mundo de la IA y el análisis de datos.

Compatibilidad con texto enriquecido y medios múltiples

Además de la programación, Google Colab permite la inclusión de texto enriquecido y distintos medios como imágenes, vídeos y tablas, facilitando la colaboración y el seguimiento de los proyectos.

Ventajas

- Bibliotecas de data science preinstaladas.
- Facilidad para compartir documentos y colaborar.
- Integración con diversas herramientas y GitHub.
- Capacidad para trabajar con datos de diversas fuentes.
- Acceso a aceleradores de hardware de ordenador.

Desventajas

- Diseñado principalmente para Python, lo que puede limitar el acceso a datos en otros lenguajes como R, SQL.
- Fluctuaciones en el acceso a GPU y TPU debido a la demanda en tiempo real.
- Las bibliotecas que se instalan son específicas para Google Colab y deben instalarse en cada equipo que se use.
- Limitación de espacio en el disco, lo que puede ser un problema para grandes conjuntos de datos.

4.2 Lenguaje de programación y librerías

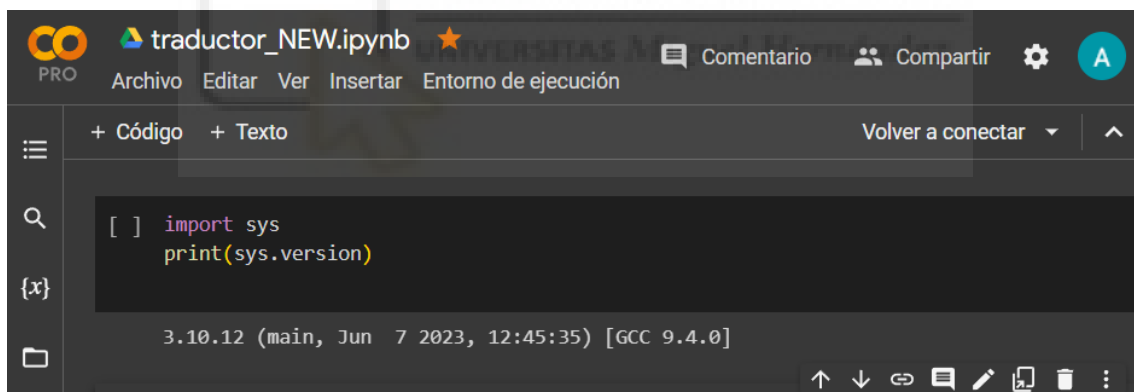
En este proyecto, se ha utilizado **Python** como lenguaje de programación por su versatilidad y facilidad de uso. Durante el desarrollo, se ha contado con el apoyo de diversas bibliotecas poderosas para optimizar el proceso y mejorar los resultados.

4.2.1 Python

Para este proyecto, se utiliza **Python** como el lenguaje de programación principal debido a su versatilidad, facilidad de uso y gran soporte comunitario. Python es ampliamente utilizado en campos como la **ciencia de datos** y el **aprendizaje automático**, principalmente debido a su eficiencia y compatibilidad con una variedad de plataformas. La versión específica de Python utilizada para este proyecto fue la **3.10.12** [43].



Ilustración 38. Logo de Python

A screenshot of a Jupyter Notebook interface. The top bar shows the file name "traductor_NEW.ipynb" and various menu options like "Archivo", "Editar", "Ver", "Insertar", and "Entorno de ejecución". The main area contains a code cell with the following Python code:

```
[ ] import sys
print(sys.version)
```

The output of the code is displayed below the cell:

```
3.10.12 (main, Jun 7 2023, 12:45:35) [GCC 9.4.0]
```

Ilustración 39. Versión de Python

Python es un lenguaje de programación interpretado de alto nivel y de propósito general. Su diseño de sintaxis clara y legibilidad hace que sea fácil de aprender y usar, permitiendo a los programadores expresar conceptos complejos en menos líneas de código que sería posible en lenguajes como **C++** o **Java**.

		Most popular in	Least popular in
JavaScript*	19.6 M	Apps for 3rd-party ecosystems, Cloud	DS/ML/AI, Embedded
Python	16.9 M	DS/ML/AI, IoT apps	Web, Mobile
Java	16.5 M	Cloud, Desktop	Web, DS/ML/AI
C/C++	12.3 M	Embedded, IoT apps	Cloud, Web
C#	10.6 M	Desktop, Games	DS/ML/AI, IoT devices
PHP	8.9 M	Web, Cloud	Mobile, DS/ML/AI
Kotlin	6.1 M	Mobile, AR/VR	Games, DS/ML/AI
Visual development tools	4.9 M	AR/VR, Games	Embedded, Cloud
Swift	4.2 M	Mobile, AR/VR	Embedded, Cloud
Go	3.8 M	Apps for 3rd-party ecosystems, Cloud	Mobile, DS/ML/AI
Objective C	3.0 M	AR/VR, IoT devices	Desktop, Apps for 3rd-party ecosystems
Rust	2.8 M	AR/VR, IoT apps	Mobile, Web
Ruby	2.4 M	IoT devices, Apps for 3rd-party ecosystems	Embedded, Web
Dart	1.9 M	Mobile, Apps for 3rd-party ecosystems	Web
Lua	1.9 M	IoT devices, AR/VR	Mobile, Embedded

Ilustración 40. Gráfico de uso de Python en la industria

Durante el desarrollo de este proyecto, se utilizaron algunas características clave de **Python**. Primero, la naturaleza interpretada de Python facilitó la depuración del código, ya que los errores se indican inmediatamente durante la ejecución. Además, Python es un lenguaje de tipado dinámico, lo que significa que no fue necesario declarar explícitamente los tipos de variables. Esto agilizó el proceso de codificación y permitió un desarrollo más rápido y fluido.

Python también apoya la **programación orientada a objetos**, un estilo de programación que se utiliza ampliamente en la ciencia de datos y el aprendizaje automático. En este proyecto, se utilizó este enfoque para estructurar el código de una manera más intuitiva y escalable.

Uno de los puntos fuertes de Python es su extenso ecosistema de librerías y frameworks. Estas bibliotecas, muchas de las cuales son de código abierto,

ofrecen funcionalidades preconstruidas para una variedad de tareas, desde la manipulación de datos hasta el aprendizaje automático y la visualización. En este proyecto, se utilizaron varias de estas bibliotecas para facilitar el desarrollo del modelo **Transformer**.

Por último, **Python** tiene una comunidad de desarrollo activa y de soporte que es invaluable para los desarrolladores. La comunidad no solo contribuye con bibliotecas y herramientas, sino que también ofrece soporte a través de foros de discusión y tutoriales en línea. Esto asegura que Python continúe siendo una opción relevante y eficaz para proyectos de aprendizaje automático y procesamiento de lenguaje natural.



Ilustración 41. Comunidad de Python en España

4.2.2 Bibliotecas

Pandas

Pandas es una biblioteca de **Python** de código abierto que proporciona estructuras de datos potentes y flexibles, ideales para manipular y analizar datos estructurados y de series temporales. Esta librería surge con el objetivo de equipar a los analistas de datos con una herramienta única para cargar, modelar,

analizar, manipular y preparar datos, simplificando el trabajo en el campo de la **ciencia de datos** y el **aprendizaje automático** [44].

Las dos estructuras de datos fundamentales de **Pandas** son las **Series** (arrays unidimensionales etiquetados) y los **DataFrame** (estructuras bidimensionales con columnas de cualquier tipo). Con **Pandas**, es posible cargar datos de diferentes formatos de archivos (csv, json, html, etc.), y realizar análisis detallados gracias a sus múltiples funciones. Además, permite la creación de gráficos a partir de un DataFrame o Series, gracias a su integración con **Matplotlib**. Su facilidad de uso, capacidad para manejar cualquier tipo de información y eficiencia en la ejecución, hacen de **Pandas** una herramienta invaluable para los entusiastas de los datos.

```
#import all libraries!  
import pandas as pd # el as pd es un alias, hace el codigo un poco mas corto  
  
#importemos (carguemos en memoria) la data  
#pandas carga la data como un dataframe o matriz, tal como si tuvieramos un spreadsheet  
data = pd.read_csv('data/titanic.csv')  
  
#previsualicemos la data  
data.head()
```

PassengerId	Survived	Pclass	Name	Sex	Age	SibSp	Parch	Ticket	Fare	Cabin	Embarked
0 1	0	3	Braund, Mr. Owen Harris	male	22.0	1	0	A/5 21171	7.2500	NaN	S
1 2	1	1	Cumings, Mrs. John Bradley (Florence Briggs Th...	female	38.0	1	0	PC 17599	71.2833	C85	C
2 3	1	3	Heikinen, Miss. Laina	female	26.0	0	0	STON/O2. 3101282	7.9250	NaN	S
3 4	1	1	Futrelle, Mrs. Jacques Heath (Lily May Peel)	female	35.0	1	0	113803	53.1000	C123	S
4 5	0	3	Allen, Mr. William Henry	male	35.0	0	0	373450	8.0500	NaN	S

Ilustración 42. Visualización de datos con Pandas

Matplotlib

Matplotlib es una poderosa biblioteca de visualización de datos de Python que produce gráficos de alta calidad en una variedad de formatos. Se utiliza ampliamente para convertir datos en gráficos de dos y tres dimensiones, facilitando su interpretación y análisis. Además de la personalización completa, los gráficos creados con Matplotlib pueden ser estáticos, animados o interactivos y pueden ser utilizados en scripts de Python, shells de Python, aplicaciones web y entornos de notebook [45].

Gracias a su versatilidad y eficiencia, Matplotlib proporciona un sólido puente entre los datos y su representación visual, ayudando a visualizar relaciones y

patrones en los datos, y facilitando la identificación de áreas que necesitan atención o la predicción de tendencias futuras.

```
In [222]: %matplotlib notebook

In [223]: X = np.arange(-10, 10, 0.25)
          Y = np.arange(-10, 10, 0.25)
          X, Y = np.meshgrid(X, Y)
          Z = np.sin(np.sqrt(X**2 + Y**2))

In [224]: fig = plt.figure(figsize = (9, 5))
          ax = fig.gca(projection='3d')
          surface = ax.plot_surface(X, Y, Z, cmap = "summer")
          fig.colorbar(surface)
          plt.show()
```

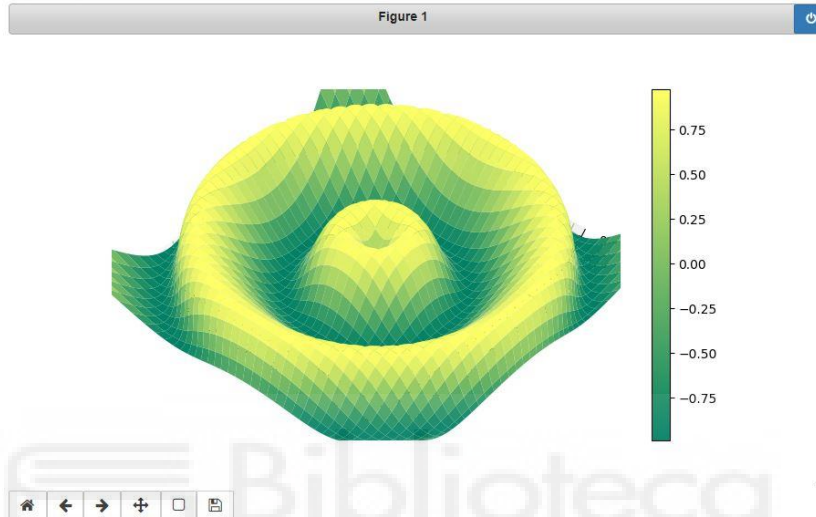


Ilustración 43. Gráfico dinámico en 3D con Matplotlib

NumPy

NumPy, abreviatura de Python Numérico, es una biblioteca de Python esencial para el cálculo y procesamiento de elementos de matriz multidimensionales y unidimensionales. Escrita en su mayoría en C, proporciona una serie de funciones para realizar cálculos numéricos de alta velocidad. NumPy ofrece diversas estructuras de datos poderosas, incluyendo arrays y matrices multidimensionales, que son utilizadas para optimizar los cálculos relacionados con estos elementos.[43]

Debido a su capacidad para manejar grandes volúmenes de datos de manera eficiente, NumPy se ha vuelto imprescindible en el campo de la ciencia de datos. Además de facilitar la multiplicación de matrices y la reconfiguración de datos, NumPy realiza computación orientada a arrays y proporciona una implementación eficaz de matrices multidimensionales.

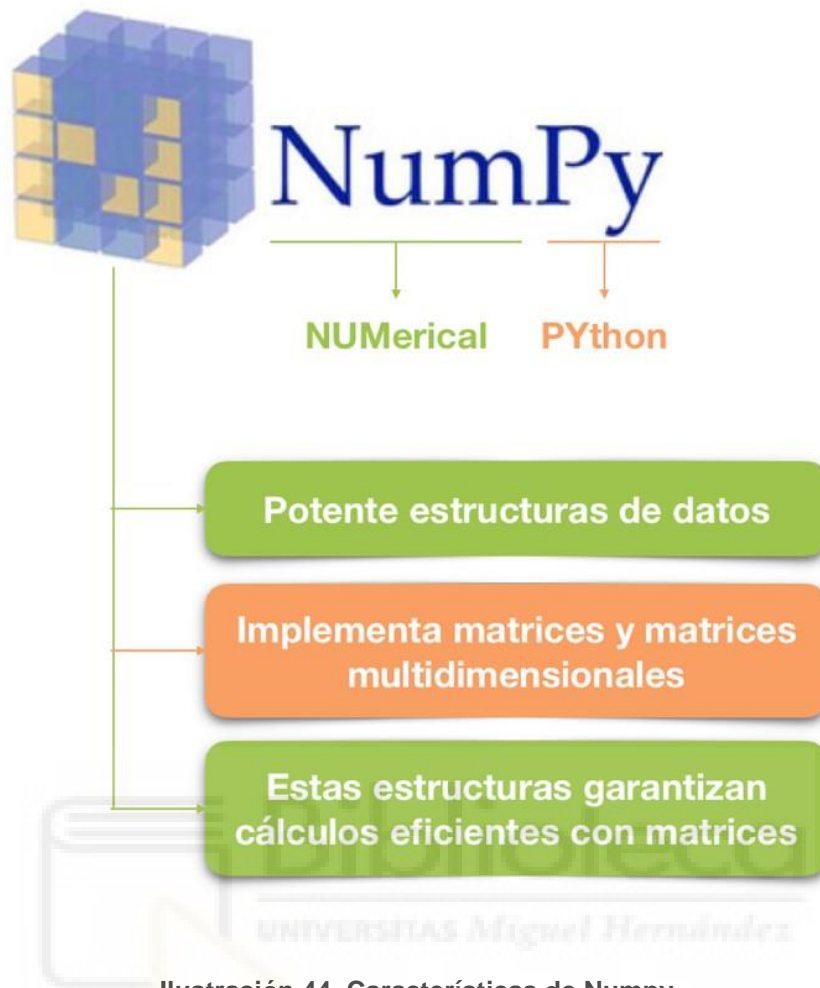


Ilustración 44. Características de Numpy

TensorFlow

TensorFlow es una biblioteca de software de código abierto desarrollada por ingenieros e investigadores del Google Brain Team, con el objetivo principal de realizar investigaciones en el campo del aprendizaje automático y las redes neuronales profundas. Su flexibilidad y arquitectura permiten implementar la computación en una o varias CPU o GPU en dispositivos de escritorio, servidores o dispositivos móviles [46].

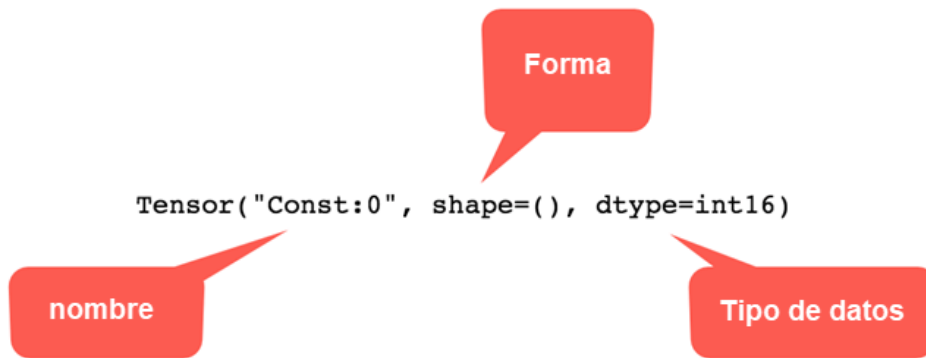


Ilustración 45. Visualización de un tensor

Scikit-Learn

Scikit-Learn es una biblioteca de Python que proporciona implementaciones eficientes de una multitud de algoritmos comunes. Desde su creación en 2007, **Scikit-Learn** se ha ganado popularidad debido a su API coherente y estandarizada, facilitando la preparación de datos, el entrenamiento y la evaluación de los modelos, y ofreciendo una amplia gama de algoritmos y herramientas para optimizar parámetros y analizar los resultados[47].



Ilustración 46. Funcionalidades Scikit-Learn

CAPÍTULO 5: METODOLOGÍA Y RESULTADOS

Este quinto capítulo tiene como objetivo describir y esclarecer la metodología adoptada para el desarrollo de este proyecto. Se presentará un desglose detallado de las diversas tareas ejecutadas, así como una explicación clara de la planificación y organización del proyecto. A continuación, se realizará una inmersión profunda en el modelo Transformer, un componente esencial y el motor de esta investigación, proporcionando detalles sobre su funcionamiento y aplicación. Finalmente, se abordarán los desarrollos concretos llevados a cabo durante el curso del proyecto, todos los cuales emplean el modelo Transformer, resaltando sus particularidades y los resultados obtenidos.

5.1 Gestión y coordinación del proyecto

En este **Trabajo de Fin de Grado** (TFG), la gestión eficiente del tiempo y los recursos se ha considerado un aspecto fundamental para el desarrollo del proyecto. Por lo tanto, en lugar de seguir un enfoque tradicional basado en hitos o entregables, se ha optado por un enfoque basado en tareas. Este enfoque ha permitido dividir el proyecto en componentes manejables y definir claramente qué se necesitaba hacer para alcanzar los objetivos establecidos.

La planificación y ejecución del proyecto se llevaron a cabo siguiendo una serie de tareas. Cada tarea fue cuidadosamente evaluada y designada con un conjunto de horas estimadas para su finalización, teniendo en cuenta su complejidad y los requisitos necesarios para llevarla a cabo.

La clave para este enfoque fue la capacidad de adaptarse y reorganizar las tareas según las necesidades del proyecto. En algunos casos, se necesitó más tiempo del esperado para completar una tarea, pero gracias a una planificación cuidadosa y un seguimiento constante del progreso, se consiguió hacer los ajustes necesarios para mantener el proyecto en curso.

Al final, el enfoque basado en tareas permitió llevar a cabo el proyecto de manera eficiente, garantizando que cada componente esencial del proyecto se tratara en

detalle y en el orden correcto. Esto condujo a la consecución de los objetivos propuestos para este TFG dentro de los plazos establecidos.

5.1.1 Desglose de tareas

La tabla a continuación proporciona un desglose detallado de las tareas necesarias para la realización de este Trabajo de Fin de Grado. Cada tarea se ha clasificado según su naturaleza (**Estudio, Análisis, Desarrollo, Entorno, Documentación**), se le ha asignado una duración estimada en horas y se ha identificado la tarea o tareas precedentes.

Este desglose detallado de tareas ha permitido una gestión eficaz del tiempo y los recursos, asegurando que todas las tareas se ejecuten en la secuencia correcta y dentro del tiempo previsto. A su vez, proporciona una base sólida para la creación del **Diagrama de Gantt**, una herramienta visual útil para seguir y controlar el progreso del proyecto. Cabe mencionar que, aunque se han asignado tiempos estimados a cada tarea, estos pueden variar en función de los desafíos y obstáculos que puedan surgir durante la ejecución del proyecto.

A continuación, se presenta el desglose detallado de tareas para el proyecto:

Numero	Tarea	Actividad	Duración (h)	Precendente
1	Estudio general de la IA	Estudio	20	-
2	Estudio del PLN y sus aplicaciones	Estudio	10	1
3	Estudio de las RNN y LSTM	Estudio	16	2
4	Estudio del Transformer	Estudio	24	3
5	Estudio de preprocesamiento de datos	Estudio	13	2
6	Estudio del entorno de desarrollo, Google Colab	Entorno	8	-
7	Estudio del lenguaje Python	Estudio	22	6
8	Estudio de las librerías empleadas	Estudio	7	6

9	Creación de Redes Neuronales Básicas	Desarrollo	25	7,8
10	Desarrollo del dataset para Q&A	Desarrollo	32	7
11	Creación de red transformer para modelo de Q&A	Desarrollo	25	10
12	Creación de red transformer para análisis de sentimientos	Desarrollo	25	11
13	Creación de red transformer para traducción automática	Desarrollo	25	12
14	Análisis de los resultados y pruebas	Análisis	9	11, 12, 13
15	Generación de memoria final	Documentación	72	14

Ilustración 47. Desglose de tareas

5.1.2 Diagrama de Gantt

El Diagrama de Gantt a continuación ilustra la programación de las tareas a lo largo del tiempo. Este diagrama ofrece una representación visual de las diferentes etapas del proyecto, las dependencias entre las tareas y su duración prevista en horas. Al ver todas las tareas en un solo gráfico, es más fácil comprender cómo se relacionan entre sí y cómo el retraso en una tarea puede afectar a otras.

Este diagrama también proporciona una excelente herramienta de seguimiento, ya que permite verificar en cualquier momento si el proyecto está avanzando según lo previsto. Si una tarea se está retrasando, es fácil identificar las tareas que podrían verse afectadas y reajustar la programación en consecuencia.

A continuación, se muestra el Diagrama de Gantt para el proyecto:

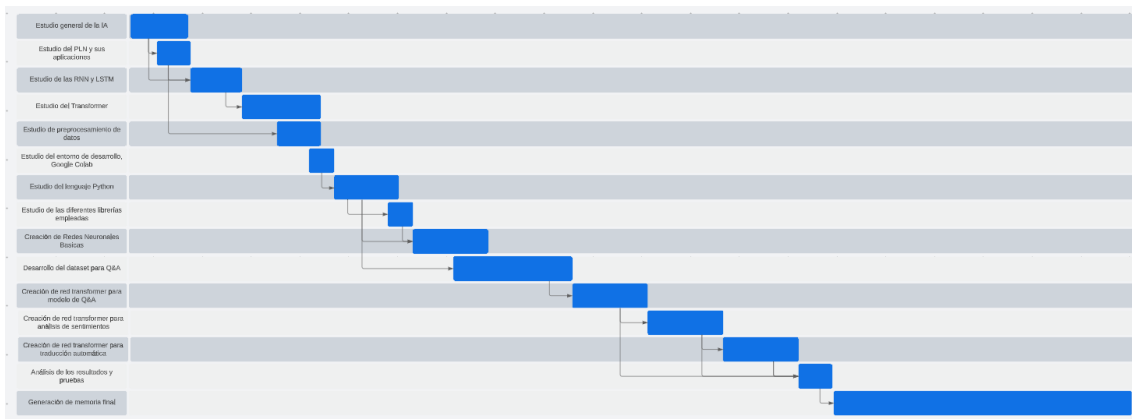


Ilustración 48. Diagrama de Gantt

5.1 Preparación de datos

El Procesamiento del Lenguaje Natural (PLN) se centra en la comprensión, el manejo y la generación de lenguaje natural por parte de las máquinas. En este sentido, se ocupa de explorar formas efectivas de comunicación entre las máquinas y las personas mediante el uso de lenguaje natural. Para posibilitar esta comunicación, es necesario que las máquinas sean capaces de comprender el texto, que representa nuestro lenguaje. Sin embargo, puesto que las máquinas operan en términos de representaciones numéricas, es imprescindible convertir el texto en números.

Antes de detallar el proceso de esta transformación numérica, es crucial abordar el preprocesamiento de los datos que se introducen en formato de texto. Este aspecto se podría considerar como uno de los más importantes para cualquier algoritmo de **machine learning**, ya que toda la arquitectura de estos sistemas se fundamenta en estos datos.

En esta sección, se discutirán algunas de las técnicas utilizadas en la **limpieza de datos** (también conocida como **data cleaning** en inglés), así como las técnicas empleadas en el preprocesamiento y en la configuración de la representación numérica del texto.

5.1.1 Limpieza del texto

La primera fase en el preprocesamiento de datos para el PLN es la limpieza del texto. Este proceso implica depurar y limpiar los datos, eliminando toda la información no textual que pueda estar presente en los textos usados para el PLN. El objetivo es obtener textos coherentes, puros y libres de errores, eliminando elementos como etiquetas HTML, URLs, emojis, etc. Algunas de las técnicas empleadas incluyen:

Corrección ortográfica

Dado que el lenguaje humano es propenso a errores, los textos de entrada pueden contener errores ortográficos. Estos errores pueden dificultar la tarea de recuperar y depurar datos, especialmente en aplicaciones como motores de búsqueda, redes sociales y chatbots. Por tanto, es crucial desarrollar técnicas de corrección ortográfica y etiquetado en el ámbito del análisis léxico. Python ofrece varios módulos para facilitar esta tarea, algunos ejemplos incluyen:

- **TextBlob:** Una biblioteca de PLN que proporciona una API intuitiva para su uso.
- **API REST de Bing Spell Check y Python:** Una aplicación de Python que envía una solicitud a la API y devuelve una lista de correcciones sugeridas.

