



Universidad de Oviedo

TÉCNICAS DE MACHINE LEARNING PARA
PROCESAMIENTO DEL LENGUAJE NATURAL

Alejandra Navarro Castillo

Dirigido por
Elías Fernández-Combarro Álvarez

UNIVERSIDAD DE OVIEDO
Facultad de Ciencias
Grado en Matemáticas

Junio de 2022

Índice general

1. Introducción	5
1.1. Motivación	5
1.2. Objetivos	6
1.3. Estructura del trabajo	6
2. Preliminares	11
2.1. Extracción y depuración del texto	12
2.2. Construcción de un vocabulario	13
2.2.1. Tokenización	14
2.2.2. Normalización de las palabras	14
2.3. Representación numérica del texto	17
2.3.1. Codificación One-hot	18
2.3.2. Modelo Bolsa de Palabras o <i>Bag-of-Words</i>	19

2.3.3.	TF-IDF	20
2.3.4.	Word embeddings	21
2.3.5.	Representaciones distribuidas más allá de las palabras	26
3.	Técnicas clásicas de PLN	30
3.1.	Introducción al <i>machine learning</i> o aprendizaje automático	30
3.2.	El algoritmo de Naive-Bayes	32
3.3.	Support Vector Machine (SVM)	37
4.	Redes Neuronales	41
4.1.	La idea detrás de la red neuronal	41
4.2.	Funcionamiento de una red neuronal artificial	42
4.2.1.	Neuronas	42
4.2.2.	Funciones de activación	43
4.2.3.	Capas en una ANN (Artificial Neural Network)	44
4.2.4.	Aprendizaje de la red neuronal	46
4.2.5.	Descenso de gradiente	47
4.2.6.	Problemas de entrenamiento	50
4.3.	Redes neuronales recurrentes (RNN)	51

4.3.1. Arquitectura de una RNN	51
4.3.2. Retropropagación a través del tiempo	53
4.3.3. Problemas de entrenamiento en las redes neuronales recurrentes	56
4.4. Long short-term memory (LSTM)	58
4.4.1. Funcionamiento de una celda LSTM	59
4.4.2. Retropropagación en el tiempo en las LSTM	64
4.5. Gated Recurrent Unit (GRU)	65
5. Los transformers	67
5.1. Introducción y origen de los transformers	67
5.2. La estructura general	71
5.2.1. Encoder de un transformer	73
5.2.2. Decoder de un transformer	82
6. Tipos de transformers	84
6.1. BERT. Aplicaciones y fine-tune.	86
6.1.1. Arquitectura del modelo BERT	86
6.2. GPT-3. Aplicaciones de GPT-3.	93

7. Conclusiones 98

Bibliografía 99

Capítulo 1

Introducción

1.1. Motivación

El procesamiento del lenguaje natural o PLN (del inglés, *Natural Language Processing* o *NLP*) se podría definir como el campo de investigación que estudia cómo los ordenadores pueden entender y procesar el lenguaje humano con el objetivo de que exista una interacción entre humano y ordenador. Por lo tanto, este campo se encuentra en la intersección de las ciencias de la computación, la inteligencia artificial y la lingüística. Se ocupa de construir sistemas que puedan procesar y comprender el lenguaje humano.

Desde sus comienzos alrededor de 1950 y hasta muy recientemente, el PLN ha sido uno de los principales campos de estudio e investigación entre la comunidad científica. Además, los grandes avances en el desarrollo de las técnicas de la década pasada han hecho que el PLN haya experimentado un auge en un amplio rango de aplicaciones como ventas, sanidad, economía, derecho, marketing, recursos humanos y muchos más. Así como el PLN se está expandiendo cada vez más, también está aumentando el número de profesionales que quieren hacer uso de sus técnicas. Por lo tanto, el PLN se está convirtiendo rápidamente en una habilidad necesaria para científicos, ingenieros, managers de producción o estudiantes.

Desde mi perspectiva de estudiante de matemáticas, y dada la notable importancia que tienen las matemáticas en la construcción de las técnicas

y modelos del aprendizaje automático o *machine learning* utilizados para realizar tareas de PLN, considero este trabajo un primer acercamiento y estudio de este campo tan importante hoy en día en el mundo tecnológico-científico.

1.2. Objetivos

Con este trabajo de fin de grado, mi objetivo principal es estudiar y analizar, desde una perspectiva matemática, las técnicas de Machine Learning (ML) y Deep Learning (DP) más innovadoras que se han desarrollado en los últimos años para aplicaciones relacionadas con el campo del procesamiento del lenguaje natural. En segundo lugar, mostrar algunas de las aplicaciones prácticas que podemos realizar con la ayuda de estas técnicas.

Dentro de las aplicaciones más populares del PLN podemos encontrar las siguientes: clasificación de textos, extracción de información, respuesta a preguntas, traducción automática, síntesis de voz, recuperación de información, comprensión del lenguaje, reconocimiento del habla o generación de texto en lenguaje natural.

En concreto, se va a estudiar principalmente la arquitectura Transformer. Esta arquitectura es, hasta el momento, la que mejor procesa secuencias de texto y mejor llega a comprender el lenguaje. Desde su desarrollo e implementación en 2017, se han desarrollado muchos tipos de modelos basados en esta arquitectura y se han mejorado en gran medida muchas de las aplicaciones del PLN.

1.3. Estructura del trabajo

Este trabajo está redactado de manera autocontenido, es decir, hace a su lector minimizar la probabilidad de que no entienda o de que tenga que acudir a otro texto para entender este.

La estructura que tiene este trabajo es análoga a la de una *pipeline*¹ genérica para tareas de procesamiento del lenguaje natural. Así, si entendemos los procedimientos más habituales a la hora de trabajar en tareas relacionadas con el PLN, adquiriremos los conocimientos necesarios para empezar a desarrollar soluciones a cualquier problema en este campo.

El siguiente esquema resume de manera gráfica lo que podría ser una *pipeline* genérica para el desarrollo de sistemas de PLN.

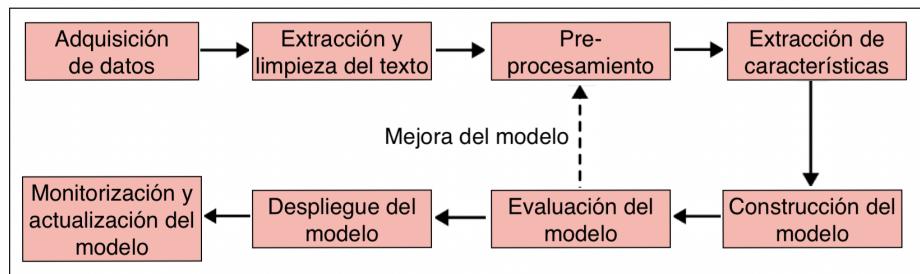


Figura 1.1: *Pipeline* genérica para tareas de PLN.

Expicaremos las etapas principales de la *pipeline* con más detalle:

1. Adquisición de los datos.

Los datos son el corazón de cualquier sistema de aprendizaje automático. De hecho, la falta de datos es normalmente la responsable de que gran parte de proyectos a nivel industrial se demoren o no se desarrollen debidamente. Para conseguir bases de datos útiles para desarrollar proyectos de PLN existen varias técnicas como: usar bases de datos públicas, hacer *web scraping*², realizar técnicas de aumentación de datos, etc.

¹El término *pipeline* se usa para referirse al progreso o desarrollo hacia un objetivo a largo plazo que involucra una serie de pasos distintos.

²*Web scraping* o raspado web es el nombre que recibe la técnica utilizada mediante programas de software para extraer información de páginas web.

Al conjunto de datos de texto obtenidos que servirán posteriormente para la aplicación de tareas de procesamiento del lenguaje natural se le denomina “corpus”.

2. **Extracción y limpieza del texto.** Esta etapa se refiere al proceso de extraer texto puro de los datos de entrada eliminando toda la información que no sea texto, como anotaciones, metadatos³, etc. Además se debe convertir el texto al formato deseado.

Este paso de la *pipeline* puede ser uno de los más costosos en tiempo y esfuerzo y, además, no se suelen emplear técnicas específicas de PLN. Sin embargo, es una etapa importante ya que tiene implicaciones en todas las etapas posteriores.

3. **Pre-procesamiento.**

Al llegar a esta etapa, si hemos completado correctamente los pasos anteriores, tendríamos un texto puro y sencillo. Sin embargo, el software que realiza tareas de PLN trabaja normalmente en contextos de frases y necesita separar el texto por lo menos en palabras (es posible obtener unidades incluso más pequeñas separando el texto en sílabas o letras directamente).

Además, algunas veces se tienen que eliminar caracteres especiales y dígitos, o convertir todo en letra minúscula porque nos dan igual las mayúsculas en una tarea concreta. Las decisiones de este tipo son las que se ejecutan durante el pre-procesamiento del texto. Algunos de los pasos más comunes serían: *tokenización*, eliminación de *stopwords*, *stemming*, lematización, conversión en minúsculas, etc, los cuales se explican en profundidad en el capítulo 2.

4. **Feature engineering, feature extraction o extracción de características.**

Hasta ahora hemos visto los diferentes pasos del pre-procesamiento de los datos. Pero cuando se usan algoritmos de aprendizaje automático o *machine learning (ML)* para construir posteriormente un modelo, se necesita todavía una forma de introducir los datos de texto al algoritmo. De esto se encarga la extracción de características: el conjunto de

³Los metadatos son datos que describen otros datos. Los metadatos dan información sobre uno o más aspectos de un conjunto de datos. El concepto de metadatos es análogo al uso de índices para localizar objetos. Por ejemplo, en una biblioteca se usan fichas que especifican autores, títulos, casas editoriales y lugares para buscar libros. En este ejemplo, las fichas serían los metadatos.

métodos que capturan las características del texto y las transforman en vectores numéricos que pueden ser entendidos y manipulados por algoritmos de ML. En este trabajo, este paso es denominado como *Representación numérica de texto* y se encuentra en la sección 2.3.

5. Construcción del modelo.

El siguiente paso en la *pipeline* de PLN es la construcción del modelo, lo cual incluye diseñar un modelo estadístico o de aprendizaje automático, ajustar sus parámetros con los datos de entrenamiento, utilizar un procedimiento de optimización y por último usarlo para realizar predicciones en nuevos conjuntos de datos.

Una de las ventajas de trabajar con características de valores numéricos es que permite elegir cualquier modelo de ML e incluso una combinación de ellos.

6. Evaluación del modelo.

Una fase clave dentro del *pipeline* de PLN es la de medir cuánto de “bueno” es el modelo que hemos construido. Medir la “bondad” del modelo se puede realizar de varias formas, pero la más común es medir el comportamiento del modelo en nuevos conjuntos de datos, lo cual depende de dos factores: (1) usar la métrica correcta para la evaluación, y (2) realizar el proceso de evaluación correcto.

Las métricas pueden variar dependiendo de la tarea de PLN que estamos resolviendo y dependiendo de qué fase estemos midiendo. Además, también hay dos tipos de evaluación: la intrínseca y la extrínseca. La intrínseca se focaliza en objetivos intermedios de la *pipeline*, y por el contrario la extrínseca se ocupa de evaluar el comportamiento del modelo en su objetivo final.

7. Despliegue de software o *Deployment*.

Una vez nuestro modelo ya ha sido diseñado, probado y funciona correctamente, nos movemos a la fase post-modelización: despliegue, monitorización y actualización del modelo.

En la mayoría de aplicaciones, el modelo de PLN que estamos implementando forma parte de un sistema más grande (por ejemplo un clasificador de correos spam dentro de una aplicación de correo electrónico). Entonces, cuando el modelo funciona correctamente, se tiene que desplegar en un ambiente de producción como parte de un sistema más grande (por ejemplo una aplicación web o móvil).

8. Monitorización y actualización del modelo.

Como en todo proyecto de desarrollo de software, el modelo se sigue monitorizando constantemente después del despliegue. Además, se van

recogiendo más datos y así se puede poner al día el modelo basándose en los nuevos datos.

Acabamos de ver y comentar todas las etapas de la *pipeline* de PLN más comunes, pero en este trabajo sólo veremos en detalle algunas de ellas. En el capítulo 2, veremos la extracción y limpieza del texto, así como las técnicas de pre-procesamiento del texto más comunes y los métodos habituales para la extracción de características. Continuaremos explicando, en el capítulo 3, algunos de modelos más clásicos de ML utilizados para resolver tareas de PLN. Y, finalmente, en los últimos capítulos, veremos en qué consisten las redes neuronales y explicaremos una arquitectura novedosa y revolucionaria en el campo del PLN: los transformers.

Capítulo 2

Preliminares

El PLN es una disciplina que se encarga de intentar unificar el lenguaje humano natural con el lenguaje de los ordenadores. Evidentemente, para que una máquina pueda llegar a procesar el lenguaje humano, tiene que existir una especie de “traducción” de un tipo de lenguaje al otro. Esta traducción consiste en convertir el texto en representaciones numéricas. Para ello, se realiza el pre-procesamiento de los datos de texto.

El pre-procesamiento de los datos es probablemente el aspecto más importante de cualquier algoritmo de aprendizaje automático, por lo que también es aplicable a las tareas del procesamiento del lenguaje natural.

En este capítulo se van a explicar y analizar algunas técnicas de extracción y depuración del texto (del inglés, *data cleaning*) así como el pre-procesamiento y representación de texto. Nos sumergiremos en las metodologías que se utilizan para representar texto de forma numérica, recogiendo la información sintáctica¹ y semántica².

¹Sintaxis: las reglas que gobiernan la combinación de las palabras para formar unidades más grandes como sintagmas o frases.

²Semántica: el estudio del significado, sentido o interpretación de unidades lingüísticas.

2.1. Extracción y depuración del texto

El primer paso en el procesamiento del texto es extraer y depurar los datos, es decir, eliminar toda la información no textual innecesaria, como por ejemplo las etiquetas HTML, para conseguir texto puro. Veamos algunos ejemplos y aspectos importantes para extraer texto y depurarlo:

- **Analizar y depurar HTML.** A menudo, los datos de texto se obtienen haciendo *web scraping* en páginas web con el objetivo de recuperar información. Como estos sitios web son, en su gran mayoría, páginas HTML, en los datos de texto se van a encontrar etiquetas HTML y por ello se necesita un procedimiento para eliminar estas etiquetas del resto del texto.

Para ello es frecuente utilizar librerías de Python³ como BeautifulSoup o Scrapy. Típicamente, todas las librerías de HTML tienen alguna función para extraer todas las etiquetas HTML y devolver sólo el contenido entre etiquetas.

Sin embargo, hay casos en los que las etiquetas también pueden aportar información específica. Por ejemplo pensemos en el caso de una página web como Amazon que usa etiquetas específicas para identificar las diferentes características de un producto, por ejemplo una etiqueta <price> que pueda estar creada para expresar el precio de cada producto. En un escenario como ese, las etiquetas HTML pueden llegar a ser de gran utilidad, pero normalmente éstas son totalmente innecesarias para la mayoría de textos usados en el PLN.

- **Normalización de Unicode.** En textos sacados de páginas web también es común encontrarse con caracteres de Unicode⁴, que incluyen símbolos, emojis y caracteres gráficos. Con el objetivo de analizar estos símbolos no textuales, se usa la normalización de Unicode.

³Python es uno de los lenguajes de programación más utilizados en el campo del procesamiento del lenguaje natural.

⁴Unicode es un estándar de codificación de caracteres diseñado para facilitar el tratamiento informático, transmisión y visualización de textos de numerosos idiomas y disciplinas técnicas, además de textos clásicos de lenguas muertas. El término Unicode proviene de los tres objetivos perseguidos: universalidad, uniformidad y unicidad.

Este proceso consiste en convertir el texto que se muestra en un tipo de representación binaria para almacenar en un ordenador. Este proceso es conocido como “codificación de caracteres” (del inglés, *text encoding*).

- **Corrección de la ortografía.** En este mundo, en el cual estamos constantemente tecleando rápidamente, los textos de entrada contienen frecuentemente errores de ortografía. Este hecho puede ser dominante en motores de búsqueda, chatbots de escritura, redes sociales y muchas otras aplicaciones. Este tipo de errores de escritura, junto con la taquigrafía⁵ en mensajes de texto, dificultan en gran medida el procesamiento del lenguaje y la comprensión del contexto.

Por esa razón existen algunos métodos para mitigar este problema, como por ejemplo una REST API desarrollada por Microsoft que puede ser usada en Python para corregir potencialmente la ortografía de las palabras.

Aunque las técnicas de PLN juegan un rol muy pequeño en la extracción y depuración de texto, hemos visto algunos ejemplos de conflictos que pueden surgir durante este proceso y posibles soluciones para resolverlos. Al acabar esta fase, nos encontramos con un texto puro con el cual se puede trabajar en las etapas posteriores.

2.2. Construcción de un vocabulario

Una vez hemos conseguido textos puros, lo siguiente que se debe tener en cuenta a la hora de procesar el lenguaje natural es cómo se representa este lenguaje de forma que un ordenador lo pueda entender y procesar. Con este objetivo, se necesita construir un vocabulario, es decir, un conjunto de palabras que el ordenador pueda procesar. A continuación se van a explicar las técnicas y métodos que se aplican a datos de texto más usuales y utilizadas entre los profesionales del campo.

⁵La taquigrafía o estenografía es todo aquel sistema de escritura rápido y conciso que permite transcribir un discurso a la misma velocidad a la que se habla. Para ello se suelen emplear trazos breves, abreviaturas y caracteres especiales para representar letras, palabras e incluso frases.

2.2.1. Tokenización

El software para el PLN normalmente analiza el texto segmentándolo en frases y palabras (unidades llamadas *tokens*), con el objetivo de construir un vocabulario. Un *token* se caracteriza por poseer significado semántico propio. Cualquier *pipeline* de PLN, por lo tanto, debe contener un sistema de separación de los textos en frases y después en palabras. A este proceso de conseguir tokens a partir de un texto más largo se le llama *tokenización*.

2.2.2. Normalización de las palabras

Con el objetivo de simplificar el vocabulario que construimos, se pueden aplicar algunas técnicas muy útiles pero a la vez muy intuitivas. Vamos a ver las diferentes técnicas con los ejemplos correspondientes:

- **Stemming.**

Stemming es el método para reducir una palabra a su raíz (en inglés, *stem*). Por ejemplo, si tenemos las palabras *biblioteca* y *bibliotecario* ambas tienen la misma raíz que sería *bibliotec*. La parte restante de las palabras se llama afijo, que en el ejemplo concreto quedarían como *a* y *ario*. Una cosa a notar es que la raíz de la palabra no tiene por qué ser una palabra en sí.

En esta técnica de normalización lleva consigo potenciales problemas como el *over-stemming* y el *under-stemming*. El *over-stemming* ocurre cuando palabras para las cuales se ha encontrado la misma raíz en realidad tendrían que haber tenido raíces diferentes. Sin embargo, en el *under-stemming* las palabras que deberían tener la misma raíz, tienen raíces diferentes.

- **Lematización.**

A diferencia del *stemming*, donde a las palabras se le quitan algunos caracteres directamente para sacar su raíz, la lematización es un proceso lingüístico que consiste en, dada una forma flexionada (es decir, en plural, en femenino, conjugada, etc), hallar el lema correspondiente. Es decir convierte la palabra en su forma básica de significado. Esta técnica es útil para agrupar palabras que tengan un lema común y así identificarlas como iguales.

Los algoritmos que realizan la lematización intentan encontrar el lema de una palabra teniendo en cuenta su contexto, su categoría gramatical, el significado de la palabra, etc. El contexto que se tiene en cuenta puede ser desde unas pocas palabras, frases en incluso documentos.

Además algunas palabras pueden tener lemas diferentes dependiendo del contexto. El lematizador identifica las características gramaticales de las palabras del contexto para extraer el lema correcto.

- **Eliminación de palabras vacías o *stopwords*.**

Palabras vacías se les llama a las palabras que no dan mucha información como los artículos, pronombres o preposiciones y que aparecen frecuentemente en textos. Por lo general estas palabras son necesarias para completar frases y hacerlas gramaticalmente correctas. Por esta razón suelen ser las palabras más comunes en una lengua y por ello se pueden filtrar para la mayoría de tareas del PLN, reduciendo consecuentemente el vocabulario.

Hay que notar que no existe una lista universal de palabras vacías, ya que dependiendo del caso de uso las palabras varían.

La eliminación de stopwords es generalmente el primer paso después de la tokenización cuando se está construyendo el vocabulario o preprocesando los datos de texto.

- **Convertir el texto en minúsculas o *Case folding*.**

Otra de las estrategias que se suelen llevar a cabo en la normalización es la conversión de todas las letras a minúscula. Por ejemplo, *El* y *el* serán tratadas iguales en el caso de haber aplicado la conversión a minúsculas. Esta técnica ayuda en sistemas de recuperación de información, como los motores de búsqueda.

Sin embargo, en algunas situaciones la conversión a minúsculas puede llegar a ser una traba. Esto es por ejemplo cuando los nombres propios derivan de nombres comunes. Por ejemplo, *General Motors* se compone de dos nombres comunes pero él en sí mismo es un nombre propio.

Otro problema frecuente es cuando los acrónimos se convierten en minúsculas. En este caso también hay una probabilidad muy alta de que estos acrónimos se consideren como nombres comunes. Por ejemplo *CAT* (Common Admission Test) sería considerado como *cat* (gato en inglés).

Una solución potencial es crear modelos de ML que utilicen las características de una frase para determinar qué palabras o tokens de la frase deben estar en minúscula y cuáles no. Sin embargo, este enfoque no siempre ayuda si los usuarios escriben todo en minúscula.

- **N-gramas.**

Hasta ahora, sólo nos hemos fijado en tokens de tamaño 1, lo que significa una sola palabra. Sin embargo, las frases contienen generalmente nombres de personas, lugares y otros nombres compuestos como *sala de estar* o *mesilla de noche*.

Estas expresiones llevan consigo un significado especial cuando se usan juntas dos o más palabras. Si se usaran individualmente, significarían algo notablemente distinto y el significado inherente del término compuesto se perdería.

Por ello, el uso de múltiples tokens para representar ese significado inherente puede ser de gran utilidad para realizar las tareas de PLN. La técnica que se usa es la de agrupar los términos en n-gramas. Cuando n es igual a 1, éstos se llaman unigramas. Los bigramas o 2-gramas se refieren a parejas de palabras como por ejemplo *Buenos Aires*. Cuando la expresión se compone de 3 palabras como en *Emiratos Árabes Unidos* se habla de trígrama o 3-gramas. En la mayoría de tareas de NLP se usan sólo hasta los trígramas, a pesar de que el concepto de n-grama se puede extender a n-gramas más grandes.