Normalización de Unicode

En el PLN, es esencial tratar a aquellos caracteres que son funcional y visualmente equivalentes, pero que tienen diferentes representaciones de código, de manera igualitaria. Ejemplos comunes incluyen símbolos, emojis y caracteres gráficos, a menudo encontrados en las interfaces web. Este problema se resuelve con un proceso llamado **normalización de Unicode**, que genera una representación binaria uniforme para todas las representaciones equivalentes de un carácter [48] .

Eliminación de etiquetas HTML

La obtención de datos de calidad es un paso crucial en cualquier proyecto basado en datos. En el contexto del PLN, el **web scraping** es una técnica comúnmente utilizada para extraer grandes cantidades de información de sitios

web, lo que a menudo implica la extracción de código HTML innecesario. Sin embargo, este código puede contener muchas etiquetas HTML las cuales no proporcionan información útil para el PLN, como “<script> y <button>”. Por tanto, es esencial eliminar estas etiquetas. Módulos de Python como **boilerpy3**, que facilita la extracción de texto eliminando etiquetas innecesarias, y **BeautifulSoup**, que es útil para extraer datos de documentos HTML y XML, son muy útiles en este proceso [49].

Este apartado destaca la importancia de la calidad de los datos a la hora de obtener información. Al obtener un texto puro y sin errores, se puede avanzar hacia las siguientes fases del PLN, que adquieren aún más importancia en todo el proceso.

5.1.2 Técnicas de PLN en la etapa de preprocesamiento

Después de limpiar los datos para obtener un texto puro, el siguiente paso es determinar cómo se puede representar el texto de manera que una computadora pueda entenderlo y procesarlo correctamente. En esta sección, se describirán algunas técnicas que pueden ser utilizadas según criterios como el idioma, el dominio del texto, y la aplicación del procesamiento. Según investigaciones previas, las técnicas más comúnmente aplicadas en PLN incluyen [50]:

Eliminación de signos de puntuación y caracteres especiales

Los signos de puntuación y caracteres especiales, como la repetición de letras en palabras, hashtags, signos de interrogación y exclamación, no influyen en el procesamiento del texto. Por lo tanto, es recomendable su eliminación o sustitución por tokens únicos, ya que podrían generar un mayor tiempo de procesamiento.

Eliminación de enlaces web y menciones de usuarios

La presencia de enlaces web es frecuente en textos que provienen de redes sociales, como los tweets, que se utilizan comúnmente para el análisis de sentimientos o clasificación del texto. Estos enlaces no afectan a las tareas

mencionadas, por lo que es recomendable reducir el número de palabras, eliminándolos.

Tokenización

El objetivo de esta técnica es la división del texto en palabras o frases significativas, denominadas tokens. Estas se delimitan por caracteres especiales como puntos, comas o espacios en blanco, permitiendo la creación de un vocabulario.

Esta técnica es común en todos los campos del PLN, desde la traducción del lenguaje hasta el análisis de sentimiento. Este amplio uso se debe a que facilita al PLN al generar conjuntos o subconjuntos de palabras que pueden ser analizadas y utilizadas de forma individual. Por lo tanto, cualquier pipeline de PLN, debe contener un sistema de tokenización, es decir, conseguir tokens a partir de un texto con mayor longitud. Ejemplo: «En adjunto, encontrarás el documento en cuestión» -> «encontrarás», «en adjunto», «el documento», «en cuestión».

Convertir el texto a minúsculas o lowerCase

Esta técnica es sencilla y se basa en la transformación de todo el texto que se encuentra en mayúsculas a minúsculas, logrando unificar las palabras y disminuir la dimensionalidad del texto.

Sin embargo, existen situaciones en las cuales esta conversión a minúsculas puede resultar confusa. Un ejemplo, serían cuando los acrónimos se convierten a minúsculas, en estos casos la probabilidad de que se consideren nombres es muy alta. Por ejemplo, el acrónimo DOG (Digital Onscreen Graphic), en minúscula sería considerado como un nombre, dog, perro en inglés, lo cual generaría una confusión.

Eliminación de palabras vacías o Stop-Word removal

Stop-word, traducido al castellano como palabra de parada, se refiere a palabras de uso común que no aportan mucha información. Es recomendable ignorar

estas palabras, ya que ocupan espacio en el conjunto de datos y requieren tiempo de procesamiento.

Existen 4 métodos de eliminación de estas palabras:

- **Método clásico:** se utiliza para eliminar las palabras vacías mediante diccionarios pre-compilados.
- **Métodos basados en la Ley de Zip:** los métodos que incluyen consisten en eliminar las palabras más frecuentes, palabras de baja frecuencia y palabras simples.
- **Métodos de Información Mutua (MII):** mediante un sistema de comprobación de información entre un término dado y el tipo de documento, facilita cuanta información el termino puede generar sobre una clase del documento, cuando la información sea baja indicara que el termino tiene bajo poder de distinción y por lo tanto se elimina.
- **Muestreo aleatorio basado en términos (TBRIS):** este método se basa en la detección de palabras vacías manualmente por medio de documentos web.

Lematización

El objetivo de esta técnica es unificar los términos que aportan la misma información de un texto, reemplazándolos por su lema. El lema de una palabra es un término que por convención se acepta como representante de todas las formas de una palabra y para encontrarlo, se eliminan todas sus conjugaciones, (genero, numero, etc.).

En PLN esta técnica resulta de utilidad por que reduce la cantidad de términos de un texto a analizar. Esta técnica de lematización necesita un diccionario para buscar e indexar, lo cual mejora su precisión en aplicaciones de extracción de características.

Estos diccionarios o tabla de búsquedas son un método que agrupa en una tabla los termino o palabras y sus lemas, de esta manera estos pueden ser buscados y encontrados en el diccionario y así conseguir aplicar la *lematización*. Para la

mencionada búsqueda se utiliza el algoritmo de árbol-b o una dispersión, puesto que es un método sencillo y rápido de utilizar, aunque tiene el inconveniente de necesitar diccionarios para cada idioma, para establecer la relación entre un término y el lema que le corresponda. Ejemplo: «encontrarás» -> “encontrar” [51].

Steaming

Al igual que la lematización, el stemming se involucra en el análisis a nivel morfológico dentro del PLN, pero a diferencia de la lematización, el stemming pretende eliminar automáticamente los sufijos o prefijos de las palabras del texto.

Una técnica muy útil en trabajos de recuperación de información (RI). Estos son de los algoritmos más importantes:

- **Eliminación de afijos:** pretende eliminar sufijos y/o prefijos de los términos para obtener su lema, por medio de algoritmo de coincidencia, como es el de Porter. Ejemplo: «encontrarás» -> «encontr»
- **Sucesor de variedad:** se basa en lingüística estructural, determinando los límites en los términos y los morfemas de una cadena.

N-GRAMAS

N-gramas es una técnica comúnmente utilizada en el procesamiento del lenguaje natural para preservar la semántica de frases que necesitan ser entendidas como una unidad. En términos sencillos, un n-grama se refiere a una secuencia de "n" palabras en un fragmento de texto.

Los n-gramas son útiles dado que permiten mantener juntas palabras que poseen un significado conjunto, en lugar de tratarlas como palabras individuales. Para una mejor comprensión, se presentarán algunos ejemplos con diferentes valores para "n":

- **Unigramas:** "n" es igual a 1, lo que significa que cada palabra se considera de manera individual. Ejemplos: "gato", "perro", "casa".
- **Bigramas:** Cuando "n" es igual a 2, se agrupan palabras de dos en dos. Un ejemplo de esto sería la frase "Estados Unidos". En este caso, se trata

"Estados Unidos" como una sola entidad, en lugar de dos palabras separadas.

- **Trigramas:** Cuando "n" es igual a 3, se toman tres palabras juntas. Por ejemplo, "Emiratos Árabes Unidos". De nuevo, en lugar de tratar "Emiratos", "Árabes" y "Unidos" como palabras independientes, se las considera como un solo elemento.

Etiquetado POS

También conocido como etiquetado gramatical, su objetivo es asignar a cada palabra de un texto una categoría gramatical, como sustantivo, adjetivo, verbo, etc. Este proceso se realiza de dos maneras, conforme a la definición de la palabra o de acuerdo con el contexto.

Finalmente, es importante destacar algunos modelos pre-entrenados e implementados en las librerías de Python para resolver las tareas mencionadas anteriormente, como son:

- **NLTK:** Natural Language Toolkit, una plataforma de código abierto, escrito en Python, siendo una de las más estables para trabajar con textos puros. Entre las principales herramientas y recursos léxicos que aportan, está la tokenización, etiquetado morfo sintáctico, análisis morfo sintáctico, reconocimiento de entidades nombradas y etiquetado de texto
- **FRRELING:** es una librería de código abierto, escrita en C++, que proporciona funcionalidades para analizar el lenguaje. Por ejemplo, análisis morfológico, etiquetado POS, análisis sintáctico, Tratamiento de sufijos, reconocimiento de palabras compuestas, etc.
- **SPACY:** es una librería de código abierto dedicada al procesamiento avanzado del PLN en Python. Fue creada para que procese y entienda grandes volúmenes de texto y su uso normalmente se aplica a la construcción de sistema de extracción de información, compresión del PLN o para pre-procesamiento de texto para ML.

En este apartado se han visto las técnicas más necesarias para obtener datos de calidad y así formar un vocabulario, dejando atrás una parte fundamental en

el preprocesamiento de datos y de las más costosas, debido a la cantidad de datos de texto que se manejan y sobre los cuales hay que realizar las técnicas mencionadas. Este esquema de a continuación servirá como guía para saber los pasos recorridos y lo siguiente por ver en el PLN.



Ilustración 49. Proceso del pre-procesamiento

5.1.3 Representación numérica

Se inició la vectorización siguiendo el esquema previo. Con el texto depurado y un vocabulario como datos de entrada, fue necesario preparar estos datos para ser procesados por las redes neuronales que soportan el aprendizaje automático. Los algoritmos de aprendizaje automático no pueden trabajar directamente con texto o caracteres; se requieren representaciones numéricas, expresadas como vectores. En esta sección, se examinan técnicas para convertir el texto en estas representaciones numéricas.

Label encoding

Es la técnica más simple para representar numéricamente cada token, asignándole un número a cada palabra. Como la siguiente frase mostrada a continuación.

Hacemos la asociación de cada palabra con la misma etiqueta.
25 7 62 8 28 20 5 7 16 12

Ilustración 50. Representación label encoding

Se puede implementar fácilmente, por ejemplo, con **LabelEncoder** de **scikit-learn** en Python. Sin embargo, puede llevar a interpretaciones erróneas, ya que el algoritmo puede interpretar el valor numérico asignado a las palabras como una medida de magnitud o distancia. Por ejemplo, si las ciudades están etiquetadas como 0, 1, 2, y 3, el algoritmo puede malinterpretar que 3 es el triple de 1. Este problema se abordará con el método de codificación one-hot, que describiremos a continuación.

Condificación one-hot

La codificación one-hot es una técnica que prepara los datos para un algoritmo de aprendizaje automático, transformando cada valor categórico en una nueva columna y asignándole un valor de 0 o 1. En otras palabras, cada palabra o token se convierte en un vector binario del tamaño del vocabulario, con todos los valores en 0, excepto en el índice que corresponde a la palabra, que será 1.



Ilustración 51. Palabra en codificación one-hot

Se toma un ejemplo: si tenemos un vocabulario de cuatro palabras, la codificación one-hot nos proporcionará un vector binario de dimensión 4 para cada una de ellas.

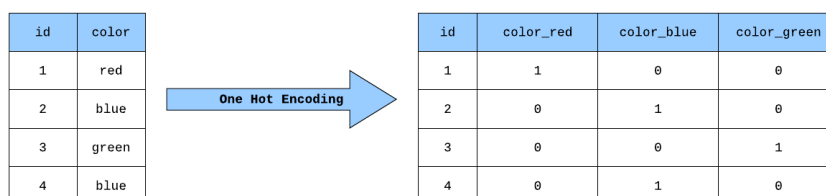


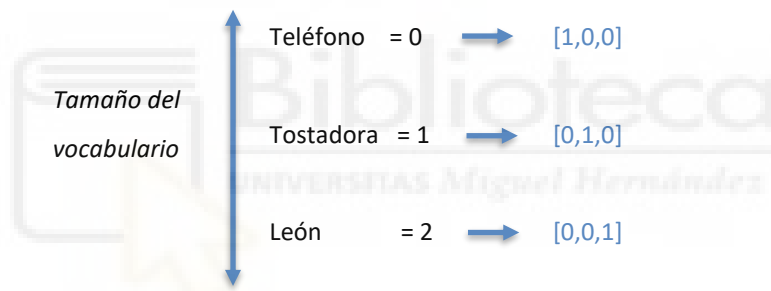
Ilustración 52. Codificación one-hot de 4 colores

Una vez que se comprende la técnica utilizada, será más fácil analizar sus ventajas y desventajas. Algunas de sus ventajas podrían ser:

- **Evita relaciones cuantitativas erróneas entre palabras:** no hay una asignación numérica para cada token, lo que soluciona el problema de interpretación errónea de la distancia en Label Encoding.
- **Facilidad de implementación e interpretación intuitiva:** es fácil de implementar y comprender.

Algunas de sus desventajas son:

- **Misma distancia geométrica entre palabras:** en un espacio tridimensional, todos los tokens tienen su propia dimensión y la distancia entre cada palabra es la misma, lo que puede llevar a interpretaciones incorrectas sobre las relaciones semánticas entre palabras.



Al colocar estos vectores en un espacio tridimensional, cada token adquiere su propia dimensión. Así, la distancia geométrica entre cada par de palabras será idéntica, independientemente de su significado. Esto significa que, según este método, palabras como "león" y "tostadora" se percibirían como igualmente distintas a "teléfono" y "tostadora". No obstante, este enfoque de medir las distancias es incorrecto en el contexto del lenguaje, ya que existen similitudes inherentes entre ciertos grupos de palabras.

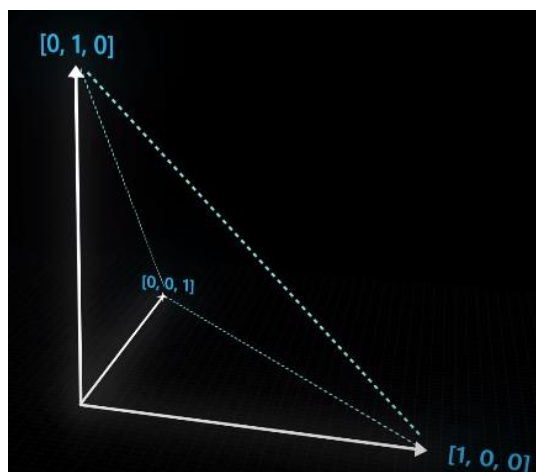


Ilustración 53. Espacio tridimensional

- **Alto número de dimensiones:** el tamaño de un vector codificado en one-hot es directamente proporcional al tamaño del vocabulario. Con vocabularios grandes, la mayoría de las coordenadas de los vectores serán ceros, lo que resulta ineficiente en almacenamiento y procesamiento.
- **Dificultad para las palabras nuevas:** no hay forma de representar palabras que no estuvieran en el conjunto de datos de entrenamiento original, un problema conocido como "fuera de vocabulario". La única solución a este problema es reentrenar el modelo para incluir la nueva palabra en el vocabulario.

Word embeddings

Los enfoques previos presentan una desventaja geométrica significativa: cada **token** o palabra es independiente del resto, cada uno ocupando su propia dimensión y manteniendo la misma distancia con las demás palabras. Sin embargo, el **lenguaje humano** es intrínsecamente relacional: se asocian palabras y, por lo tanto, se esperan diferentes distancias entre ellas.

Por eso, es vital tener en cuenta el **contexto** de cada palabra. El contexto aporta información crucial sobre su significado, lo que nos permite establecer similitudes entre conjuntos de palabras.

Un método que aborda estas preocupaciones es el uso incrustaciones, conocidas en inglés como "**embeddings**". Este método captura el contexto, la similitud semántica y sintáctica de las palabras al reducir su dimensionalidad, es decir, pasa de representar palabras en miles de dimensiones a unas pocas. Los embeddings son útiles para reducir la dimensionalidad de las variables categóricas y representarlas significativamente en el espacio transformado.

La **reducción de la dimensionalidad** se logra mediante una arquitectura de red neuronal que agrega capas de menor tamaño para obtener un vector denso de menor dimensionalidad, como se muestra en la siguiente imagen.

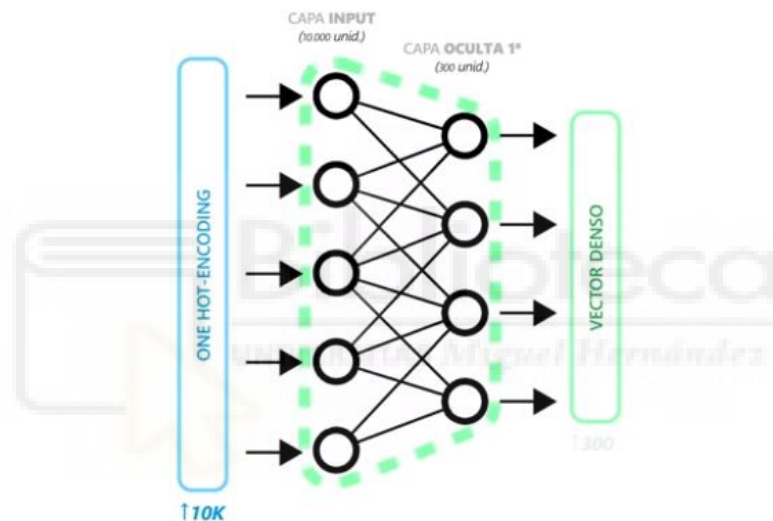


Ilustración 54. Arquitectura de embedding

Entendiendo la arquitectura, la red tiene que aprender a comprimir y organizar las palabras de manera óptima para resolver la tarea dada. Por ejemplo, si el objetivo es determinar si un texto tiene un **sentimiento positivo o negativo**, la red aprenderá a agrupar palabras con sentimientos similares en el proceso de creación de **embeddings**. Este proceso ocurre en la primera capa de la red neuronal, como se muestra a continuación.

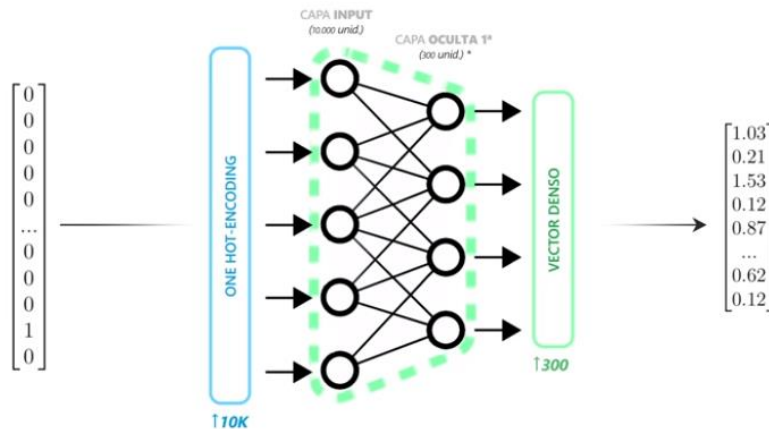


Ilustración 55. Embedding

El resultado final es que las palabras con contextos similares se agrupan espacialmente. Matemáticamente, el coseno del ángulo entre los vectores de palabras similares debería ser cercano a 1, lo que implica un ángulo cercano a 0. Esto se evidencia en la siguiente imagen que muestra la similitud entre las palabras "manzana" y "pera" [52].

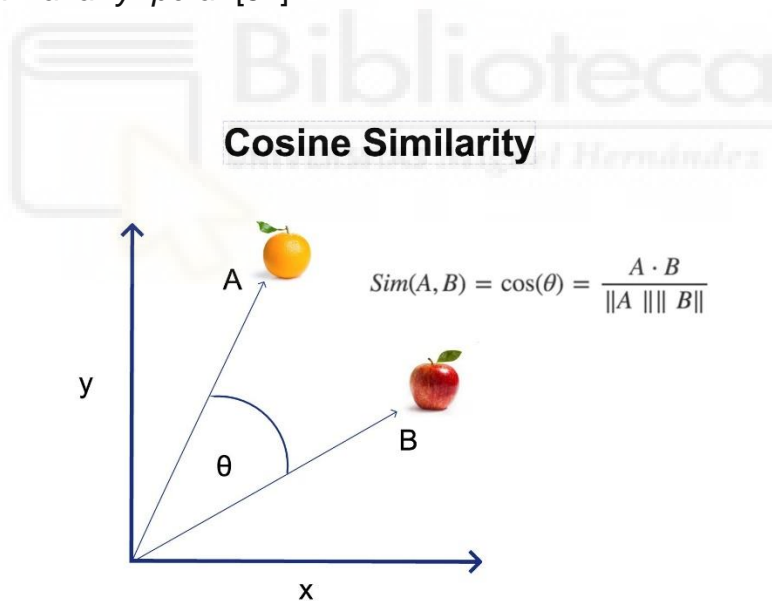


Ilustración 56. Relación vectorial

A menudo, se parte de una **red neuronal preentrenada** que ya ha aprendido los patrones básicos para estas relaciones entre palabras. Uno de los métodos de embeddings preentrenados más utilizados es **Word2Vec**, desarrollado por Tomas Mikolov en Google en 2013. Word2Vec es eficiente y se ha convertido en un estándar para desarrollar **embeddings** de palabras preentrenadas.

Word2Vec es especialmente eficaz para capturar regularidades sintácticas y semánticas en el lenguaje. Cada relación se caracteriza por un desplazamiento vectorial específico. Por ejemplo, las relaciones de género como hombre/mujer se aprenden automáticamente, y con los embeddings vectoriales "Rey - Hombre + Mujer", se obtiene un vector resultante muy cercano a "Reina".

Word2Vec se puede implementar mediante dos modelos de aprendizaje distintos: la **Bolsa Continua de Palabras (CBOW)** y el **Modelo Skip-Gram**. El modelo CBOW predice la palabra actual en función de su contexto, mientras que el modelo Skip-Gram predice las palabras vecinas dada una palabra actual. Ambos modelos se centran en aprender palabras usando su contexto local, definido por una ventana de palabras vecinas.

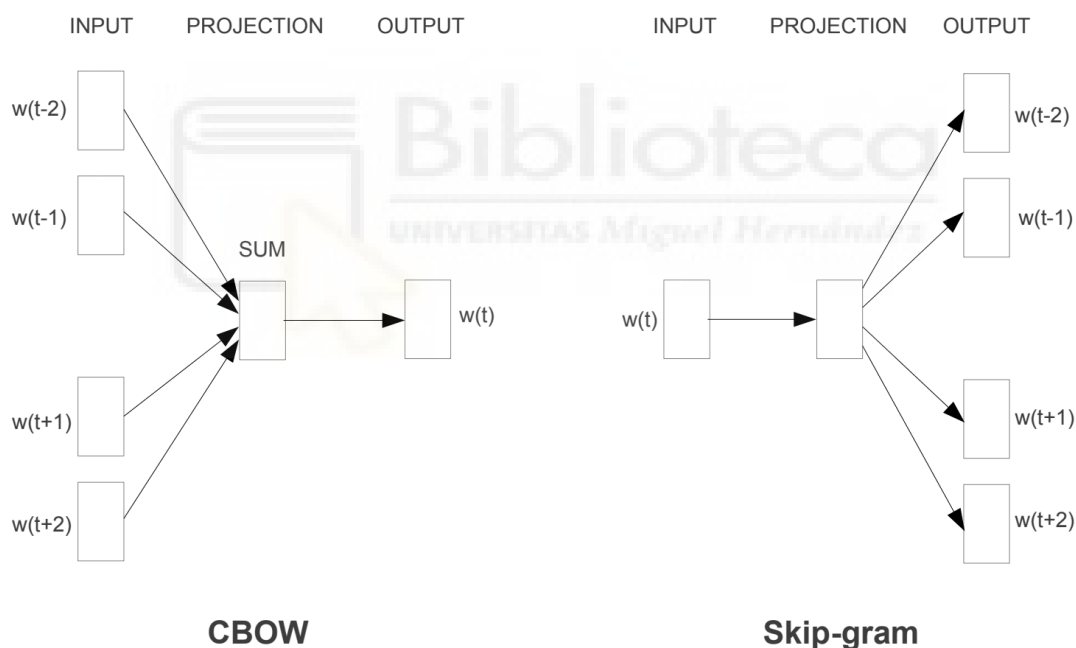


Ilustración 57. Modelos de entrenamiento de Word2Vec

Según Mikolov, Skip-G Según Mikolov, el modelo **Skip-Gram** se comporta bien con una pequeña cantidad de datos y representa bien las palabras raras, mientras que **CBOW** es más rápido y tiene mejores representaciones para palabras más comunes.

La principal ventaja de Word2Vec es su **escalabilidad**, ya que permite aprender embeddings de palabras de calidad de manera eficiente.

Finalmente, la siguiente imagen ilustra la estructura tridimensional de las palabras después de reducir su dimensionalidad con Word2Vec. Se pueden observar diferentes estructuras de palabras relacionadas por su proximidad. Por ejemplo, la palabra "football" tendrá una relación de proximidad con otras palabras como "soccer", "rugby", "fifa", etc [53].