- **Etiquetado gramatical o *POS-tagging*.**

En algunos casos particulares, por ejemplo si nuestra tarea es identificar nombres de personas en una colección de documentos, una etapa de pre-procesamiento más avanzado es el etiquetado gramatical de las palabras. No se va a explicar cómo están construidos los algoritmos encargados de esta tarea, pero existen modelos ya pre-entrenados implementados en librerías de PLN como NLTK o spaCy.

Acabamos de ver muchas de las técnicas necesarias para construir un vocabulario de lenguaje natural. Así, hemos estudiado la parte más crucial del pre-procesamiento de los datos de texto. En el siguiente esquema se muestra un orden lógico en los pasos a seguir para el pre-procesamiento del texto.

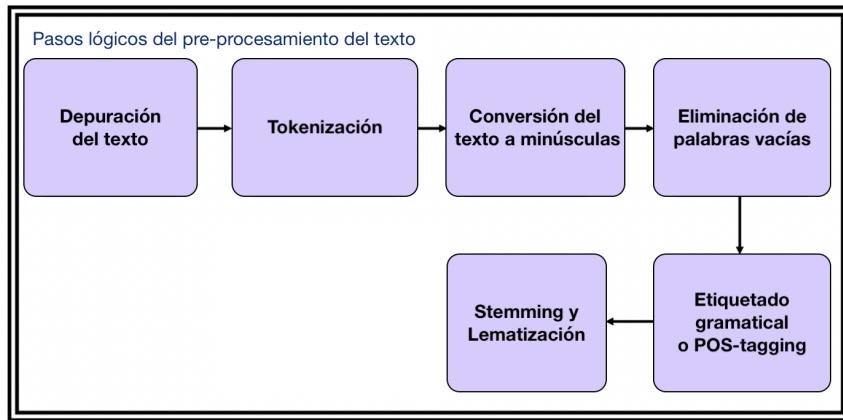


Figura 2.1: Pasos en el pre-procesamiento del texto.

2.3. Representación numérica del texto

Cuando se usan métodos de aprendizaje automático para ejecutar y construir un modelo, se necesita que los datos sean interpretables por el algoritmo. Sin embargo, los datos en forma de texto o cadena de caracteres no tienen ninguna representación directa o intuitiva en forma de números para poder ser procesados por un ordenador. Por ello, se deben estudiar diferentes técnicas y enfoques para representar numéricamente la máxima información del texto.

Dependiendo del enfoque (sintáctico o semántico), las técnicas que se van a describir van a ser las siguientes:

- Codificación One-hot
- Bolsa de Palabras o *Bag-of-Words*
- TF-IDF
- *Word embeddings* o Representaciones distribuidas de palabras
- Representaciones distribuidas más allá de las palabras

2.3.1. Codificación One-hot

En la codificación one-hot, a cada palabra w del vocabulario del corpus se le atribuye un número entero como identificador w_{id} que está entre 1 y $|V|$, donde V es el conjunto de palabras que forman el vocabulario del corpus dado. Entonces cada palabra es representada por un vector binario $|V|$ -dimensional de ceros y unos. La forma del vector one-hot serán ceros en todas sus entradas salvo en el índice del identificador de la palabra, donde habrá un 1. La forma de representar frases a través de la codificación one-hot es entonces combinar los vectores de las palabras que forman esa frase.

Vamos a entender la técnica mediante un ejemplo. Tomemos el corpus de ejemplo que se muestra en la tabla 2.1 con sólo cuatro documentos D_1 , D_2 , D_3 y D_4 .

D_1	El perro muerde al hombre.
D_2	El hombre muerde al perro.
D_3	El perro come carne.
D_4	El hombre come comida.

Tabla 2.1: Nuestro ejemplo de corpus.

Primero, asignamos a cada palabra un ID: $El = 1$, $perro = 2$, $muerde = 3$, $al = 4$, $hombre = 5$, $come = 6$, $carne = 7$, $comida = 8$. Hay que notar que esta es una asignación arbitraria; cualquier otra funcionaría igualmente. Por construcción, cada palabra es un vector de dimensión 8, representado en la siguiente tabla:

El	[1,0,0,0,0,0,0,0]
$perro$	[0,1,0,0,0,0,0,0]
$muerde$	[0,0,1,0,0,0,0,0]
al	[0,0,0,1,0,0,0,0]
$hombre$	[0,0,0,0,1,0,0,0]
$come$	[0,0,0,0,0,1,0,0]
$carne$	[0,0,0,0,0,0,1,0]
$comida$	[0,0,0,0,0,0,0,1]

Tabla 2.2: Representación one-hot de cada palabra del vocabulario.

Por lo tanto si consideremos el documento D_3 , “El perro come carne”, su representación one-hot es [[1,0,0,0,0,0,0], [0,1,0,0,0,0,0], [0,0,0,0,1,0,0], [0,0,0,0,0,1,0]]. Los demás documentos del corpus se pueden representar de forma análoga.

Ahora que ya se ha entendido el esquema, discutamos algunos de sus pros y contras. La mayor ventaja es que la codificación one-hot es muy intuitiva de entender y directa para implementar. Sin embargo, tiene algunos inconvenientes:

- La dimensión de un vector one-hot es directamente proporcional al tamaño del vocabulario (además la mayoría de corpora del mundo real tienen grandes vocabularios). Por lo tanto, esto resulta en representaciones dispersas (del inglés *sparse*), donde la gran parte de las coordenadas de los vectores son ceros, haciendo que sea ineficiente almacenarlos y manipularlos computacionalmente.
- La representación de cada texto no tiene una longitud fijada, es decir, si un documento tiene 10 palabras, su representación tendrá 10 vectores pero un documento de 5 palabras tendrá 5 vectores para representarlo.
- Trata a las palabras como unidades indivisibles y no considera la similitud o disimilitud entre las palabras en términos de significado.
- No hay forma de representar palabras que no estuvieran previamente en el conjunto de entrenamiento. Este problema se conoce como el problema “fuera de vocabulario” (del inglés *out of vocabulary (OOV)*). La única forma de solventar el problema sería reentrenando el modelo para incluir la nueva palabra en el vocabulario.

2.3.2. Modelo Bolsa de Palabras o *Bag-of-Words*

Una forma muy intuitiva de representar un texto puede ser usando la frecuencia de las palabras que aparecen en ese mismo texto. Esta técnica solo aporta información sintáctica, ya que no se tiene en cuenta la componente semántica, es decir, el significado del texto.

El objetivo del modelo *Bag-of-Words (BoW)* consiste en construir un vocabulario dada una lista de frases. Su funcionamiento consiste en representar cada frase por un vector de longitud el tamaño del vocabulario. Cada

coordenada del vector corresponde a un término del vocabulario y el valor de cada coordenada es la frecuencia del término en la frase considerada. El valor mínimo sería 0, e indicaría que el término no se encuentra en la frase representada.

Este modelo simple cuenta con un gran número de limitaciones, como era de esperar, dado que se basa sólo en el recuento de palabras y no tiene para nada en cuenta el contexto o significado de las mismas:

- El *BoW* reduce la importancia de los tokens o las frases que aparecen raramente. Sin embargo, que un token o frase no tenga un número de apariciones alto no quiere decir que no sea importante en el texto general.
- Otro de los grandes problemas de esta técnica es que no tiene en cuenta la semántica. Tampoco incluye la posibilidad de representar el contexto de una palabra o frase.
- Por último, el modelo *BoW* se vuelve inabordable si el vocabulario que se está utilizando contiene un gran número de tokens. Esto puede llevar al hecho de tener que manejar vectores de grandes dimensiones, lo que puede causar un deterioro en la ejecución del modelo.

2.3.3. TF-IDF

La técnica de TF-IDF es la más utilizada a la hora de medir la relevancia de una palabra o término en un documento de una colección. Su nombre viene del inglés “Term frequency - Inverse document frequency” (frecuencia de término - frecuencia inversa de documento).

El valor del TF-IDF aumenta proporcionalmente al número de veces que un término aparece en un documento y a su vez disminuye con la frecuencia de dicho término en la colección de documentos, lo que permite distinguir las palabras que son generalmente más comunes que otras.

Matemáticamente, el valor TF-IDF es el producto de dos medias: frecuencia del término en un documento y frecuencia inversa de documento. Para ello, se utilizan las fórmulas siguientes:

$$TF(t, d) = \frac{\text{No. veces que el término } t \text{ se encuentra en el documento } d}{\text{No. total de términos en el documento } d} = \\ = \frac{f(t, d)}{|d|}$$

$$IDF(t, C) = \log \frac{\text{Número total de documentos en la colección } C}{\text{Número de documentos que contienen el término } t} = \\ = \log \frac{|C|}{|\{d \in C : t \in d\}|}$$

$$TFIDF(t, d, C) = TF(t, d) \times IDF(t, C)$$

Por lo tanto, un peso alto en la medida TF-IDF se alcanza cuando la frecuencia del término en el documento es alta y cuando la ocurrencia del término en la colección de documentos es baja. Hay que notar que el valor del TF-IDF es siempre mayor o igual que 0.

La representación vectorial de tokens a través de esta técnica cuenta con algunos inconvenientes, ya que, aunque mejora la técnica de *Bag-of-Words* ponderando la relevancia de los términos, todavía sigue basándose en el recuento de palabras y no llega a tener en cuenta por ejemplo la coocurrencia⁶ de términos, la semántica, el contexto asociado a cada término o su posición en un documento. Además, como en el caso de *Bag-of-Words*, es muy sensible al tamaño del vocabulario y la manipulación de vectores de grandes dimensiones puede ralentizar la ejecución del modelo.

2.3.4. Word embeddings

Los métodos de representación de tokens o palabras vistos hasta ahora reflejan básicamente sólo el aspecto sintáctico de un texto en la forma de presencia o ausencia del token en un documento, lo cual conlleva una serie

⁶Coocurrencia: en lingüística, se refiere a la utilización conjunta de dos unidades léxicas (por ejemplo palabras) en una unidad superior, como una palabra o documento.

de limitaciones a la hora de representar el significado del texto. Es importante, por lo tanto, prestarle atención al contexto de una palabra ya que nos proporciona mucha información acerca de la semántica o significado de ésta. Esta idea es la que trata de recoger la técnica de la que se ocupa esta sección: word-embeddings.

Word-embedding se le llama a la representación aprendida de un token usando un vector en un espacio vectorial n-dimensional. La idea de estas representaciones vectoriales es que palabras con significados parecidos deben tener asociados vectores cercanos en el espacio vectorial. Esta técnica usa la arquitectura de red neuronal (se explicará más adelante en el capítulo 4) para crear las representaciones n-dimensionales y densas de palabras o textos.

El trabajo que dio pie a esta idea de word-embedding fue el paper *Efficient Estimation of Word Representations in Vector Space* [6], publicado en 2013 por Mikolov et al., el cual mostraba como su modelo de representación de palabras basado en redes neuronales conocido como “Word2vec” podía capturar relaciones de analogía entre palabras como por ejemplo:

$$\text{Rey} - \text{Hombre} + \text{Mujer} \approx \text{Reina}$$

El modelo Word2vec aprende las relaciones semánticas entre palabras y además garantiza que la representación aprendida de la palabra es un vector de baja dimensión (se obtienen vectores de dimensiones 50 a 500, en vez de miles tal y como se ha visto previamente en este capítulo) y es un vector denso (es decir, la gran parte de sus entradas son distintas de cero). Así se consigue que las tareas de aprendizaje automático sean más manejables y eficientes.

Veamos de forma intuitiva cómo funciona Word2vec. El objetivo es aprender el embedding para cada una de las palabras del corpus tal que su vector capture lo mejor posible el significado de la palabra en el espacio vectorial. Para “obtener” el significado de la palabra, se usa la semántica distribucional⁷, es decir, el significado de una palabra se deriva de su contexto (palabras

⁷La semántica distribucional es un área de investigación que desarrolla y estudia las teorías y métodos para cuantificar y categorizar las similitudes semánticas entre elementos lingüísticos, según sus propiedades distribucionales en grandes muestras de datos. La idea básica de la semántica distribucional se puede resumir en la llamada hipótesis distribucio-

que aparecen en su entorno). Técnicamente, Word2vec proyecta el significado de las palabras en un espacio vectorial donde las palabras con significados parecidos van a tender a agruparse, y las palabras con significados muy distintos estarán lejos entre sí.

Conceptualmente, Word2vec coge un corpus de texto grande como entrada y “aprende” a representar las palabras en un espacio vectorial común basado en los contextos en los cuales han aparecido en el corpus. Dada una palabra w y las palabras que aparecen en su contexto C , ¿cómo se calcula el vector que mejor representa el significado de la palabra? Para cada palabra w en el corpus, se empieza con un vector v_w inicializado con valores aleatorios. El modelo Word2vec refina los valores de v_w , dados los vectores de las palabras del contexto C . Esto lo realiza mediante una red neuronal de dos capas. Las redes neuronales se explicarán en detalle en el capítulo 4.

Entrenar nuestros propios embeddings.

Para entrenar nuestros propios embeddings, estudiaremos las variantes propuestas según el enfoque del paper original: (1) Continuous bag of words (CBOW) o Bolsa de palabras continuo, y (2) Skip-Gram.

Continuous Bag of Words (CBOW).

En el modelo CBOW, la tarea principal es construir un modelo de lenguaje que prediga correctamente una palabra central dadas las palabras de su entorno.

Pero primero, ¿qué es un modelo de lenguaje? Se trata de un modelo estadístico que intenta obtener una distribución de probabilidad para secuencias de palabras. Es decir, dada una frase de m palabras, el modelo asigna la probabilidad $P(w_1, w_2, \dots, w_m)$ a toda la frase. El objetivo de un modelo de lenguaje es el de asignar probabilidades para que las frases “buenas” (frases correctas semánticamente y sintácticamente) tengan probabilidades altas y

nal: «elementos lingüísticos con distribuciones similares tienen significados similares».

las “malas” (frases incorrectas) probabilidades bajas.

Veamos un ejemplo ilustrativo de cómo funciona este modelo. Empezamos explicando cómo obtiene las muestras de entrenamiento. Si tomamos la frase: *El zorro veloz saltó sobre el perro vago*, y tomamos como palabra central la palabra “saltó”, entonces las palabras de su contexto son las que se encuentran alrededor de la palabra central. Podemos elegir el tamaño del contexto mediante el hiper-parámetro k que representa cuántas palabras a cada lado de la palabra central se deben considerar como contexto. Si tomamos $k = 2$, entonces tendremos un tamaño de ventana de $2k + 1$ palabras, es decir 5 palabras.

Los datos obtenidos del corpus se representan como (X, Y) , donde X es el contexto e Y es la palabra objetivo. Para obtener el siguiente dato, simplemente se desplaza la ventana una palabra a la derecha y se repite el proceso. Cuando la ventana se ha desplazado por todo el corpus, entonces hemos finalizado de crear el conjunto de datos de entrenamiento. Veamos un ejemplo gráfico:

Texto fuente		Muestras de entrenamiento (contexto, objetivo)
El zorro veloz saltó sobre el perro vago.	————→	((zorro, veloz), El)
El zorro veloz saltó sobre el perro vago.	————→	((El, veloz, saltó), zorro)
El zorro veloz saltó sobre el perro vago.	————→	((El, zorro, saltó, sobre), veloz)
El zorro veloz saltó sobre el perro vago.	————→	((zorro, veloz, sobre, el), saltó)

Figura 2.2: Preparación del dataset de entrenamiento para CBOW.

Una vez obtenidos los datos de entrenamiento, vamos a estudiar la arquitectura del modelo. Para ello, se construye una red neuronal de dos capas como se muestra en la figura 2.3. Suponemos que queremos generar los embeddings de dimensión N y sea V el vocabulario del corpus de texto.

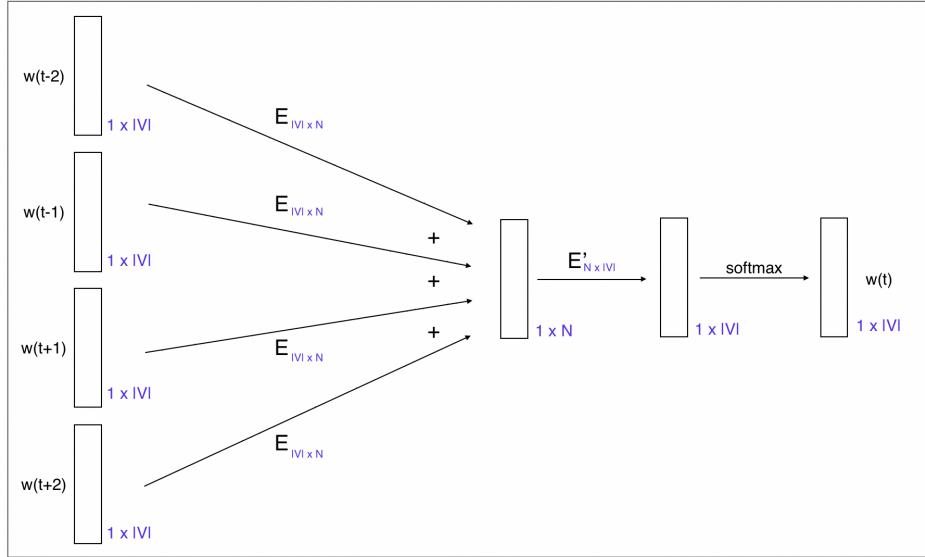


Figura 2.3: Arquitectura del modelo CBOW.

El objetivo es aprender la matriz de embeddings $E_{|V| \times N}$. Se trata de una red neuronal en la que los valores de entrada son los vectores one-hot de las palabras del contexto. Éstos se multiplican con la matriz de embeddings y posteriormente se suman para obtener el vector de dimensión N que pasará a la siguiente capa de la red. La siguiente capa toma este vector y lo multiplica con otra matriz $E'_{N \times |V|}$ para dar un vector $1 \times |V|$. Este último vector de salida pasa a través de la función *softmax*⁸ para determinar la distribución de probabilidad de las palabras del vocabulario. Esta distribución se compara con el vector one-hot de la palabra objetivo y así se calcula la función de pérdida y se ejecuta la retropropagación (explicada en detalle en el capítulo 4) para actualizar ambas matrices E y E' . Al final de la fase de entrenamiento la matriz E es la matriz de embeddings aprendida⁹ que deseábamos obtener.

⁸La función softmax o función exponencial normalizada es una generalización de la función logística. Se emplea para comprimir un vector d-dimensional de valores reales arbitrarios en un vector d-dimensional de valores reales en el rango [0,1]. La función está dada por $\sigma : \mathbb{R}^K \rightarrow [0, 1]^K$, donde $\sigma(\mathbf{z})_j = \frac{e^{z_j}}{\sum_{k=1}^K e^{z_k}}$.

⁹En realidad, técnicamente ambas E y E' son dos matrices de embeddings aprendidas

Skip-Gram.

La arquitectura del modelo Skip-Gram es muy similar a CBOW. La principal diferencia está en que mientras que el objetivo de CBOW es predecir la palabra central dadas las palabras del contexto, en Skip-Gram el modelo intenta predecir una palabra del contexto dada la palabra central como input al modelo.

Hay que añadir que, mientras CBOW es más rápido, Skip-Gram desempeña mejor las predicciones en palabras poco frecuentes.

2.3.5. Representaciones distribuidas más allá de las palabras

Nivel de caracteres.

Hasta ahora hemos visto ejemplos de embeddings de palabras, lo cual nos da una representación compacta y densa de las palabras de nuestro vocabulario. Pero se tiene que tener en cuenta que los embeddings de palabras que obtenemos con modelos pre-entrenados o con nuestro propio modelo dependen del vocabulario que hayan visto en los datos de entrenamiento. Sin embargo, no se tiene ninguna garantía de que los datos de producción para la aplicación que estemos construyendo estén en el vocabulario obtenido. A pesar de la facilidad con la que Word2vec realiza la extracción de características de los textos, todavía no tenemos un sistema para manejar las palabras fuera del vocabulario.

Un posible enfoque para tratar con el problema OOV es modificar el proceso de entrenamiento para tener en cuenta los caracteres y otras estructuras lingüísticas más pequeñas que las palabras, como morfemas, lexemas, afijos, sílabas o conjuntos de caracteres. Uno de los algoritmos más populares que sigue este enfoque fue desarrollado por *Facebook AI research* y se conoce como *fastText*.

diferentes. Se puede usar ambas o también una combinación como la media de las dos.

La idea de este modelo es que una palabra puede estar representada por sus n-gramas constituyentes. Es decir, si tomamos la palabra “espectacular”, podemos dividirla en sus n-gramas (por ejemplo con $n = 3$) - esp, spe, pec, ..., lar - y así combinar los embeddings aprendidos de cada n-grama para obtener el embedding de la palabra “espectacular”. Así, lo que realiza este algoritmo es aprender los embeddings de palabras y de n-gramas de forma conjunta y considera el vector embedding de una palabra como una agregación de sus propios n-gramas.

Se pueden encontrar modelos de fastText ya pre-entrenados en su web o entrenar tus propios modelos usando fastText de una forma similar a Word2vec.

Nivel de frases, párrafos o textos.

Llegados a este punto hemos visto embeddings a nivel de las palabras y a nivel de n-gramas. Pero ambos enfoques tienen un problema potencial que es que no tienen en cuenta el contexto de las palabras. Por ejemplo las frases “El perro muerde al hombre” y “El hombre muerde al perro” tendrían la misma representación, pero sin embargo tienen significados muy diferentes. Necesitamos entonces una forma de representar numéricamente frases o incluso textos más largos. ¿Se puede conseguir esto a través de los embeddings?

Vamos a estudiar brevemente el modelo Doc2vec que permite aprender directamente la representación de textos arbitrariamente largos (frases, párrafos, y documentos) tomando el contexto de las palabras del texto. El modelo Doc2vec está basado en la idea de “vector párrafo” propuesta en el paper publicado en 2014 por Le y Mikolov *Distributed Representations of Sentences and Documents* [5].

El modelo Doc2vec es similar a Word2vec en términos de la arquitectura general, excepto que, además de los vectores de las palabras, el modelo también aprende un “vector párrafo” que representa el texto entero. Cuando el entrenamiento se realiza con un corpus de muchos textos, los vectores párrafo son únicos para cada texto (por texto se entiende cualquier escrito de longitud arbitraria), mientras que los vectores de cada palabra son iguales en todos los textos.

Las redes neuronales usadas para entrenar los embeddings de Doc2vec son muy parecidas a las arquitecturas de CBOW y Skip-Gram. En el caso de Doc2vec, estas arquitecturas tienen el nombre de *Distributed Memory Model of Paragraph Vectors (PV-DM)* y *Distributed Bag-of-Words Model of Paragraph Vectors (PV-DBOW)*.