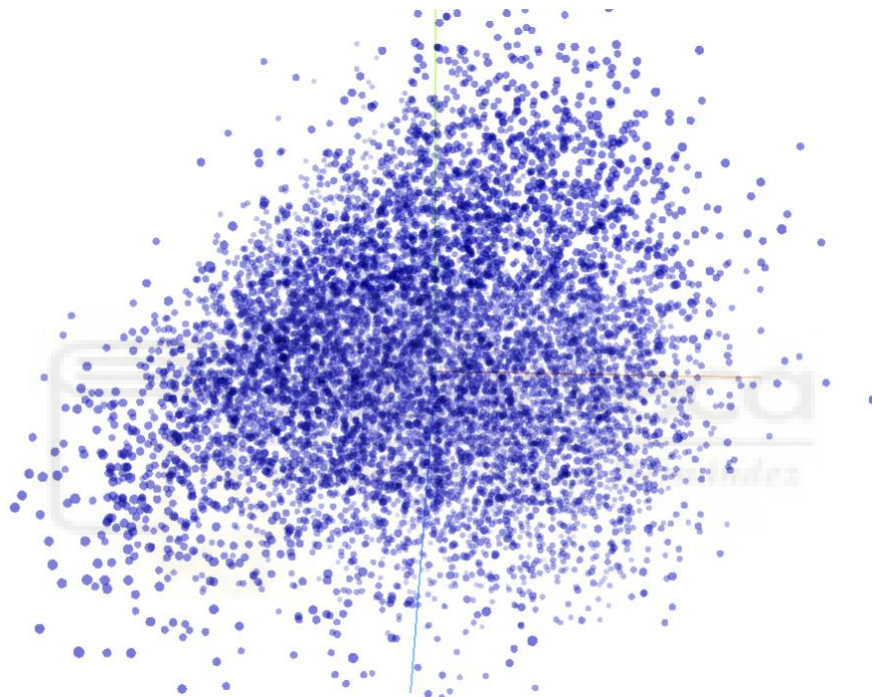


Ilustración 58. Representación tridimensional de embeddings

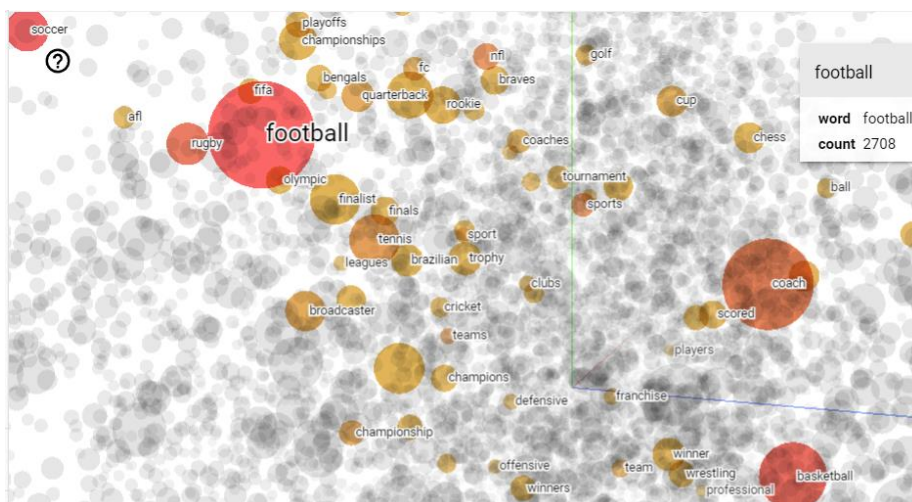


Ilustración 59. Estructura de palabras relacionadas

Hasta este punto, se ha recorrido el proceso desde la obtención, limpieza y depuración de datos hasta su representación numérica, cómo se entienden el contexto y se forman relaciones entre palabras de manera similar a cómo lo hace nuestro cerebro. Todo esto sirve para mejorar la precisión en el procesamiento del texto y obtener mejores resultados. A continuación, se analizarán los algoritmos de aprendizaje automático, basados en redes neuronales, de mayor relevancia en el **Procesamiento del Lenguaje Natural (PLN)**.

5.2 Redes Transformers

En apartados anteriores se ha detallado el **procesamiento del lenguaje natural** mediante machine learning, centrándose en el uso de las **redes neuronales** más utilizadas en este campo, como las **RNN** y las **LSTM**. Todo ese proceso ha sido necesario para una mejor comprensión de este apartado, en el cual se centrará en una red neuronal innovadora en el PLN, la conocida **red neuronal Transformer**.

Esta red es una arquitectura novedosa que tiene como objetivo resolver tareas de **secuencia a secuencia** mientras maneja **dependencias a largo alcance**, es decir, es capaz de entender el contexto y recordarlo. El **Transformer** se ha convertido en la técnica de vanguardia actual para este campo. Su aplicación práctica se encuentra en herramientas de uso común como Google, que utiliza **BERT**, un modelo basado en los **transformers**, en su motor de búsqueda para entender de forma óptima las consultas de los usuarios [40].

5.2.1 Breve historia

Las redes neuronales conocidas como **Transformers** y su mecanismo de atención fueron presentados por primera vez por un equipo de Google en 2017, en un artículo que generó gran repercusión, titulado "**Attention is all you need**". Antes de la aparición de estas redes, las tareas relacionadas con secuencias se manejaban en su mayoría con **Redes Neuronales Recurrentes (RNN)** y algunas de sus variantes.

Las RNN, creadas por David Rumelhart en 1986, presentaban limitaciones importantes en su aplicación práctica en su versión original. Estos problemas surgían cuando se trataba con secuencias largas, lo que generaba el fenómeno del desvanecimiento o explosión del gradiente, problemas que fueron analizados y estudiados en apartados anteriores. La solución que se encontró fueron unas modificaciones en el diseño original, dando lugar a las **LSTM** y las **GRU**. Ambas variantes utilizaban componentes similares a las puertas lógicas para mitigar el problema del gradiente y permitir que estas redes pudieran retener información a lo largo del tiempo.

En 2017, a pesar de que las LSTM y las GRU habían revolucionado el reconocimiento de voz y la traducción automática, seguían presentando limitaciones importantes, como la falta de paralelización, lo que provocaba retrasos considerables en el entrenamiento. Además, no se utilizaba todo el contexto al codificar palabras. Estos problemas fueron abordados y resueltos por los creadores de la red Transformer en su artículo, con la introducción del mecanismo de atención de múltiples cabezas. La idea central del artículo era que, si la red se basaba en mecanismos de atención, ya no era necesario utilizar una arquitectura recurrente, lo que relegaba a este tipo de redes a un segundo plano frente a las nuevas redes, que permitían modelos más estables y fáciles de entrenar.

Más tarde, en 2018, Google presentó **BERT**, un modelo de lenguaje en **TensorFlow** basado en Transformers. En 2019, **OpenAI** se sumó a los avances en PLN y lanzó **GPT-2**, un modelo basado en una arquitectura de transformer ligeramente diferente. Desde entonces, la creación y mejora de modelos basados en Transformers ha sido constante, dando lugar a modelos más grandes alimentados con enormes corpus de texto, como se verá en el siguiente capítulo [54].

5.2.2 Precursores y fundamentos de las redes Transformers

RNN y LSTM

Para comprender mejor la arquitectura de la red neuronal Transformer, se repasará brevemente las Redes Neuronales Recurrentes (RNN) y sus variantes.

Las **RNN** pueden recordar información de entradas anteriores mientras generan salidas. Estos modelos se pueden clasificar en:

1. **Modelos de secuencia de vectores:** se toma un vector de tamaño fijo como entrada y proporcionan una secuencia de cualquier longitud como salida. Un ejemplo es el título de una imagen, donde la entrada es una imagen y la salida es la descripción de la imagen.
2. **Modelo vectorial de secuencia:** pueden tomar una secuencia de cualquier tamaño como entrada y generar un vector de tamaño fijo como salida. Un ejemplo es el análisis de sentimientos, como en las reseñas de películas, donde la entrada es la reseña y la salida es una calificación positiva o negativa.
3. **Modelo de secuencia a secuencia:** son los más utilizados, ya que toman una secuencia como entrada y generan otra secuencia de tamaño variable como salida. Un ejemplo de esto es la traducción de idiomas.

Las RNN son conocidas por su lentitud para entrenar y por los problemas que enfrentan al tratar con secuencias largas, que producen el conocido desvanecimiento del gradiente y el gradiente explosivo. A medida que la distancia entre la información crece, las RNN se vuelven incapaces de conectar, ya que su memoria se desvanece con la distancia.

Para resolver el problema del desvanecimiento del gradiente y el gradiente explosivo, se desarrollaron las **Long Short-Term Memory (LSTM)**, una evolución de las RNN. Las LSTM son capaces de aprender dependencias a largo plazo, es decir, son capaces de recordar. Funcionan bien con secuencias de palabras pequeñas, pero el problema surge cuando el tamaño de las palabras es grande, lo que hace que el entrenamiento sea computacionalmente costoso y lento. Al procesar las entradas de manera secuencial, no pueden beneficiarse del uso de las GPU, que están diseñadas para cálculos paralelos. Este problema de paralelización se resolverá con los Transformers.

Mecanismos de atención

La forma en que los seres humanos enfocan su atención es similar a cómo lo hace una red neuronal. A continuación, se puede observar un modelo simple de secuencia a secuencia. Este procesa sus entradas y genera sus salidas para cada paso de tiempo del codificador y del decodificador, actualizando su estado oculto en función de las entradas y las salidas vistas anteriormente. El estado oculto es lo que se conoce como **vector de contexto**, que se pasa al decodificador.

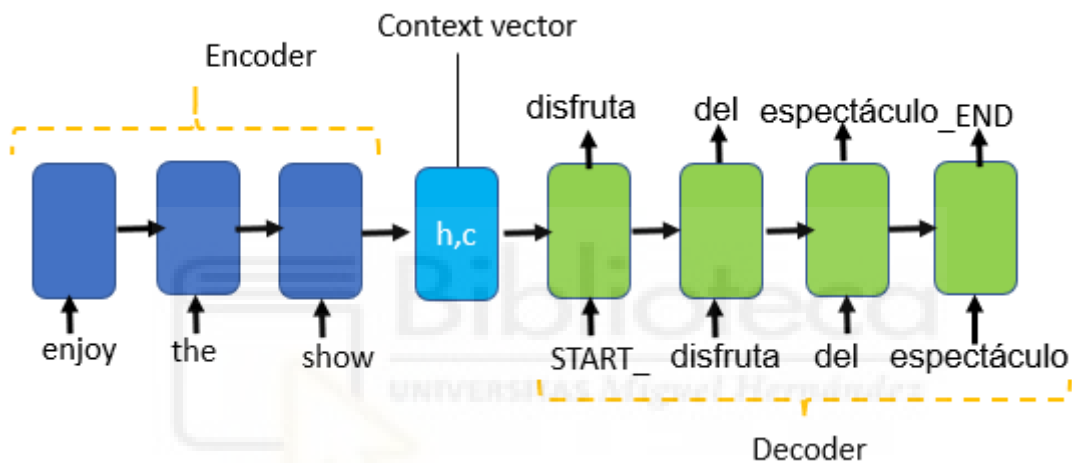


Ilustración 60. Modelo de secuencia a secuencia

Este vector de contexto seguía presentando problemas al tratar con secuencias largas para su entrenamiento. La solución provino de los mecanismos de atención, que permiten al modelo enfocarse en la parte más relevante de la secuencia de entrada según sea necesario.

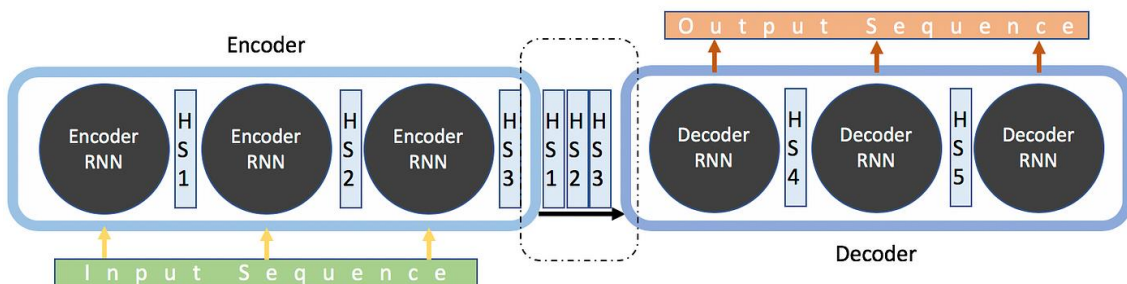


Ilustración 61. Modelo de secuencia a secuencia con atención

Las diferencias entre el modelo de secuencia a secuencia clásico y el modelo de secuencia a secuencia con atención son las siguientes:

1. **El volumen de datos:** en el modelo de atención, a diferencia del clásico, el codificador pasa todos los estados ocultos al decodificador. Anteriormente, el codificador solo enviaba el último estado final al decodificador.
2. **El proceso del decodificador:** cómo se puede observar claramente en la siguiente imagen, el decodificador verifica cada estado oculto recibido, ya que están asociados con una palabra de la frase de entrada. Luego asigna puntuaciones a cada estado oculto y las pasa a la función softmax, que genera una salida entre 0 y 1, amplificando las puntuaciones altas y reduciendo la importancia de las bajas. El resto del proceso se puede observar en la imagen.

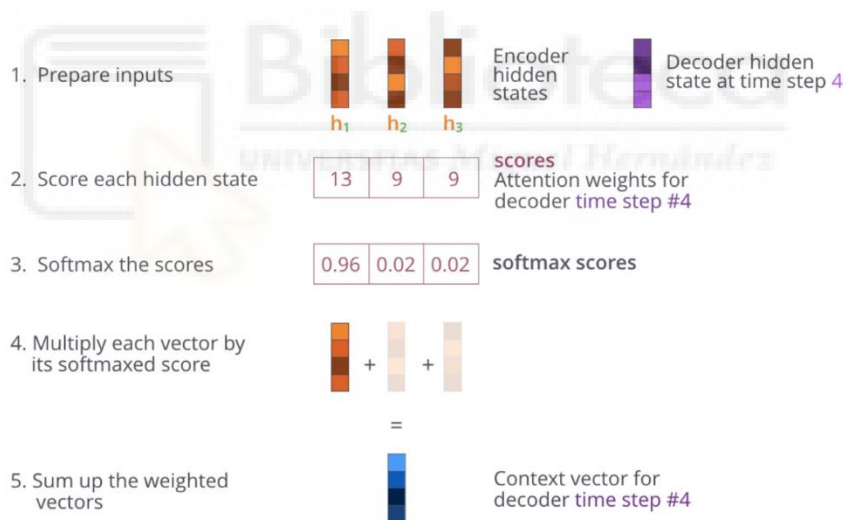


Ilustración 62. Proceso de atención

El proceso completo del decodificador es el siguiente:

- 2.1 Se genera una salida utilizando un vector de estado oculto, h_4 . Esta salida es resultado de la combinación del token **<END>** y un estado oculto inicial del decodificador.
- 2.2 Se realiza el paso de atención, que genera el vector de contexto usando los estados ocultos del codificador y el vector h_4 .
- 2.3 A través de un cálculo matemático, se concatena el vector h_4 y C_4 en un único vector.

2.4 Finalmente, el vector obtenido en el paso anterior se pasa a través de una red neuronal de avance, que indica la palabra de salida para este paso de tiempo. Este conjunto de pasos se repetirá para todos los pasos de tiempo.

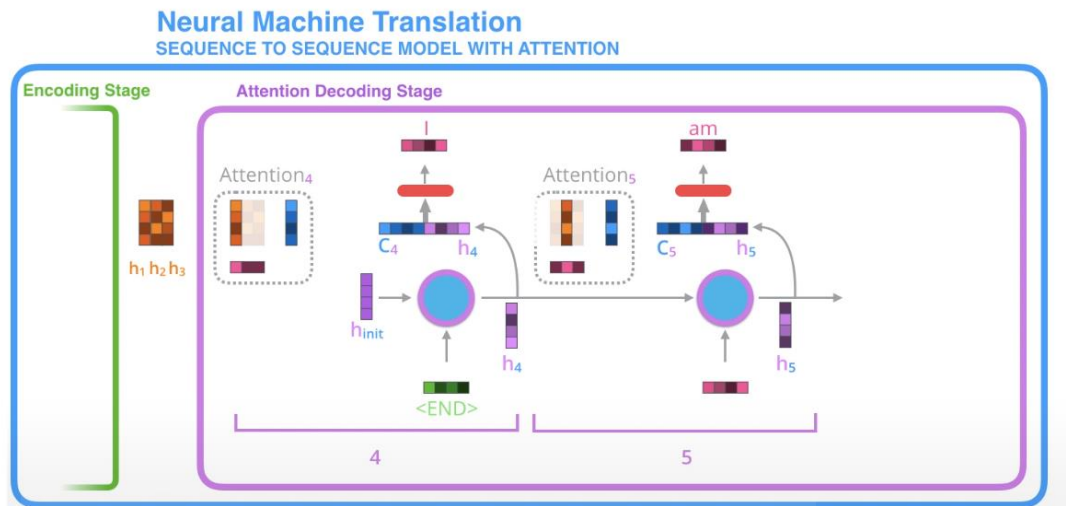


Ilustración 63. Ultimo decodificador

Ahora, se ha resuelto el problema de obtener el contexto y recordar cada palabra en una frase, superando uno de los problemas de las RNN de manera óptima mediante los mecanismos de atención. Sin embargo, todavía es necesaria la paralelización de los datos secuenciales para resolver el problema de la lentitud en el entrenamiento que ocurrió con las RNN. Este concepto se abordará en la siguiente sección [55].

5.2.3 Arquitectura de la Red Neuronal Transformer

La red neuronal **Transformer** posee una arquitectura que, a primera vista, podría ser percibida como complicada. Sin embargo, una característica crucial de esta arquitectura es que se compone de capas repetitivas, lo que, en realidad, simplifica su entendimiento. A lo largo de esta sección, se va a desglosar cada componente de esta arquitectura detalladamente, apoyándose en la imagen que

se presenta a continuación para obtener una visión más completa de su estructura.

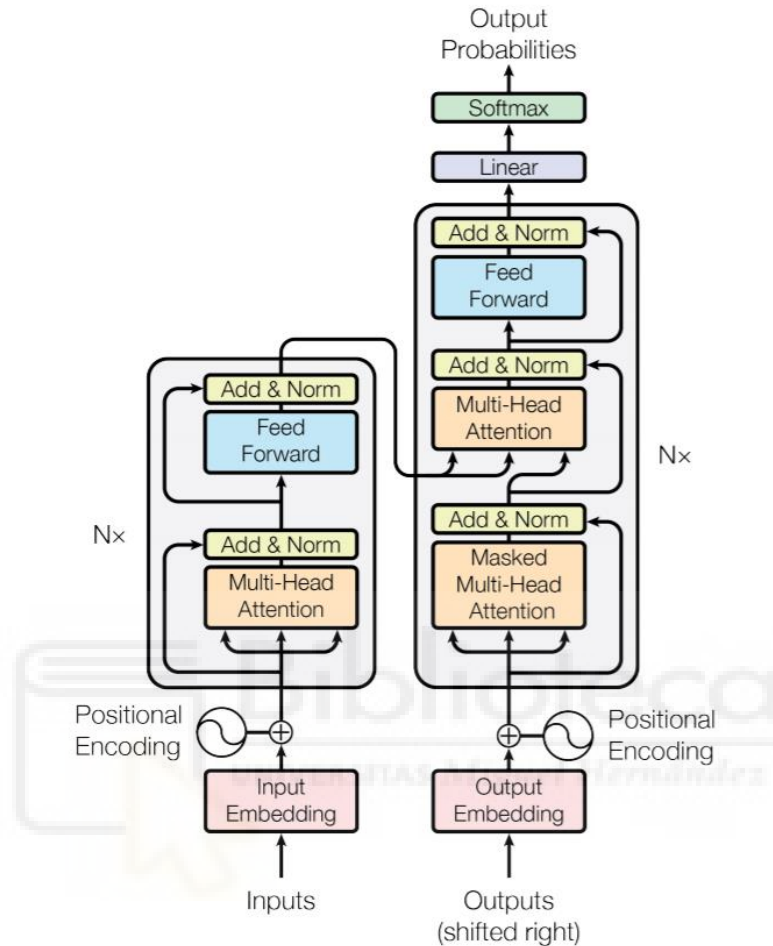


Figure 1: The Transformer - model architecture.

Ilustración 64. Arquitectura de un Transformer

BLOQUE CODIFICADOR

La primera sección que explorar en la arquitectura Transformer es el bloque del codificador. Para entender su funcionamiento, es esencial recordar cómo se introducen los datos en una red neuronal, un concepto conocido como **incrustación o "embedding"**. Este proceso consiste en convertir palabras o frases en vectores numéricos, lo que facilita su manipulación por la red neuronal. La explicación detallada de este proceso se encuentra en un apartado anterior.

Una vez que hemos transformado las frases en vectores numéricos, surge un desafío. Cada palabra puede tener un significado diferente dependiendo de su posición en la frase. Para abordar este problema, se utilizan **codificadores posicionales**, que generan un vector que proporciona contexto según la posición de la palabra en una frase.

Es importante entender que en esta red Transformer, todas las palabras se procesan simultáneamente, un proceso conocido como paralelización. Por lo tanto, se necesitaba un mecanismo que pudiera mantener la información sobre el orden de las palabras en la frase.

La solución propuesta fue agregar un segundo vector del mismo tamaño al vector original de la palabra. En lugar de una codificación binaria, se utiliza una técnica llamada **codificación posicional** que involucra funciones de seno y coseno para representar la posición de la palabra en la frase. Esta codificación posicional puede conceptualizarse como una serie de patrones que podrían parecerse a las ondas si se visualizan, con diferentes posiciones de las palabras creando diferentes patrones. A través de este método, la red neuronal puede entender la secuencia de las palabras en las entradas, lo cual es fundamental para entender el significado del texto en muchos contextos.

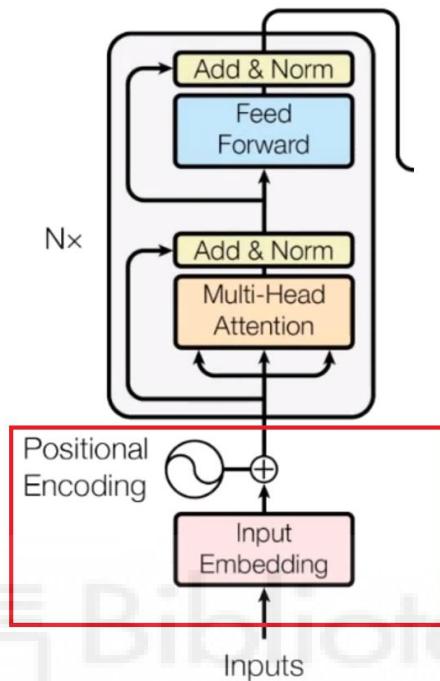


Ilustración 65. Bloque codificador

Parte de atención de varios cabezales o *Multi-Head Attention Part*

Este segmento, también conocido como Auto-Atención, es el núcleo tanto del bloque codificador como de la totalidad de la arquitectura Transformer.

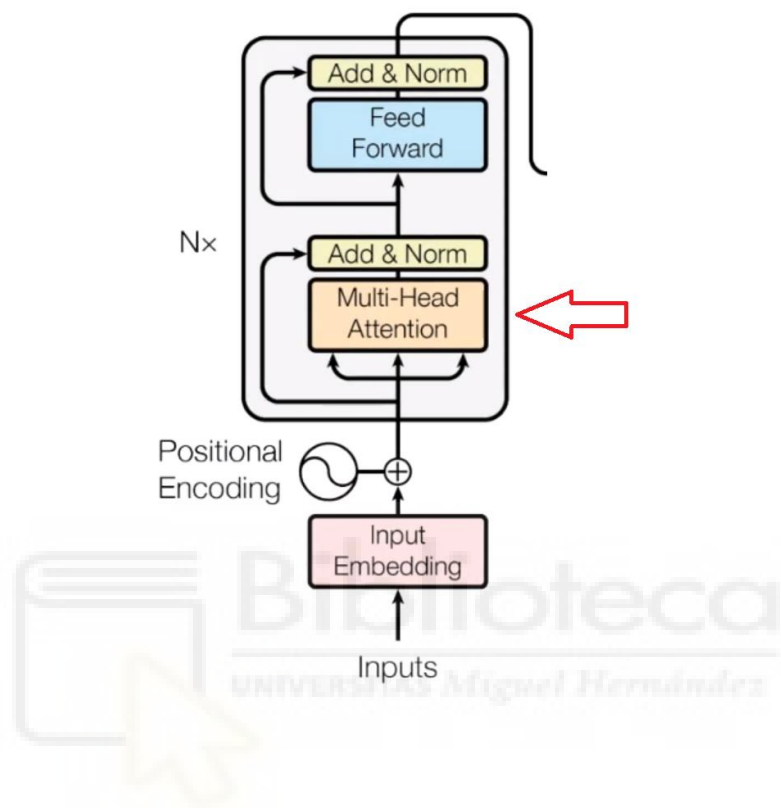


Ilustración 66. Multi-Head Attention Part

La **Auto-Atención** básicamente funciona estudiando qué tan relevante es una palabra en comparación con las otras palabras en una frase. Esta relevancia se captura en lo que se conoce como un **vector de atención**. Para cada palabra, se genera un vector de atención que establece su relación con las otras palabras en la frase. Sin embargo, hay un desafío: la importancia de cada palabra en sí misma tiende a ser más significativa que su relación con las otras palabras.

Para solucionar esto, se crean varios vectores de atención para cada palabra y luego se calcula un promedio ponderado de estos. Este promedio ponderado es el vector de atención final para esa palabra. Esta es la razón por la que se refiere a esta parte como **atención de múltiples cabezales**.

Attention : What part of the input should we focus?

	Focus	Attention Vectors
The	→ The big red dog	$[0.71 \ 0.04 \ 0.07 \ 0.18]^T$
big	→ The big red dog	$[0.01 \ 0.84 \ 0.02 \ 0.13]^T$
red	→ The big red dog	$[0.09 \ 0.05 \ 0.62 \ 0.24]^T$
dog	→ The big red dog	$[0.03 \ 0.03 \ 0.03 \ 0.91]^T$

Ilustración 67. Vector de atención

El bloque de atención de múltiples cabezales es conocido por el uso de múltiples vectores de atención, como se muestra en la imagen.

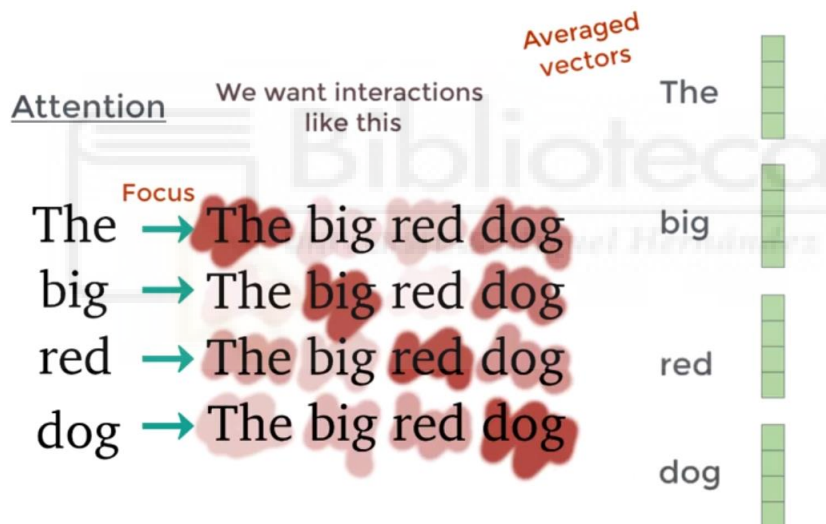


Ilustración 68. Múltiples vectores de atención

Red de alimentación directa o Feed Forward Network

El último componente del bloque codificador es la red de propagación hacia delante. Este bloque es relevante por la presencia de una red neuronal de avance que se aplica a cada vector de atención individualmente. Esta red se encarga de transformar el vector de atención de tal manera que la siguiente capa, ya sea de codificación o decodificación, pueda procesarlo.

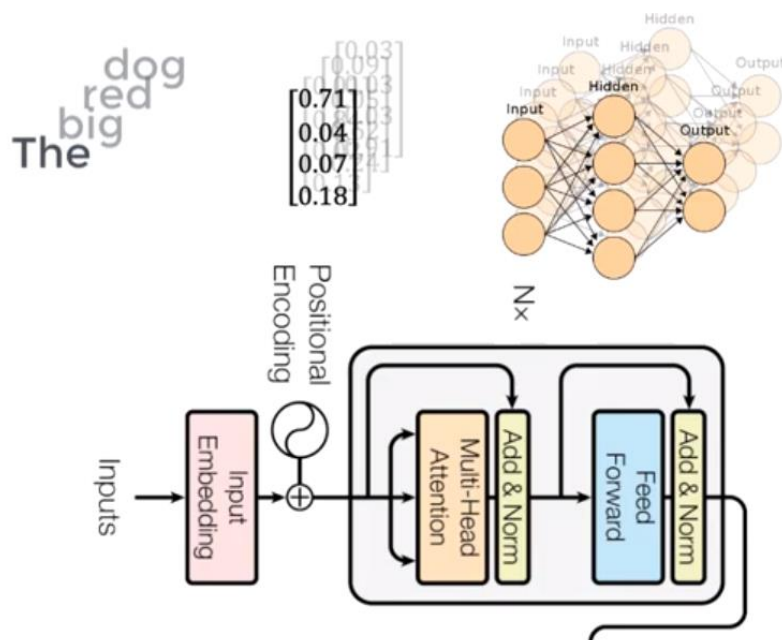


Ilustración 69. Feed Forward Network

Este bloque procesa los vectores de atención de manera individual. Debido a que estos vectores de atención no dependen unos de otros, se puede aplicar la estrategia de paralelización. Esto significa que todas las palabras pueden ser introducidas simultáneamente al bloque de codificación, generando los vectores codificados en paralelo. Este enfoque optimiza la velocidad de cálculo durante el proceso de entrenamiento, permitiendo que se complete más rápido.

BLOQUE DECODIFICADOR

Al observar el bloque decodificador, uno puede notar que es bastante similar al bloque codificador, ya que las salidas de la predicción son las entradas para el decodificador, permitiendo que este realice su entrenamiento. En las siguientes secciones, se detallarán las partes del bloque decodificador que difieren de las del bloque codificador y se omitirá la explicación de las partes que son iguales y, por ende, ya fueron analizadas en el apartado anterior.

Inicialmente, tenemos el embedding de los datos de entrada y la manera de determinar la posición de las palabras a través del **codificador posicional**.

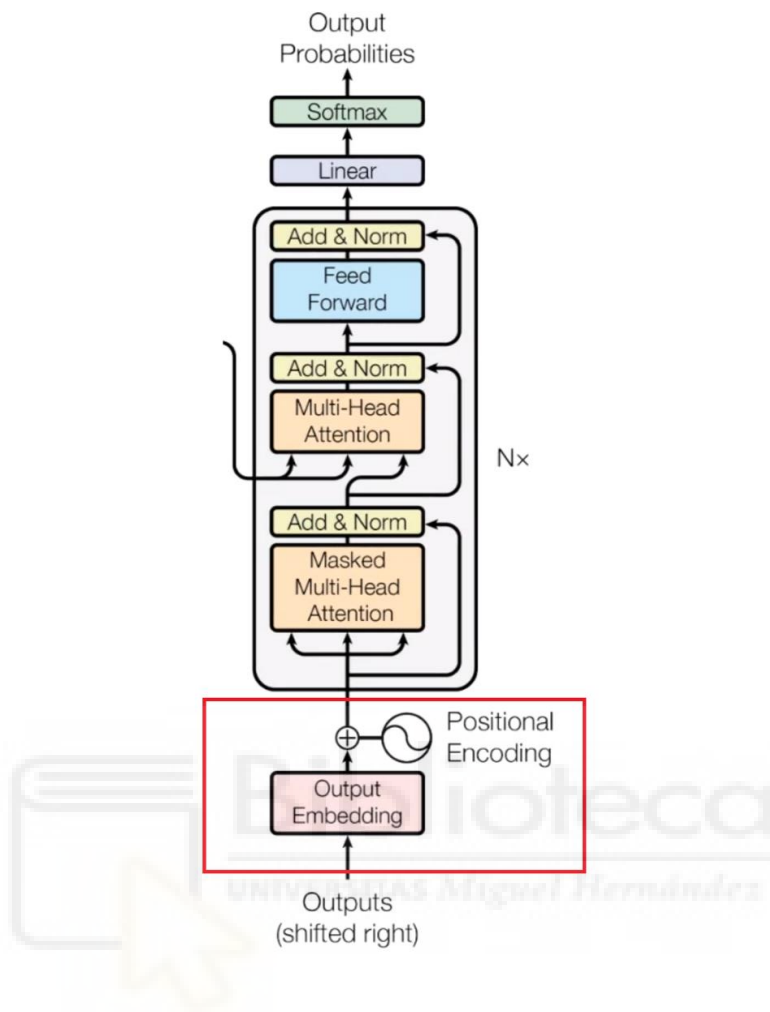


Ilustración 70. Bloque decodificador

Parte de atención de múltiples cabezas enmascarada o Masked Multi-Head Attention

Parte de atención de múltiples cabezas enmascarada o Masked Multi-Head Attention

En términos sencillos, esta etapa se centra en aprender de las palabras anteriores para predecir la siguiente en la secuencia. Para hacer esto, enmascara, es decir, "oculta" la palabra que está tratando de predecir, permitiendo que la red aprenda basándose únicamente en las palabras previas.

Este "enmascaramiento" se logra convirtiendo ciertos elementos de la matriz en ceros durante el proceso de paralelización. Esta acción asegura que la red de atención no pueda hacer uso de los elementos enmascarados.

Multi-headed Attention

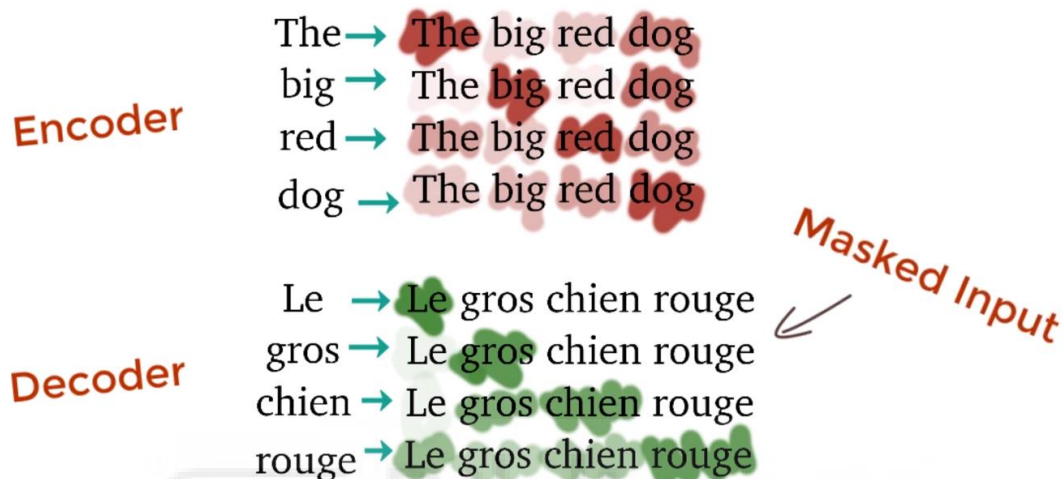


Ilustración 71. Masked Multi-Head Attention

En el bloque siguiente, los vectores de atención generados en la capa previa se combinan con los vectores del bloque codificador en un bloque conocido como **atención codificador-decodificador**. Este bloque establece una correlación entre las palabras que se introducen tanto en el codificador como en el decodificador.

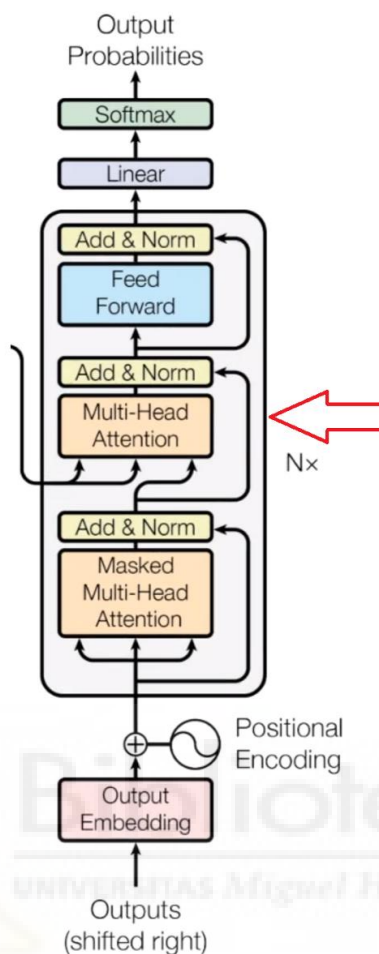


Ilustración 72. Bloque de atención codificador-decodificador

Finalmente, cada vector de atención pasa por las unidades de avance previamente mencionadas, con el objetivo de ser transformados de manera que otro bloque decodificador o una capa lineal pueda aceptarlos.

Después, el resultado pasa por una capa lineal que expande las dimensiones en términos de cantidad de palabras en el resultado obtenido. Finalmente, la capa Softmax convierte el resultado en una distribución de probabilidades, que es más fácil de interpretar para los humanos [55].

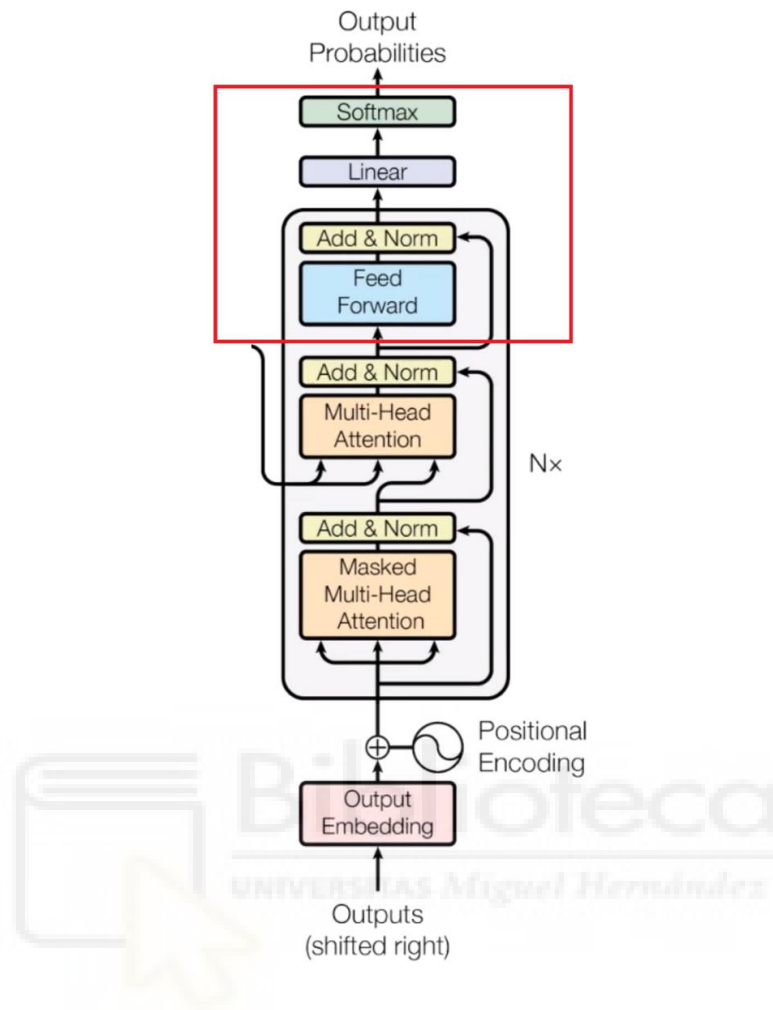


Ilustración 73. Bloque de salida del decodificador

5.3 Desarrollos y desafíos

En esta sección se presentarán cuatro implementaciones distintas del modelo **Transformer**, cada una centrada en la resolución de problemas específicos dentro del ámbito del procesamiento del lenguaje natural. No solo se destacará el uso práctico y las soluciones que aporta el modelo **Transformer**, sino que se proporcionará una visión honesta y transparente de los retos y obstáculos encontrados en el camino. A pesar de las dificultades inherentes a la aplicación de estos avanzados modelos de IA, se han realizado esfuerzos para superar estos desafíos y maximizar la eficacia de las soluciones propuestas con las herramientas y conocimientos disponibles. A lo largo de esta sección, se

explorarán estos desarrollos y se discutirán en profundidad las lecciones aprendidas, los éxitos alcanzados y las dificultades superadas.

5.3.1 Desarrollo 1: traducción automática

En esta primera implementación, se aborda el desafío de la traducción automática, un problema notoriamente complejo dentro del **Procesamiento del Lenguaje Natural**. La solución propuesta se basa en el uso completo de la **arquitectura Transformer**, reconocida por su excepcional desempeño en tareas similares. A través de la selección cuidadosa de un conjunto de datos apropiado, un preprocesamiento de datos preciso y un entrenamiento adecuado, se alcanzan los resultados más destacados de todos los desarrollos presentados, los cuales serán detallados en la sección correspondiente.

Uno de los desafíos más significativos a los que se enfrentó en este desarrollo fue la elección de un conjunto de datos y la necesidad de preprocesar esos datos de manera eficaz. A pesar de estos desafíos, este desarrollo explora cómo, con la utilización de toda la estructura del **Transformer**, es posible superar estas dificultades y conseguir buenos resultados en la traducción automática.

Como objetivo principal de este desarrollo, se buscó mostrar la implementación completa de un Transformer, lo que se logró con éxito y contribuyó significativamente a los resultados obtenidos.

5.3.1.1 Dataset Utilizado

Para el desarrollo de este proyecto se utiliza un conjunto de datos disponible en **ManyThings.org**, una plataforma online destinada al aprendizaje de idiomas. El conjunto de datos seleccionado se titula "**Spanish - English**" y es apropiado para tareas de traducción automática.

Este conjunto de datos consta de 139,705 registros y 2 columnas:

- **Inglés:** La frase en inglés.
- **Español:** La frase correspondiente en español.

La base de datos se puede encontrar y descargar gratuitamente en el siguiente enlace: <https://www.manythings.org/anki/>

Los datos, alojados en un archivo .txt, son importados y convertidos en un **DataFrame de pandas** para su posterior manipulación y análisis. El conjunto de datos se encuentra en buen estado, sin valores nulos ni duplicados, lo cual garantiza su idoneidad para la tarea de traducción automática.

El volumen considerable del conjunto de datos permite realizar una división equilibrada en segmentos de entrenamiento, validación y prueba, asegurando que cada segmento contenga información suficiente para la construcción y validación del modelo.

5.3.1.2 Visualización de los datos

La construcción de un modelo efectivo de **traducción automática**, como la red **Transformer**, requiere un profundo entendimiento de los datos de entrada. A través de la visualización y análisis detallado de estos datos, se puede obtener información valiosa que puede ser crucial para el entrenamiento y rendimiento del modelo. Este análisis se centrará en dos aspectos fundamentales: la distribución de la longitud de las frases y los n-gramas más comunes en los conjuntos de datos en inglés y español.

Distribución de la longitud de las frases

Los resultados del análisis de longitud de las frases son cruciales para la implementación de la red Transformer para la traducción automática. La mayoría de las frases en ambos idiomas tienen entre 1 y 8 palabras, lo que es un rango óptimo para el rendimiento del Transformer, dado que estos modelos manejan eficazmente secuencias más cortas. No obstante, las frases más largas podrían plantear desafíos, ya que los Transformers pueden tener dificultades para capturar las dependencias de largo alcance entre las palabras.

En cuanto a las diferencias entre los dos idiomas, es relevante tener en cuenta que la longitud media de las frases en español tiende a ser ligeramente mayor que en inglés. Esto puede afectar la efectividad del modelo **Transformer**, ya que el español es estructuralmente más complejo y puede requerir una comprensión más profunda de las relaciones de las palabras. Por lo tanto, estas diferencias deberían considerarse durante el ajuste y entrenamiento del modelo para garantizar una **traducción de alta calidad**.

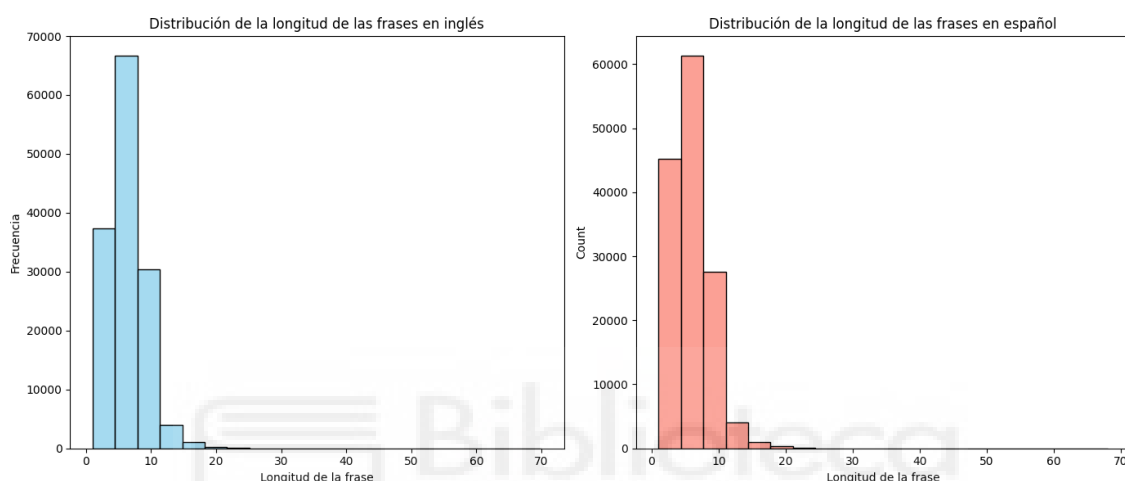


Ilustración 74. Comparación de la longitud de las frases

Análisis de N-gramas más comunes

La lista de n-gramas más comunes en inglés y español proporciona una visión sobre las palabras más recurrentes en ambos idiomas dentro de tu conjunto de datos. En ambos idiomas, las palabras más comunes suelen ser artículos, preposiciones y pronombres, lo cual es bastante estándar para la mayoría de los conjuntos de datos textuales. El nombre "Tom" destaca en ambas listas, sugiriendo que el dataset podría contener una gran cantidad de oraciones o diálogos relacionados con un personaje llamado Tom.

En el contexto de la construcción de un modelo Transformer para la traducción automática, estos **n-gramas comunes** juegan un papel crítico. Dado que los Transformers aprenden a traducir basándose en patrones en los datos de entrenamiento, estas palabras comunes tendrán un impacto significativo en las predicciones del modelo. Esta información puede guiar el proceso de preprocesamiento, asegurándose de que estas palabras frecuentes no sean

consideras como palabras de parada (stop words), ya que son vitales para el significado de muchas frases.

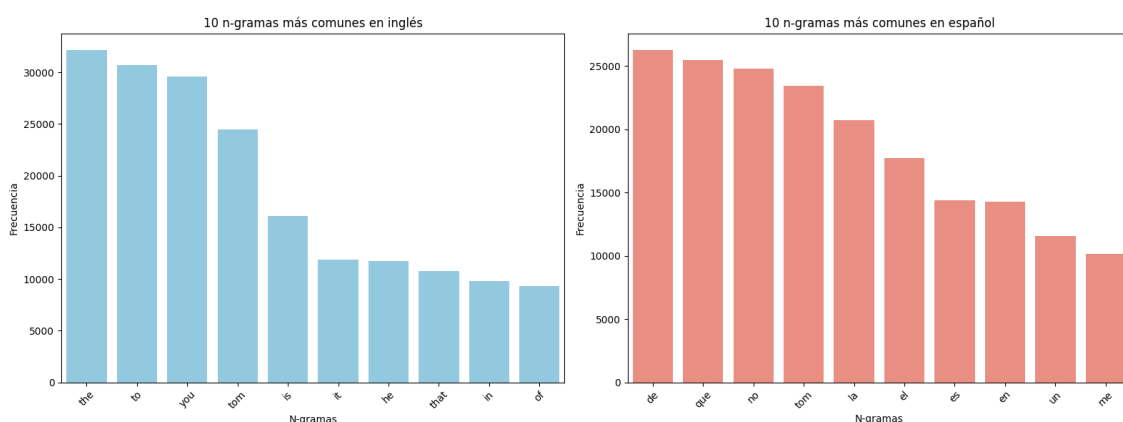


Ilustración 75. Distribución de N-gramas más Comunes en Inglés y Español

5.3.1.3 Preprocesamiento de los datos

En esta sección, se trabaja en la preparación de nuestros datos para el entrenamiento del modelo. Estos pasos son esenciales para garantizar que nuestros datos están en el formato correcto y son adecuados para el **entrenamiento del modelo**.

Normalización de los datos

En este paso, se toma el conjunto de datos y se realiza una serie de transformaciones para garantizar que los datos se encuentren en un formato uniforme y adecuado para el procesamiento posterior. Las tareas realizadas durante la normalización incluyen:

- **Eliminar caracteres no deseados y convertir todas las letras a minúsculas.**

Esta conversión a minúsculas asegura que se traten de manera uniforme las palabras, independientemente de su posición en la oración. Esto ayuda a reducir la dimensionalidad de los datos y mejora la eficacia del modelo.

- **Dividir el texto en inglés y español.**

Esto se hace para preparar los datos para la tarea de traducción bidireccional. Cada idioma se procesará por separado en las etapas posteriores.

- **Agregar tokens de inicio y fin a las oraciones en español.**

Estos tokens ayudan al modelo a identificar cuándo comienza y termina una oración. Esto es crucial para el proceso de traducción ya que proporciona al modelo la estructura necesaria para generar traducciones precisas.

Estos pasos aseguran que los datos estén en un formato adecuado para los pasos de procesamiento posteriores.

```
# Normaliza el texto, divide en inglés y español, y añade tokens de inicio y fin a las oraciones en español.
def normalize(line):
    line = unicodedata.normalize("NFKC", line.strip().lower())
    line = re.sub(r"^(^[^ \w])?!\\s)", r" \1 ", line)
    line = re.sub(r"(\\s[^\w])?!\\s)", r" \1 ", line)
    line = re.sub(r"(?!\\s)([^\w])$", r" \1", line)
    line = re.sub(r"(?!\\s)([^\w]\\s)", r" \1", line)
    eng, spa = line.split("\\t", 1)
    spa = spa.split("\\t")[0]
    spa = "[start] " + spa + " [end]"
    return eng, spa

# Aplicar la normalización en el dataset
with open(dataset_path) as fp:
    text_pairs = [normalize(line) for line in fp]
```

Ilustración 76. Proceso de Normalización de los Datos

Vectorización de los datos

La vectorización es el proceso de convertir el texto en una **representación numérica** que pueda ser entendida por el modelo de aprendizaje automático.