- **Distributed Memory Model of Paragraph Vectors (PV-DM):** Este modelo es parecido al enfoque CBOW en Word2vec. Sin embargo, el algoritmo tiene dos etapas: (1) Durante el entrenamiento se intentan aprender los vectores palabra \mathbf{W} , los pesos de la capa softmax, \mathbf{U} y \mathbf{b} , y los vectores párrafo \mathbf{D} de los párrafos en el conjunto de entrenamiento; y (2) “la etapa de inferencia” para obtener los vectores \mathbf{D} de nuevos párrafos (nunca antes vistos) añadiendo más columnas a la matriz con los vectores párrafo y realizando el descenso de gradiente (explicado en detalle en el capítulo 4) en \mathbf{D} mientras se mantienen fijos todos los vectores palabra \mathbf{W} (es decir, la matriz de vectores palabra ya construida), \mathbf{U} y \mathbf{b} .

El siguiente diagrama muestra como se entrena el modelo PV-DM.

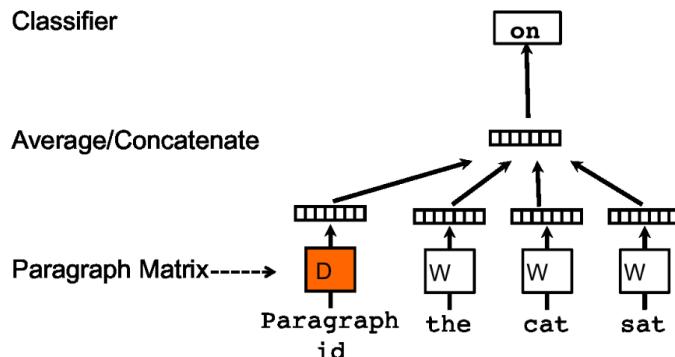


Figura 2.4: Estructura del entrenamiento de vectores párrafo según el enfoque de *Distributed memory*. Imagen extraída del paper *Distributed Representations of Sentences and Documents*[5].

Durante el entrenamiento, como se ve en la figura anterior, se concatena o se calcula la media del vector párrafo con los vectores de las primeras palabras para predecir la palabra siguiente. El vector párrafo representa la información ausente del contexto actual y puede actuar como memoria del tema general del párrafo.

- **Distributed Bag-of-Words Model of Paragraph Vectors (PV-DBOW):** En este enfoque, las palabras no se tienen en cuenta como

valores de entrada del modelo. Esto es, sólo el vector párrafo se usa para predecir una muestra aleatoria de palabras del párrafo. En la fase de entrenamiento, usando el descenso de gradiente y la retropropagación (técnicas vistas en el capítulo 4), los vectores párrafo van siendo ajustados y aprendiendo basándose en su comportamiento haciendo predicciones. Este enfoque es análogo al modelo Skip-Gram usado en Word2vec.

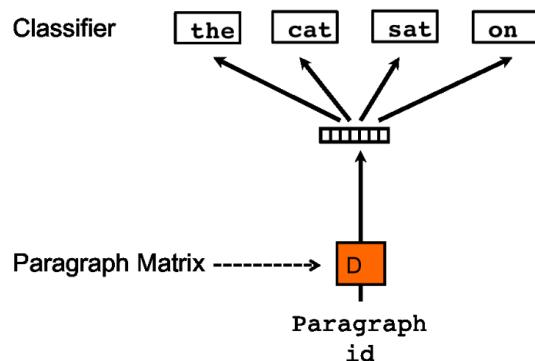


Figura 2.5: Versión vector párrafo de *Distributed Bag of Words*. Imagen extraída del paper *Distributed Representations of Sentences and Documents*[5].

Además de ser conceptualmente más simple, este modelo no necesita almacenar tantos datos como el anterior. Sólo se necesita almacenar los pesos de la capa softmax \mathbf{U} y \mathbf{b} , a diferencia del modelo PV-DM que tiene que almacenar tanto estos mismos pesos como los vectores palabra.

En resumen, Doc2vec modela una especie de contexto y codifica textos de longitud arbitraria en un vector fijo, denso y de pequeña dimensión. Además, fue probablemente la primera implementación para obtener representaciones de embedding para textos completos en vez de usar una combinación de los embeddings individuales de cada palabra. Por todo esto, ha encontrado un gran rango de aplicaciones en el PLN, como clasificación de textos, sistemas de recomendación de textos y simples chatbots para FAQs¹⁰.

¹⁰FAQ son las siglas de la expresión inglesa *Frequently Asked Questions*, que en español podemos traducir como “preguntas frecuentes”. Como tal, es una lista de las preguntas más frecuentes con sus respectivas respuestas sobre un tema en particular.

Capítulo 3

Técnicas clásicas de PLN

En el capítulo anterior se han explicado diversas técnicas de representación de texto, las cuales ayudan significativamente a mejorar la precisión en el procesamiento del texto. En este capítulo, sin embargo, vamos a explorar algunas aplicaciones de algoritmos de *machine learning* (ML) en el campo del procesamiento del lenguaje natural.

3.1. Introducción al *machine learning* o aprendizaje automático

El aprendizaje automático (del inglés, *Machine learning*) es un subcampo de la inteligencia artificial cuyo objetivo es construir sistemas que sean capaces de realizar tareas sin que hayan sido explícitamente programados para llevarlas a cabo. Los algoritmos de ML emplean modelos matemáticos que aprenden de datos ya existentes para realizar tareas como las de predicción, clasificación, toma de decisiones, etc. La parte de aprendizaje del modelo es la llamada *fase de entrenamiento*, donde el algoritmo analiza un gran número de datos para identificar patrones. Este proceso es normalmente costoso computacionalmente, ya que el modelo necesita realizar un gran número de cálculos. Sin embargo, con los continuos avances en la potencia de los nuevos ordenadores existentes, el entrenamiento y uso de modelos de ML se ha vuelto una práctica más fácil y común. En el procesamiento del lenguaje natural, también se necesita analizar un gran volumen de da-

tos, por lo que los algoritmos de ML son bastante utilizados en términos de procesamiento de textos.

Los algoritmos de aprendizaje automático se pueden dividir en tres categorías:

- **Aprendizaje supervisado.** Los algoritmos de aprendizaje supervisado entran el modelo usando datos de entrenamiento ya etiquetados. En el conjunto de datos de entrenamiento se encuentran las variables independientes y los valores correspondientes de las variables dependientes (las que queremos predecir). El algoritmo analiza estos valores e intenta detectar una función que prediga el valor de la variable dependiente en función de los valores de las demás variables. Algunos algoritmos pertenecientes a esta categoría son: regresión lineal, kNN, árboles de decisión, *random forest*, Máquinas de vectores de soporte, Naive-Bayes, etc.
- **Aprendizaje no supervisado.** Este tipo de algoritmos realizan el entrenamiento con datos sin clasificar. El proceso de entrenamiento trata de estudiar el conjunto de datos para intentar comprender la estructura subyacente de los datos. El aprendizaje no supervisado se usa sobretodo para agrupar datos y realizar detección de anomalías analizando un punto con respecto a los demás puntos del conjunto de entrenamiento. Además, también existen técnicas de aprendizaje automático para disminuir la dimensionalidad de los datos (como por ejemplo el análisis de componentes principales). Algunos algoritmos populares de aprendizaje no supervisado son: *k-means*, *k-medoids*, BIRCH, DBSCAN, etc.
- **Aprendizaje por refuerzo.** Este tipo de algoritmos se entrena basándose en simulaciones donde el modelo aprende mediante recompensas que recibe cuando ha realizado ciertas acciones. Ejemplos de algoritmos de aprendizaje por refuerzo son: *Q-learning*, SARSA, etc.

A pesar de que los tres tipos de algoritmos de ML se hayan utilizado en algunas aplicaciones en el campo del procesamiento del lenguaje natural, los algoritmos más populares para el PLN son los basados en el aprendizaje supervisado. Por ello, vamos a ver con más detalle el algoritmo de Naive-Bayes y las Máquinas de vectores de soporte.

3.2. El algoritmo de Naive-Bayes

Naive-Bayes es un algoritmo muy popular de aprendizaje automático, basado en el **teorema de Bayes**, que expresa la probabilidad condicional de un suceso aleatorio A dado B , en términos de la distribución de probabilidad condicional del suceso B dado A y la distribución de probabilidad marginal de solo A . Matemáticamente, el teorema de Bayes se expresa mediante la fórmula

$$P(A|B) = \frac{P(B|A) \cdot P(A)}{P(B)} \quad (3.1)$$

siendo A y B dos sucesos aleatorios.

El algoritmo de Naive-Bayes se trata de un algoritmo de clasificación, ya que se basa en la idea de calcular la probabilidad de que un dato pertenezca a una clase y así asignarle la clase con mayor probabilidad. Además, se entrena con un conjunto de datos etiquetados, es decir, se entrena el modelo y éste aprende a clasificar o etiquetar nuevos datos.

Pongamos que tenemos un conjunto de datos donde cada dato está representado por N variables (X_1, \dots, X_N) y queremos clasificar un dato obteniendo el valor de la variable predictiva Y , que sería en este caso la clase a la que pertenece. Para realizar esto, al igual que en el teorema de Bayes, el algoritmo de Naive-Bayes computa la probabilidad de cada dato de pertenecer a cada clase y finalmente asigna al dato la clase que tiene mayor probabilidad.

Por lo tanto, la fórmula que computa el algoritmo sería la siguiente:

$$P(Y|(X_1, \dots, X_N)) = \frac{P((X_1, \dots, X_N)|Y) \cdot P(Y)}{P((X_1, \dots, X_N))} \quad (3.2)$$

Sin embargo, para grandes conjuntos de datos, calcular la probabilidad condicionada puede llegar a ser complicado. Para evitar este problema, se asume que todas las características o variables de cada dato son independientes entre sí, así que la probabilidad condicionada es simplemente el producto de las probabilidades independientes.

$$P(Y|(X_1, \dots, X_N)) = \frac{P(X_1|Y) \cdot \dots \cdot P(X_N|Y) \cdot P(Y)}{P(X_1) \cdot \dots \cdot P(X_N)} \quad (3.3)$$

Esta suposición es muy ingenua (*naive* en inglés) ya que en la mayoría

de los casos es una suposición errónea, es poco probable que todas las variables de un conjunto de datos sean totalmente independientes. Por ello, al algoritmo de clasificación, se le da ese nombre de *Naive-Bayes*.

Ejemplo práctico en Python

Veamos un ejemplo práctico muy simple de la utilización de este algoritmo en Python. Vamos a clasificar un conjunto de reseñas en términos de si son positivas o negativas. Para este ejemplo, se ha creado manualmente un dataset de reseñas con dos columnas: el texto de la reseña y su clasificación en “positiva” o “negativa”.

Queremos entonces construir un modelo para que dada una reseña nueva fuera del dataset, el modelo sepa clasificarla en “positiva” o “negativa”, es decir, el modelo clasifique correctamente el texto. Se trata por tanto de un clasificador de textos.

Primero se tiene que importar el texto puro y pasarlo a un dataframe. En este caso se ha creado manualmente una lista con los textos de las reseñas y sus clasificaciones directamente en la variable `dataset`.

```
dataset = [[ "Me gustó la peli", "positiva" ],
           [ "Es una buena película. Buena historia", "positiva" ],
           [ "La actuación del héroe es mala pero la heroína era guapa. En general, buena película", "positiva" ],
           [ "Buenas canciones. Pero final realmente aburrido.", "negativa" ],
           [ "Peli triste y aburrida", "negativa" ],
           [ "Es la peor película que he visto en mucho tiempo", "negativa" ],
           [ "En general divertida, pero un poco rara", "positiva" ],
           [ "Al principio prometía, pero después resultó ser aburrida", "negativa" ],
           [ "Una película llena de acción y final feliz", "positiva" ],
           [ "Me sorprendió gratamente por su historia increíble", "positiva" ],
           [ "Demasiado lenta y aburrida, además el argumento es bastante pobre", "negativa" ],
           [ "Cuando empezó parecía guay, pero pronto resultó ser muy mala", "negativa" ],
           [ "Los actores no sabían actuar y el argumento era malísimo", "negativa" ],
```

```
["El actor principal muy guapo. Entretenida de  
inicio a fin", "positiva"]]
```

```
import pandas as pd  
dataset = pd.DataFrame(dataset)  
dataset.columns = ["Texto", "Reseña"]  
display(dataset)
```

El output de este trozo de código se muestra en la tabla siguiente de los textos con sus correspondientes etiquetas:

	Texto	Reseña
0	Me gustó la peli	positiva
1	Es una buena película. Buena historia	positiva
2	La actuación del héroe es mala pero la heroína...	positiva
3	Buenas canciones. Pero final realmente aburrido.	negativa
4	Peli triste y aburrida	negativa
5	Es la peor película que he visto en mucho tiempo	negativa
6	En general divertida, pero un poco rara	positiva
7	Al principio prometía, pero después resultó se...	negativa
8	Una película llena de acción y final feliz	positiva
9	Me sorprendió gratamente por su historia incre...	positiva
10	Demasiado lenta y aburrida, además el argument...	negativa
11	Cuando empezó parecía guay, pero pronto result...	negativa
12	Los actores no sabían actuar y el argumento er...	negativa
13	El actor principal muy guapo. Entretenida de i...	positiva

Figura 3.1: Dataframe de los datos.

A continuación se deben separar la columna que contiene los textos y de la que contiene las etiquetas.

```
X = dataset.iloc[:,0] # extraer columna con texto  
y = dataset.iloc[:,1] # extraer columna con etiqueta
```

Hacemos esto para pre-procesar los datos de texto y que los pueda entender el modelo de *machine learning*. Los pasos que vamos a realizar en el pre-procesamiento del texto serán: tokenizar el texto, eliminar las palabras vacías o *stopwords*, y representar cada texto de forma vectorial mediante la técnica del bolsa de palabras. Todo esto lo realizamos gracias a la clase `CountVectorizer`¹ del módulo `feature_extraction.text` de `sklearn`². Hay que tener en cuenta que para la eliminar las palabras vacías, necesitamos darle como argumento al `CountVectorizer` una lista de palabras. Para ello previamente se ha creado una lista de las palabras vacías más comunes en español llamada `stopwordlist`.

```
from sklearn.feature_extraction.text import CountVectorizer
vectorizer = CountVectorizer(stop_words=stopwordlist)
X_vec = vectorizer.fit_transform(X)
X_vec = X_vec.todense() # convertir matriz en sparse (dispersa)
                        en matriz densa
X_vec
```

A continuación representaremos cada texto en forma de vector pero esta vez mediante la técnica TF-IDF que nos da una idea mejor de la relevancia de una palabra o término en un documento.

```
from sklearn.feature_extraction.text import TfidfTransformer
tfidf = TfidfTransformer()
X_tfidf = tfidf.fit_transform(X_vec)
X_tfidf = X_tfidf.todense()
X_tfidf
```

Con esto, hemos completado la parte del pre-procesamiento y estamos preparados para entrenar el modelo. Sin embargo, antes de esto, se necesita dividir el conjunto de datos en los datos de entrenamiento y los datos de prueba, para poder evaluar la ejecución del modelo. Para ello vamos a utilizar la clase `train_test_split` del módulo `model_selection` de `sklearn`. En la propia función se debe además pasar el argumento del ratio de la división.

```
from sklearn.model_selection import train_test_split
X_train, X_test, y_train, y_test = train_test_split(X_tfidf, y,
                                                    test_size = 0.25, random_state = 0)
```

¹Documentación de la clase `CountVectorizer` [10]

²Scikit-learn (o `sklearn`) es una biblioteca para aprendizaje automático de software libre para el lenguaje de programación Python.

Ahora está todo preparado para entrenar nuestro modelo. Para ello, importamos la clase `MultinomialNB` del módulo `naive_bayes` de `sklearn`.

```
from sklearn.naive_bayes import MultinomialNB  
clf = MultinomialNB()  
clf.fit(X_train, y_train)
```

Entrenar el modelo gracias a los datos de entrenamiento quiere decir que nuestro clasificador Naive-Bayes ha aprendido las probabilidades y es capaz de calcular nuevas probabilidades de nuevos datos. Entonces, dada una nueva frase fuera del corpus (como: “Me quedé muy insatisfecha con la película”), el clasificador es capaz de calcular las probabilidades sobre si esa reseña es positiva o negativa. Así es como se obtienen los valores predichos por el clasificador para los datos de prueba:

```
y_pred = clf.predict(X_test)
```

Por último, para determinar cuánto de bien ha funcionado el modelo, podemos crear una matriz de confusión³ para mostrar el número de predicciones correctas en cada grupo. Así mismo podemos calcular en porcentaje la exactitud total en las predicciones del modelo.

```
# Matriz de confusión  
from sklearn.metrics import confusion_matrix  
cm = confusion_matrix(y_test, y_pred)  
print(cm)  
  
# Evaluación del modelo  
from sklearn import metrics  
accuracy_score = metrics.accuracy_score(y_pred, y_test)  
print(str(':{04.2f}'.format(accuracy_score*100))+'%)
```

El output es el siguiente:

```
[[2 0]  
 [1 1]]  
75.00 %
```

³En el campo de la inteligencia artificial y, en especial en el problema de la clasificación estadística, una matriz de confusión es una herramienta que permite la visualización del desempeño de un algoritmo que se emplea en aprendizaje supervisado. Cada columna de la matriz representa el número de predicciones de cada clase, mientras que cada fila representa a las instancias en la clase real. Uno de los beneficios de las matrices de confusión es que facilitan ver si el sistema está confundiendo dos clases.

La matriz de confusión se interpreta de la siguiente manera: las filas representan los valores reales de cada clase, mientras que las columnas representan los valores predichos. Entonces, pequeño modelo de ejemplo ha predicho 3 (2 + 1) muestras con la etiqueta “negativa”, de los cuales 2 estaban bien clasificados y 1 no. Igualmente, el modelo ha clasificado 1 muestra como “positiva”, clasificándola correctamente.

Por último comentar que existen varias formas de evaluar el modelo. Por ejemplo existen métricas como la **exactitud** (*accuracy*), la **precisión** (*precision*) o la **sensibilidad** (*recall*).

La exactitud representa el porcentaje de predicciones correctas frente al total. Por tanto, es el cociente entre los casos bien clasificados por el modelo (es decir, los valores en la diagonal de la matriz de confusión) y la suma de todos los casos. En nuestro ejemplo esto sería: $(2+1)/(2+1+1) = 3/4 = 0.75$. Es decir, nuestro modelo tiene una exactitud del 75 %. Esta exactitud es bastante buena, pero hay que tener en cuenta que la base de datos de nuestro modelo es muy pequeña, por lo que el modelo es bastante inestable.

3.3. Support Vector Machine (SVM)

Support Vector Machine o Máquinas de Vectores de Soporte son un conjunto de algoritmos de aprendizaje supervisado cuyo objetivo es resolver problemas de clasificación y regresión. Más concretamente, dado un conjunto de datos de entrenamiento, se pueden identificar las clases y entrenar una SVM para construir un modelo que prediga la clase de una nueva muestra.

El objetivo de este algoritmo es el de, representando los puntos de muestra en un espacio n-dimensional, encontrar el hiperplano que mejor separa los datos en distintas clases. Cada punto del conjunto de datos es considerado un vector en un plano n-dimensional, donde cada dimensión representa una variable de los datos. SVM identifica los puntos frontera (o los puntos más cercanos a otra clase), también llamados **vectores de soporte** (del inglés *support vectors*), y después trata de encontrar el hiperplano en el espacio N-dimensional que se encuentra lo más lejos posible de los vectores de soporte de cada clase. Es decir, trata de encontrar una frontera que diferencie bien las clases entre sí.

Veamos las matemáticas detrás de este algoritmo. Supongamos que tenemos un conjunto de datos (x_1, \dots, x_p) el cual queremos que diferenciar en dos clases (se diría que se trata de un problema de clasificación *One-vs-One*), basándonos en las características de cada punto dato. Supongamos que cada punto tiene N características. Entonces cada punto se representa como un vector de N dimensiones $x_i \in \mathbb{R}^N$ para $i = 1, \dots, p$. Supongamos además que los dos grupos son perfectamente separables, es decir, que un hiperplano es capaz de separar las clases con un 100 % de exactitud.

Una vez que tenemos todos los puntos del conjunto de datos representados en el espacio vectorial N -dimensional, el siguiente objetivo es separar esos puntos encontrando la frontera más óptima entre clases. Es decir, se debe encontrar un hiperplano $H : w^T \cdot x - c = 0$, donde $w \in \mathbb{R}^N$ son los coeficientes del hiperplano, $x \in \mathbb{R}^N$ es un punto genérico del hiperplano y $c \in \mathbb{R}$ es el término independiente.

Es evidente que hay muchos hiperplanos que pueden separar las dos clases de este ejemplo con un 100 % de exactitud. Sin embargo, el algoritmo de SVM intenta calcular el valor óptimo de los coeficientes w y de la constante c para que el hiperplano esté a la máxima distancia de ambos vectores de soporte.

Para ello se debe calcular la distancia del hiperplano H a un punto dado x como

$$d_H(x) = \frac{|w^T \cdot x - c|}{\|w\|}$$

donde $\|\cdot\|$ representa la norma euclídea.

¿Pero, cómo determina el algoritmo cuáles son los vectores de soporte? Éstos son los puntos de cada clase a menor distancia del hiperplano. Tenemos entonces que distinguir los puntos de cada clase matemáticamente. Un hiperplano divide siempre un espacio N -dimensional en dos mitades definidas como $w^T \cdot x - c \geq 0$ y $w^T \cdot x - c \leq 0$.

Así que teniendo un hiperplano que separase las dos clases perfectamente, los puntos de una clase estarían en el semiespacio $w^T \cdot x - c \geq 0$ y los de la otra clase en el otro semiespacio $w^T \cdot x - c \leq 0$.

Además, si consideramos las etiquetas de las clases de los datos de entrenamiento (y_i con $i = 1, \dots, p$), a una clase se le puede asignar un valor positivo

y a la otra clase un valor negativo. Es decir, que el producto de una etiqueta predicha y y la verdadera siempre dará mayor o igual a 0 si la predicción es correcta, y menor que cero si no lo es. Es decir,

$$y_i (w^T \cdot x_i - c) = \begin{cases} \geq 0 & \text{si es correcto} \\ < 0 & \text{si es incorrecto} \end{cases}$$

Por lo tanto, los vectores de soporte son los puntos de cada clase más cercanos al hiperplano, es decir, los puntos que son el argumento del mínimo de $\frac{y_i(w^T \cdot x_i - c)}{\|w\|}$.

Teniendo en cuenta todo lo anterior, los parámetros del hiperplano óptimo que clasifica todos los puntos de forma correcta, se calculan como sigue:

$$w^* = \arg_w \max \left[\min_i \frac{|w^T \cdot x_i - c|}{\|w\|} \right] = \arg_w \max \left[\min_i \frac{y_i(w^T \cdot x_i - c)}{\|w\|} \right]$$

donde $\arg_w \max$ es la abreviación para referirnos a los argumentos del máximo, es decir, los puntos del dominio donde la función alcanza su máximo.

Notar que este algoritmo es el SVM lineal pero, con transformaciones previas a los datos, también puede ser aplicado en problemas de naturaleza no lineal.

Ejemplo práctico en Python

Veamos el ejemplo anterior de las reseñas de cine aplicando esta vez el algoritmo de SVM. Todos los pasos del pre-procesamiento de datos son totalmente idénticos, así como la división del conjunto de datos en datos de entrenamiento y datos de prueba. Veamos entonces directamente el ajuste del modelo: importamos la clase SVC (**Support Vector Classification**) del módulo `svm` de `sklearn` y entrenamos el modelo.

```
# Entrenar el algoritmo de SVM #
from sklearn.svm import SVC
classifier = SVC(kernel='linear')
classifier.fit(X_train, y_train)
```

Entrenar el modelo quiere decir que el clasificador a encontrado el hiperplano óptimo que divide mejor los grupos. Para medir la exactitud del

hiperplano, utilizamos los datos de prueba para ver el signo de la ecuación (es decir, para ver cómo el modelo clasifica estos datos de prueba)

```
# Prediccion del conjunto de datos de prueba
y_pred = classifier.predict(X_test)
```

Y de nuevo, usaremos la matriz de confusión para medir cuánto de exacto ha sido el modelo en sus predicciones.

```
# Making the Confusion Matrix
from sklearn.metrics import confusion_matrix
cm = confusion_matrix(y_test, y_pred)
print(cm)

# Evaluacion del modelo
from sklearn import metrics
accuracy_score = metrics.accuracy_score(y_pred, y_test)
print(str('{:04.2f}'.format(accuracy_score*100))+'%)
```

El output de este último trozo de código es la matriz de confusión y el porcentaje de exactitud del modelo:

```
array([[0, 2],
       [0, 2]])
50.00 %
```

Interpretando la matriz de confusión, observamos que ha clasificado todas las reseñas del conjunto de prueba en la clase “positiva”, habiendo acertado 2 y habiéndose equivocado en otras 2. Esto quiere decir que este modelo tendría una exactitud del 50% ($\frac{2}{2+2} = 0.5$).

Igualmente como en el algoritmo de Naive-Bayes, hay muchas mejoras que se podrían hacer para incrementar el porcentaje de exactitud del modelo, empezando por ampliar la base de datos. En la etapa del pre-procesamiento de los datos se podrían también aplicar técnicas como la lematización o *stemming*.

En este capítulo se han explicado algunos ejemplos de algoritmos clásicos de *machine learning*, así como ejemplos en la práctica dentro del campo del PLN. Sin embargo, dentro del campo del procesamiento natural no podemos dejar de mencionar las **redes neuronales** que veremos en el siguiente capítulo, ya que su uso ha sufrido un crecimiento exponencial y ha supuesto un avance notable en la mejora de las aplicaciones durante las últimas dos décadas del siglo XXI.

Capítulo 4

Redes Neuronales

Más allá de las técnicas más clásicas de Machine Learning, debemos estudiar en profundidad las redes neuronales artificiales, ya que éstas han sido la arquitectura que ha permitido resolver una amplia variedad de problemas que son difíciles de solucionar con la programación clásica basada en reglas o con métodos de aprendizaje automático como los vistos en el capítulo anterior.

4.1. La idea detrás de la red neuronal

Las redes neuronales están basadas en el funcionamiento del sistema nervioso humano. Es decir, se puede hacer una analogía clara entre el funcionamiento de las neuronas (células del sistema nervioso) y las neuronas artificiales que componen una red neuronal artificial.

Podríamos describir el funcionamiento de la red neuronal biológica como un proceso matemático que comienza cuando las dendritas de una neurona reciben señales de las neuronas vecinas. Cada dendrita tiene “asociado” un valor o peso que marca la importancia de la señal que llega a través de cada una de ellas. Así, es como si la señal que llega a cada dendrita se “multiplicara” por su correspondiente peso. Las señales entrantes se “sumarían” en el núcleo de la neurona y si la señal total alcanza un umbral específico, ésta sería enviada a través del axón y propagada más allá a otras neuronas.

4.2. Funcionamiento de una red neuronal artificial

Análogamente a la red neuronal biológica, una red neuronal artificial consiste en un conjunto de unidades básicas de procesamiento llamadas *neuronas* que se conectan entre sí para transmitir señales de unas a otras. La información de entrada a la red neuronal se somete a diversas operaciones produciendo así unos valores de salida. En definitiva, matemáticamente se podría decir que una red neuronal es una función diferenciable que transforma una variable en otra.

La gran fortaleza de este modelo computacional es el “entrenamiento” de la red neuronal para que los valores de salida que obtengamos sean los deseados. ¿A qué nos referimos con “entrenamiento”? Es el proceso por el cual el sistema aprende y se forma a sí mismo, en lugar de ser programado explícitamente. Para realizar este aprendizaje automático, normalmente se intenta minimizar una función llamada *función de pérdida*. Este proceso se lleva a cabo mediante la denominada *retropropagación*. Explicaremos todos estos procesos a lo largo de este capítulo.

4.2.1. Neuronas

Cada neurona en una red neuronal tiene n conexiones de entrada a través de las cuales reciben n valores de entrada representados por $\vec{x} = (x_1, \dots, x_n)$. Este número n no es fijo para todas las neuronas, es decir, puede variar de neurona a neurona.

La primera operación que realiza una neurona es una suma ponderada de sus valores de entrada. La ponderación de cada valor viene dado por el peso que se le asigna a cada una de las conexiones de entrada de la neurona representados por $\vec{w} = (w_1, \dots, w_n)$. Entonces, la operación que realiza la neurona se reduce a

$$z = w_1 x_1 + \dots + w_n x_n + b , \quad (4.1)$$

donde el término independiente b es el *sesgo*, el cual se puede interpretar como una propiedad intrínseca de esa neurona y es un parámetro que se entrena como todos los demás parámetros de la red.

Por último, la neurona realiza una operación no lineal llamada *función de*

activación cuyo objetivo es introducir la no-linealidad a la red neuronal. Sin esta no-linealidad la red estaría simplemente realizando operaciones lineales con los valores de entrada, lo que equivaldría a una sola ecuación lineal multivariable.

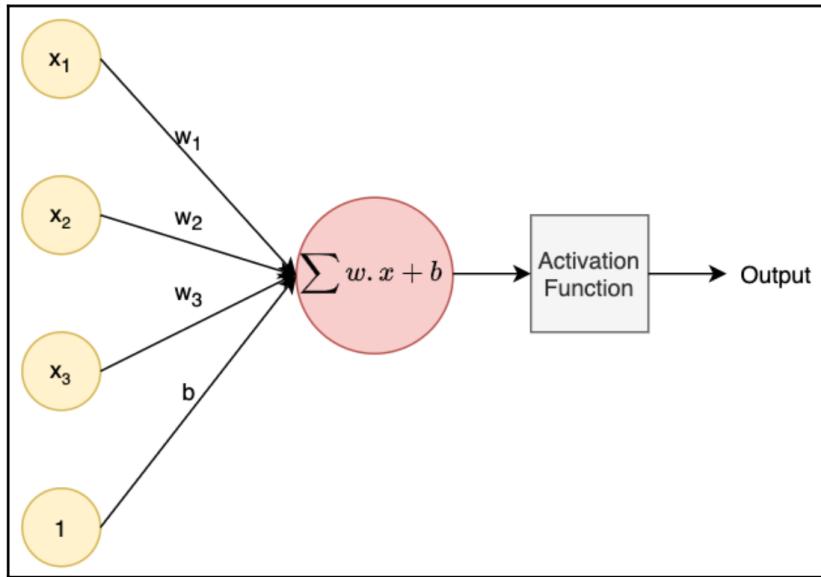


Figura 4.1: Estructura una neurona con 3 conexiones de entrada (x_1, x_2, x_3). Imagen extraída del libro *Hands-On Python Natural Language Processing*[4].

4.2.2. Funciones de activación

Volviendo a la analogía con la redes neuronales biológicas, las funciones de activación de cada neurona serían las encargadas de controlar ese umbral que se describía en la idea biológica de una red neuronal. Matemáticamente, la función de activación es simplemente una función $f(z)$ donde z es el resultado de la ecuación 4.1. Las funciones de activación más conocidas son las siguientes:

- **Sigmoide:** esta función restringe la salida entre el 0 y el 1.

$$\text{sigmoid}(z) = \frac{1}{1 + e^{-z}} \quad (4.2)$$

Esta función lleva valores muy grandes a 1 y valores pequeños a 0.

Es muy común cuando se trabaja en tareas que precisan de salidas binarias. Sin embargo, debido al hecho de que esta función no está centrada en cero, su uso ha disminuido significativamente.

- **Tangente hiperbólica (Tahn):**

$$\tanh(z) = \frac{e^z - e^{-z}}{e^z + e^{-z}} \quad (4.3)$$

Esta función se usa, principalmente, en lugar de la sigmoide en la mayoría de redes hoy en día ya que solventa el problema de estar centrada en el cero. Sin embargo, esta función, así como la sigmoide, se satura rápidamente para la mayoría de los valores, lo que deriva en gradientes extremadamente pequeños.

- **ReLU:** esta función es probablemente la más utilizada hoy en día. Se define por

$$ReLu(z) = \max(0, z) \quad (4.4)$$

La gran ventaja de esta función de activación es su simplicidad, la cual hace que no se requieran operaciones computacionales muy complejas. A su vez, ReLu vence los problemas asociados con las funciones sigmoide y tangente hiperbólica.

4.2.3. Capas en una ANN (Artificial Neural Network)

Ya hemos visto el funcionamiento de una neurona aislada. Pero las redes neuronales se basan también en las conexiones entre neuronas para su funcionamiento. Una red neuronal artificial consiste en tres grupos de capas en las cuales se encuentran cada una de las neuronas.

- **Capa de entrada.** Esta es la primera capa de la red y el número de nodos o neuronas equivale al número de variables de entrada a la red.
- **Capa oculta.** Una red neuronal puede tener una o varias capas ocultas. El número de capas ocultas es un hiper-parámetro¹, así como el

¹En el apredizaje automático, un hiper-parámetro es un parámetro cuyo valor es utili-

número de nodos en cada capa oculta. Estas capas son las encargadas de encontrar las relaciones y patrones en los datos.

- **Capa de salida.** Esta es la última capa en una red neuronal artificial y devuelve el output obtenido tras todo el proceso. El número de nodos de esta capa depende del tipo de problema que estemos resolviendo.

Hay que notar que, en una red neuronal artificial, las neuronas están interconectadas entre capas pero las neuronas de una misma capa no tienen ninguna conexión.

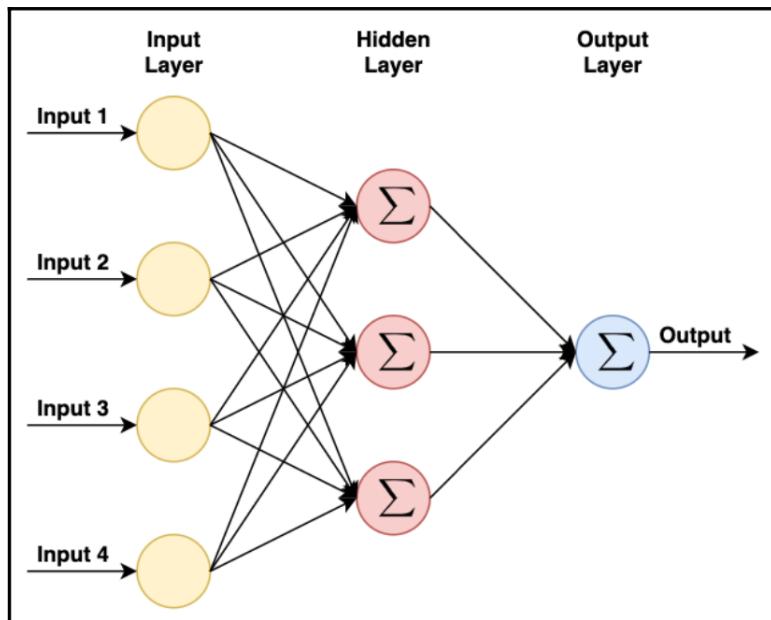


Figura 4.2: Diagrama de una red neuronal con capa de entrada de 4 nodos que proveen 4 valores de entrada. La capa oculta consiste en 3 neuronas y por último la capa de salida tiene un solo nodo. El sumatorio dibujado en el diagrama representa las operaciones que se llevan a cabo en cada neurona, seguidas de la función de activación. Imagen extraída del libro *Hands-On Python Natural Language Processing*[4].

zado para controlar el proceso de aprendizaje o entrenamiento. Por el contrario, los valores de los otros parámetros (generalmente los pesos de la red neuronal) se obtienen durante la fase de entrenamiento.

4.2.4. Aprendizaje de la red neuronal

Una vez explicado el funcionamiento de una red neuronal, podemos empezar a entender cómo aprenden por sí mismos este tipo de modelos. Veamos paso a paso cómo se tramita la información en una red neuronal:

1. Los valores de entrada (x_1, \dots, x_n) llegan a la capa de entrada.
2. Estos valores son trasladados a las capas ocultas, multiplicados por los pesos inicializados aleatoriamente y sumados al término de sesgo b en cada neurona de la capa oculta.
3. Al resultado z de esta suma ponderada se le aplica la función de activación.
4. Tras las capas ocultas, los valores llegan a la capa de salida donde las neuronas de esta capa procesan los datos y emiten el valor de salida o respuesta y . Este valor y es la predicción hecha por el modelo para los valores de entrada dados.

A este proceso se le denomina **propagación hacia delante**. Para llegar a entrenar la red, ahora se debe comparar el valor de salida predicho y con el valor correcto \hat{y} . Esto se lleva a cabo calculando la *función de pérdida*. Este tipo de funciones nos indican cómo de bien el modelo ha sabido predecir el valor de salida para unos valores de entrada particulares. El objetivo de la red es, entonces, minimizar la función de pérdida ya que de esta manera mejorará sus predicciones.

Veamos, a continuación, algunas de las funciones de pérdida más utilizadas. Las dos primeras se utilizan para problemas de regresión y las dos últimas para problemas de clasificación.

- Error cuadrático medio (ECM) para N predicciones.

$$ECM = \frac{1}{N} \sum_{i=1}^N (\hat{y}_i - y_i)^2 \quad (4.5)$$

- Error absoluto medio (EAM) para N predicciones.

$$EAM = \frac{1}{N} \sum_{i=1}^N |\hat{y}_i - y_i| \quad (4.6)$$

- Entropía cruzada binaria (es decir, para 2 clases).

$$ECB = -(\hat{y} \log(y) + (1 - \hat{y}) \log(1 - y)) \quad (4.7)$$

- Entropía cruzada categórica general (para c clases).

$$EC = - \sum_{i=1}^c (\hat{y}_i \log(p_i)) \quad (4.8)$$

donde \hat{y}_i es la etiqueta correcta y p_i es la probabilidad calculada para la clase i .

¿Cómo se consigue minimizar la función de pérdida? El objetivo es que la red neuronal artificial sea capaz de hacer predicciones lo más certeras posible. Para ello, los parámetros de la red neuronal, es decir los pesos y los términos de sesgo, deben cambiar. Es decir, los pesos asociados a cada conexión w_i y el vector de sesgos \mathbf{b} deben ser actualizados en cada capa de la red neuronal para encontrar los valores óptimos que mejoran la ejecución del modelo, es decir, que minimizan la función de pérdida. Esto se realiza con la ayuda de la técnica del **descenso de gradiente**, un algoritmo iterativo de optimización que sirve para encontrar el mínimo local de una función diferenciable.

4.2.5. Descenso de gradiente

Para entender mejor cómo funciona el aprendizaje automático de un modelo de red neuronal, vamos a explicar el proceso mediante un ejemplo básico. La red tendrá dos variables de entrada, una capa oculta con tres nodos y un solo nodo en la capa de salida.

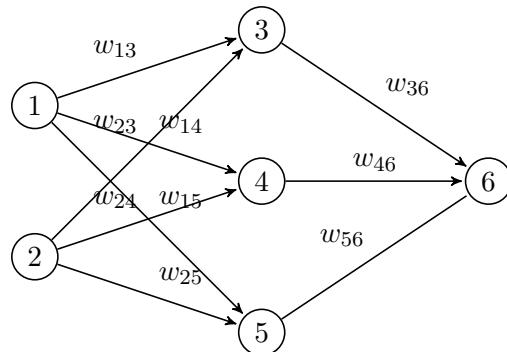


Figura 4.3: Ejemplo de red neuronal.

Los nodos etiquetados con 1 y 2 son las neuronas de la capa de entrada. 3, 4 y 5 son las neuronas de la capa oculta. Por último la capa de salida tiene una única neurona etiquetada por 6.

Como ya se ha explicado anteriormente, cada neurona de la capa oculta y la neurona de la capa de salida, tendrán asociado un término de sesgo b_i con $i = 3, 4, 5, 6$. Además cada conexión entre neuronas de capas diferentes tendrá un peso asociado w_{ij} , que indica que la neurona i se conecta con la neurona j de la capa siguiente.

Representamos todos los pesos y términos de sesgo de la red neuronal por el vector \mathbf{W} . Es decir $\mathbf{W} = (w_{13}, w_{14}, b_3, \dots, b_6)^T$. En este caso contamos con 13 parámetros. La función de pérdida es la función $C(\mathbf{W})$, una función multivariable que depende de los parámetros de la red² y además debe ser diferenciable.

Como la función de pérdida depende de los 13 parámetros de la red neuronal, lo que queremos hacer es encontrar los valores de los parámetros que minimizan la función pérdida $C(\mathbf{W})$. Es decir, calculamos las derivadas parciales de la función pérdida con respecto a cada peso y término de sesgo por separado. Así se calcula cuánto cambia la función de pérdida, y en consecuencia la predicción del modelo, con respecto a los cambios que hagamos en los parámetros. Matemáticamente, tendríamos que

$$\mathbf{W}_{new} = \mathbf{W}_{old} - \alpha \frac{\partial C}{\partial \mathbf{W}_{old}} = \mathbf{W}_{old} - \alpha \nabla C(\mathbf{W}_{old}), \quad (4.9)$$

donde \mathbf{W}_{new} son los nuevos valores de los parámetros de la red tras realizar el descenso de gradiente, \mathbf{W}_{old} son los valores de los parámetros previos al descenso de gradiente y α es la tasa de aprendizaje (del inglés, *learning rate*).

La tasa de aprendizaje, α , controla cuánto se actualizan los valores. Idealmente, la tasa de aprendizaje se debería modificar durante el entrenamiento para que tuviera valores grandes cuando se está lejos de alcanzar el mínimo de la función de pérdida y valores pequeños cuando se está cerca del valor

²En realidad, la función de pérdida depende directamente de la variable a predecir y , es decir, $C(y)$. Lo que ocurre es que la variable a predecir y , depende a su vez de los parámetros de la red neuronal (\mathbf{W}).

mínimo de ésta.

En resumen, este proceso se denomina **retropropagación**, ya que la señal se transmite hacia atrás en la red.

Con el objetivo de comprender mejor la idea del descenso de gradiente, pongamos un ejemplo concreto. En este ejemplo nuestra función de pérdida va a ser el error cuadrático, es decir, $C(y) = (\hat{y} - y)^2$, donde \hat{y} es el valor correcto y y es el valor predicho por la red.

Además, vamos a tomar el parámetro w_{25} de nuestra red neuronal como ejemplo. Veamos entonces cómo se calcula el nuevo valor del parámetro concreto w_{25} . Nuestro objetivo es calcular cuánto de sensible es la función perdida con respecto al parámetro deseado. Es decir, queremos calcular $\frac{\partial C(\mathbf{W})}{\partial w_{25}}$.

Recordamos que el valor de salida predicho ha sido calculado por la neurona de salida mediante

$$z_6 = w_{36}y_3 + w_{46}y_4 + w_{56}y_5 + b_6$$

$$y = y_6 = \sigma_6(z_6)$$

donde σ_6 es la función de activación de la neurona de salida.

Igualmente, el valor de salida de la neurona 5 (y_5) ha sido calculada como

$$z_5 = w_{15}x_1 + w_{25}x_2 + b_5$$

$$y_5 = \sigma_5(z_5)$$

Aplicando la regla de la cadena³, se tiene que

$$\frac{\partial C}{\partial w_{25}} = \frac{\partial C}{\partial y} \frac{\partial y}{\partial z_6} \frac{\partial z_6}{\partial y_5} \frac{\partial y_5}{\partial z_5} \frac{\partial z_5}{\partial w_{25}}$$

³En cálculo, la regla de la cadena es una fórmula para obtener la derivada de funciones compuestas, esto es, si f y g son funciones diferenciables entonces la regla de la cadena expresa la derivada de la composición $f \circ g$ en términos de la derivada de f y g y el producto de funciones como $(f \circ g)' = (f' \circ g) \cdot g'$.

Sustituyendo cada derivada parcial por su valor nos queda

$$\frac{\partial C}{\partial w_{25}} = -2(\hat{y} - y) \sigma'_6(z_6) w_{56} \sigma'_5(z_5) x_2 ,$$

que es la expresión que utilizaremos para actualizar w_{25} .

4.2.6. Problemas de entrenamiento

Durante la fase de entrenamiento de una red neuronal hay dos principales problemas que pueden ocurrir: *underfitting* y *overfitting*.

- **Underfitting.** Este fenómeno ocurre cuando nuestro modelo es muy simplista y no predice bien ni con los datos de entrenamiento ni con los datos de test. En este caso, el modelo es incapaz de capturar los patrones o tendencias en los datos y entonces no puede generalizar a la hora de trabajar con nuevos datos. Para resolver este problema en el caso de redes neuronales, se puede aumentar el número de capas y así crear una red más grande para que el modelo pueda capturar mejor los patrones complejos de los datos.
- **Overfitting o sobreajuste.** En este caso el modelo actúa muy bien sobre los datos de entrenamiento pero no generaliza por lo que no predice bien sobre los datos de prueba. Lo que ocurre cuando tenemos un problema de este tipo es que el modelo ha memorizado los datos de entrenamiento pero no ha sabido realmente aprender los patrones o tendencias. Para resolver este problema podemos usar técnicas como la regularización⁴ o el *dropout*⁵.