Para la vectorización, se utiliza la capa **TextVectorization** de TensorFlow. Esta capa convierte el texto en secuencias de números enteros que representan las palabras. Cada número entero corresponde a una palabra específica en el vocabulario del conjunto de datos.

La capa TextVectorization de TensorFlow se adapta al conjunto de entrenamiento, aprendiendo el vocabulario y mapeando cada palabra a un número entero correspondiente. Este proceso de mapeo se conoce como **codificación de palabras (word encoding)**.

La vectorización se realiza de la siguiente manera:

- **División de los pares de frases normalizadas** en conjuntos de entrenamiento, prueba y validación.

```
random.shuffle(text_pairs)
n_val = int(0.15 * len(text_pairs))
n_train = len(text_pairs) - 2 * n_val
train_pairs = text_pairs[:n_train]
val_pairs = text_pairs[n_train:n_train+n_val]
test_pairs = text_pairs[n_train+n_val:]
```

Ilustración 77. Proceso de División de Datos para Vectorización

Los pares de texto se barajan inicialmente para prevenir cualquier sesgo durante la selección de datos para entrenamiento, validación y prueba. Después, se designa el 15% del total de pares de texto para la validación (n_{val}) y se deduce el 70% para el entrenamiento (n_{train}) restando dos veces la cantidad de datos de validación del total. Finalmente, con estas cantidades definidas, se escogen los conjuntos de datos correspondientes a **entrenamiento, validación y prueba**.

- **Creación de la capa de vectorización y entrenamiento** con el conjunto de entrenamiento.

```
# Crea el vectorizador
eng_vectorizer = TextVectorization(
    max_tokens=vocab_size_en,
    standardize=None,
    split="whitespace",
    output_mode="int",
    output_sequence_length=seq_length,
)
spa_vectorizer = TextVectorization(
    max_tokens=vocab_size_sp,
    standardize=None,
    split="whitespace",
    output_mode="int",
    output_sequence_length=seq_length + 1
)
```

Ilustración 78. Creación y Entrenamiento de la Capa de Vectorización

```
# Entrena la capa de vectorización usando el conjunto de entrenamiento
train_eng_texts = [pair[0] for pair in train_pairs]
train_spa_texts = [pair[1] for pair in train_pairs]
eng_vectorizer.adapt(train_eng_texts)
spa_vectorizer.adapt(train_spa_texts)
```

Ilustración 79. Incorporación de la Capa de Vectorización al Modelo

- **Aplicación de la capa de vectorización** a las frases en inglés y español para convertirlas en representaciones numéricas.

```
def format_dataset(eng, spa):
    eng = eng_vectorizer(eng)
    spa = spa_vectorizer(spa)
    source = {"encoder_inputs": eng, "decoder_inputs": spa[:, :-1]}
    target = spa[:, 1:]
    return (source, target)
```

Ilustración 80. Transformación de Frases a Formato Numérico

Después de vectorizar los datos, se crean **conjuntos de datos de TensorFlow** que se utilizan para alimentar el modelo durante el entrenamiento y la validación. Cada conjunto de datos contiene entradas para el codificador y el decodificador, así como las etiquetas objetivo para el entrenamiento.

```
def make_dataset(pairs, batch_size=64):
    eng_texts, spa_texts = zip(*pairs)
    dataset = tf.data.Dataset.from_tensor_slices((list(eng_texts), list(spa_texts)))
    return dataset.shuffle(2048).batch(batch_size).map(format_dataset).prefetch(16).cache()

train_ds = make_dataset(train_pairs)
val_ds = make_dataset(val_pairs)
```

Ilustración 81. Datos para Entrenamiento y Validación

Este proceso asegura que los datos estén en el formato correcto y sean adecuados para el **entrenamiento del modelo**. La **manipulación de datos** es un paso crucial en cualquier proyecto de aprendizaje automático, ya que la calidad y la preparación de los datos pueden tener un gran impacto en el rendimiento del modelo.

Análisis de los datos

Antes de procesar los datos, es útil obtener una visión general de la distribución y la estructura de estos. Para ello, se calcula **la longitud de las frases en términos de tokens para el inglés y el español** y se traza un histograma de estas longitudes. Se proporciona información sobre la cantidad de ejemplos en función de la longitud de los tokens.

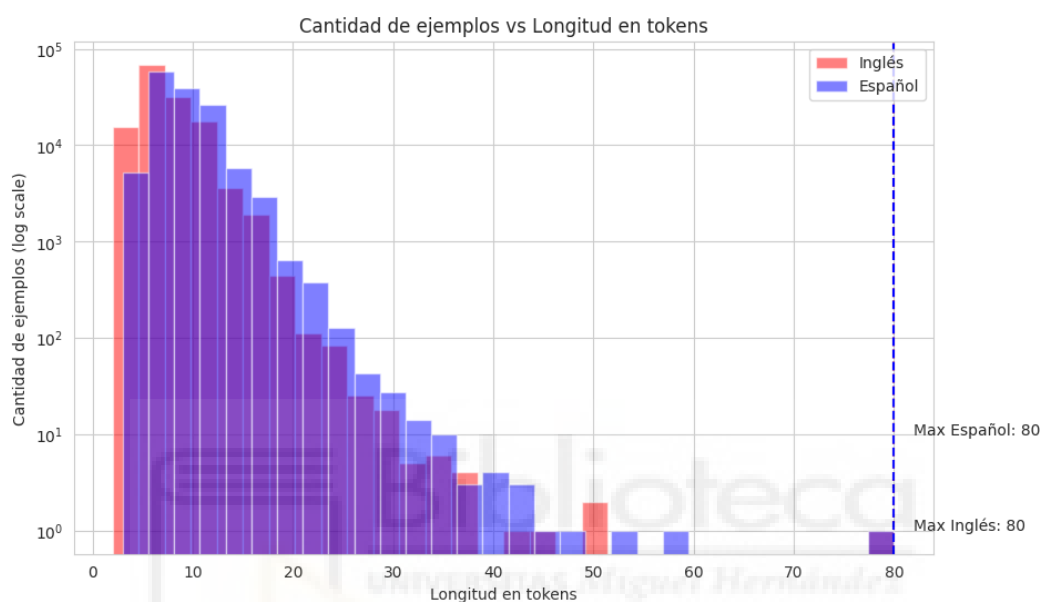


Ilustración 82. Distribución de Longitud de Frases en Términos de Tokens

Los datos del histograma muestran la frecuencia de las longitudes de las frases en inglés y español en términos de la cantidad de tokens. En ambos idiomas, la mayoría de las frases tienen entre 2 y 10 tokens, siendo las frases de 6 tokens las más comunes en inglés y las de 8 tokens en español. Esto indica que la mayoría de las frases son relativamente cortas, lo que es beneficioso para el rendimiento de los modelos Transformer, que pueden tener dificultades para manejar frases más largas debido a limitaciones en la **capacidad de atención**.

Sin embargo, también se observa que algunas frases tienen una longitud significativamente mayor, llegando hasta 80 tokens en ambos idiomas. Estas frases más largas pueden representar un desafío para el modelo Transformer, especialmente debido a las dependencias de largo alcance que pueden estar presentes en ellas. Además, se observa que la distribución de longitudes de frases en español es ligeramente más ancha que en inglés, lo que indica que las

frases en español tienden a ser más largas en promedio. Esto puede ser un factor para tener en cuenta durante el entrenamiento y ajuste del modelo.

5.3.1.4 Creación del modelo Transformer

Matriz de codificación posicional

La matriz de **codificación posicional** es esencial para mantener un registro de la posición de cada palabra en la secuencia de entrada. Los Transformers, a diferencia de los modelos de redes neuronales recurrentes como las **RNN y las LSTM**, no tienen una noción inherente de orden secuencial en los datos de entrada. Por lo tanto, se agrega información adicional en forma de codificaciones posicionales para compensar esta carencia. Cada posición en la secuencia de entrada recibe una codificación única y esta codificación se agrega a la representación vectorial de la palabra (**embedding**) correspondiente a esa posición.

Capa de codificación posicional

En el aprendizaje automático y especialmente en los modelos de lenguaje como los Transformers, es esencial tener en cuenta la posición de cada palabra en una oración. Aquí es donde la función "**pos_enc_matrix**" entra en juego. Esta función crea una matriz única que asigna un valor a cada palabra basado en su posición en la oración. Por ejemplo, podría dar a la primera palabra el valor 1, a la segunda palabra el valor 2, y así sucesivamente.

Por otro lado, la clase "**PositionalEmbedding**" se encarga de combinar esta información posicional con la representación de las palabras, también conocida como embeddings. Estos embeddings son vectores de números que representan el significado de una palabra. La clase "**PositionalEmbedding**" simplemente suma el **embedding** de una palabra con su valor posicional, para obtener un nuevo vector que contiene información tanto sobre el significado de la palabra como sobre su ubicación en la oración.

Por último, la función "**compute_mask**" dentro de "**PositionalEmbedding**" ayuda al modelo a ignorar las palabras de relleno o padding, que son palabras

sin significado añadidas para hacer que todas las oraciones tengan la misma longitud.

Entonces, en resumen, "**pos_enc_matrix**" y "**PositionalEmbedding**" trabajan juntas para asegurar que el modelo comprenda tanto el significado de cada palabra como su posición en la oración, lo que mejora la interpretación del modelo de las oraciones.

```
def pos_enc_matrix(L, d, n=10000):  
    assert d % 2 == 0  
    d2 = d//2  
    P = np.zeros((L, d))  
    k = np.arange(L).reshape(-1, 1)  
    i = np.arange(d2).reshape(1, -1)  
    denom = np.power(n, -i/d2)  
    args = k * denom  
    P[:, ::2] = np.sin(args)  
    P[:, 1::2] = np.cos(args)  
    return P
```

Ilustración 83. Generación de la Matriz de Codificación Posicional


```

class PositionalEmbedding(tf.keras.layers.Layer):
    def __init__(self, sequence_length, vocab_size, embed_dim, **kwargs):
        super().__init__(**kwargs)
        self.sequence_length = sequence_length
        self.vocab_size = vocab_size
        self.embed_dim = embed_dim
        self.token_embeddings = tf.keras.layers.Embedding(
            input_dim=vocab_size, output_dim=embed_dim, mask_zero=True
        )
        matrix = pos_enc_matrix(sequence_length, embed_dim)
        self.position_embeddings = tf.constant(matrix, dtype="float32")

    def call(self, inputs):
        embedded_tokens = self.token_embeddings(inputs)
        return embedded_tokens + self.position_embeddings

    def compute_mask(self, *args, **kwargs):
        return self.token_embeddings.compute_mask(*args, **kwargs)

    def get_config(self):
        config = super().get_config()
        config.update({
            "sequence_length": self.sequence_length,
            "vocab_size": self.vocab_size,
            "embed_dim": self.embed_dim,
        })
        return config

```

Ilustración 84. Implementación de la Capa de Embedding Posicional

Atención de múltiples cabezas

La **atención de múltiples cabezas** es una técnica clave en los modelos de Transformers que permite que el modelo se enfoque en diferentes partes de la entrada al mismo tiempo. Esto es útil para entender mejor la relación entre las palabras y su contexto.

La función "**self_attention**" implementa la atención de múltiples cabezas. Esta función toma una secuencia de entrada y aplica la atención a sí misma. Esto significa que el modelo examina la secuencia y calcula una puntuación para cada palabra basada en su relación con las demás. Estas puntuaciones se utilizan luego para recalibrar la importancia de cada palabra en la secuencia. Además, la función "**self_attention**" incluye una normalización de capa y una adición para evitar que las puntuaciones se salgan de control y mantener la estabilidad numérica.

```

def self_attention(input_shape, prefix="att", mask=False, **kwargs):
    inputs = tf.keras.layers.Input(shape=input_shape, dtype='float32',
                                    name=f"{prefix}_in1")
    attention = tf.keras.layers.MultiHeadAttention(name=f"{prefix}_attn")
    norm = tf.keras.layers.LayerNormalization(name=f"{prefix}_norm1")
    add = tf.keras.layers.Add(name=f"{prefix}_add1")
    attout = attention(query=inputs, value=inputs, key=inputs,
                      use_causal_mask=mask)
    outputs = norm(add([inputs, attout]))
    model = tf.keras.Model(inputs=inputs, outputs=outputs, name=f"{pref
    return model

seq_length = 20

```

Ilustración 85. Implementación de la Atención de Múltiples Cabezas

Por otro lado, la función "**cross_attention**" es similar a la "**self_attention**", pero en lugar de aplicar la atención a la misma secuencia, toma en consideración otra secuencia, conocida como contexto. Esto permite que el modelo entienda mejor cómo se relaciona una secuencia con otra. Por ejemplo, en una tarea de traducción, la secuencia de entrada podría ser una oración en inglés y el contexto podría ser la correspondiente oración en español.

```

def cross_attention(input_shape, context_shape, prefix="att", **kwargs):
    context = tf.keras.layers.Input(shape=context_shape, dtype='float32',
                                    name=f"{prefix}_ctx2")
    inputs = tf.keras.layers.Input(shape=input_shape, dtype='float32',
                                    name=f"{prefix}_in2")
    attention = tf.keras.layers.MultiHeadAttention(name=f"{prefix}_attn2", **kwargs)
    norm = tf.keras.layers.LayerNormalization(name=f"{prefix}_norm2")
    add = tf.keras.layers.Add(name=f"{prefix}_add2")
    attout = attention(query=inputs, value=context, key=context)
    outputs = norm(add([attout, inputs]))
    model = tf.keras.Model(inputs=[context, inputs], outputs=outputs,
                            name=f"{prefix}_cross")
    return model

```

Ilustración 86. Implementación de la Atención Cruzada

Para resumir, tanto "**self_attention**" como "**cross_attention**" permiten al modelo prestar atención a diferentes partes de la secuencia de entrada y del contexto, respectivamente, lo que ayuda a entender mejor las relaciones entre las palabras y su entorno.

Para el modelo de “**self_attention**”, se observa cómo la secuencia de entrada se procesa a través de una capa de **Atención de Múltiples Cabezas**, luego se suma a la entrada original, y finalmente se normaliza. Esta ilustración visualiza cómo cada elemento en la secuencia recalcula su representación en relación con los demás, mejorando así su **comprensión del contexto**.

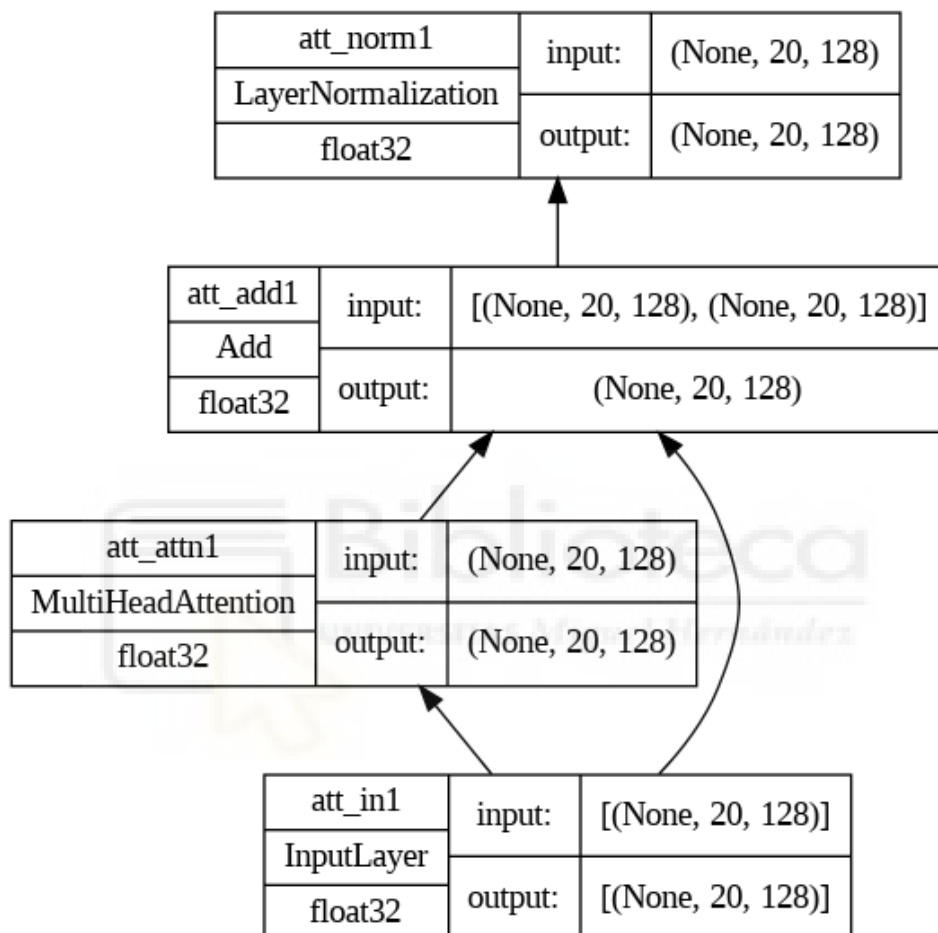


Ilustración 87. Auto-Atención en la Secuencia de Entrada

En cuanto al modelo de **cross_attention**, se muestran dos secuencias: la entrada y el contexto. Ambas se procesan a través de la **capa de Atención de Múltiples Cabezas**, pero la secuencia contextual se utiliza como **clave y valor para la atención**, mientras que la secuencia de entrada se utiliza como consulta. La representación resultante, que se suma a la secuencia de entrada y se normaliza, refleja cómo la entrada se revisa en el contexto de la secuencia contextual.

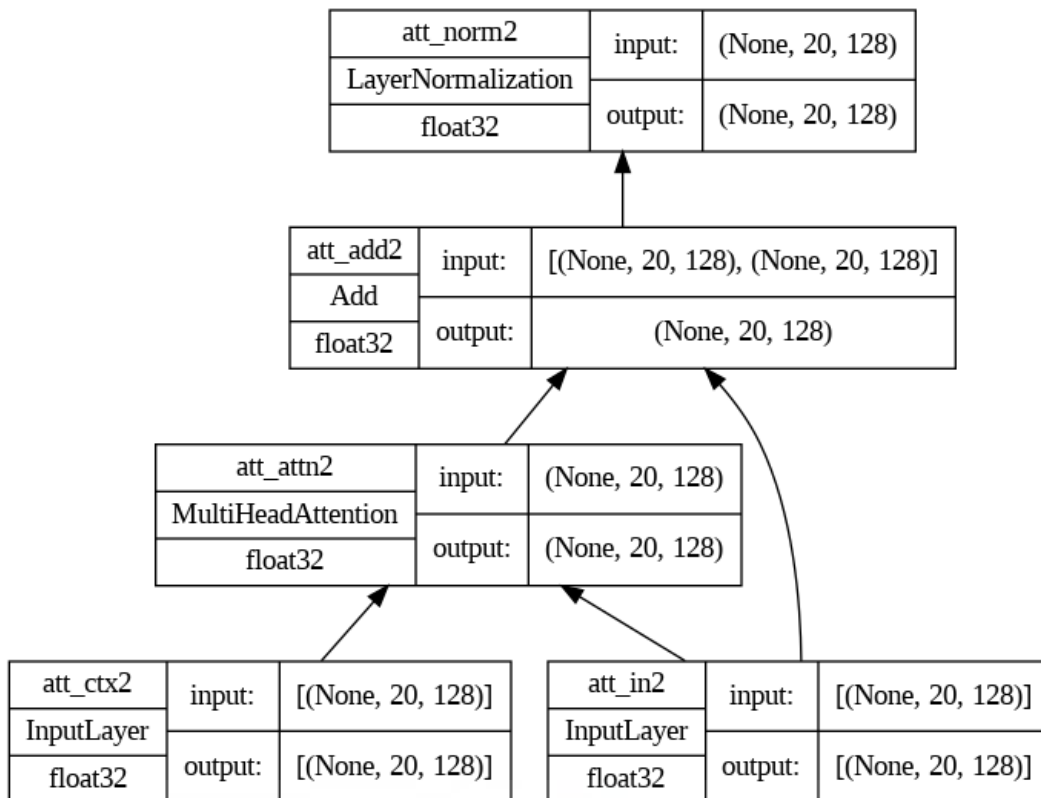


Ilustración 88. Atención Cruzada entre Secuencias de Entrada y Contexto

Red de avance punto a punto (Feed Forward Network)

La Red de Avance Punto a Punto, conocida en inglés como **Feed Forward Network**, es una etapa clave en los Transformers que toma las representaciones mejoradas de la capa de **MultiHeadAttention** y las transforma aún más. Esencialmente, es una red neuronal de dos capas que se aplica de forma independiente a cada posición en la secuencia.

La función "**feed_forward**" construye una red de avance punto a punto. Esta función utiliza dos capas densas: la primera con activación **ReLU** y la segunda sin activación. Entre estas dos capas densas, se aplica una capa de **dropout** para la regularización y la prevención del sobreajuste.

La entrada a esta función es una secuencia, que pasa a través de la primera capa densa, que tiene un número mayor de neuronas (**ff_dim**). Esta ampliación del espacio dimensional permite que la red capture más características de los datos.

Después, los datos pasan por la segunda capa densa, que reduce las dimensiones nuevamente al tamaño original (**model_dim**). Este tipo de estructura, donde las dimensiones se expanden y luego se contraen, a menudo permite que la red capture interacciones complejas en los datos.

La salida de la segunda capa densa se somete a un **dropout** y luego se suma a la entrada original. Finalmente, esta suma pasa por una normalización de capa. El propósito de la suma y la normalización es permitir que la red aprenda a mantener la información de la entrada original si esa es la mejor acción.

```
def feed_forward(input_shape, model_dim, ff_dim, dropout=0.1, prefix="ff"):
    inputs = tf.keras.layers.Input(shape=input_shape, dtype='float32', name=f"{prefix}_in3")
    dense1 = tf.keras.layers.Dense(ff_dim, name=f"{prefix}_ff1", activation="relu")
    dense2 = tf.keras.layers.Dense(model_dim, name=f"{prefix}_ff2")
    drop = tf.keras.layers.Dropout(dropout, name=f"{prefix}_drop")
    add = tf.keras.layers.Add(name=f"{prefix}_add3")
    ffout = drop(dense2(dense1(inputs)))
    norm = tf.keras.layers.LayerNormalization(name=f"{prefix}_norm3")
    outputs = norm(add([inputs, ffout]))
    model = tf.keras.Model(inputs=inputs, outputs=outputs, name=f"{prefix}_ff")
    return model
```

Ilustración 89. Red Neuronal de Avance Punto a Punto

La imagen de la **Red de Avance Punto a Punto** visualiza cómo la secuencia de entrada se procesa a través de las dos capas densas, se aplica el dropout, se suma a la entrada original y finalmente se normaliza.

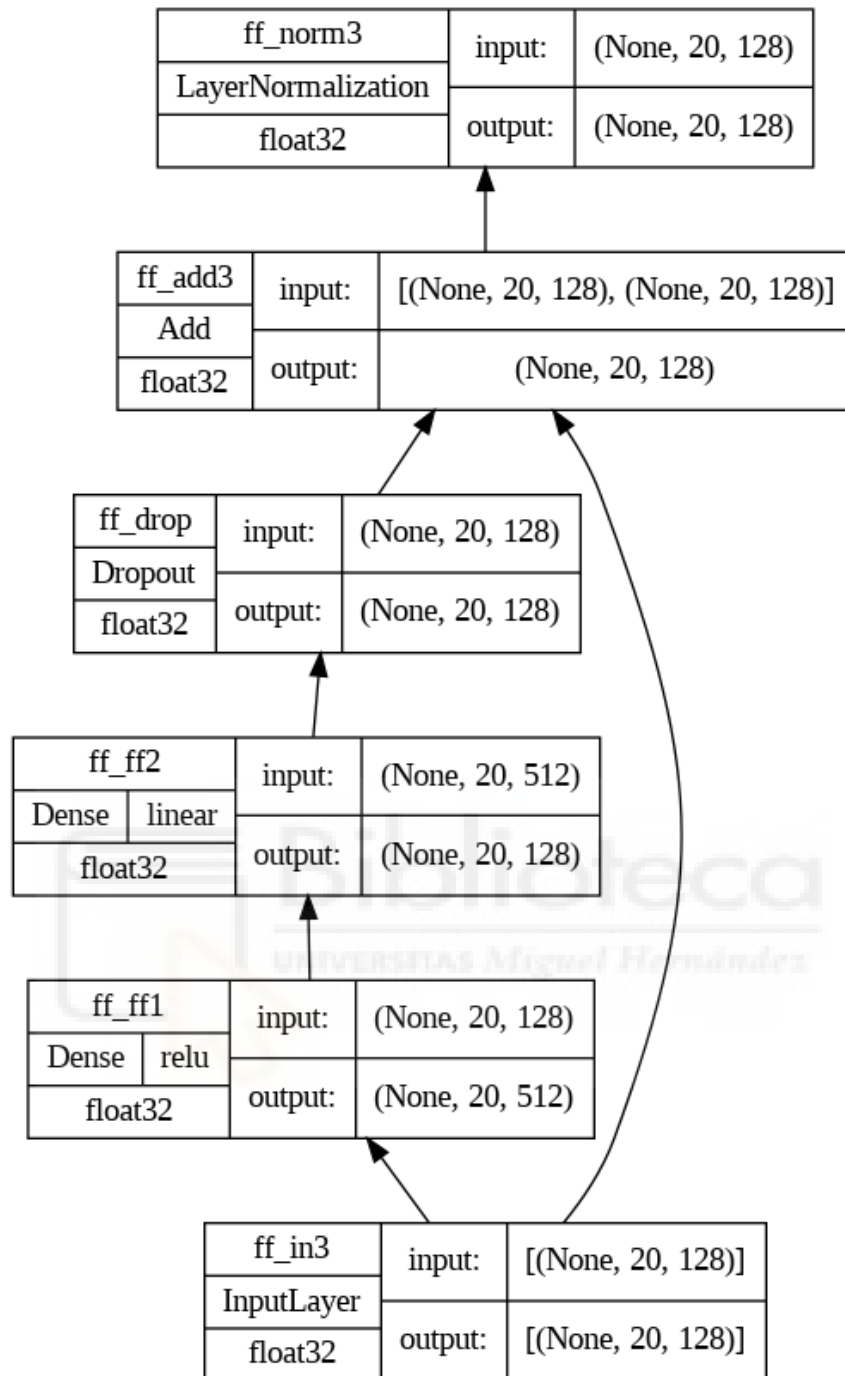


Ilustración 90. Visualización de la Red de Avance Punto a Punto

Capa de Codificación

La unidad de codificación, definida por la función "**encoder**" es una etapa esencial en el modelo **Transformer**. Esta unidad recibe una secuencia de entrada y la procesa a través de una serie de transformaciones para generar una representación más rica y contextualizada de la secuencia. Concretamente, la función "**encoder**" define una unidad de codificación que consta de una capa de

atención de múltiples cabezas, realizada por la función "**self_attention**", seguida de una **red de avance punto a punto**, ejecutada por la función "**feed_forward**".