⁴La regularización evita el problema del *overfitting* penalizando al modelo cuando éste ha predicho demasiado bien sobre el conjunto de datos de prueba. Esto se consigue gracias a un “término de castigo” que es sumado a la función de pérdida.

⁵El *dropout* es una técnica en la cual cada neurona de una red neuronal está activa con una probabilidad p . Por lo tanto, hay un porcentaje de señales que no pasan a las siguientes capas. Esto ayuda a prevenir el *overfitting*, ya que el modelo no llega a generalizar bien porque cada vez se utilizan unos pesos diferentes correspondientes a las neuronas activas.

4.3. Redes neuronales recurrentes (RNN)

Anteriormente, hemos visto cómo en el caso de las redes neuronales artificiales prealimentadas (en inglés, *feedforward* o FNN) las entradas eran totalmente independientes entre sí. Sin embargo, sabemos que en el caso del lenguaje natural, la relaciones entre las palabras de una frase o texto son muy importantes a nivel gramatical y semántico. Por ejemplo, si consideramos la frase “Sara fue esta mañana a dar una vuelta con su perro”, vemos que el término “su” depende directamente de “Sara” que se encuentra totalmente al inicio de la frase.

Las relaciones entre palabras se pueden extraer mirando la posición de cada palabra con respecto a las demás. A un nivel más técnico, estas relaciones se pueden interpretar como una serie temporal⁶, donde las palabras o tokens constituirían el conjunto de datos de una serie temporal.

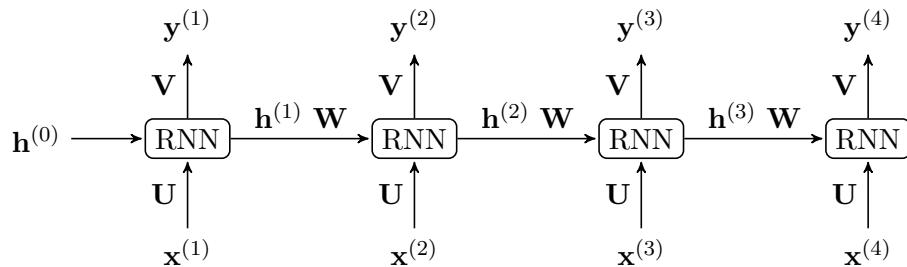
Las redes neuronales recurrentes (del inglés, *recurrent neural network* o *RNN*) son un tipo de red neuronal artificial especializada en procesar datos secuenciales o series temporales cuya arquitectura permite, además, que la red obtenga una especie de “memoria” artificial. Con la introducción, en el campo del PLN, de este tipo de redes neuronales se logra entonces capturar el contexto y las relaciones dentro de una secuencia de palabras. Más concretamente, las redes neuronales recurrentes pueden transformar secuencias enteras de datos en vectores y viceversa, capturando así el contexto general de la secuencia.

4.3.1. Arquitectura de una RNN

Las redes neuronales recurrentes son simplemente un conjunto de varias copias de redes neuronales prealimentadas donde cada copia representa la misma red neuronal en un instante o estado de tiempo concreto.

⁶Una serie temporal o cronológica es una sucesión de datos medidos en determinados momentos y ordenados cronológicamente.

La intuición detrás de las redes neuronales recurrentes está en los sistemas dinámicos discretos, en los cuales el estado de un sistema en un cierto instante depende del estado en el instante anterior. Sin embargo, en el aprendizaje automático, como es el caso de las redes neuronales, además se tiene siempre un valor de entrada para el cual se quiere hacer una predicción. Entonces vamos a considerar un sistema dinámico con valores de entrada externos. Cada neurona recibe dos valores de entrada: uno es el valor de entrada actual y el otro es el llamado “estado oculto” (del inglés *hidden state*) que se trata del valor de salida de esa misma neurona en un estado de tiempo anterior. Veamos un esquema representativo de una red neuronal recurrente genérica:



Vamos a desarrollar matemáticamente las ecuaciones de la propagación hacia delante de la red neuronal recurrente. La propagación hacia delante comienza con un estado inicial $\mathbf{h}^{(0)}$. Después, para cada etapa de tiempo de $t = 1$ a $t = \tau$ se utilizan las ecuaciones

$$\mathbf{a}^{(t)} = \mathbf{U}\mathbf{x}^{(t)} + \mathbf{W}\mathbf{h}^{(t-1)} + \mathbf{b} \quad (4.10)$$

$$\mathbf{h}^{(t)} = \phi(\mathbf{a}^{(t)}) \quad (4.11)$$

$$\mathbf{y}^{(t)} = \psi(\mathbf{V}\mathbf{h}^{(t)} + \mathbf{c}) \quad (4.12)$$

donde $\mathbf{x}^{(t)}$ es el valor de entrada en el instante de tiempo t , $\mathbf{h}^{(t)}$ es el estado de memoria en el instante de tiempo t e $\mathbf{y}^{(t)}$ es la salida en el instante de tiempo t . Además, ϕ y ψ son las funciones de activación no lineales y \mathbf{U} , \mathbf{V} , \mathbf{W} , \mathbf{b} y \mathbf{c} son los parámetros de la red de la regresión lineal que precede la activación no lineal. Nótese que los parámetros anteriores son los mismos a lo largo de toda la arquitectura.

4.3.2. Retropropagación a través del tiempo

Ahora que hemos entendido la arquitectura básica de una red neuronal recurrente, vamos a explicar cómo la red en efecto aprende mediante la retropropagación a través de los varios instantes de tiempo.

La retropropagación a través del tiempo (del inglés *Backpropagation through time (BPTT)*) es uno de los conceptos clave para entender las redes neuronales recurrentes. Ya hemos discutido la retropropagación en la sección 4.2.5 con redes neuronales artificiales en general, donde observábamos que para cada valor de entrada se tenía un valor de salida correcto con el cual el modelo podía computar la función de pérdida o error en la predicción. Ese error se propagaba hacia atrás en la red y los parámetros de ésta se modificaban en función de cuánto eran responsables de ese error. En ese caso teníamos un valor de entrada por cada valor de salida.

Sin embargo, en el caso de las redes neuronales recurrentes, cada token de una frase es un valor de entrada pero como valor de salida sólo tenemos uno. Además todos los parámetros de la red neuronal recurrente son los mismos durante todos los instantes de tiempo. Entonces, ¿cómo funciona la retropropagación en este caso?

La retropropagación a través de tiempo consiste en calcular los gradientes de la función de pérdida con respecto a los diferentes parámetros y después aplicar el algoritmo de descenso de gradiente para actualizarlos. En el caso de redes neuronales recurrentes, los gradientes de cada parámetro no sólo dependen de los valores en ese mismo instante de tiempo sino que también se debe tener en cuenta los valores en los instantes anteriores.

Vamos a considerar la predicción de un valor de salida para una secuencia particular $\mathbf{x} = (x^{(1)}, \dots, x^{(N)})$. En cada momento de tiempo $t = 1, \dots, N$, se computa lo siguiente:

$$\mathbf{a}^{(t)} = \mathbf{U}\mathbf{x}^{(t)} + \mathbf{W}\mathbf{h}^{(t-1)} + \mathbf{b} \quad (4.13)$$

$$\mathbf{h}^{(t)} = \phi(\mathbf{a}^{(t)}) \quad (4.14)$$

$$\mathbf{y}^{(t)} = \psi(\mathbf{V}\mathbf{h}^{(t)} + \mathbf{c}) \quad (4.15)$$

Como sabemos ya, los parámetros \mathbf{U} , \mathbf{V} , \mathbf{W} , \mathbf{b} y \mathbf{c} no varían a lo largo de toda la red neuronal.

Para calcular cuánto y cómo varía el valor de salida con respecto a los

parámetros de la red, debemos calcular el vector gradiente de la función de pérdida $L(\mathbf{U}, \mathbf{W}, \mathbf{V}, \mathbf{b}, \mathbf{c})$. En el caso de redes neuronales recurrentes se calcula la función de pérdida $\mathcal{L}(\hat{\mathbf{y}}^{(t)}, \mathbf{y}^{(t)})$ sobre todas las etapas de tiempo ($t = 1, \dots, N$) mediante

$$L(\mathbf{U}, \mathbf{W}, \mathbf{V}, \mathbf{b}, \mathbf{c}) = \sum_{t=1}^N \mathcal{L}(\hat{\mathbf{y}}^{(t)}, \mathbf{y}^{(t)})$$

donde $\hat{\mathbf{y}}^{(t)}$ son los valores correctos e $\mathbf{y}^{(t)}$ son los valores de salida predichos por la red.

Entonces el vector gradiente de L sería

$$\nabla L(\mathbf{U}, \mathbf{W}, \mathbf{V}, \mathbf{b}, \mathbf{c}) = \left[\frac{\partial L}{\partial \mathbf{U}}, \frac{\partial L}{\partial \mathbf{W}}, \frac{\partial L}{\partial \mathbf{V}}, \frac{\partial L}{\partial \mathbf{b}}, \frac{\partial L}{\partial \mathbf{c}} \right]^T$$

A continuación, hallamos el valor de las derivadas parciales de L con respecto a las variables anteriores; que resultan ser

$$\frac{\partial L}{\partial \mathbf{U}} = \sum_{t=1}^N \frac{\partial \mathcal{L}(\hat{\mathbf{y}}^{(t)}, \mathbf{y}^{(t)})}{\partial \mathbf{U}} = \sum_{t=1}^N \frac{\partial \mathcal{L}(\hat{\mathbf{y}}^{(t)}, \mathbf{y}^{(t)})}{\partial \mathbf{y}^{(t)}} \frac{\partial \mathbf{y}^{(t)}}{\partial \mathbf{h}^{(t)}} \frac{\partial \mathbf{h}^{(t)}}{\partial \mathbf{U}} \quad (4.16)$$

$$\frac{\partial L}{\partial \mathbf{W}} = \sum_{t=1}^N \frac{\partial \mathcal{L}(\hat{\mathbf{y}}^{(t)}, \mathbf{y}^{(t)})}{\partial \mathbf{W}} = \sum_{t=1}^N \frac{\partial \mathcal{L}(\hat{\mathbf{y}}^{(t)}, \mathbf{y}^{(t)})}{\partial \mathbf{y}^{(t)}} \frac{\partial \mathbf{y}^{(t)}}{\partial \mathbf{h}^{(t)}} \frac{\partial \mathbf{h}^{(t)}}{\partial \mathbf{W}} \quad (4.17)$$

$$\frac{\partial L}{\partial \mathbf{V}} = \sum_{t=1}^N \frac{\partial \mathcal{L}(\hat{\mathbf{y}}^{(t)}, \mathbf{y}^{(t)})}{\partial \mathbf{V}} = \sum_{t=1}^N \frac{\partial \mathcal{L}(\hat{\mathbf{y}}^{(t)}, \mathbf{y}^{(t)})}{\partial \mathbf{y}^{(t)}} \frac{\partial \mathbf{y}^{(t)}}{\partial \mathbf{V}} \quad (4.18)$$

$$\frac{\partial L}{\partial \mathbf{b}} = \sum_{t=1}^N \frac{\partial \mathcal{L}(\hat{\mathbf{y}}^{(t)}, \mathbf{y}^{(t)})}{\partial \mathbf{b}} = \sum_{t=1}^N \frac{\partial \mathcal{L}(\hat{\mathbf{y}}^{(t)}, \mathbf{y}^{(t)})}{\partial \mathbf{y}^{(t)}} \frac{\partial \mathbf{y}^{(t)}}{\partial \mathbf{h}^{(t)}} \frac{\partial \mathbf{h}^{(t)}}{\partial \mathbf{b}} \quad (4.19)$$

$$\frac{\partial L}{\partial \mathbf{c}} = \sum_{t=1}^N \frac{\partial \mathcal{L}(\hat{\mathbf{y}}^{(t)}, \mathbf{y}^{(t)})}{\partial \mathbf{c}} = \sum_{t=1}^N \frac{\partial \mathcal{L}(\hat{\mathbf{y}}^{(t)}, \mathbf{y}^{(t)})}{\partial \mathbf{y}^{(t)}} \frac{\partial \mathbf{y}^{(t)}}{\partial \mathbf{c}} \quad (4.20)$$

Las ecuaciones 4.18 y 4.20 son fáciles de calcular, ya que no hay ningún problema de recursión en sus factores. Los factores de estas ecuaciones serían los siguientes:

$$\frac{\partial \mathbf{y}^{(t)}}{\partial \mathbf{V}} = \psi'(\mathbf{V}\mathbf{h}^{(t)} + \mathbf{c}) \cdot \mathbf{h}^{(t)} \quad , \quad \frac{\partial \mathbf{y}^{(t)}}{\partial \mathbf{c}} = \psi'(\mathbf{V}\mathbf{h}^{(t)} + \mathbf{c})$$

Así mismo, los factores primero y segundo de las ecuaciones 4.16, 4.17 y 4.19, se calculan directamente:

$$\frac{\partial \mathbf{y}^{(t)}}{\partial \mathbf{h}^{(t)}} = \psi'(\mathbf{V}\mathbf{h}^{(t)} + \mathbf{c}) \cdot \mathbf{V}$$

Sin embargo, en los factores $\frac{\partial \mathbf{h}^{(t)}}{\partial \mathbf{U}}$, $\frac{\partial \mathbf{h}^{(t)}}{\partial \mathbf{W}}$ y $\frac{\partial \mathbf{h}^{(t)}}{\partial \mathbf{b}}$, hay problemas de recursión. En estos casos tenemos que calcular el efecto de los parámetros \mathbf{U} , \mathbf{W} y \mathbf{b} respectivamente en $\mathbf{h}^{(t)}$ de forma recurrente. Entonces, como hemos visto en las fórmulas 4.13 y 4.14, $h^{(t)}$ depende tanto de $h^{(t-1)}$ como de \mathbf{U} , \mathbf{W} y \mathbf{b} y a su vez $h^{(t-1)}$ también depende de los estados ocultos en instantes anteriores y de \mathbf{U} , \mathbf{W} y \mathbf{b} .

Vamos a calcular únicamente $\frac{\partial \mathbf{h}^{(t)}}{\partial \mathbf{W}}$ ya que el cálculo de los otros dos factores es totalmente análogo. Si tenemos en cuenta la regla de la cadena y la derivada de un producto, se tiene que

$$\frac{\partial \mathbf{h}^{(t)}}{\partial \mathbf{W}} = \frac{\partial \phi(\mathbf{a}^{(t)})}{\partial \mathbf{a}^{(t)}} \frac{\partial \mathbf{a}^{(t)}}{\partial \mathbf{W}} = \frac{\partial \phi(\mathbf{a}^{(t)})}{\partial \mathbf{W}} + \frac{\partial \phi(\mathbf{a}^{(t)})}{\partial \mathbf{h}^{(t-1)}} \frac{\partial \mathbf{h}^{(t-1)}}{\partial \mathbf{W}}$$

¿Cómo podemos calcular esta fórmula recursiva? Con el objetivo de calcular el gradiente anterior, vamos a suponer que tenemos tres series a_t , b_t y c_t tales que satisfacen que $a_0 = 0$ y $a_t = b_t + c_t a_{t-1}$ para $t = 1, 2, 3, \dots$. Entonces, para $t \geq 1$ se comprueba por inducción que

$$a_t = b_t + \sum_{i=1}^{t-1} \left(\prod_{j=i+1}^t c_j \right) b_i$$

En efecto: para $t = 1$ se tiene que $a_1 = b_1$. Suponemos cierto el resultado para t , vamos a comprobar que se cumple para $t+1$:

$$\begin{aligned} a_{t+1} &= b_{t+1} + c_{t+1} a_t = \\ b_{t+1} + c_{t+1} \left(b_t + \sum_{i=1}^{t-1} \left(\prod_{j=i+1}^t c_j \right) b_i \right) &= \\ b_{t+1} + b_t c_{t+1} + c_{t+1} \sum_{i=1}^{t-1} \left(\prod_{j=i+1}^t c_j \right) b_i &= \\ b_{t+1} + b_t c_{t+1} + \sum_{i=1}^{t-1} \left(\prod_{j=i+1}^{t+1} c_j \right) b_i &= \\ b_{t+1} + \sum_{i=1}^t \left(\prod_{j=i+1}^{t+1} c_j \right) b_i & \end{aligned}$$

Ahora, sustituyendo a_t , b_t y c_t respectivamente por

$$a_t = \frac{\partial \mathbf{h}^{(t)}}{\partial \mathbf{W}} \quad , \quad b_t = \frac{\partial \phi(\mathbf{a}^{(t)})}{\partial \mathbf{W}} \quad , \quad c_t = \frac{\partial \phi(\mathbf{a}^{(t)})}{\partial \mathbf{h}^{(t-1)}}$$

Nos queda que

$$\frac{\partial \mathbf{h}^{(t)}}{\partial \mathbf{W}} = \frac{\partial \phi(\mathbf{a}^{(t)})}{\partial \mathbf{W}} + \sum_{i=1}^{t-1} \left(\prod_{j=i+1}^t \frac{\partial \phi(\mathbf{a}^{(j)})}{\partial \mathbf{h}^{(j-1)}} \right) \frac{\partial \phi(\mathbf{a}^{(i)})}{\partial \mathbf{W}}$$

Como último paso en la retropropagación, debemos actualizar cada parámetro proporcionalmente a su gradiente. Veamos cómo se actualiza el parámetro \mathbf{U} .

$$\mathbf{U}_{new} = \mathbf{U} - \mu \left(\frac{\partial L}{\partial \mathbf{U}} \right) \quad (4.21)$$

donde \mathbf{U}_{new} es el nuevo valor del parámetro \mathbf{U} tras realizar el descenso de gradiente y μ es la tasa de aprendizaje o *learning rate*. Nótese que la ecuación para actualizar el resto de los parámetros de la red neuronal es análoga a 4.21.

Acabamos de ver cómo funciona la retropropagación a través del tiempo en una red neuronal recurrente, que junto con la propagación hacia delante de la red son los mecanismos que hacen “aprender” a la red neuronal.

Además, hemos visto cómo las redes neuronales recurrentes pueden ayudarnos a realizar un análisis mejor a la hora de tratar con datos secuenciales y capturar las relaciones entre datos espaciales o temporales como son las palabras en una frase. Sin embargo, a este tipo de redes vienen asociados ciertos problemas como el problema del desvanecimiento del gradiente o el problema de gradiente explosivo.

4.3.3. Problemas de entrenamiento en las redes neuronales recurrentes

Como ya sabemos, para entrenar una red neuronal se tienen que actualizar los parámetros de la red de forma que la función de pérdida llegue a un mínimo. Los gradientes calculados en la retropropagación nos ayudan a recalcular los parámetros de la red en la dirección y cantidad correctas.

Pero, ¿qué pasaría si los valores de los gradientes se vuelven muy pequeños? En ese caso, los parámetros se actualizarían muy lentamente y entonces el aprendizaje dejaría de llevarse a cabo. Este problema es conocido

como el problema del **desvanecimiento del gradiente** (en inglés *vanishing gradient*) y afecta particularmente a redes neuronales con un gran número de capas ocultas o a RNN donde ir hacia atrás en cada etapa de tiempo equivale a retropropagar el error de la capa anterior en una ANN.

Análogamente, si el gradiente asociado a un parámetro se vuelve extremadamente grande, los cambios en los parámetros también serían grandes. Esto causa inestabilidad en los gradientes lo que puede llevar a que el algoritmo no converja. Este problema se conoce como el problema del **gradiente explosivo** (en inglés *exploding gradient*).

¿Por qué ocurren estos problemas? En el caso del desvanecimiento del gradiente, hay que tener en cuenta que algunas funciones de activación, como la sigmoide, hacen que la salida de la neurona esté acotada entre 0 y 1. Entonces, un gran cambio en la entrada de la función sigmoide causa un cambio muy pequeño en la salida, así que la derivada es también pequeña.

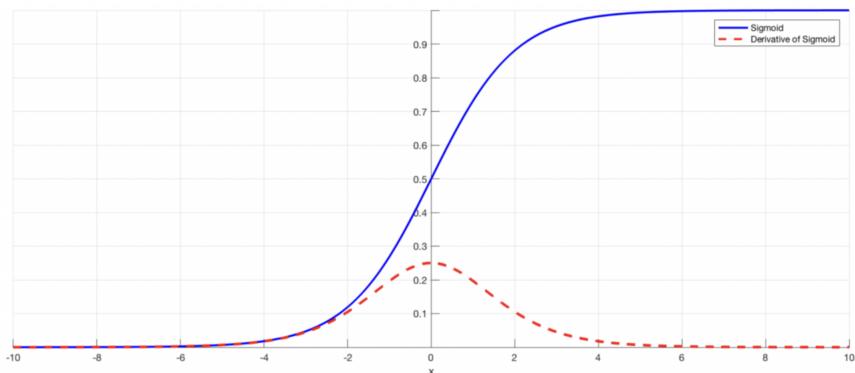


Figura 4.4: Función sigmoide y su derivada.

Lo que ocurre entonces en redes neuronales muy profundas o en redes neuronales recurrentes es que, para calcular los gradientes de la retropropagación se multiplica varias veces la derivada de la función de activación, lo que hace que el gradiente decrezca exponencialmente cuando retropropagamos el error a las capas iniciales. En consecuencia, esto produce que los parámetros de la red se actualicen muy lentamente.

Sin embargo, la razón detrás del problema del gradiente explosivo suele ser el valor de los parámetros de la red y no las funciones de activación. Si estos parámetros son grandes, como vienen multiplicados varias veces para

el cálculo de los gradientes, éstos se vuelven extremadamente grandes.

Existen diversos métodos para solventar estos dos problemas, uno de los cuales explicaremos a continuación.

4.4. Long short-term memory (LSTM)

Debido al problema del desvanecimiento del gradiente, las redes neuronales recurrentes no son capaces de capturar las dependencias entre términos lejanos en la secuencia. Sin embargo, si por ejemplo la palabra en la posición octava de una frase tiene una relación de causa con la palabra primera de la frase, es esencial capturar y recordar la relación entre ambas.

Las celdas LSTM han sido diseñadas para recordar dependencias entre datos lejanos en la secuencia y guardar la información necesaria en la memoria todo el tiempo que se necesite. Se implementaron también con el objetivo de mitigar el problema del desvanecimiento del gradiente y del gradiente explosivo.

De manera más técnica, la LSTM es una arquitectura de red neuronal recurrente que se diferencia por tener puertas (o *gates*) que ayudan a que la celda LSTM recuerde valores en intervalos de tiempo arbitrarios y que regulan el flujo de información que entra y sale de ella. De hecho esta arquitectura pertenece a las llamadas “Gated Recurrent Neural Networks”.

La arquitectura LSTM consiste en una red neuronal recurrente en la que cada unidad oculta es reemplazada por una celda LSTM. Además, existe una nueva conexión de celda en celda llamada “el estado de la celda” (del inglés *cell state*). Veamos un diagrama representativo de esta arquitectura:

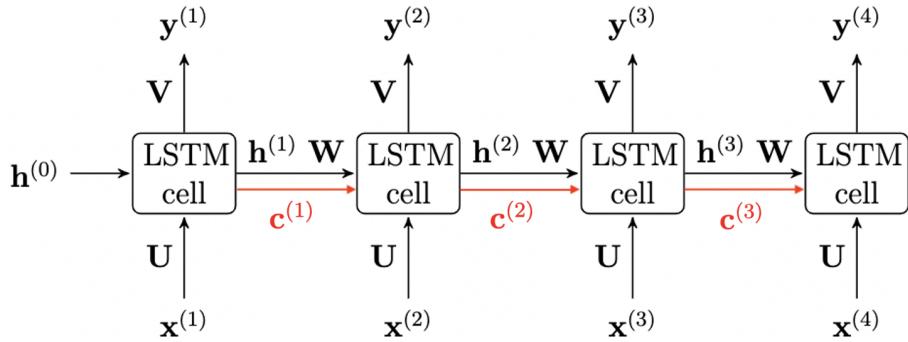


Figura 4.5: Arquitectura LSTM, en este caso con 4 etapas de tiempo.

4.4.1. Funcionamiento de una celda LSTM

El valor de entrada para una celda LSTM es una concatenación del valor de entrada en ese paso de tiempo $\mathbf{x}^{(t)}$ y el estado oculto del paso de tiempo anterior $\mathbf{h}^{(t-1)}$. Estos vectores son manipulados según las distintas puertas en la celda, que no son nada más que redes neuronales prealimentadas (*Feedforward neural network o FNN*) con una función de activación. Estas puertas son: **puerta olvidar o forget** (*forget gate*), **puerta de entrada** (*input gate*) y **puerta de salida** (*output gate*).

Las redes neuronales de cada una de estas puertas se entranan mediante retro-propagación y permiten a la señal fluir con diferentes estados de memoria. Éstas deciden qué información debe ser recordada, olvidada o descartada en cada paso de tiempo. Vamos a estudiar individualmente en detalle el funcionamiento de estas puertas.

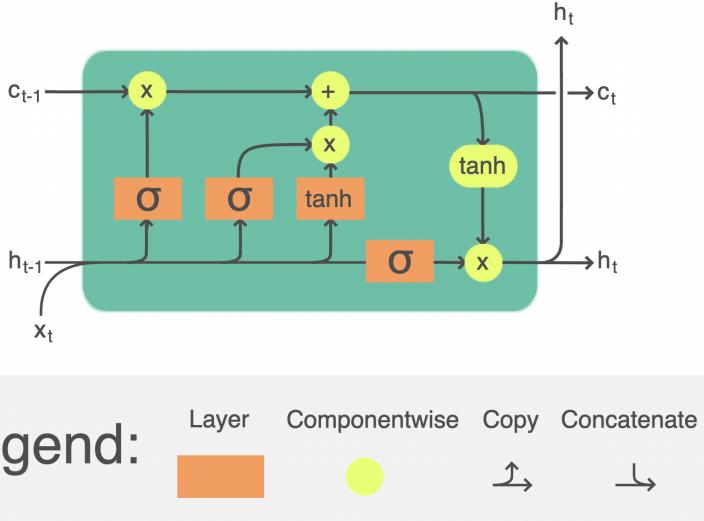


Figura 4.6: Celda LSTM. Imagen extraída de Wikipedia[15].

FORGET GATE O PUERTA OLVIDAR

La primera puerta es la *forget gate*. La misión de esta puerta es decidir cuánta información debe ser eliminada de la memoria. Esto se realiza porque, para que las celdas LSTM puedan ser eficientes en su tarea de recordar dependencias, éstas deben diferenciar entre lo que es más importante recordar y lo que no. Básicamente, nuestra red debe ser capaz de olvidar dependencias a largo término para que, cuando lleguen nuevas dependencias que valgan la pena recordar, éstas tengan espacio para ser almacenadas.

Como ya se ha mencionado anteriormente, la *forget gate* es una FNN (red neuronal prealimentada) cuya función de activación es la función sigmoide que nos da un valor de salida entre 0 y 1, lo cual nos da una idea de cuánta información debe ser olvidada. Un valor de salida de 0 de la puerta *forget gate* nos indica que se debe olvidar toda la información anterior. Por el contrario, si el valor de salida es 1, todo el estado de memoria anterior debe ser retenido.

Posteriormente, cuando tenemos el valor de salida de la *forget gate*, éste es multiplicado por los valores del estado de memoria de la celda anterior

para retener solo la información más importante de los estados pasados.

$$\text{Matemáticamente, } \mathbf{f}^{(t)} = \sigma (\mathbf{U}_f \mathbf{x}^{(t)} + \mathbf{W}_f \mathbf{h}^{(t-1)} + \mathbf{b}_f)$$

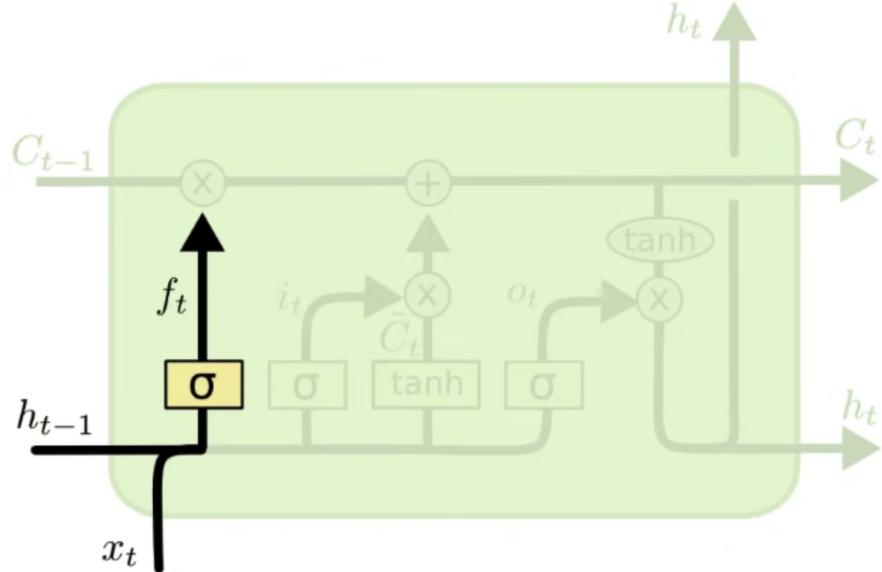


Figura 4.7: Celda LSTM, con la parte *forget gate* marcada. Imagen de la página web *colah's blog*[7].

INPUT GATE O PUERTA DE ENTRADA

El objetivo de la puerta de entrada o *Input Gate* es saber cuáles son las nuevas dependencias a recordar y cuánto recordar de ellas. Para ello, esta puerta se compone de dos partes:

- La primera parte de la puerta de entrada usa una función de activación *sigmoide* para precisar qué partes de los valores de entrada tiene que ser recordadas. Este proceso lo realiza creando una especie de máscara con valores entre 0 y 1. Un valor de 0 indicaría que los inputs de ese estado no vale la pena ser recordados y, por el contrario, un valor de 1 indicaría que toda la información de valor de entrada en ese estado debe ser recordada. Matemáticamente, $\mathbf{i}^{(t)} = \sigma (\mathbf{U}_i \mathbf{x}^{(t)} + \mathbf{W}_i \mathbf{h}^{(t-1)} + \mathbf{b}_i)$

- La segunda parte usa una función de activación *tangente hiperbólica*

para detectar cuál es potencialmente la información más relevante que la celda puede actualizar en el estado de memoria presente. Esta parte es también conocida como “el vector candidato”, ya que recoge los valores con los que la celda LSTM puede llegar a ser actualizada. En este caso, los valores de salida van de -1 a 1. Matemáticamente, $\bar{C}^{(t)} = \tanh(\mathbf{U}_C x^{(t)} + \mathbf{W}_C h^{(t-1)} + \mathbf{b}_C)$

Después de obtener los valores de salida de la primera y segunda parte, éstos dos se multiplican elemento por elemento. En esencia, lo que se ha hecho ha sido entender cuánto de relevantes son los componentes de la parte 2 en función de los valores de salida de la parte 1.

Posteriormente, el output resultante se suma al vector de memoria obtenido en la *forget gate*, actualizando así la información o memoria de esa particular celda LSTM. Matemáticamente, $C^{(t)} = f^{(t)} C^{(t-1)} + i^{(t)} \bar{C}^{(t)}$

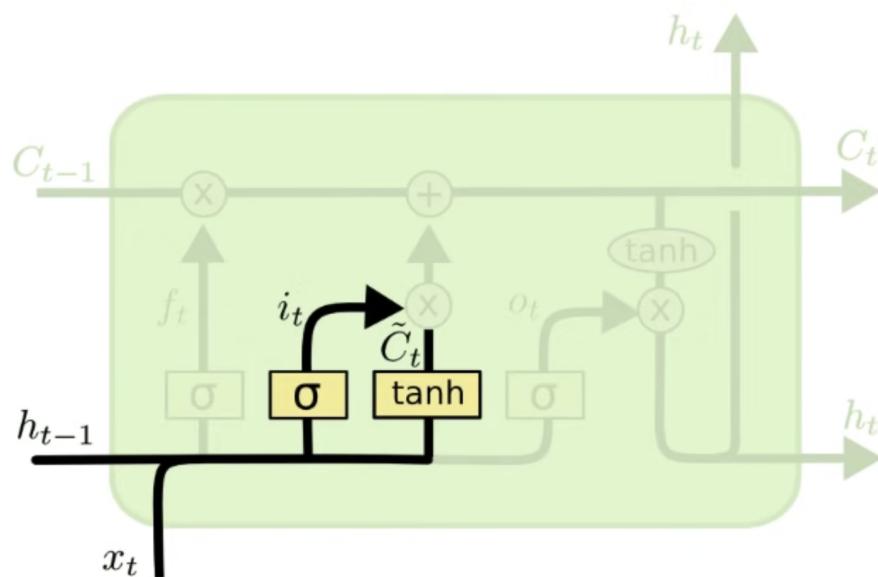


Figura 4.8: Celda LSTM, con la parte *input gate* marcada. Imagen de la página web *colah's blog*[7].

OUTPUT GATE O PUERTA DE SALIDA

La tarea de la puerta de salida o *Output gate* es la de controlar qué partes de la información de ese estado de memoria se van a enviar como salida de esa celda LSTM. En esta etapa de la celda LSTM, ocurren dos cosas:

1. Primero, la puerta de salida recibe el input de la celda y estos valores pasan por una red neuronal prealimentada y por una función de activación sigmoide. Matemáticamente, $\mathbf{o}^{(t)} = \sigma(\mathbf{U}_o \mathbf{x}^{(t)} + \mathbf{W}_o \mathbf{h}^{(t-1)} + \mathbf{b}_o)$
2. En segundo lugar, el estado de memoria en este punto ya ha sido actualizado en las puertas anteriores en base a lo que debe ser olvidado y recordado. Así, a estos valores se les aplica una función de activación de tipo tangente hiperbólica.

Finalmente, los valores del estado de memoria tras la aplicación de la función de activación tangente hiperbólica se multiplican elemento por elemento a los valores de salida de la función de activación sigmoide para conseguir el output final de esta celda LSTM de la red. Este output es, además, el estado oculto para el siguiente instante de tiempo de la celda LSTM. Matemáticamente, $\mathbf{h}^{(t)} = \tanh(\mathbf{C}^{(t)}) \circ \mathbf{o}^{(t)}$, donde \circ denota el producto de Hadamard entre matrices, también conocido como producto elemento a elemento.

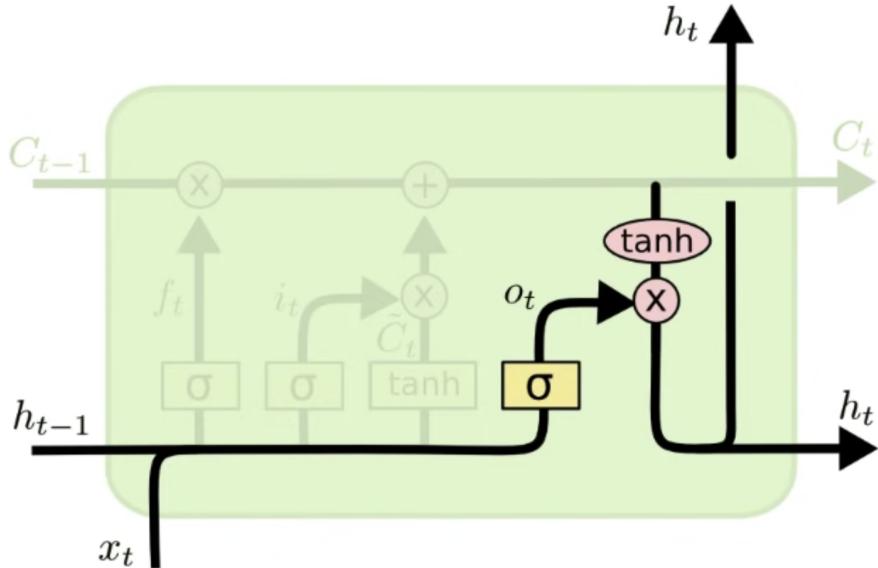


Figura 4.9: Celda LSTM, con la parte *output gate* marcada. Imagen de la página web *colah's blog*[7].

4.4.2. Retropropagación en el tiempo en las LSTM

A la hora de entrenar este modelo mediante retropropagación, hay que tener en cuenta que hay 3 puertas por cada celda LSTM, lo cual implica que tenemos bastantes parámetros de la red neuronal que deben ser actualizados.

La retropropagación en la arquitectura LSTM funciona de forma similar a las redes neuronales recurrentes. Sin embargo, en comparación con las redes neuronales recurrentes, en la arquitectura LSTM no nos encontramos con el problema del desvanecimiento de gradiente o del gradiente explosivo gracias a la componente de memoria introducida en estas celdas LSTM. Los parámetros de las redes neuronales de cada una de las puertas fueron usados para actualizar la memoria de las celdas. Estos mismos parámetros se actualizan en la retropropagación usando los gradientes de la función pérdida con respecto a los mismos, es decir, usando las varias derivadas parciales de las funciones aplicadas en la propagación hacia delante.

4.5. Gated Recurrent Unit (GRU)

Debido a la gran cantidad de parámetros a modelar en la arquitectura LSTM que hacen que el proceso de entrenamiento sea mucho más lento y costoso computacionalmente, uno se pregunta si realmente es necesaria una estructura tan compleja.

Las *Gated Recurrent Unit* o GRU resuelven este problema ya que contienen dos puertas en vez de tres. Combinan la *forget gate* y la parte del “vector candidato” de la *input gate* en una sola puerta llamada la *update gate*. La otra puerta se denomina *reset gate* y es la encargada de decidir cómo se actualiza la memoria con la nueva información. En base a los valores de salida de estas dos puertas, se decide cuál es el valor de salida de la celda GRU y cuál es el estado oculto en ese instante de tiempo. Este último proceso se lleva a cabo mediante el “content state”.

Vamos a explicar el funcionamiento de una celda GRU más detalladamente. Los valores de entrada a la celda en el instante t son el vector de entrada $\mathbf{x}^{(t)}$ y el vector estado oculto de la etapa anterior $\mathbf{h}^{(t-1)}$. Gracias a estos, se pueden calcular el vector *update gate* $\mathbf{z}^{(t)}$ y el vector *reset gate* $\mathbf{r}^{(t)}$ mediante las ecuaciones 4.22 y 4.23. Por último, se calcula el vector candidato $\hat{\mathbf{h}}^{(t)}$ y con éste se calcula el vector de salida $\mathbf{h}^{(t)}$.

Las ecuaciones que rigen el funcionamiento de una celda GRU son

$$z^{(t)} = \sigma \left(U_z x^{(t)} + W_z h^{(t-1)} + b_z \right) \quad (4.22)$$

$$r^{(t)} = \sigma \left(U_r x^{(t)} + W_r h^{(t-1)} + b_r \right) \quad (4.23)$$

$$\hat{h}^{(t)} = \tanh \left(U_h x^{(t)} + W_h (r^{(t)} \circ h^{(t-1)}) + b_h \right) \quad (4.24)$$

$$h^{(t)} = (1 - z^{(t)}) \circ h^{(t-1)} + z^{(t)} \circ \hat{h}^{(t)} \quad (4.25)$$

siendo $h^{(0)} = 0$, σ la función sigmoide, \tanh la función tangente hiperbólica y \circ el producto de Hadamard.

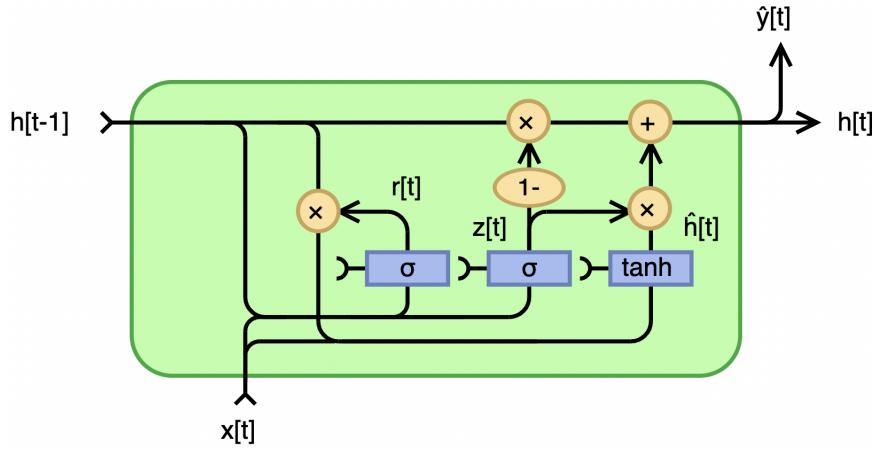


Figura 4.10: Celda GRU. Imagen extraída de Wikipedia [14].

Como consecuencia de la existencia de las GRUs, el número de parámetros de la red se reduce drásticamente y por tanto son más rápidas en el proceso de entrenamiento del modelo.

En este capítulo hemos revisado toda la arquitectura de redes neuronales, desde las más básicas hasta las redes neuronales recurrentes, LSTM y GRU. Hemos visto cómo éstas nos permiten capturar las relaciones entre datos secuenciales como los textos. En el próximo capítulo vamos a ir un paso más allá y vamos a estudiar los modelos *sequence-to-sequence (seq2seq)*⁷ (es decir, secuencia a secuencia) que son uno de los últimos avances en cuanto a técnicas de PLN. Éstos usan codificadores (o *encoders*) y decodificadores (o *decoders*) dentro de sus arquitecturas.

⁷Seq2seq es una familia de algoritmos de aprendizaje automático para tareas de PLN. Este tipo de modelos transforma una secuencia (de texto) en otra, mediante el uso de una RNN, más a menudo de LSTM o GRU para evitar el problema del desvanecimiento del gradiente.

Capítulo 5

Los transformers

5.1. Introducción y origen de los transformers

Hasta ahora, se han visto varias técnicas clásicas de PLN y también se ha estudiado la esencia detrás de las redes neuronales. En este capítulo, se va a estudiar la arquitectura Transformer¹, que es uno de los modelos de última generación en *machine learning* y más concretamente en *deep learning*.

Desde su introducción en 2017, los transformers se han convertido en los modelos estándar para abordar un amplio rango de tareas en el campo del procesamiento del lenguaje natural, tanto en investigación como en industria. Probablemente hoy mismo ya has interactuado con un transformer. Por ejemplo, Google usa BERT en su motor de búsqueda para entender mejor las consultas de los usuarios.

Pero, ¿qué tienen de especial los transformers para haber cambiado tan

¹Notación: se va a utilizar la palabra “Transformer” para referirnos a la red neuronal original que fue introducida en el famoso paper *Attention is All You Need* [13]. Sin embargo para referirnos en general a los modelos basados en la arquitectura Transformer se utilizará el término “transformers”.

radicalmente el campo del PLN? Como muchos otros avances científicos, fue la culminación de varias ideas nuevas como los mecanismos de atención, el *transfer learning* y la mejora continua de las redes neuronales la que animó a la comunidad de investigadores a dar con esta nueva arquitectura.

Para empezar, ¿qué entendemos por Transformer? En 2017 investigadores de Google publicaron un *paper*[13] en el cual se proponía una novedosa arquitectura de *deep learning* basada en redes neuronales llamada Transformer que estaba diseñada para trabajar con datos secuenciales (como es el caso del lenguaje natural). Vamos a entender lo que realmente fue la parte más novedosa sobre esta arquitectura:

- **La estructura *encoder-decoder*.**

Este tipo de estructura ya existía anteriormente, como en el caso de algunas redes neuronales recurrentes. Como su nombre sugiere, la tarea del *encoder* es la de codificar la información de la secuencia de entrada en una representación numérica a la que normalmente se le llama “último estado oculto” (en inglés *last hidden state*). Este estado pasa al *decoder*, a partir del cual se genera la secuencia de salida. En general, las partes del *encoder* y *decoder* pueden ser cualquier tipo de arquitectura basada en redes neuronales que se adapte bien para modelar secuencias.

- **Los mecanismos de atención.**

La idea principal detrás de los mecanismos de atención es que en vez de producir un único *hidden state* para toda la secuencia de entrada al modelo (ver Figura 5.1), el encoder genera un *hidden state* para cada etapa de tiempo al cual el *decoder* puede acceder. Sin embargo, en este caso es necesario crear un mecanismo para priorizar unos estados frente a otros y aquí es donde los mecanismos de atención entran en juego. Éstos permiten al *decoder* asignar un peso o “atención” más fuerte a los estados pasados específicos que tienen más relevancia a la hora de producir el siguiente elemento en la secuencia de salida (Figura 5.2). Este proceso es, además, diferenciable por lo que los mecanismos de atención también pueden ser entrenados.

La arquitectura Transformer tomó esta idea de “atención” y la llevó un paso más allá reemplazando las unidades dentro del *encoder* y *decoder* por redes con capas de *self-attention* (sección 5.2.1) y simples capas prealimentadas (o en inglés *feed-forward*), como se ve en la Figura 5.3.

En las figuras 5.1, 5.2 y 5.3 se observa gráficamente cómo se desarrolló la idea de los mecanismos de atención con un ejemplo aplicado a la

traducción automática de textos, donde el objetivo es transformar una secuencia de palabras de una lengua a otra.