La atención de múltiples cabezas ayuda al modelo a centrarse en diferentes partes de la secuencia simultáneamente, capturando las interacciones entre las palabras y su contexto. Luego, la red de avance punto a punto transforma estas representaciones atendidas para capturar interacciones más complejas. El resultado es una representación más rica y contextualizada de la secuencia de entrada.

```
def encoder(input_shape, key_dim, ff_dim, dropout=0.1, prefix="enc", **kwargs):
    model = tf.keras.models.Sequential([
        tf.keras.layers.Input(shape=input_shape, dtype='float32', name=f"{prefix}_in0"),
        self_attention(input_shape, prefix=prefix, key_dim=key_dim, mask=False, **kwargs),
        feed_forward(input_shape, key_dim, ff_dim, dropout, prefix),
    ], name=prefix)
    return model
```

Ilustración 91. Unidad de Codificación en el Modelo Transformer

La imagen del codificador muestra cómo la secuencia de entrada se procesa a través de la capa de atención ("**self_attention**") y la red de avance punto a punto ("**feed_forward**") para generar la secuencia de salida.

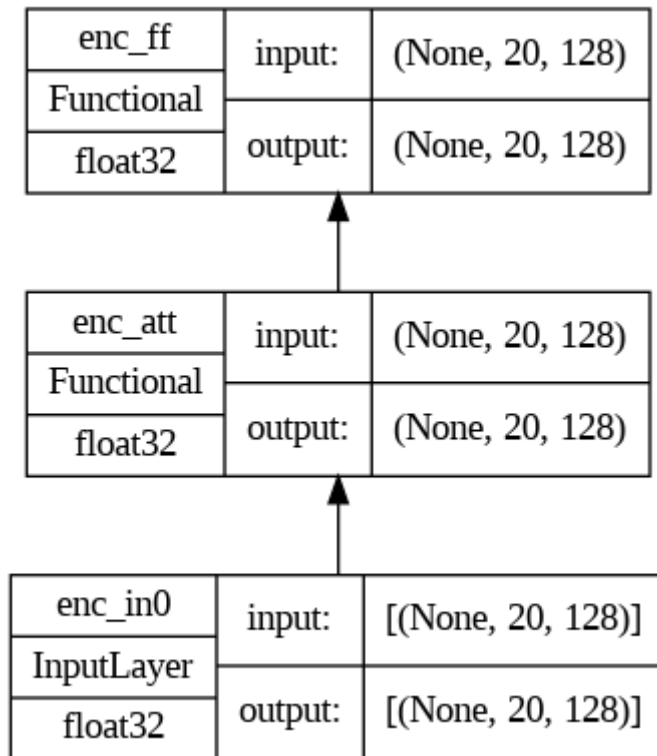


Ilustración 92. Flujo de Datos a través de la Unidad de Codificación

Capa de Decodificación

La unidad de decodificación, definida por la función "**decoder**", es la etapa final en el modelo **Transformer**. A diferencia de la unidad de codificación, la unidad de decodificación toma dos entradas: una secuencia de entrada y un vector de contexto. Este último es el resultado de la codificación de otra secuencia, y proporciona el contexto para la decodificación de la secuencia de entrada.

Al igual que la unidad de codificación, la unidad de decodificación consta de una capa de atención de múltiples cabezas y una red de avance punto a punto. Pero, además, entre estas dos, se añade una capa de atención cruzada, realizada por la función "**cross_attention**", que permite al modelo relacionar la secuencia de entrada con el vector de contexto. La atención cruzada ayuda al modelo a entender cómo la secuencia de entrada se relaciona con el contexto proporcionado, lo cual es crucial para tareas como la **traducción automática**, donde el contexto es la representación codificada de la oración fuente.


```
def decoder(input_shape, key_dim, ff_dim, dropout=0.1, prefix="dec", **kwargs):
    inputs = tf.keras.layers.Input(shape=input_shape, dtype='float32', name=f"{prefix}_in0")
    context = tf.keras.layers.Input(shape=input_shape, dtype='float32', name=f"{prefix}_ctx0")
    attmodel = self_attention(input_shape, key_dim=key_dim, mask=True, prefix=prefix, **kwargs)
    crossmodel = cross_attention(input_shape, input_shape, key_dim=key_dim, prefix=prefix, **kwargs)
    ffmodel = feed_forward(input_shape, key_dim, ff_dim, dropout, prefix)
    x = attmodel(inputs)
    x = crossmodel([(context, x)])
    output = ffmodel(x)
    model = tf.keras.Model(inputs=[(inputs, context)], outputs=output, name=prefix)
    return model
```

Ilustración 93. Implementación de la Unidad de Decodificación

La imagen del decodificador muestra cómo la secuencia de entrada y el vector de contexto se procesan a través de las capas de atención ("self_attention", "cross_attention") y la red de avance punto a punto ("feed_forward") para generar la secuencia de salida.

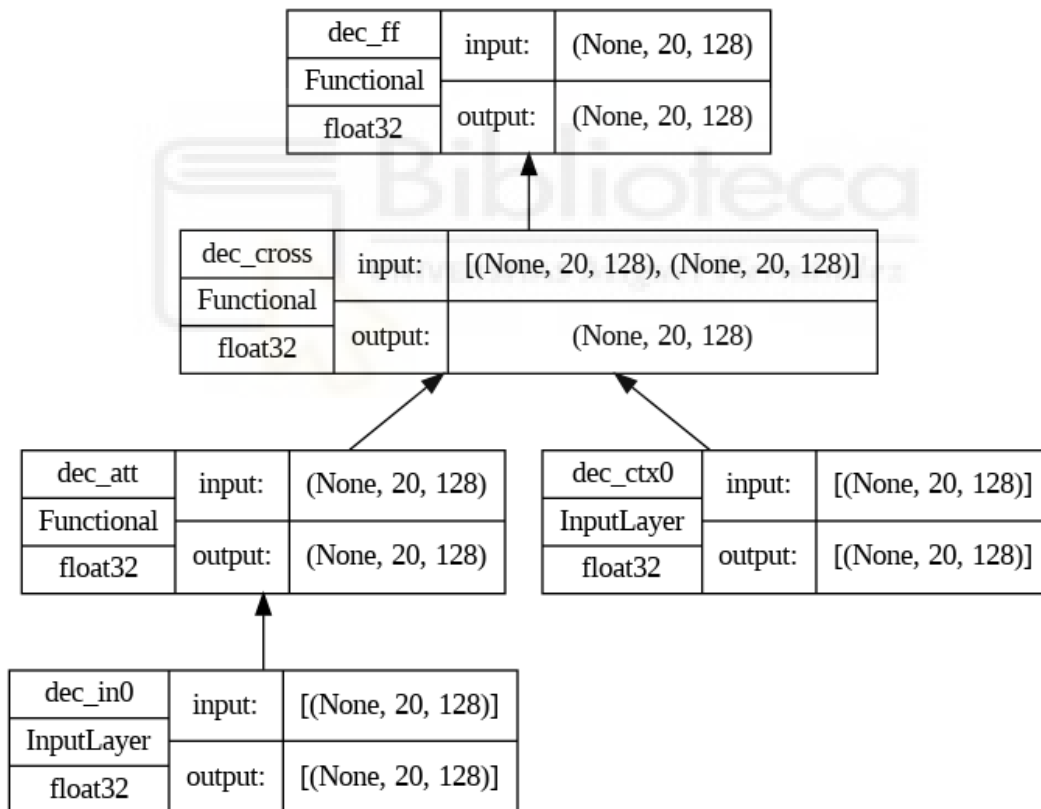


Ilustración 94. Visualización del Decodificación en el Transformer

Arquitectura del Transformer

La información inicialmente comienza en forma de secuencias de texto que se introducen en las capas de entrada para el **codificador** y el **decodificador**. Estas secuencias de texto están formadas por índices numéricos que

representan palabras específicas en el vocabulario. En el caso del codificador, su entrada es la secuencia de texto de origen, mientras que la entrada del decodificador es la secuencia de texto de destino hasta el paso temporal actual.

Una vez que las secuencias de texto ingresan al modelo, son transformadas por las capas de "**PositionalEmbedding**". Estas capas convierten los índices numéricos en vectores densos que representan el significado de las palabras. Además, también incorporan información sobre la posición de cada palabra en la secuencia, ya que la posición puede afectar el significado de la palabra en ciertos contextos.

Las representaciones de "**PositionalEmbedding**" luego ingresan a los bloques de codificadores y decodificadores. En el codificador, la secuencia de entrada se procesa a través de múltiples bloques, cada uno de los cuales consiste en una **atención auto-regulada** y una red de avance, permitiendo que el modelo capture interacciones complejas entre las palabras de la secuencia de entrada.

La salida del último bloque de codificador se alimenta a cada bloque de decodificador, además de su propia entrada de "**PositionalEmbedding**". En el decodificador, la atención está enmascarada para prevenir que la posición actual observe las posiciones futuras, asegurando que la predicción para la posición 'i' solo dependa de las posiciones conocidas antes de 'i'.

Finalmente, la salida del último bloque del decodificador se alimenta en una capa densa final, que proyecta los vectores de alta dimensión a las dimensiones del tamaño del vocabulario. Esta capa utiliza una función de **activación softmax** para convertir las puntuaciones en las dimensiones de las palabras del vocabulario en probabilidades. El índice con la mayor probabilidad se elige la predicción de la palabra para la posición actual en la secuencia de salida.

```

def transformer(num_layers, num_heads, seq_len, key_dim, ff_dim, vocab_size_src, vocab_size_tgt, dropout=0.1, name="transformer"):
    embed_shape = (seq_len, key_dim)
    input_enc = tf.keras.layers.Input(shape=(seq_len,), dtype="int32", name="encoder_inputs")
    input_dec = tf.keras.layers.Input(shape=(seq_len,), dtype="int32", name="decoder_inputs")
    embed_enc = PositionalEmbedding(seq_len, vocab_size_src, key_dim, name="embed_enc")
    embed_dec = PositionalEmbedding(seq_len, vocab_size_tgt, key_dim, name="embed_dec")
    encoders = [encoder(input_shape=embed_shape, key_dim=key_dim, ff_dim=ff_dim, dropout=dropout, prefix=f"enc{i}", num_heads=num_heads) for i in range(num_layers)]
    decoders = [decoder(input_shape=embed_shape, key_dim=key_dim, ff_dim=ff_dim, dropout=dropout, prefix=f"dec{i}", num_heads=num_heads) for i in range(num_layers)]
    final = tf.keras.layers.Dense(vocab_size_tgt, name="linear")
    x1 = embed_enc(input_enc)
    x2 = embed_dec(input_dec)
    for layer in encoders:
        x1 = layer(x1)
    for layer in decoders:
        x2 = layer([x2, x1])
    output = final(x2)
    try:
        del output._keras_mask
    except AttributeError:
        pass
    model = tf.keras.Model(inputs=[input_enc, input_dec], outputs=output, name=name)
    return model

```

Ilustración 95. Arquitectura y Proceso del Modelo Transformer

En la imagen se puede apreciar como el flujo de información en un Transformer comienza con secuencias de texto, que son transformadas por las capas de "**PositionalEmbedding**". Esta transformación convierte los índices numéricos de palabras en vectores densos y agrega información de posición.

Estos vectores se introducen en los bloques del codificador y decodificador. El codificador procesa la secuencia de entrada, capturando interacciones complejas entre palabras. Su salida se alimenta en cada bloque de decodificador, además de su propia entrada de "**PositionalEmbedding**". Los decodificadores están enmascarados para prevenir que las posiciones futuras sean visibles durante la predicción de una palabra en la posición actual.

Finalmente, la salida del último decodificador se proyecta a las dimensiones del tamaño del vocabulario utilizando una capa densa, y la **función de activación softmax** convierte las puntuaciones de las palabras en probabilidades, seleccionando la palabra con la mayor probabilidad como predicción.

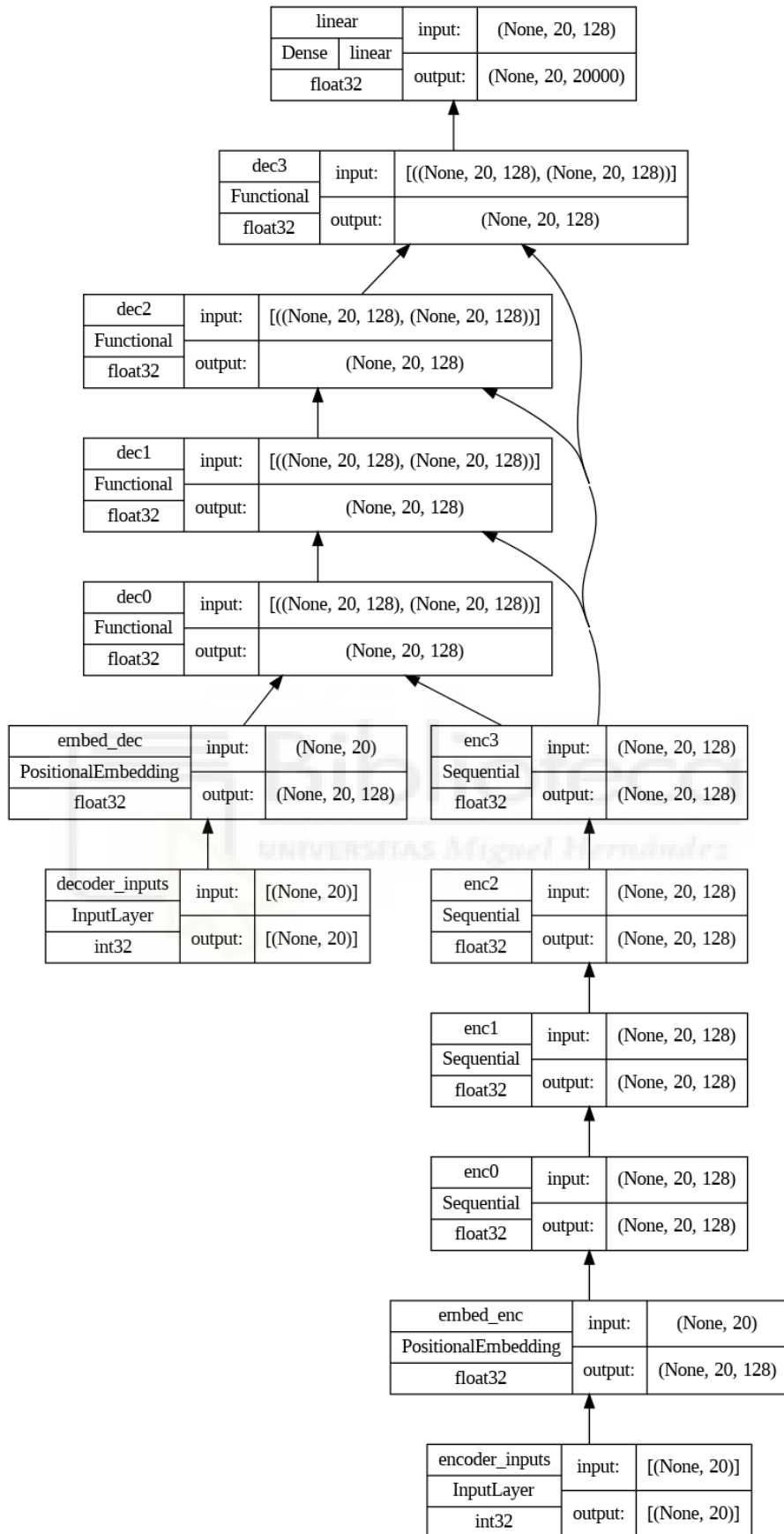


Ilustración 96. Proceso de Transformación en un Modelo Transformer

5.3.1.5 Configuración y Entrenamiento de la Red Neuronal

Preparación del modelo

Primero, se define una programación de tasa de aprendizaje personalizada para el optimizador Adam, que incluye un periodo de calentamiento inicial y una disminución proporcional a la raíz cuadrada inversa del número de pasos. Esto se realiza a través de la clase **CustomSchedule**.

```
class CustomSchedule(tf.keras.optimizers.schedules.LearningRateSchedule):
    def __init__(self, key_dim, warmup_steps=4000):
        super().__init__()
        self.key_dim = key_dim
        self.warmup_steps = warmup_steps
        self.d = tf.cast(self.key_dim, tf.float32)

    def __call__(self, step):
        step = tf.cast(step, dtype=tf.float32)
        arg1 = tf.math.rsqrt(step)
        arg2 = step * (self.warmup_steps ** -1.5)
        return tf.math.rsqrt(self.d) * tf.math.minimum(arg1, arg2)

    def get_config(self):
        config = {
            "key_dim": self.key_dim,
            "warmup_steps": self.warmup_steps,
        }
        return config
```

Ilustración 97. Ajuste en el optimizador

Luego, se instancia un objeto de la clase **CustomSchedule** y lo utilizamos para definir nuestro **optimizador Adam**. Se puede visualizar una gráfica de **Matplotlib** la programación de la **tasa de aprendizaje**.

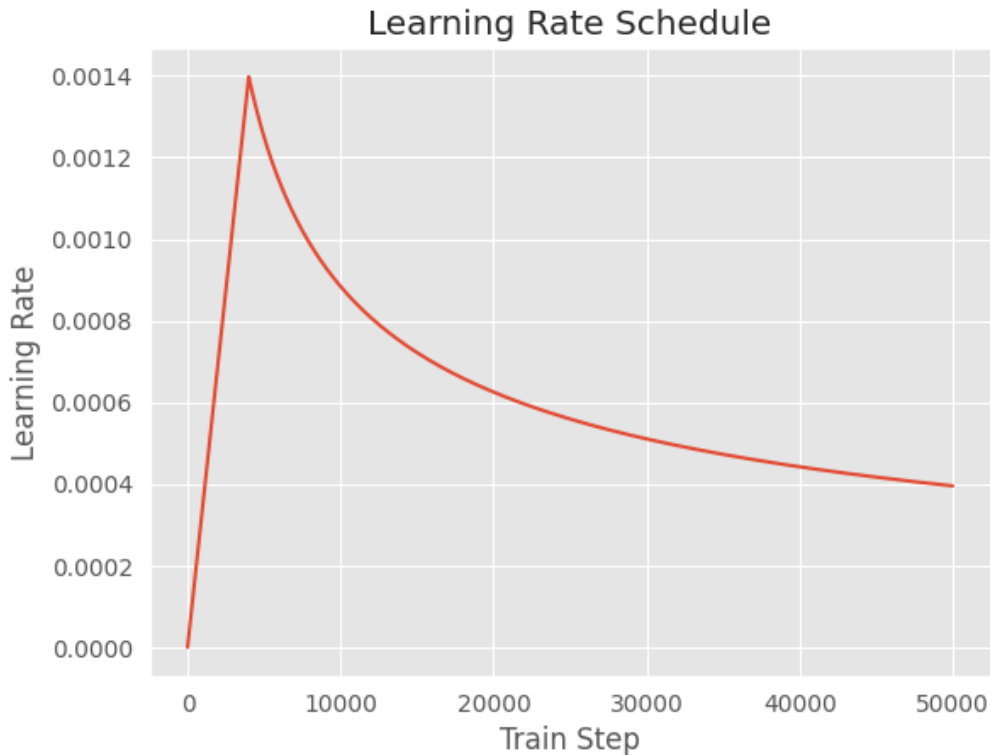


Ilustración 98. Programación de la Tasa de Aprendizaje

Cálculo de la pérdida y precisión

Se definen dos funciones: **masked_loss** y **masked_accuracy**. Estas funciones son necesarias para calcular la pérdida y la precisión durante el entrenamiento, respectivamente. Ambas funciones tienen en cuenta las máscaras de los **tokens** de relleno, que son tokens que se agregan a las secuencias para que todas las secuencias tengan la misma longitud.

```
def masked_loss(label, pred):
    mask = label != 0
    loss_object = tf.keras.losses.SparseCategoricalCrossentropy(from_logits=True, reduction='none')
    loss = loss_object(label, pred)
    mask = tf.cast(mask, dtype=loss.dtype)
    loss *= mask
    loss = tf.reduce_sum(loss)/tf.reduce_sum(mask)
    return loss

def masked_accuracy(label, pred):
    pred = tf.argmax(pred, axis=2)
    label = tf.cast(label, pred.dtype)
    match = label == pred
    mask = label != 0
    match = match & mask
    match = tf.cast(match, dtype=tf.float32)
    mask = tf.cast(mask, dtype=tf.float32)
    return tf.reduce_sum(match)/tf.reduce_sum(mask)
```

Ilustración 99. Funciones para el Cálculo de la Pérdida y Precisión

Configuración de las capas

- **vocab_size_en:** Se define la cantidad de palabras únicas en nuestro vocabulario para el idioma inglés. En este caso, nuestro modelo puede manejar hasta 10,000 palabras diferentes en inglés.
- **vocab_size_es:** Se define la cantidad de palabras únicas en nuestro vocabulario para el idioma español. En este caso, nuestro modelo puede manejar hasta 20,000 palabras diferentes en francés.
- **seq_len:** Se representa la longitud máxima de las secuencias de entrada que nuestro modelo puede procesar. Si una secuencia es más larga, se trunca; si es más corta, se rellena. En este caso, la longitud máxima de las secuencias de entrada es de 20 tokens.
- **num_heads:** Se indica el número de "cabezas" en la capa de atención multi-cabeza del modelo Transformer. Cada cabeza aprende a prestar atención a diferentes partes de la secuencia de entrada. En este modelo, se utilizan 8 cabezas.
- **key_dim:** Se especifica la dimensión de las claves y los valores en las capas de atención del Transformer. En este caso, las claves y los valores tienen 128 dimensiones.
- **ff_dim:** Se especifica la dimensión del espacio interno de la capa densa en el codificador Transformer. En este caso, el espacio interno de la capa densa tiene 512 dimensiones.
- **dropout:** Este es el porcentaje de nodos que se "abandonarán" o desactivarán durante el entrenamiento en cada iteración. Esto ayuda a prevenir el sobreajuste.

Compilación y entrenamiento del modelo

Se construye y compila el modelo **Transformer**, utilizando la pérdida enmascarada, el optimizador Adam con una tasa de aprendizaje personalizada, y la precisión enmascarada como métrica. Especificamos un total de **20 épocas**

para el entrenamiento. Finalmente, se entrena el modelo en **train_ds** y se valida en **val_ds** en cada época.

```

model = transformer(num_layers, num_heads, seq_len, key_dim, ff_dim, vocab_size_en, vocab_size_fr, dropout)
lr = CustomSchedule(key_dim)
optimizer = tf.keras.optimizers.Adam(lr, beta_1=0.9, beta_2=0.98, epsilon=1e-9)

model.compile(loss=masked_loss, optimizer=optimizer, metrics=[masked_accuracy])
epochs = 20
history = model.fit(train_ds, epochs=epochs, validation_data=val_ds)

```

Ilustración 100. Configuración del Modelo Transformer

5.3.1.6 Evaluación del Modelo

Tras el entrenamiento del modelo Transformer utilizando el conjunto de datos definido, se obtienen los siguientes resultados. Las métricas usadas son las que se describieron en la sección anterior.

Para el caso de nuestro modelo Transformer, el cual cuenta con una cantidad significativa de parámetros entrenables, se observan los resultados siguientes durante las épocas de entrenamiento y validación:

Loss y Masked Accuracy:

Epoch	Training Loss	Training Masked Accuracy	Validation Loss	Validation Masked Accuracy
1	5.6178	0.2805	3.7916	0.4088
2	3.1101	0.5077	2.5830	0.5722
...
19	1.0355	0.8251	1.4630	0.7550
20	1.0121	0.892	1.4454	0.7583

Ilustración 101. Evaluación de Resultados del Transformer

El modelo se sometió a un proceso de entrenamiento durante **20 épocas**, durante las cuales se vigiló de cerca tanto la pérdida como la **precisión enmascarada** para los conjuntos de entrenamiento y validación. Se observa una disminución constante y significativa en la pérdida de entrenamiento desde una cifra inicial de 5.6178 en la primera época hasta un mínimo de 1.0121 en la última

época. Paralelamente, la precisión enmascarada mostró un aumento notable, partiendo de un valor de **0.2805** y llegando a un impresionante **0.8292**.