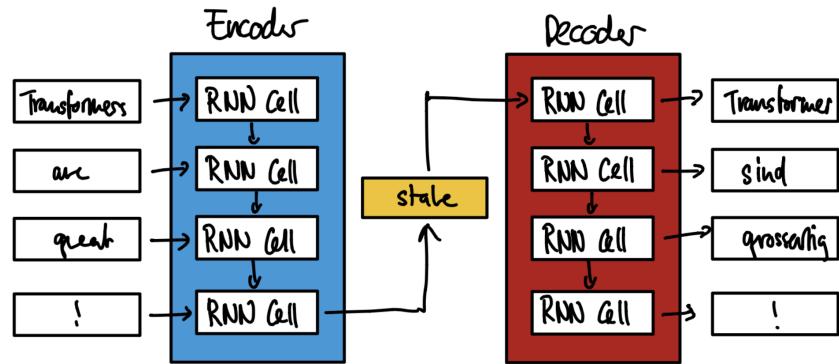


Figura 5.1: Arquitectura encoder-decoder con un par de RNNs. Imagen del libro *Natural Language Processing with Transformers*. [11]

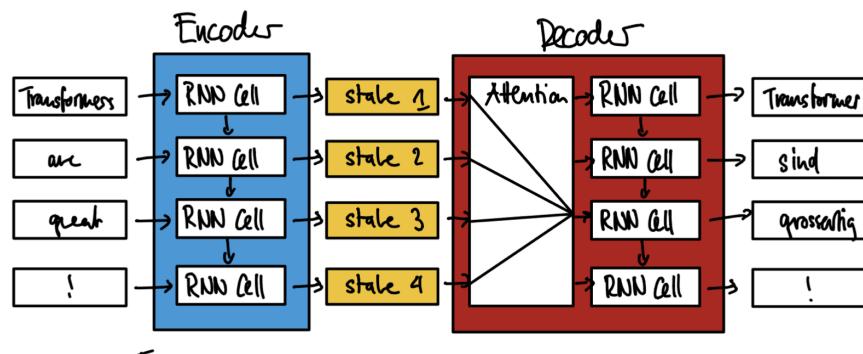


Figura 5.2: Arquitectura encoder-decoder con un mecanismo de atención para dos RNNs. Imagen del libro *Natural Language Processing with Transformers*. [11]

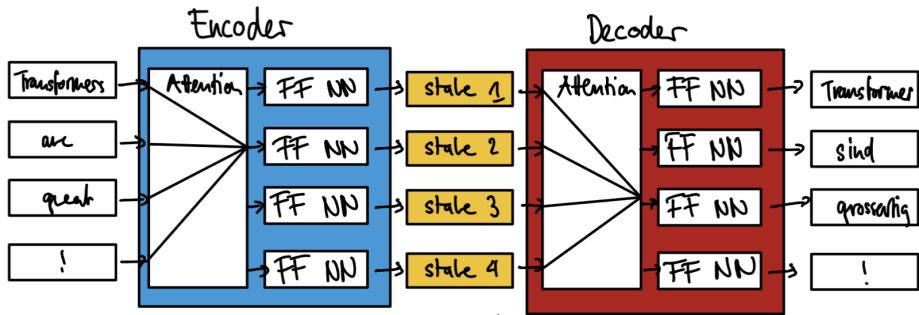


Figura 5.3: Arquitectura encoder-decoder con un par de transformers. Imagen del libro *Natural Language Processing with Transformers*. [11]

- ***Transfer learning* o aprendizaje por transferencia.**

El *transfer learning* consiste en reutilizar un modelo de machine learning ya pre-entrenado en un problema distinto pero relacionado con la tarea en la que ha sido pre-entrenado inicialmente. De esta manera, se hace uso del conocimiento ya adquirido por el modelo inicialmente. Estructuralmente, esto funciona normalmente dividiendo el modelo en dos partes: cuerpo y cabeza (en inglés *body and head*). El cuerpo del modelo es la parte común y sus pesos se actualizan durante la fase del pre-entrenamiento, aprendiendo las características más generales del campo al cual pertenezca el problema a resolver. Posteriormente, dependiendo de la tarea específica a resolver, se utiliza una cabeza u otra, la cual es inicializada con los pesos calculados en el pre-entrenamiento y actualizada para resolver la nueva tarea específica.

Antes del desarrollo y uso de los transformers, los modelos para el procesamiento del lenguaje natural de última generación eran las *gated RNNs* como las unidades LSTM o las GRU con mecanismos de atención. Estas arquitecturas contienen un bucle de retroalimentación en las conexiones de la red que permite a la información propagarse de una etapa a otra, lo cual las vuelve idóneas para modelar datos secuenciales como lo es el lenguaje. Sin embargo, gracias al desarrollo de la arquitectura Transformer, se vio que los mecanismos de atención por sí solos eran suficientemente efectivos y que el procesamiento secuencial de los datos no es algo necesario para lograr los resultados que alcanzan las redes neuronales recurrentes con mecanismos de atención.

Además, el hecho de no procesar los datos en orden, como es el caso

de los transformers, les aporta grandes ventajas como la posibilidad de una computación paralela² (en inglés *parallelization*), reduciéndose así los tiempos de entrenamiento.

5.2. La estructura general

Los modelos Transformer se basan en la idea de convertir un conjunto de secuencias de entrada en un conjunto de estados ocultos que son posteriormente decodificados y transformados de nuevo en un conjunto de secuencias de salida.

Por lo tanto, se puede decir que los transformers se basan en la arquitectura codificador-decodificador (del inglés *encoder-decoder*) que se usa extensamente en las tareas de traducción automática de textos o resumen de textos. Esta arquitectura consta de dos componentes (ver figura 5.4):

- **Encoder:** lee la secuencia de tokens de entrada y la codifica en una secuencia de vectores de longitud fija que normalmente se denomina “vector de contexto” (o *hidden state*).
- **Decoder:** lee el vector de contexto o *hidden state* y genera un token de salida iterativamente, formando así toda una secuencia de tokens.

²Computación paralela: es una forma de cómputo en la que muchas instrucciones se ejecutan simultáneamente.

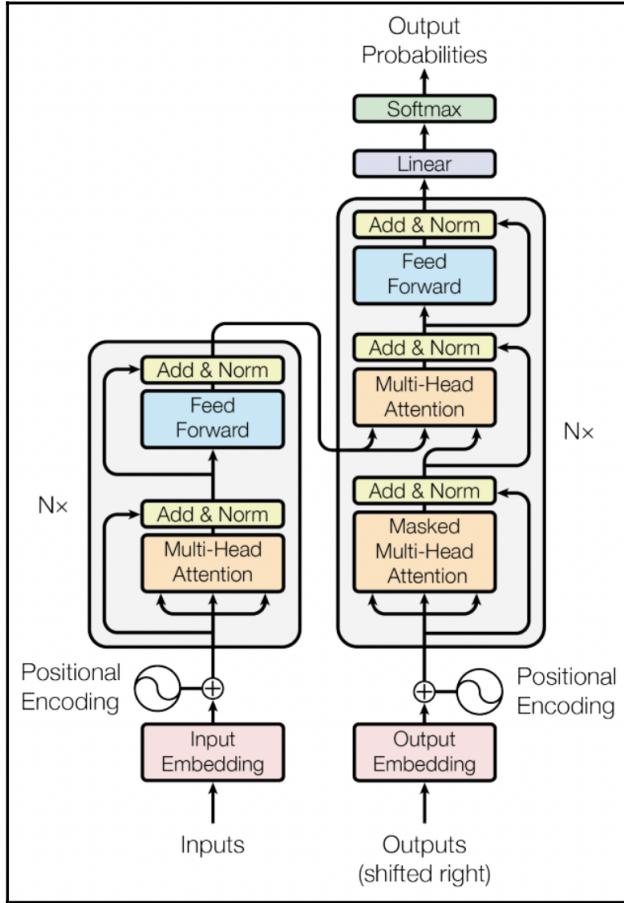


Figura 5.4: El modelo Transformer del paper original *Attention is all you need* [13]. El bloque de la izquierda es el *encoder* y el de la derecha, el *decoder*.

La arquitectura Transformer fue diseñada originalmente para tareas secuencia-a-secuencia (del inglés *sequence-to-sequence*) como la traducción automática, pero pronto se adaptó a otro tipo de tareas utilizando solo la componente *encoder* o la componente *decoder* como modelos en sí mismos. Así, aunque existen cientos de modelos transformer, la mayoría de ellos se pueden categorizar en uno de los siguientes grupos:

- **Solo encoder:** este tipo de modelos se utilizan sobretodo para tareas de clasificación de textos o *name entity recognition (NER)*. Ejemplos de esta clase de modelos son BERT y sus variantes como RoBERTa y

DistilBERT.

- **Solo decoder:** modelos de este tipo pueden ser útiles para, dada una frase incompleta, completarla automáticamente prediciendo iterativamente la palabra siguiente más probable. Un ejemplo de esta familia es GPT.
- **Encoder-decoder:** se usa para modelar una secuencia de texto en otra, como por ejemplo se hace en la traducción automática de textos o en el resumen de textos. Modelos de este tipo son el Transformer, BART y T5.

5.2.1. Encoder de un transformer

La parte *encoder* de la arquitectura Transformer consiste en muchas capas *encoder* apiladas una sobre otra. En particular, en el modelo original Transformer hay 6 capas apiladas. La estructura de estas capas es idéntica pero no comparten los pesos entre cada una de ellas. El input consiste en una secuencia de embeddings. Estos embeddings pueden ser vectores one-hot o otras formas de embeddings como por ejemplo los aprendidos con el modelo Word2Vec.

Cada capa a su vez contiene dos sub-capas: la *multi-head self-attention* y la *position-wise fully connected FNN*. Cada una de las sub-capas tiene también una capa residual (del inglés, *skip or residual connection*) y una capa de normalización (del inglés, *layer normalization*), que son básicamente trucos estándar para llegar a entrenar redes neuronales profundas de una manera más efectiva.

La salida de cada capa del *encoder* es un conjunto de embeddings de la misma dimensión que los inputs, ya que básicamente la función del bloque de *encoders* es la de actualizar los embeddings de entrada para que no sólo representen al token en sí, sino que además incluyan información contextual de la secuencia.

Para entender realmente la arquitectura, tenemos que estudiar por separado todas las partes que la forman. Empezamos por la capa *self-attention*.

SELF-ATTENTION

Self-attention es el mecanismo por el cual las redes neuronales tienen la capacidad de asignar una cantidad diferente de peso o “atención” a cada elemento de la secuencia. La parte llamada *self* se refiere al hecho de que estos pesos se calculan para todos los estados ocultos del mismo conjunto a la vez, es decir, todos los estados ocultos del *encoder*. Esta es una de las mayores diferencias con los modelos de redes neuronales recurrentes en las cuales se computa la relevancia de cada estado oculto del *encoder* en un instante de tiempo particular del *decoder*.

La idea principal detrás de la *self-attention* es que en vez de usar un solo embedding fijo para cada token, se usa la secuencia entera para calcular una media ponderada de cada embedding. Dicho de otro modo, dada una secuencia de embeddings de tokens $\{\mathbf{x}_1, \dots, \mathbf{x}_n\}$, el mecanismo de *self-attention* produce una secuencia de nuevos embeddings $\{\mathbf{y}_1, \dots, \mathbf{y}_n\}$ donde cada \mathbf{y}_i con $i = 1, \dots, n$ es una combinación lineal de los embeddings de entrada \mathbf{x}_i como en

$$\mathbf{y}_i = \sum_{j=1}^n w_{ji} \mathbf{x}_j.$$

Los coeficientes w_{ji} son los llamados “pesos de atención” (en inglés, *attention weights*) y están normalizados de modo que $\sum_j w_{ji} = 1$.

¿Por qué se realiza todo este proceso de ponderar los embeddings de los tokens? La respuesta está en que mediante este proceso los nuevos embeddings generados contienen nueva información sobre el contexto de cada token y así pueden llegar a representar de manera mucho más exacta ese token particular. A los embeddings generados de esta manera se les denomina *contextualized embeddings*.

Otra pregunta que surge es, ¿cómo se calculan los pesos de atención? Hay diversos modos para implementar este mecanismo de atención, pero el más común es el llamado **Scaled Dot-Product Attention** propuesto en el paper donde fue introducida la arquitectura Transformer [13]. A continuación se explica el funcionamiento.

1. Se crean los vectores *query*, *key* y *value* para cada token. Su-

pongamos que tenemos los embeddings de dimensión d_{model} ³ de una secuencia de tokens $\{\mathbf{x}_1, \dots, \mathbf{x}_n\}$. Veamos lo que ocurre para un embedding cualquiera \mathbf{x}_j de la secuencia. En el primer paso, el embedding de entrada \mathbf{x}_j se multiplica por tres matrices, W_Q , W_K y W_V . Esta multiplicación genera otros tres embeddings, llamados *query*, *key* y *value*. Los dos primeros son de dimensión d_k y el último, de dimensión d_v . El modo por el cual se han obtenido esas tres matrices ha sido mediante el entrenamiento del modelo, usando la técnica de la retropropagación.

Un esquema ilustrativo es el representado en la tabla 5.1.

		W_Q	W_K	W_V
Token 1	Embedding 1 (\mathbf{x}_1)	query 1 (\mathbf{q}_1)	key 1 (\mathbf{k}_1)	value 1 (\mathbf{v}_1)
Token 2	Embedding 2 (\mathbf{x}_2)	query 2 (\mathbf{q}_2)	key 2 (\mathbf{k}_2)	value 2 (\mathbf{v}_2)
Token 3	Embedding 3 (\mathbf{x}_3)	query 3 (\mathbf{q}_3)	key 3 (\mathbf{k}_3)	value 3 (\mathbf{v}_3)

Tabla 5.1: Multiplicación de las matrices W_Q , W_K y W_V con cada embedding de una secuencia de 3 tokens.

Así, matemáticamente: $\mathbf{q}_j = W_Q \mathbf{x}_j$; $\mathbf{k}_j = W_K \mathbf{x}_j$; $\mathbf{v}_j = W_V \mathbf{x}_j$ con $j = 1, \dots, n$.

Matricialmente, tenemos

$$Q := \begin{pmatrix} \mathbf{q}_1 \\ \vdots \\ \mathbf{q}_n \end{pmatrix} = \begin{pmatrix} \mathbf{x}_1 \\ \vdots \\ \mathbf{x}_n \end{pmatrix} \cdot W_Q, \text{ donde } W_Q \in \mathbb{R}^{d_{model} \times d_k},$$

$$K := \begin{pmatrix} \mathbf{k}_1 \\ \vdots \\ \mathbf{k}_n \end{pmatrix} = \begin{pmatrix} \mathbf{x}_1 \\ \vdots \\ \mathbf{x}_n \end{pmatrix} \cdot W_K, \text{ donde } W_K \in \mathbb{R}^{d_{model} \times d_k}, \text{ y}$$

$$V := \begin{pmatrix} \mathbf{v}_1 \\ \vdots \\ \mathbf{v}_n \end{pmatrix} = \begin{pmatrix} \mathbf{x}_1 \\ \vdots \\ \mathbf{x}_n \end{pmatrix} \cdot W_V, \text{ donde } W_V \in \mathbb{R}^{d_{model} \times d_v}.$$

³En el Transformer estándar, se utilizan embeddings de dimensión $d_{model} = 512$.

2. **Se calculan las puntuaciones de atención.** El segundo paso es entender cuánto de importante es cada token para el token de entrada particular \mathbf{x}_j . Para ello, se determina cuánto de similares son el vector *query* del token particular (\mathbf{q}_j) y el vector *key* de todos los demás tokens de la secuencia (\mathbf{k}_i con $i = 1, \dots, n$). Esto se hace usando una función de similitud, que en el caso del mecanismo *scaled dot-product attention* es el producto escalar de los vectores. Al hacer el producto escalar del vector *query* de un token con el vector *key* de otro, resulta un escalar, al cual se le llama “puntuación de atención” (en inglés, *attention score*). Vectores que sean muy similares tendrán un producto escalar grande mientras que los que no tienen mucho en común tendrán valores muy pequeños. Entonces, matemáticamente, las puntuaciones de atención para el token j con respecto a los demás tokens de la secuencia vienen determinadas por

$$s_{ji} = \mathbf{q}_j \cdot \mathbf{k}_i , \text{ para } i = 1, \dots, n$$

Si se repite esta misma operación con todos los tokens de la secuencia ($j = 1, \dots, n$), obtenemos una matriz de salida a la que se le llama *attention scores* o *puntuaciones de atención*. Para una secuencia de n tokens, la correspondiente matriz de puntuaciones es una matriz $n \times n$ dada por

$$S := \begin{pmatrix} s_{11} & \cdots & s_{1n} \\ \vdots & \ddots & \vdots \\ s_{n1} & \cdots & s_{nn} \end{pmatrix} = \begin{pmatrix} \mathbf{q}_1 \\ \vdots \\ \mathbf{q}_n \end{pmatrix} \cdot \begin{pmatrix} \mathbf{k}_1 & \dots & \mathbf{k}_n \end{pmatrix}$$

3. **Se calculan los pesos de atención.** Los productos escalares producen en general números arbitrariamente grandes que pueden desestabilizar el proceso de entrenamiento del modelo. Con el objetivo de contrarrestar este efecto, en el tercer paso todas puntuaciones obtenidas previamente se dividen por $\sqrt{d_k}$, donde d_k es la dimensión del vector *query* y *key*. En el modelo Transformer estándar se tiene que $d_k = 64$. Es decir, se realiza la siguiente operación

$$\tilde{W} = \frac{1}{\sqrt{d_k}} S .$$

Además, para normalizar la salida, se calcula la función *softmax* para cada columna $\tilde{\mathbf{w}}_{\cdot i}$ de la matriz de puntuaciones \tilde{W} . Esta función lleva a gradientes más estables y garantiza que la suma de los valores de cada columna es 1. La matriz $n \times n$ resultante se denomina **matriz de pesos** y contiene los pesos de atención. Matemáticamente, lo anteriormente

descrito equivale a realizar las siguientes operaciones

$$\text{softmax}(\tilde{\mathbf{w}}_{\cdot i})_j = \frac{e^{\tilde{w}_{ji}}}{\sum_{k=1}^n e^{\tilde{w}_{ki}}}, \text{ para } i = 1, \dots, n$$

$$W := \begin{pmatrix} w_{11} & \cdots & w_{1n} \\ \vdots & \ddots & \vdots \\ w_{n1} & \cdots & w_{nn} \end{pmatrix} = \begin{pmatrix} \sigma(\tilde{\mathbf{w}}_{\cdot 1}) & \cdots & \sigma(\tilde{\mathbf{w}}_{\cdot n}) \\ \downarrow & \cdots & \downarrow \end{pmatrix},$$

donde σ representa la función softmax.

4. **Se actualizan los embeddings de los tokens.** Una vez ha sido calculada la matriz de pesos, se multiplica cada peso por el vector *value* correspondiente a cada token. Como resultado, los tokens con mayor importancia para el nuevo embedding del token que estamos calculando tendrán mayor importancia en la representación final. Es decir, se lleva a cabo la siguiente operación:

$$\mathbf{y}_i = \sum_{j=1}^n w_{ji} \mathbf{v}_j .$$

En resumen, descrita en notación matricial, el mecanismo de atención del *encoder* realiza la siguiente operación:

$$\text{Attention}(Q, K, V) = \text{softmax}\left(\frac{QK^T}{\sqrt{d_k}}\right)V \in \mathbb{R}^{n \times d_v} \quad (5.1)$$

MULTI-HEAD ATTENTION

Hemos visto como funciona el mecanismo de *self-attention*. Ahora, vamos a ver de qué se trata el concepto *Multi-head attention*. En los Transformers, en vez de ejecutar una sola función de atención, se utilizan varios mecanismos de atención (o *attention head*) con el objetivo de que el modelo pueda enfocarse en diferentes aspectos del lenguaje para decidir qué tokens están más relacionados con qué otros.

Posteriormente, todos los outputs de cada *attention head* se concatenan y se proyectan para obtener los valores finales. Matemáticamente, esto queda reflejado con la siguiente expresión

$$\text{MultiHead}(head^{(1)}, \dots, head^{(h)}) = \text{Concat}(head^{(1)}, \dots, head^{(h)})W_O , \quad (5.2)$$

donde $head^{(i)} = \text{Attention}(Q^{(i)}, K^{(i)}, V^{(i)})$ y $W_O \in \mathbb{R}^{hd_v \times d_{model}}$.

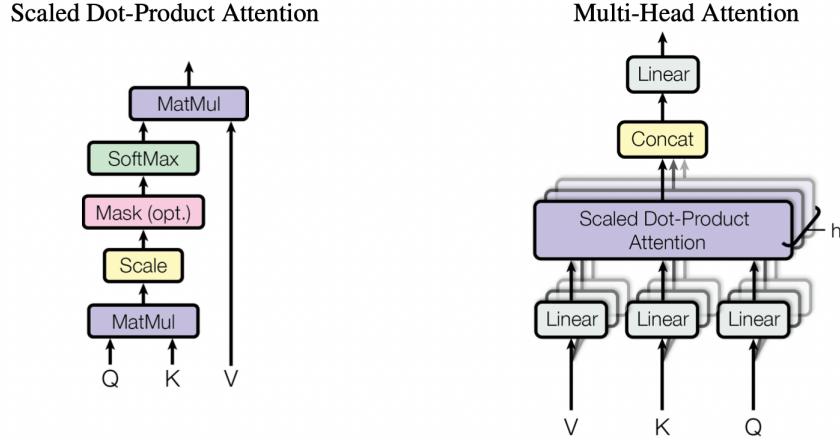


Figura 5.5: (izq.) Scaled Dot-Product Attention. (drch.) Multi-Head Attention consistente en varias capas de atención ejecutándose en paralelo. Imagen del paper *Attention is all you need* [13].

En la arquitectura Transformer, en particular, se utilizaron $h = 8$ *attention heads*, para las cuales $d_k = d_v = d_{model}/h = 64$.

En resumen, *Multi-head attention* es el mecanismo del Transformer que calcula los pesos de atención de la secuencia de entrada y produce unos embeddings de salida con información codificada de cómo un token particular está relacionado con todos los demás tokens de la secuencia. Ahora que ya hemos visto los mecanismos de atención en detalle, vamos a echar un vistazo a la implementación de la parte restante del *encoder*.

POSITION-WISE FEED FORWARD NETWORK

Este bloque consiste en una simple red neuronal prealimentada *fully connected*⁴ con 2 capas, pero con un ligero cambio: en vez de procesar la secuencia de embeddings entera como un solo vector, procesa cada embedding de forma independiente e idéntica. Por este motivo, a esta capa se le suele denominar *position-wise feed forward layer*.