Estas tendencias son indicativas de un **aprendizaje efectivo y sistemático** del modelo con cada iteración de entrenamiento. Sin embargo, también es esencial destacar la necesidad de prestar atención a la posible aparición de **sobreajuste**, dada la observación de cierta estabilización de la precisión enmascarada y un ligero incremento de la pérdida en el conjunto de validación en las últimas épocas.

Las siguientes gráficas ilustran estos hallazgos de manera más clara y detallada, y se evidencia el rendimiento sólido del modelo y su capacidad de aprender de manera efectiva a lo largo del **proceso de entrenamiento**.

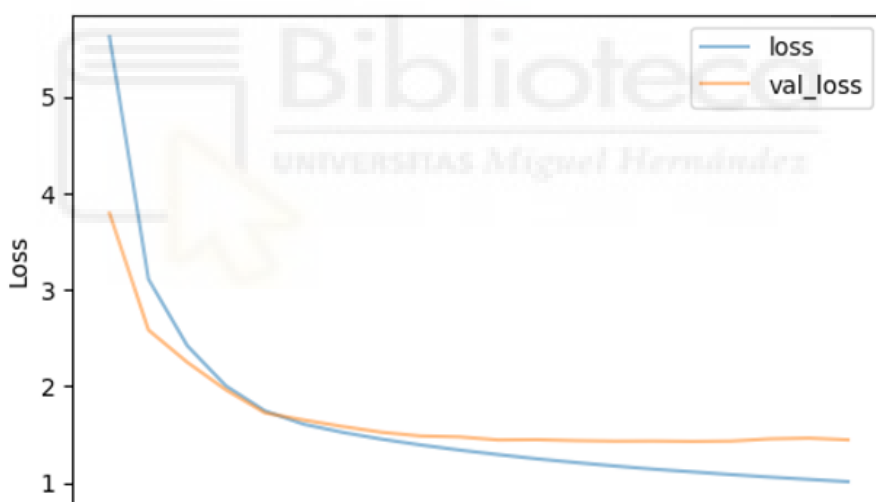


Ilustración 102. Evolución de la Pérdida Durante el Entrenamiento

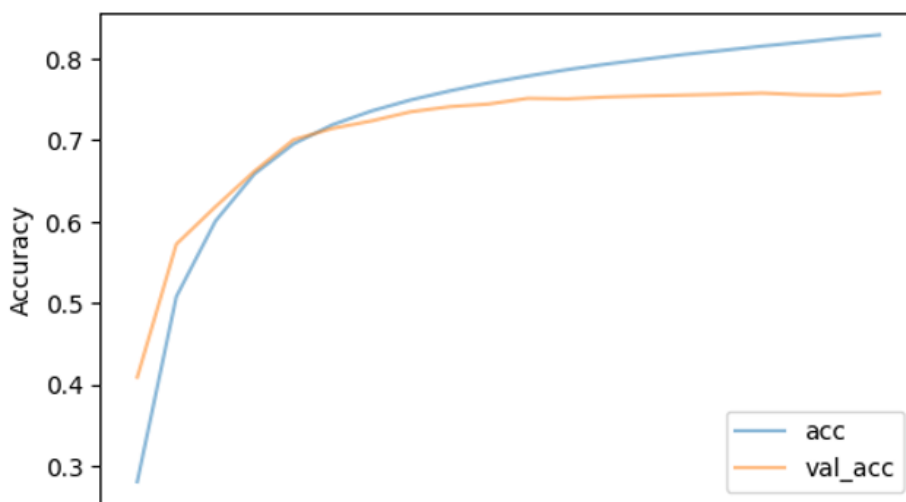


Ilustración 103. Evolución de la Precisión Durante el Entrenamiento

Finalmente, se destacan algunos de los resultados obtenidos con el modelo de **traducción automática** Transformer. Estos resultados representan la capacidad del modelo para traducir frases del inglés al español de una manera coherente y gramaticalmente correcta. En cada caso de prueba, se presentan tres partes:

1. **Frase en inglés:** Esta es la frase original en inglés que el modelo debe traducir. Se selecciona aleatoriamente del conjunto de datos de prueba.
2. **Traducción correcta:** Esta es la traducción correcta de la frase en inglés al español, tal como aparece en nuestro conjunto de datos. Se utiliza como un **punto de referencia** para evaluar el rendimiento del modelo. Por convención, la frase se delimita con las etiquetas "[start]" y "[end]" para indicar el comienzo y el final de la frase respectivamente.
3. **Traducción del modelo:** Esta es la traducción generada por el modelo de Transformer. Es el resultado de alimentar la frase en inglés al modelo y dejar que genere su propia traducción al español.

Esto muestra que el modelo Transformer puede generar traducciones coherentes y significativas en la tarea de traducción automática, incluso cuando las traducciones exactas pueden variar. Esto es crucial para el Procesamiento del Lenguaje Natural, donde a menudo hay más de una forma correcta de expresar una idea o concepto.

```
Test 3:
is it bad ?
== [start] ¿ es grave ? [end]
-> [start] ¿ es malo ? [end]

Test 4:
tom might get sick .
== [start] tom podría caer enfermo . [end]
-> [start] tom podría estar enfermo . [end]

Test 5:
he liked making decisions .
== [start] a él le gustaba tomar decisiones . [end]
-> [start] le gustó tomar decisiones . [end]
```

Ilustración 104. Resultado traducción automática

5.3.2 Desarrollo 2: análisis de sentimientos

En este segundo desarrollo, se profundiza en el desafío del análisis de sentimientos. A diferencia del primer desarrollo, que requería la estructura completa del **Transformer**, el **análisis de sentimientos** puede llevarse a cabo eficazmente utilizando solo la capa de codificación del modelo Transformer. Esto subraya la versatilidad y eficiencia de esta arquitectura, que puede ser adaptada y modificada para resolver diferentes problemas en función de las necesidades concretas.

A pesar de la simplificación de la arquitectura, el Transformer sigue demostrando su eficacia y robustez, proporcionando resultados sólidos en esta tarea. Sin embargo, cabe señalar que, para optimizar los resultados y mejorar la eficacia del modelo en la detección y clasificación de sentimientos, se han requerido algunas adaptaciones y optimizaciones adicionales.

Para poder evaluar de forma adecuada el rendimiento de este modelo modificado, se han incorporado nuevas métricas que proporcionan una visión más completa de su capacidad para analizar y clasificar los sentimientos. Esta combinación de modificaciones estructurales y metodológicas demuestra una vez más la adaptabilidad del Transformer, y su aptitud para abordar eficazmente una amplia gama de tareas en el **dominio del PLN**.

5.3.2.1 Presentación de los datos

Por supuesto, entiendo que quieres proporcionar una visión clara de los datos que se utilizarán en este desarrollo, lo cual es un elemento crucial para entender y valorar los resultados obtenidos. A continuación, incluyo los párrafos adaptados para tu contexto.

El primer paso para abordar nuestro objetivo en este desarrollo es profundizar en el entendimiento del conjunto de datos que vamos a utilizar. Para ello, se emplean diversas bibliotecas de Python especializadas en visualización de datos, esencialmente útiles en el ámbito del **Procesamiento del Lenguaje Natural** (PLN). Aquí, se presenta una serie de gráficos que nos aportan un conocimiento detallado de la naturaleza de nuestros datos. Esta interpretación visual permite adquirir una percepción más precisa y comprensiva de la estructura y las características distintivas de nuestros datos.

Histograma de distribución de calificaciones

Este gráfico muestra la frecuencia de cada calificación en el conjunto de datos, y se observa un patrón interesante: hay una alta frecuencia de puntuaciones perfectas (10/10), seguida de **calificaciones en el extremo más bajo** (1/10). Esto parece evidenciar una polarización en las opiniones de los medicamentos, posiblemente indicando que las personas están más inclinadas a escribir reseñas cuando tienen experiencias extremas, ya sean positivas o negativas.

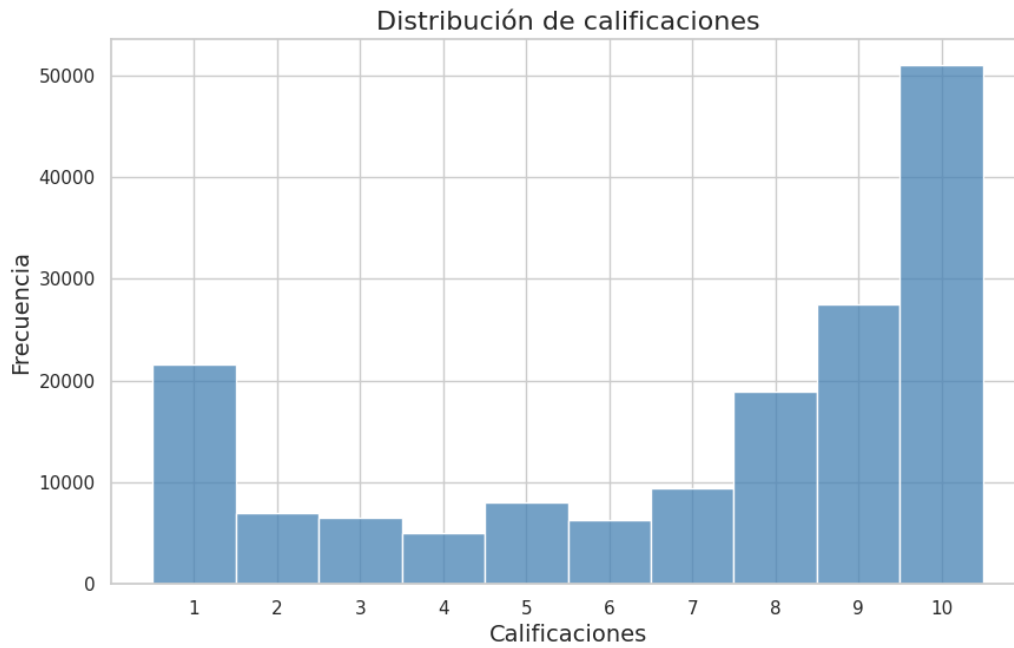


Ilustración 105. Distribución de calificaciones

Histograma de distribución de longitud de reseñas

Este gráfico indica la cantidad de reseñas que contienen un número específico de palabras. Encontramos que la media de palabras por reseña es de aproximadamente 85, aunque la distribución es bastante amplia, con longitudes que varían desde 1 hasta 1894 palabras. El 50% de las reseñas contienen entre 48 y 126 palabras, lo que sugiere una variabilidad considerable en la cantidad de información que los usuarios proporcionan en sus reseñas.

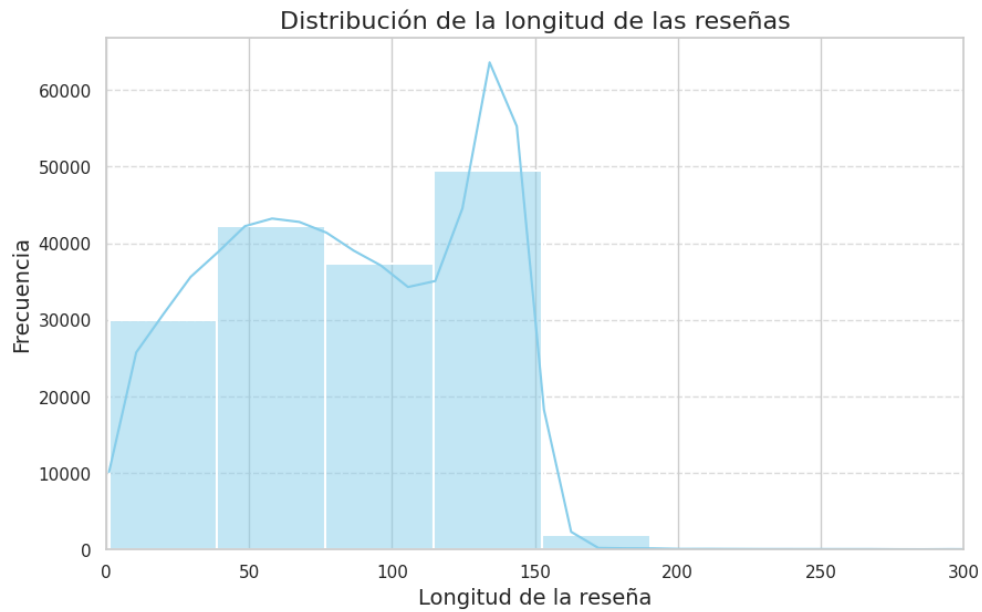


Ilustración 106. Longitud de reseñas

Gráfico circular de proporción de calificaciones positivas y negativas.

Este gráfico proporciona una representación clara de la proporción de reseñas con calificaciones por encima y por debajo de 5. La visualización muestra que hay una proporción significativamente mayor de calificaciones positivas (superiores a 5) en comparación con las negativas, con aproximadamente el 70% de las reseñas otorgando calificaciones positivas. Esta observación podría señalar una tendencia general positiva en las experiencias con los medicamentos mencionados en el conjunto de datos.

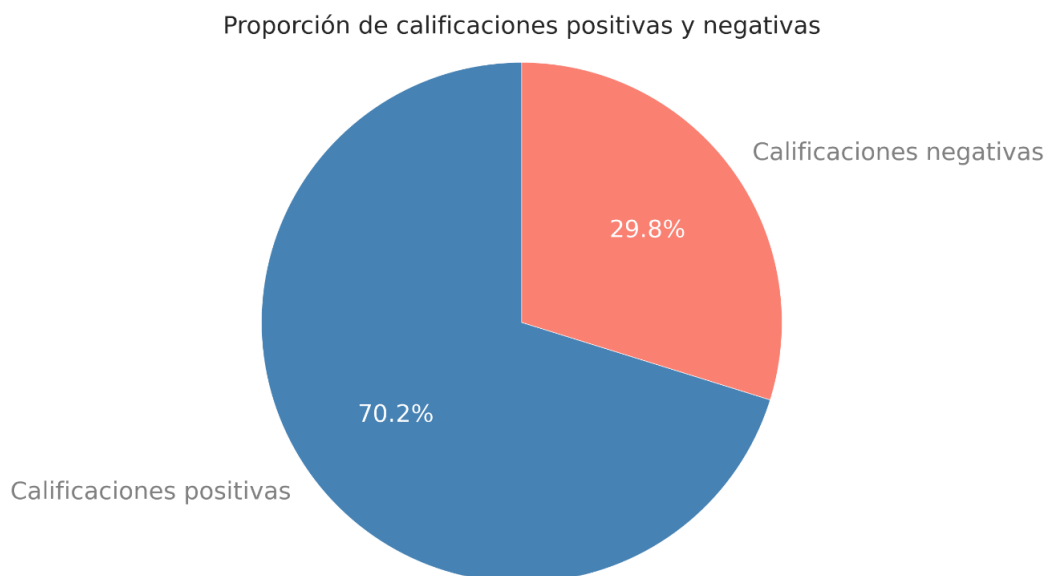


Ilustración 107. Calificaciones positivas y negativas

Estos análisis preliminares de los datos proporcionan un valioso contexto para la interpretación de los resultados obtenidos a partir de nuestro modelo de análisis de sentimientos basado en la capa de **codificación del Transformer**.

5.3.2.2 Diferencias en el modelo Transformer *Hernández*

En este segundo desarrollo, se presenta la necesidad de adaptar nuestro modelo Transformer a la tarea de clasificación de texto, específicamente para el **análisis de sentimientos**. A diferencia del primer desarrollo, donde se utiliza la estructura completa del Transformer, en esta ocasión es suficiente con implementar la porción del **codificador**, dejando de lado el decodificador. Esto se debe a la naturaleza de la tarea en cuestión: no se necesita generar secuencias de salida (como traducciones o respuestas de chatbot), sino simplemente clasificar el texto de entrada en una de varias categorías posibles.

El codificador del Transformer, con su mecanismo de atención, es perfectamente adecuado para esta tarea. Al establecer relaciones entre palabras en un texto, se captura el significado y el sentimiento del texto de manera más efectiva que los **modelos más simples o tradicionales**.

El Transformer modificado incluye dos capas principales, cada una con su propia **funcionalidad y propósito**.

La primera, llamada **TransformerEncoder**, utiliza una capa de atención **multi-cabeza** para prestar atención a diferentes partes del texto de entrada simultáneamente, lo que nos permite captar la interacción entre las palabras y las frases en nuestro texto. A continuación, una capa densa procesa la salida de la atención multi-cabeza, proyectándola en un espacio de representación interno que es útil para nuestro modelo. La suma de estas salidas, una vez normalizada, genera la salida final del **codificador**.

Además, la capa **TransformerEncoder** también puede manejar máscaras de entrada, que permiten ignorar ciertas partes del texto de entrada, como el relleno añadido durante el preprocesamiento.

```
class TransformerEncoder(layers.Layer):
    def __init__(self, embed_dim, dense_dim, num_heads, **kwargs):
        super().__init__(**kwargs)
        # Establece las dimensiones de incrustación, densas y el número de cabezas para la
        self.embed_dim = embed_dim
        self.dense_dim = dense_dim
        self.num_heads = num_heads
        # Define la capa de atención multi-cabeza
        self.attention = layers.MultiHeadAttention(num_heads=num_heads, key_dim=embed_dim)
        # Define la proyección densa (Feed-Forward) en el codificador Transformer
        self.dense_proj = keras.Sequential(
            [layers.Dense(dense_dim, activation="relu"),
             layers.Dense(embed_dim),]
        )
        # Define las capas de normalización de capa
        self.layernorm_1 = layers.LayerNormalization()
        self.layernorm_2 = layers.LayerNormalization()

    def call(self, inputs, mask=None):
        # Aplica la atención multi-cabeza a las entradas
        if mask is not None:
            mask = mask[:, tf.newaxis, :]
        attention_output = self.attention(inputs, inputs, attention_mask=mask)
        # Aplica la normalización de capa y la proyección densa
        proj_input = self.layernorm_1(inputs + attention_output)
        proj_output = self.dense_proj(proj_input)
        # Retorna la salida después de la segunda normalización de capa
        return self.layernorm_2(proj_input + proj_output)

    def get_config(self):
        # Configuración personalizada para guardar y cargar el modelo
        config = super().get_config()
        config.update({
            "embed_dim": self.embed_dim,
            "num_heads": self.num_heads,
            "dense_dim": self.dense_dim,
        })
        return config
```

Ilustración 108. Función de codificador

La segunda capa principal, **PositionalEmbedding**, implementa los **embeddings** de posición, una característica clave del Transformer que permite al modelo tener en cuenta el orden de las palabras en el texto de entrada. Esta capa combina los embeddings de palabras, las cuales representan el significado semántico de

cada palabra, con los embeddings de posición, que representan el orden de las palabras en la secuencia. Juntos, estos embeddings forman la entrada final que se alimenta al **codificador del Transformer**.

Cada una de estas capas personalizadas también incluye un método `get_config`, que permite guardar y cargar la capa para su posterior uso. Este método devuelve un diccionario con la **configuración de la capa**, lo que permite recrear exactamente la capa cuando sea necesario.

```
class PositionalEmbedding(layers.Layer):
    def __init__(self, sequence_length, vocab_size, embed_dim, **kwargs):
        super().__init__(**kwargs)
        # Establece la longitud de secuencia, el tamaño del vocabulario y las dimensiones de incrustación
        self.embed_dim = embed_dim
        self.vocab_size = vocab_size
        self.sequence_length = sequence_length
        # Define las incrustaciones de tokens y posiciones
        self.token_embeddings = layers.Embedding(input_dim=vocab_size, output_dim=embed_dim)
        self.position_embeddings = layers.Embedding(input_dim=sequence_length, output_dim=embed_dim)

    def call(self, inputs):
        # Combina las incrustaciones de tokens y posiciones
        length = tf.shape(inputs)[-1]
        positions = tf.range(start=0, limit=length, delta=1)
        embedded_tokens = self.token_embeddings(inputs)
        embedded_positions = self.position_embeddings(positions)
        return embedded_tokens + embedded_positions

    def get_config(self):
        # Configuración personalizada para guardar y cargar el modelo
        config = super().get_config()
        config.update({
            "embed_dim": self.embed_dim,
            "vocab_size": self.vocab_size,
            "sequence_length": self.sequence_length,
        })
        return config
```

Ilustración 109. Función para los embedding de posición

En resumen, esta versión simplificada del Transformer, utilizando solo el codificador, está perfectamente adaptada para la tarea de análisis de sentimientos. Al centrarse en las relaciones entre las palabras y su orden en el texto, siendo capaces de clasificar eficazmente los textos según su **sentimiento**.

5.3.2.3 Estrategias para evitar el sobreajuste

Para mitigar el sobreajuste en nuestro modelo, se han realizado varias modificaciones. Primero, se redujo a la mitad el número de neuronas en la capa densa interna del **codificador Transformer**, pasando de 1024 a 512. Esto puede reducir la capacidad del modelo para representar información compleja, pero

también hace que el modelo sea más eficiente y menos propenso al sobreajuste. Adicionalmente, se introdujo una capa de "**dropout**" con una tasa del 60%, un incremento respecto a la tasa común del 50%. Este ajuste aumenta la regularización y disminuye el **riesgo de sobreajuste**.

En la capa final del modelo, se agregó una regularización L2 para evitar que los pesos del modelo se vuelvan demasiado grandes, lo que ayuda a evitar el sobreajuste. Durante el entrenamiento, se utiliza "**callbacks**" para la detención temprana del entrenamiento si la pérdida de validación deja de mejorar durante tres épocas, que es otra estrategia efectiva para prevenir el sobreajuste.

5.3.2.4 Implementación de nuevas métricas para el análisis de resultados

Para validar la eficacia y eficiencia de nuestros modelos de clasificación binaria, es crucial comprender y aplicar adecuadamente diversas métricas de rendimiento. En el presente capítulo, se profundizará en las métricas fundamentales usadas para evaluar dichos modelos, comenzando por la función de pérdida, seguido por la precisión, y luego se profundizará en el concepto de la matriz de confusión para explicar métricas derivadas como **la precisión, la sensibilidad y la puntuación F1**. Estas métricas permitirán obtener una imagen completa y precisa del rendimiento de los modelos, permitiendo identificar áreas de mejora y tomar decisiones informadas sobre posibles ajustes.

- **Loss:** Es la función de pérdida que se utiliza para optimizar el modelo durante el entrenamiento. Mide la discrepancia entre las predicciones del modelo y los valores verdaderos. En este caso, se utiliza la pérdida de entropía cruzada binaria, común en problemas de clasificación binaria.
- **Accuracy (Exactitud):** Se proporciona una medida general de cómo el modelo se desempeña correctamente, calculada como la proporción de predicciones correctas sobre el total. Sin embargo, puede ser engañosa si las clases están desbalanceadas.

$$Accuracy = \frac{N \text{ predicciones correctas}}{N \text{ predicciones totales}}$$

Ecuación 11. Formula Accuracy

- **Confusion Matrix (Matriz de confusión):** Es una tabla que desglosa el rendimiento del modelo mostrando los Verdaderos Positivos (VP), Verdaderos Negativos (VN), Falsos Positivos (FP), y Falsos Negativos (FN). No es una métrica en sí, pero proporciona la base para calcular otras métricas.
- **Precision (Precisión):** Mide la proporción de identificaciones positivas que fueron efectivamente correctas. Un modelo con alta precisión tiene un bajo porcentaje de falsos positivos.

$$Precision = \frac{VP}{VP + FN}$$

Ecuación 12. Formula precision

- **Recall (Sensibilidad):** Se mide la proporción de positivos actuales que fueron identificados correctamente. Un modelo con alta sensibilidad tiene un bajo porcentaje de falsos negativos.

$$Recall = \frac{VP}{VP + FN}$$

Ecuación 13. Formula Recall

- **F1 Score:** Es la media armónica de la **precisión** y la **sensibilidad**. Es útil cuando se quiere balancear estas dos métricas. Una puntuación F1 alta indica un equilibrio adecuado entre precisión y sensibilidad.

$$F_1 = \frac{Precision \times Recall}{Precision + Recall}$$

Ecuación 14. Formula F1

5.3.2.5 Análisis de resultados

En este apartado, se realizara un análisis detallado de los resultados obtenidos con el modelo **Transformer Encoder** y se evaluarán las métricas adicionales calculadas para una comprensión más profunda del rendimiento del modelo.

Nuestro modelo Transformer Encoder fue entrenado durante **20 épocas**, evidenciando un aprendizaje efectivo a medida que avanzaban las iteraciones. Inicialmente, la pérdida se situaba en 0.5209, disminuyendo a 0.1693 en la última época. Por otro lado, la precisión mostró una mejora de 0.7558 a 0.9396. Sin embargo, tras la sexta época, se observó un incremento en la pérdida de validación, sugiriendo un posible sobreajuste. Gracias a las técnicas de regularización implementadas previamente, se pudo mitigar este problema, mejorando la capacidad de generalización del modelo.

En términos de eficiencia computacional, el entrenamiento total del modelo requirió alrededor de 71.7 minutos (4301.46 segundos). A pesar de esta inversión de tiempo, el rendimiento obtenido justifica dicho coste, obteniendo una precisión del 0.844 en un conjunto de pruebas independiente.

Más allá de la precisión, se calcularon métricas adicionales para profundizar en el desempeño del modelo. Estas incluyeron la precisión con un valor de 0.8577, la sensibilidad (recall) con un valor de **0.9305** y la puntuación F1 con un valor de 0.8926. Cada una de estas métricas arroja luz sobre diferentes aspectos del **rendimiento del modelo** y en conjunto demuestran un rendimiento sólido en la clasificación de los datos de prueba.

Estos resultados y métricas adicionales refuerzan la efectividad del Transformer Encoder en la tarea de análisis de sentimientos.

	Training	Validation	Test
Loss	0.1693	0.4247	0.3637
Accuracy	0.9396	0.8456	0.844
Precision			0.8577
Recall			0.9305
F1			0.8926

Ilustración 110. Resultados análisis de sentimientos

Para finalizar, se puede apreciar como el ejemplo de predicción presentado muestra cómo el modelo identifica una reseña como negativa. La reseña de texto es procesada, convertida en una secuencia numérica, y pasada a través del modelo para obtener una predicción (en este caso, cerca de 0, indicando una reseña negativa). La salida booleana "False" también refuerza que el modelo ha clasificado correctamente la reseña como negativa. Esta interpretación directa de los resultados del modelo sirve para entender mejor su **desempeño** y su toma de decisiones.

- **Review text:** I had high hopes for this product, but it turned out to be a huge disappointment. Not only did it not work as advertised, but it also caused several unwanted side effects. I experienced headaches, nausea, and even a rash after using it. I would not recommend this product to anyone. Save your money and look for something else.

```
1/1 [=====] - 0s 20ms/step
Review text: I had high hopes for this product, but it turned out to be a huge disappointment.
Label (1 - Positive, 0 - Negative): 0
Prediction: [[0.00782174]]
Boolean result: [[False]]
```

Ilustración 111. Prueba con reseña negativa

5.3.3 Desarrollo 3: Modelo Q&A para Hacienda

En este proyecto, aunque el uso de un **modelo preentrenado de Transformer** no era el objetivo principal, se empleó el modelo **T5** en una tarea de preguntas y respuestas para demostrar una lección crítica: la importancia de un buen conjunto de datos. A pesar de la potencia del modelo T5, los resultados no fueron óptimos debido a limitaciones en el conjunto de datos utilizado. Esta experiencia subraya que incluso los modelos más avanzados dependen en gran medida de la calidad y cantidad del **dataset** para su efectividad. Por lo tanto, el propósito principal de este desarrollo es resaltar la importancia de contar con un dataset adecuado al utilizar modelos de Transformer.

5.3.3.1 Motivación del desarrollo

La motivación principal para este desarrollo fue la creación de **un modelo de Q&A** que, mediante un **dataset** creado por el autor, pudiera entrenarse para responder adecuadamente a preguntas relacionadas con la legislación y regulaciones de Hacienda. El objetivo era que este modelo pudiera asistir tanto a asesores fiscales como a usuarios comunes, proporcionando respuestas rápidas y precisas a consultas que puedan surgir en su día a día. La intención era proporcionar una herramienta que pudiera ayudar a aumentar la eficiencia en el **trabajo de los asesores fiscales**, así como facilitar la comprensión de los temas fiscales para los usuarios generales.

5.3.3.2 Dataset empleado

Para este desarrollo, se decidió asumir el desafío de crear un propio dataset desde cero, sin estar completamente consciente de la complejidad que esto implicaría. Se comenzó por establecer un **dataframe vacío** con las columnas que se consideraron necesarias: **"Question"**, **"Answer"** y **"Context"**. Para generar las entradas, se basó en una ley lo suficientemente extensa como para extraer al menos 500 preguntas. Cada pregunta incluía su correspondiente respuesta y contexto. A pesar de que este **dataset** permitió el entrenamiento del modelo T5, es importante tener en cuenta que se trata de un dataset de tamaño pequeño. Además, la tarea de formular preguntas relevantes sobre temas legales y encontrar un contexto adecuado para cada una de ellas representó un desafío adicional. La creación de este dataset sirvió como un importante recordatorio de la importancia y la dificultad de recopilar y preparar datos para el entrenamiento de modelos de machine learning.

```

Question \
0 ¿Cuál es la naturaleza del Impuesto sobre la R...
1 ¿Qué principios rigen el Impuesto sobre la Ren...
2 ¿Qué se considera como renta del contribuyente...
3 ¿Importa el lugar donde se generan los rendimi...
4 ¿Cuál es la relación entre el Impuesto sobre l...
..
498 ¿Cuál es el objetivo del Título VIII en el apa...
499 ¿Qué establece el Título IX en el apartado "Co...
500 ¿Cuál es el propósito del Título X en el apart...
501 ¿Qué aspectos cubre el Título XI en el apartad...
502 ¿Cuál es el objetivo del Título XII en el apar...

Answer \
0 La naturaleza del Impuesto sobre la Renta de l...
1 Los principios que rigen el Impuesto sobre la ...
2 La renta del contribuyente se considera como l...
3 No, el objeto del impuesto incluye la renta de...
4 El Impuesto sobre la Renta de las Personas Fís...
..
498 El Título VIII trata de las deducciones estata...
499 El Título IX se refiere a la gestión censal, l...
500 El Título X aborda las infracciones y sancione...
501 El Título XI se ocupa de la revisión de actos ...
502 El Título XII trata sobre la colaboración soci...

Context
0 El Artículo 1 establece que el Impuesto sobre ...
1 El Artículo 1 menciona que el Impuesto sobre l...
2 El Artículo 2 establece que el objeto del Impu...
3 El Artículo 2 menciona que el objeto del Impue...
4 El Artículo 3 establece que el Impuesto sobre ...
..
498 El Título VIII se centra en las deducciones es...
499 El Título IX se enfoca en la gestión censal, l...
500 El Título X se centra en las infracciones y sa...
501 El Título XI aborda la revisión de actos y rec...
502 El Título XII se enfoca en la colaboración soc...

[503 rows x 3 columns]

```

Ilustración 112. Dataset Q&A

5.3.3.3 OPLModelo utilizado

En este desarrollo, se optó por utilizar el modelo **Transformer T5** (Text-To-Text Transfer Transformer), un modelo preentrenado desarrollado por Google. Este modelo se obtiene de la librería de Hugging Face Transformers, una popular plataforma para el entrenamiento y la implementación de modelos Transformer. El siguiente código ilustra cómo preparar los datos para su uso con **Hugging Face Transformers**:

```
from transformers import T5ForConditionalGeneration, T5Tokenizer

model = T5ForConditionalGeneration.from_pretrained("./results/checkpoint-100")
tokenizer = T5Tokenizer.from_pretrained("t5-base")
```

Ilustración 113. modelo T5

El **modelo T5** es un excelente candidato para tareas de **preguntas y respuestas (Q&A)**, gracias a su capacidad para generar texto en lenguaje natural de alta calidad. Este modelo ha demostrado un desempeño excepcional en una amplia variedad de tareas de procesamiento del lenguaje natural. Su diseño es universal, lo que significa que puede manejar cualquier tarea de procesamiento de lenguaje natural que se le asigne, siempre y cuando los datos de entrada y salida estén en forma de texto. Además, su arquitectura de codificador-decodificador lo hace particularmente útil para tareas generativas como la generación de respuestas a preguntas.

Dicho esto, aunque el T5 es un modelo potente, su eficacia depende en gran medida de la calidad y cantidad de datos de entrenamiento disponibles. Al tratarse de un modelo de aprendizaje profundo, T5 necesita **grandes cantidades de datos** para aprender efectivamente. Si se entrena con un dataset pequeño o mal estructurado, sus capacidades se verán limitadas. En este caso, el principal desafío fue el tamaño y la complejidad del dataset creado, que resultó ser insuficiente para entrenar adecuadamente al modelo. Este es un punto crucial que destaca la importancia de contar con un buen dataset para entrenar estos modelos.

5.3.3.4 Resultados

Para evaluar el desempeño del modelo, se utilizaron una pregunta y un contexto específicos para probar la capacidad del **modelo para generar respuestas**. Aquí está el código y los resultados obtenidos:


```
# Ejemplo de pregunta y contexto
question = "¿Describeme que pasos se deben de seguir para cantificar la base imponible según el artículo 15?"
context = "El artículo 15 establece el orden en el que se deben seguir los pasos para cuantificar la base imponible,"

# Generar la respuesta
answer = generate_answer(model, tokenizer, question, context)
answer = answer.replace("<pad>", "").replace("</s>", "").strip()
print("Respuesta:", answer)

Respuesta: los pasos
```

Ilustración 114. Resultado modelo de Q&A

El código ejecuta una función llamada **generate_answer**, la cual toma como argumentos el modelo, el **tokenizador**, una pregunta y un contexto. Esta función está diseñada para procesar la pregunta y el contexto utilizando el tokenizador y el modelo, y luego generar una respuesta.

La respuesta generada por el modelo fue "los pasos", lo cual no es una respuesta informativa ni útil en el contexto de la pregunta hecha, que pide una descripción detallada de los pasos a seguir según lo establecido en el artículo 15.

Esto indica que el modelo no se entrenó lo suficientemente bien para generar respuestas útiles y relevantes. Los resultados son una evidencia clara de que un buen modelo, como el **T5** en este caso, no es suficiente por sí solo para obtener buenos resultados en tareas de procesamiento del lenguaje natural como Q&A. El entrenamiento con un **dataset** pequeño y limitado impacta negativamente la capacidad del modelo para generar respuestas correctas y pertinentes.

En conclusión, la generación de un modelo de preguntas y respuestas para temas específicos, como los aspectos legales relacionados con la Hacienda, es un desafío que requiere no solo un modelo potente y eficaz, sino también un dataset de alta calidad y suficientemente amplio. Este proyecto resalta la importancia de ambos componentes para obtener resultados óptimos.



CAPÍTULO 7: CONCLUSIONES

7.1 Conclusión

A lo largo de la realización de este **Trabajo de Fin de Grado**, hemos obtenido varias conclusiones significativas.

Primero, hemos confirmado cómo los modelos basados en la **arquitectura Transformer** son una alternativa prometedora en el campo del procesamiento del lenguaje natural. Concretamente, en tareas de análisis de sentimiento de texto y traducción automática, hemos obtenido precisiones notables que rondan el 84.3% y 82.92% respectivamente. Estos resultados son notables para tareas de estas características, especialmente si consideramos la complejidad inherente al **procesamiento del lenguaje natural**.

Hemos podido observar también cómo los datos de entrada de nuestro problema, los textos, deben ser transformados a un formato adecuado que pueda ser procesado por nuestros **modelos basados en Transformers**. Este proceso de transformación y codificación es fundamental para el rendimiento de los modelos y nos ha permitido analizar el impacto de diferentes **estrategias de representación de texto**.

Es interesante notar que los modelos que tienen una mayor cantidad de parámetros tienden a ofrecer mejores resultados. Sin embargo, esto viene con un costo asociado en términos de recursos computacionales requeridos para el entrenamiento y la evaluación de estos modelos. Esto subraya la importancia de la eficiencia y la optimización en la elección y diseño de modelos para tareas de procesamiento del lenguaje natural.

Por último, es importante mencionar que no existe una correlación perfecta entre la complejidad del modelo y el rendimiento final. A pesar de las diferencias en la arquitectura y la cantidad de parámetros entre el Transformer parcial y el Transformer completo, los modelos han mostrado rendimientos sólidos en sus respectivas tareas, lo que resalta la flexibilidad y versatilidad de la arquitectura Transformer.

En resumen, a través de este proyecto, hemos logrado desarrollar una comprensión más profunda de la arquitectura **Transformer** y su aplicabilidad en varias tareas de procesamiento del lenguaje natural. Los resultados obtenidos refuerzan el potencial de estos modelos para seguir avanzando en el campo del procesamiento del lenguaje natural.

7.2 Líneas Futuras

Para el futuro, vemos una serie de direcciones prometedoras para la expansión y continuación de este trabajo:

- **Enriquecimiento del conjunto de datos:** Al trabajar con un conjunto de datos más extenso y diverso, podríamos entrenar modelos que sean más robustos y tengan una mayor capacidad de generalización. Esto podría implicar la recolección y anotación de más datos, o la utilización de técnicas de aumentación de datos para expandir nuestro conjunto de datos existente.
- **Implementación de modelos de lenguaje avanzados:** La exploración y aplicación de modelos de lenguaje preentrenados de última generación, como **GPT-4** o **PaLM 2**, pueden permitirnos mejorar considerablemente la precisión de nuestras inferencias y respuestas. Estos modelos representan lo último en innovación y desarrollo en **inteligencia artificial**, y su implementación podría suponer un salto cualitativo importante en nuestras capacidades de procesamiento del lenguaje natural.
- **Desarrollo de una aplicación integral:** Nuestro objetivo último es la creación de una aplicación completamente operativa, que se conecte mediante una API y emplee modelos **Transformer** para llevar a cabo funciones relevantes vinculadas con el procesamiento del lenguaje natural. Esta aplicación podría ser utilizada para una variedad de tareas, desde la traducción automática hasta el **análisis de sentimientos**, proporcionando soluciones prácticas a problemas del mundo real.

Además, es importante destacar la relevancia de explorar aplicaciones prácticas de los grandes modelos de lenguaje disponibles en la actualidad. No solo buscamos comprender y utilizar estas tecnologías, sino también investigar cómo pueden ser aplicadas a nuevos proyectos y desafíos, y cómo pueden ser adaptadas para cumplir con requisitos específicos en diferentes contextos.



BIBLIOGRAFÍA



- [1] Rodrigo Alonso, «Explorando la IA: Aprendizaje Automático vs. Aprendizaje Profundo», *harzone*, 2023.
<https://hardzone.es/tutoriales/rendimiento/diferencias-ia-deep-machine-learning/> (accedido 21 de junio de 2022).
- [2] Sonia Margarita, «Teoría de la inteligencia artificial para la calidad», *Gestiopolis*, 2008. <https://www.gestiopolis.com/teoria-inteligencia-artificial-calidad/> (accedido 26 de junio de 2022).
- [3] «Machine Learning y Deep Learning», *IIC*. <https://www.iic.uam.es/inteligencia-artificial/machine-learning-deep-learning/> (accedido 21 de junio de 2022).
- [4] Juan Lerma, «Cómo se comunican las neuronas», *CSIC*, 2010.
- [5] «¿Qué es una red neuronal? Guía de IA y ML», *AWS*.
<https://aws.amazon.com/es/what-is/neural-network/> (accedido 27 de junio de 2022).
- [6] D. Guillermo Garcia Murillo, C. Alberto Ruiz Marta Susana Basualdo Autor, y D. Jorge Matich, «Redes Neuronales: Conceptos Básicos y Aplicaciones», 2001.
- [7] «Conceptos básicos».
<http://grupo.us.es/gtocoma/pid/pid10/RedesNeuronales.htm> (accedido 3 de julio de 2022).
- [8] Ligdi González, «¿Qué es el Perceptrón? Perceptrón Simple y Multicapa», *Aprende IA*, 2021. <https://aprendeia.com/que-es-el-perceptron-simple-y-multicapa/> (accedido 12 de julio de 2022).
- [9] C. Santana, «Dot CSV», *Youtube*. <https://www.youtube.com/c/DotCSV> (accedido 12 de julio de 2022).
- [10] Frank La La, «Inteligencia artificial: ¿Cómo aprenden las redes neuronales?», *Microsoft Docs*, 2019. <https://docs.microsoft.com/es-es/archive/msdn-magazine/2019/april/artificially-intelligent-how-do-neural-networks-learn> (accedido 12 de julio de 2022).

- [11] «Función de pérdida en Machine Learning», *Sitio big data*, 2019.
<https://sitiobigdata.com/2019/12/24/funciones-comunes-de-perdida-en-el-aprendizaje-automatico/> (accedido 26 de mayo de 2023).
- [12] Farid Murzone, «Funciones de activación para redes neuronales», *Medium*, 2021. <https://medium.com/escueladeinteligenciaartificial/funciones-de-activaci%C3%B3n-para-redes-neuronales-de00fefb7150> (accedido 29 de junio de 2023).
- [13] P. Ramachandran, B. Zoph, y Q. V. Le, «Searching for Activation Functions», oct. 2017, [En línea]. Disponible en: <http://arxiv.org/abs/1710.05941>
- [14] D. Misra, «Mish: A Self Regularized Non-Monotonic Activation Function», ago. 2019, [En línea]. Disponible en: <http://arxiv.org/abs/1908.08681>
- [15] Miguel Sotaquirá, «La Función de Activación», *Codificando Bits*, 2018.
<https://www.codificandobits.com/blog/funcion-de-activacion/> (accedido 12 de julio de 2022).
- [16] Renu Khandelwal, «Overview of different Optimizers for neural networks», *Medium*, 2019. <https://medium.datadriveninvestor.com/overview-of-different-optimizers-for-neural-networks-e0ed119440c3> (accedido 27 de mayo de 2023).
- [17] R. M. Schmidt, «Recurrent Neural Networks (RNNs): A gentle Introduction and Overview», nov. 2019, [En línea]. Disponible en:
<http://arxiv.org/abs/1912.05911>
- [18] Miguel Sotaquirá, «Redes Neuronales Recurrentes: Explicación Detallada», *Codificando Bits*, 2019. <https://www.codificandobits.com/blog/redes-neuronales-recurrentes-explicacion-detallada/> (accedido 3 de agosto de 2022).
- [19] R. C. Staudemeyer y E. R. Morris, «Understanding LSTM -- a tutorial into Long Short-Term Memory Recurrent Neural Networks», sep. 2019, [En línea].
Disponible en: <http://arxiv.org/abs/1909.09586>

- [20] S. F. Fred Cummins, «Understanding LSTM Networks», *Colah's blog*, 2015. <https://colah.github.io/posts/2015-08-Understanding-LSTMs/> (accedido 4 de agosto de 2022).
- [21] Vijay Choubey, «Understanding Recurrent Neural Network (RNN) and Long Short Term Memory(LSTM)», *Medium*, 2020. <https://medium.com/analytics-vidhya/undestanding-recurrent-neural-network-rnn-and-long-short-term-memory-lstm-30bc1221e80d> (accedido 29 de junio de 2023).
- [22] M. E. Peters *et al.*, «Deep contextualized word representations», feb. 2018, [En línea]. Disponible en: <http://arxiv.org/abs/1802.05365>
- [23] Jacob Devlin and Ming-Wei Chang, «Open Sourcing BERT: State-of-the-Art Pre-training for Natural Language Processing», *Google AI Blog*, 2018. <https://ai.googleblog.com/2018/11/open-sourcing-bert-state-of-art-pre.html> (accedido 6 de junio de 2023).
- [24] C. Raffel *et al.*, «Exploring the Limits of Transfer Learning with a Unified Text-to-Text Transformer», oct. 2019, [En línea]. Disponible en: <http://arxiv.org/abs/1910.10683>
- [25] OpenAI, «GPT-4 Technical Report», mar. 2023, [En línea]. Disponible en: <http://arxiv.org/abs/2303.08774>
- [26] Atulanand, «Generative Pre-trained Transformer-4 (GPT-4)», *Medium*, 2023. <https://medium.com/@atulanand7535/generative-pre-trained-transformer-4-gpt-4-979353861f20> (accedido 6 de junio de 2023).
- [27] R. Thoppilan *et al.*, «LaMDA: Language Models for Dialog Applications», ene. 2022, [En línea]. Disponible en: <http://arxiv.org/abs/2201.08239>
- [28] Eli Collins, «LaMDA: our breakthrough conversation technology», *Goole Blog*, 2021. <https://blog.google/technology/ai/lamda/> (accedido 6 de junio de 2023).
- [29] A. Chowdhery *et al.*, «PaLM: Scaling Language Modeling with Pathways», abr. 2022, [En línea]. Disponible en: <http://arxiv.org/abs/2204.02311>

- [30] Sharan Narang and Aakanksha Chowdhery, «Pathways Language Model (PaLM): Scaling to 540 Billion Parameters for Breakthrough Performance», *Google AI Blog*, 2022. <https://ai.googleblog.com/2022/04/pathways-language-model-palm-scaling-to.html> (accedido 6 de junio de 2023).
- [31] B. Workshop *et al.*, «BLOOM: A 176B-Parameter Open-Access Multilingual Language Model», nov. 2022, [En línea]. Disponible en: <http://arxiv.org/abs/2211.05100>
- [32] Cristian Arteaga, «Understand BLOOM, the Largest Open-Access AI, and Run It on Your Local Computer», *Towards Data Science*, 2022. <https://towardsdatascience.com/run-bloom-the-largest-open-access-ai-model-on-your-desktop-computer-f48e1e2a9a32> (accedido 6 de junio de 2023).
- [33] R. Anil *et al.*, «PaLM 2 Technical Report», may 2023, [En línea]. Disponible en: <http://arxiv.org/abs/2305.10403>
- [34] Zoubin Ghahramani, «Google AI: Introducing PaLM 2», *Google Blog*, 2023. <https://blog.google/technology/ai/google-palm-2-ai-large-language-model/> (accedido 29 de junio de 2023).
- [35] Molly Ruby, «How ChatGPT Works: The Model Behind The Bot», *Towards Data Science*, 2023. <https://towardsdatascience.com/how-chatgpt-works-the-models-behind-the-bot-1ce5fca96286> (accedido 30 de junio de 2023).
- [36] Eva Rodriguez, «GPT-4, guía a fondo», *Genbeta*, 2023. <https://www.genbeta.com/a-fondo/gpt-4-guia-a-fondo-que-como-funcionara-cuando-se-lanzara-que-novedades-integrara-respecto-a-chatgpt-3-5> (accedido 30 de junio de 2023).
- [37] Sabrina Ortiz, «What is Bing Chat? Here's everything you need to know», *ZDNET*, 2023. <https://www.zdnet.com/article/what-is-the-new-bing-heres-everything-you-need-to-know/> (accedido 30 de junio de 2023).

- [38] Sundar Pichai, «Google AI updates: Bard and new AI features in Search», 2023. <https://blog.google/technology/ai/bard-google-ai-search-updates/> (accedido 30 de junio de 2023).
- [39] Yúbal Fernández, «PaLM 2: qué es, cómo funciona», *Xataka*, 2023. <https://www.xataka.com/basics/palm-2-que-como-funciona-que-puede-hacer-como-probar-este-modelo-lenguaje-inteligencia-artificial> (accedido 30 de junio de 2023).
- [40] A. Vaswani *et al.*, «Attention Is All You Need», jun. 2017, [En línea]. Disponible en: <http://arxiv.org/abs/1706.03762>
- [41] Tamal das, «Google Colab: todo lo que necesita saber», *Geekflare*, 2022. <https://geekflare.com/es/google-colab/> (accedido 19 de junio de 2023).
- [42] «NVIDIA T4 Tensor Core GPU for AI Inference», *NVIDIA Data Center*. <https://www.nvidia.com/en-us/data-center/tesla-t4/> (accedido 19 de junio de 2023).
- [43] «Python Numpy: Tutorial, What It is, Library», *Javatpoint*. <https://www.javatpoint.com/numpy-tutorial> (accedido 19 de junio de 2023).
- [44] «Introducción a Pandas, la librería de Python para trabajar con datos». <https://profile.es/blog/pandas-python/> (accedido 19 de junio de 2023).
- [45] «Matplotlib Tutorial», *javatpoint*. <https://www.javatpoint.com/matplotlib> (accedido 19 de junio de 2023).
- [46] «Introduction to TensorFlow», *GeeksforGeeks*. <https://www.geeksforgeeks.org/introduction-to-tensorflow/> (accedido 19 de junio de 2023).
- [47] «Scikit-Learn : Descubre la biblioteca de Python dedicada al Machine Learning», *DataScientest*, 2023. <https://datascientest.com/es/scikit-learn-decubre-la-biblioteca-python> (accedido 19 de junio de 2023).

- [48] «Usar la normalización Unicode para representar cadenas».
<https://docs.microsoft.com/es-es/windows/win32/intl/using-unicode-normalization-to-represent-strings> (accedido 2 de agosto de 2022).
- [49] «Limpieza de datos HTML en Python para PNL».
<https://manualestutor.com/ciencia-de-datos/limpieza-de-datos-html-en-python-para-pnl/> (accedido 2 de agosto de 2022).
- [50] «Espacio de recursos de ciencia de datos», *Universitat Oberta de Catalunya*.
<http://datascience.recursos.uoc.edu/es/procesamiento-del-lenguaje-natural-nlp/?filter=.dprocesamiento> (accedido 2 de agosto de 2022).
- [51] Lino Alberto Urdaneta Fernández, «Reducir el número de palabras de un texto: lematización y radicalización (stemming) con Python», *Medium*, 2029.
<https://medium.com/qu4nt/reducir-el-n%C3%BAmero-de-palabras-de-un-texto-lematizaci%C3%B3n-y-radicalizaci%C3%B3n-stemming-con-python-965bfd0c69fa> (accedido 2 de agosto de 2022).
- [52] Dhruvil Karani, «Introducción a Word Embedding y Word2Vec», *Towards Data Science*, 2018. <https://towardsdatascience.com/introduction-to-word-embedding-and-word2vec-652d0c2060fa> (accedido 3 de agosto de 2022).
- [53] «Embedding projector - visualization of high-dimensional data».
<http://projector.tensorflow.org/> (accedido 17 de julio de 2022).
- [54] Thomas Wood, «Transformer Neural Network Definition», *DeepAI*.
<https://deepai.org/machine-learning-glossary-and-terms/transformer-neural-network> (accedido 4 de agosto de 2022).
- [55] Thomas Wood, «What is Transformer Network», *Towards Data Science*, 2020.
<https://towardsdatascience.com/transformer-neural-network-step-by-step-breakdown-of-the-beast-b3e096dc857f> (accedido 4 de agosto de 2022).