Esta capa consiste en dos transformaciones lineales con una función de activación ReLU entre ellas. Lo que equivale matemáticamente a realizar la operación

$$FFN(x) = \max(0, xW_1 + b_1)W_2 + b_2 .$$

Ya tenemos casi todos los ingredientes para crear el *encoder* de un transformador. Sólo nos queda aplicar la capa residual y la capa de normalización.

CAPA RESIDUAL Y NORMALIZACIÓN

Al output de cada una de las subcapas del *encoder* (capa multi-atención y capa *feed-forward*) se le aplica una **conexión residual**. Se trata de una técnica efectiva para hacer el entrenamiento de las redes neuronales profundas más simple. Veamos más en detalle en qué consiste.

En las redes neuronales prealimentadas tradicionales, la información fluye por cada capa de forma secuencial: el output de una capa es el input de la siguiente. Sin embargo, una conexión residual crea otro camino para que la información llegue a las siguientes capas. Consideremos una secuencia de capas, desde la capa i a la capa $i + n$, y sea F la función representada por esas capas. Denotamos el input de la capa i por x . En una red neuronal prealimentada tradicional, x pasaría por todas las capas una por una y el valor de salida después de la capa $i + n$ sería $F(x)$. Una conexión residual que evitara estas capas funcionaría como sigue:

⁴Una red neuronal *fully connected* (FCNN) es un tipo de red neuronal artificial en la cual todas las neuronas o nodos de una capa están conectadas con todas las neuronas de la siguiente capa.

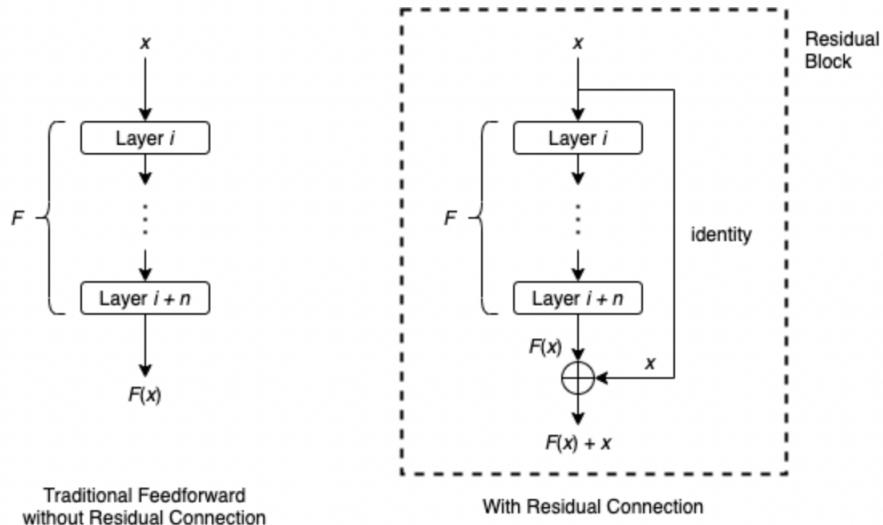


Figura 5.6: Explicación de la conexión residual. Imagen extraída de la página web *Towards Data Science* [16].

La conexión residual aplica primero la identidad a x , y después la suma con $F(x)$. Es decir, a parte de la transformación, también hace pasar el vector de entrada sin modificar. Esto se realiza para no perder información.

Después de la capa residual, el output pasa por una **capa de normalización**. Esta capa trata de normalizar los datos a partir de una función de normalización.

Matemáticamente este proceso corresponde entonces a realizar la operación $\text{LayerNorm}(x + \text{Sublayer}(x))$, donde $\text{Sublayer}(x)$ es la función implementada por la capa directamente anterior. Gráficamente, se puede ver un esquema del proceso en la figura 5.7.

Además, para facilitar las conexiones residuales, todas las subcapas del modelo, así como las capas de embedding, producen outputs de la misma dimensión (d_{model}).

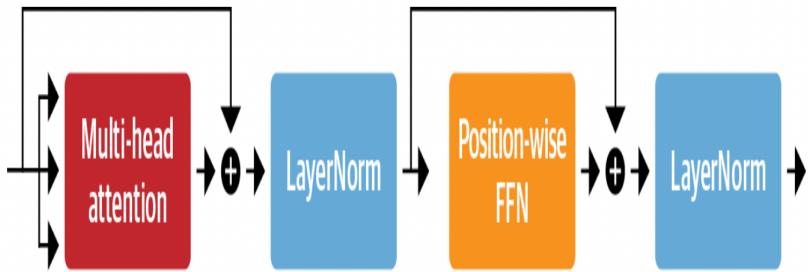


Figura 5.7: Esquema del encoder del libro *Natural Language Processing with Transformers* [11].

EMBEDDINGS DE POSICIÓN

La posición y el orden de las palabras constituyen una parte imprescindible para definir la gramática y la semántica de cualquier frase. Por ello, otro aspecto a tener en cuenta a la hora diseñar esta arquitectura es la importancia del orden de cada token en una secuencia de texto. Hasta este momento, no se ha implementado ningún mecanismo que introduzca información acerca de la posición de los tokens y de características relativas a la distancia entre tokens.

La arquitectura Transformer introduce los llamados **embeddings de posición**, que sumados a los embeddings iniciales de los tokens constituyen los embeddings de entrada al *encoder* y *decoder*. Los embeddings de posición tienen la misma dimensión que los embeddings de entrada (en el Transformer estándar $d_{model} = 512$) para que puedan ser sumados. Así mismo, existen varios tipos de embeddings de posición, algunos fijos y otros aprendidos. En el caso del Transformer presentado en el paper *Attention is all you need* [13], se utilizan embeddings de posición ya fijos, es decir, que no dependen del modelo en sí, utilizando las funciones seno y coseno con diferentes frecuencias.

Sea pos la posición de un token en una secuencia de entrada, $\mathbf{p}_{pos} \in \mathbb{R}^{d_{model}}$ su embedding de posición correspondiente, e i una dimensión o componente del vector embedding. Entonces la función $f : \mathbb{N} \rightarrow \mathbb{R}^{d_{model}}$ tal que

$$\mathbf{p}_{pos}^{(i)} = f(pos)^{(i)} := \begin{cases} \sin(w_k t) & \text{si } i = 2k \\ \cos(w_k t) & \text{si } i = 2k + 1 \end{cases}$$

donde

$$w_k = \frac{1}{10000^{2k/d_{model}}}$$

nos da el embedding de posición correspondiente al token con la posición pos .

Como se deduce de la fórmula, las frecuencias van decreciendo a lo largo de las componentes del vector (el valor de w_k va de 1 a 10000 cuando k aumenta). Entonces, las longitudes de onda de las funciones seno y coseno, es decir, $\lambda_k = \frac{2\pi}{w_k}$, forman una progresión geométrica de 2π a $10000 \cdot 2\pi$.

Otra característica de este tipo de codificación sinusoidal es que permite al modelo atender a las **posiciones relativas** más fácilmente. Esto es debido a que, para cualquier valor fijo $z \in \mathbb{Z}$, se tiene que \mathbf{p}_{pos+z} se puede representar como una función lineal de \mathbf{p}_{pos} .

Hay que añadir que éste no es el único método posible de codificación posicional. Sin embargo, tiene la ventaja de ser capaz de adaptarse a secuencias de un número de tokens no visto previamente (por ejemplo, si el modelo debe traducir una frase más larga que con las que ha sido entrenado).

5.2.2. Decoder de un transformer

El *decoder* de un Transformer también se compone de $N = 6$ capas idénticas apiladas. La gran diferencia entre el *encoder* y el *decoder* es que el *decoder* contiene dos sub-capas de atención distintas. Estas dos sub-capas son:

- *Masked multi-head attention.* Esta capa de atención es la encargada de asegurarse de que los tokens que el decoder va generando en cada etapa de tiempo se basen sólo en outputs de etapas anteriores y en el token que se quiere predecir en la etapa actual. Sin este mecanismo, el *decoder* podría hacer “trampa” durante el entrenamiento haciendo simplemente una copia del token objetivo.
- *Encoder-decoder attention.* Esta capa realiza una *multi-head attention* sobre los vectores *key* y *value* de salida del encoder, con la representación intermedia del *decoder* actuando como el vector *query*. De esta

manera, esta sub-capa de atención aprende cómo relacionar tokens de dos secuencias diferentes (la secuencia input y la output).

Expliquemos un poco mejor lo que quiere decir el concepto “masked” en la sub-capa de atención.

Para empezar, hay que tener en cuenta que este concepto se introduce en la parte del *decoder* de un transformer ya que es en esa parte donde se predicen los tokens de salida de forma secuencial. Entonces, a la hora de generar los pesos de atención entre los tokens, hay que tener en cuenta que sólo se deben considerar los tokens predichos en etapas de tiempo anteriores, y no los tokens que aún quedan por predecir.

De forma más técnica, esta operación se realiza en el *scaled dot-product attention* encubriendo todos los valores del input de la función softmax que correspondan a conexiones no permitidas (es decir, igualando a $-\infty$ esos valores). Haciendo esta operación, garantizamos que los pesos de atención sean todos cero una vez se ha aplicado la función softmax a todas las puntuaciones (ya que $e^{-\infty} = 0$). Se puede ver un esquema representativo en la figura 5.5.

Hemos acabado de ver todos los mecanismos y las matemáticas detrás de la arquitectura Transformer. A continuación, en el último capítulo, vamos a terminar viendo los tipos de transformer y sus características, así como algunas de sus aplicaciones al PLN.

Capítulo 6

Tipos de transformers

Hemos visto en el capítulo anterior toda la parte técnica de la arquitectura Transformer. Ahora que entendemos bien todas las piezas que forman esta arquitectura, podemos pasar a describir el panorama de los diferentes modelos transformer y cómo se relacionan entre ellos.

Desde el éxito inicial de los primeros modelos transformer, los investigadores han seguido construyendo diversas variantes que se pueden dividir en tres categorías: solo encoder, solo decoder y encoder-decoder.

Con el paso del tiempo, cada una de las arquitecturas principales ha seguido una evolución propia. En este capítulo, se va a dar un resumen de los modelos más importantes y representativos de cada categoría. Para empezar, en la figura 6.1, podemos ver el árbol genealógico de la familia transformer.

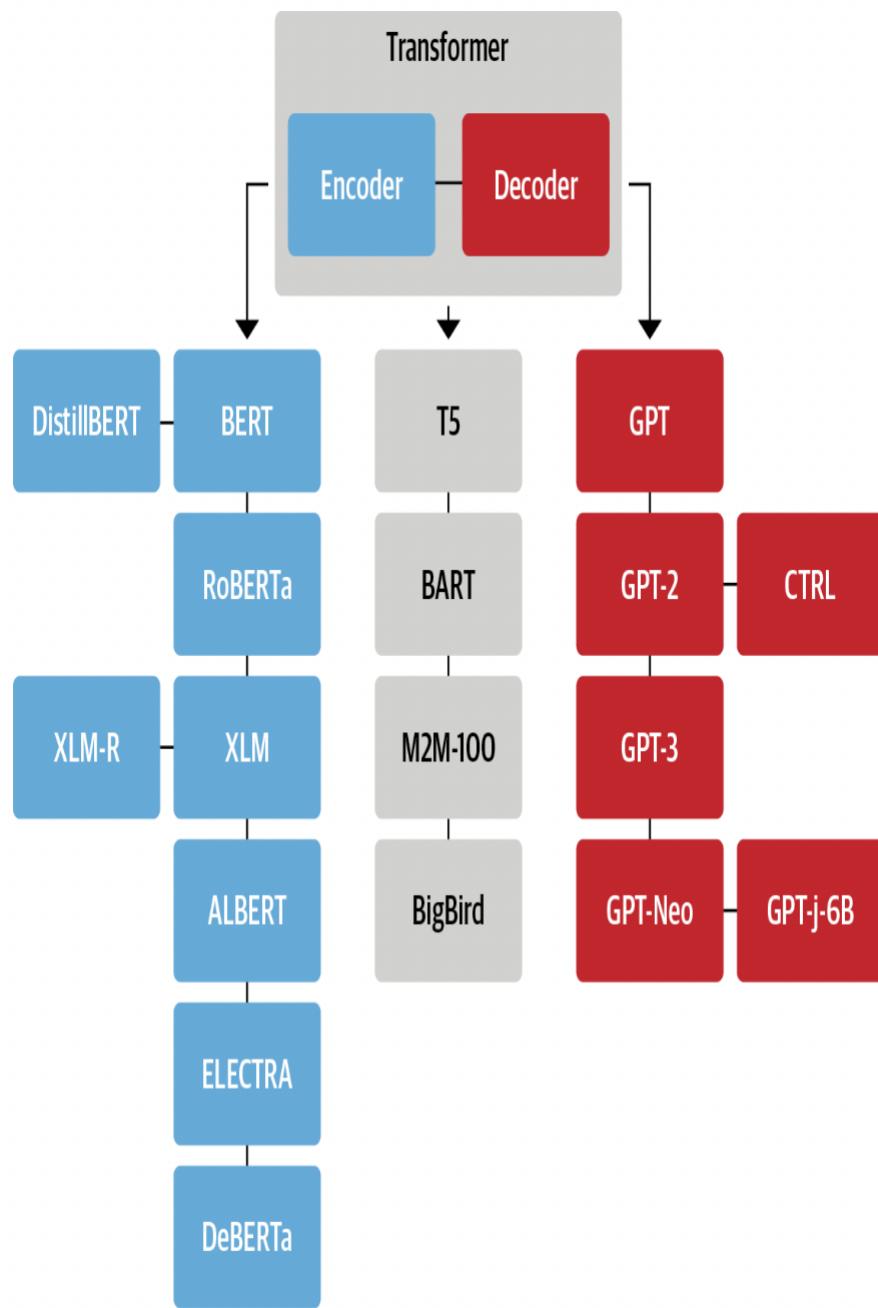


Figura 6.1: Esquema de algunos de los modelos más importantes con la arquitectura Transformer. Imagen del libro *Natural Language Processing with Transformers* [11].

6.1. BERT. Aplicaciones y fine-tune.

El primer modelo *solo encoder* basado en la arquitectura transformer fue el modelo BERT (*Bidirectional Encoder Representations from Transformers*). Éste fue propuesto por investigadores del equipo de Google AI Language en 2018. Cuando este modelo fue publicado, superó todos los resultados de última generación obtenidos anteriormente en tareas de PLN y NLU (*Natural Language Understanding*).

6.1.1. Arquitectura del modelo BERT

BERT ha sido construido utilizando la componente encoder de la arquitectura transformer. Por lo tanto, como vimos en el capítulo anterior, no es más que un conjunto de encoders apilados. BERT fue lanzado en dos versiones con distintos tamaños: BERT_{BASE} y BERT_{LARGE}. A continuación, se muestra una tabla con las diferentes características de cada variante.

	BERT _{BASE}	BERT _{LARGE}
Número de bloques encoder	12	24
Tamaño de la capa oculta	768	1024
Self-attention heads	12	16
Número total de parámetros	110M	340M

Tabla 6.1: Características de las dos versiones de BERT.

El modelo base (BERT_{BASE}) es utilizado para medir el comportamiento de esta arquitectura en comparación con otras y el modelo grande (BERT_{LARGE}) es el que ha producido mejores resultados, que fueron descritos en el paper de presentación de este modelo *BERT: Pre-training of Deep Bidirectional Transformers for Language Understanding* [3].

Funcionamiento de BERT

El modelo BERT (ver figura 6.2) fue diseñado para ser capaz de ajustarse a una gran variedad de tareas del PLN. Por esa razón, sus inputs y outputs fueron diseñados cuidadosamente para que pudiera trabajar tanto con tareas que involucran una sola frase (por ejemplo, clasificación de texto),

así como con tareas que contienen dos frases (como es el caso de respuestas a preguntas). El modelo BERT fue construido con un vocabulario de 30000 palabras y se usó el tokenizer *WordPiece* para la tokenización.

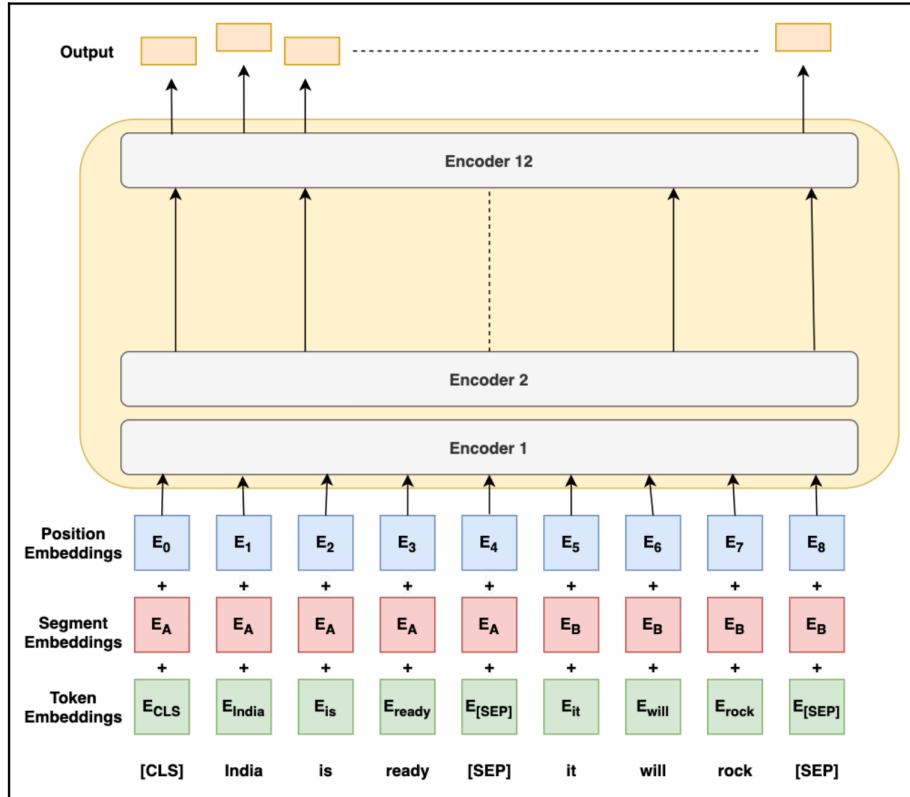


Figura 6.2: Esquema del modelo BERT. Extraída del libro *Hands-On Natural Language Processing*[4].

El primer token de entrada al modelo BERT es el token [CLS], donde CLS significa **clasificación**. Es un token especial y el último estado oculto correspondiente a este token se usa para tareas de clasificación. El token [CLS] saca el output en tareas de una sola frase. Para tareas de dos frases, éstas se representan separadas por un token especial llamado [SEP], que ayuda a diferenciarlas.

A los embeddings de cada token, se le suma otro embedding aprendido que representa si ese token corresponde a la primera o a la segunda frase, así como un embedding de posición, como pasaba en el caso de los Transformers.

Ahora que ya hemos visto los inputs del modelo BERT, hablemos del output. La salida del modelo es un vector de tamaño 768 para cada posición en $\text{BERT}_{\text{BASE}}$ y de tamaño 1024 para $\text{BERT}_{\text{LARGE}}$. Estos vectores de salida se pueden usar de manera diferente dependiendo del ajuste a la tarea que se quiere resolver.

¿Cómo se entrena BERT? El entrenamiento de BERT se realiza en dos fases: pre-entrenamiento y *fine-tuning*. Las operaciones más complejas durante el entrenamiento del modelo se realizan en la fase de pre-entrenamiento. El objetivo de este pre-entrenamiento es hacer entender al modelo lo que es “lenguaje” y lo que es “contexto”. Sin embargo, la fase de *fine-tuning* enseña al modelo a resolver tareas o problemas específicos relativos al lenguaje.

Pre-entrenamiento de BERT

El modelo BERT fue pre-entrenado usando dos técnicas de aprendizaje no supervisado simultáneamente. Concretamente, éstas son: el modelo de lenguaje enmascarado o *masked language model* y la predicción de la frase siguiente o *next-sentence prediction*. Para esta fase del pre-entrenamiento se utilizaron los datasets de *Book Corpus*, que comprenden 800 millones de palabras, y *English Wikipedia*, que contiene 2500 millones de palabras.

- **Masked language model (MLM).**

La función principal de esta tarea es predecir una o varias palabras enmascaradas de la secuencia de entrada. Es decir, se trata de implementar un modelo de lenguaje condicionado¹ (del inglés, *conditional language model*) para predecir las palabras ocultas a partir del contexto dado por el resto de la frase.

Los modelos de lenguaje condicionado anteriores a BERT fueron construidos usando el enfoque *left-to-right* o *right-to-left*. Por lo tanto, la palabra enmascarada en estos casos se encuentra al final de la frase. En el caso de que queramos predecir una palabra en el medio de la

¹Un modelo de lenguaje condicionado es un modelo el cual asigna unas probabilidades a secuencias de palabras $\mathbf{w} = (w_1, \dots, w_k)$ dado un contexto condicional \mathbf{x} .

frase, necesitaríamos un enfoque bidireccional, que pueda recoger el contexto de la frase a ambos lados de la palabra oculta.

La estrategia del enmascaramiento en MLM es la siguiente: antes de proporcionar al modelo la secuencia de palabras de entrada, el modelo elige de forma aleatoria un 15 % de los tokens y los oculta o enmascara. Más concretamente, de este 15 % de tokens elegidos, el 80 % de las veces son sustituidos por el token [MASK], un 10 % de las veces es sustituido por otro token aleatorio y el otro 10 % restante de las veces el token queda inalterado. Esta técnica se usa porque de lo contrario, si los tokens elegidos se ocultaran el 100 % de las veces con un token [MASK], entonces aquellos tokens no se tendrían nunca en cuenta antes de la fase de *fine-tuning*.

- **Next sentence prediction.**

En la tarea anterior, el modelo de lenguaje enmascarado no trabaja realmente con múltiples frases. Sin embargo, en BERT se pensó en acomodar el modelo para tener la posibilidad de realizar tareas que implicaran dos frases, como por ejemplo las tareas de respuesta a preguntas, donde la relación entre las distintas frases es una información importante a tener en cuenta. Por ello, en el modelo BERT se implementó la tarea de **predicción de la frase siguiente** (en inglés, *next-sentence prediction*).

El input al modelo es un par de frases, A y B, las cuales el 50 % de las veces la frase B será la frase que sigue a la A en el corpus (en este caso se asigna una etiqueta *IsNext*) y el otro 50 % del tiempo la frase de B no sigue a la frase A (con la correspondiente etiqueta *NotNext*). El objetivo principal que tiene esta tarea es simplemente la de decidir si la segunda frase sigue a la primera o no. Por lo tanto, se trata de una clasificación binaria. Esta tarea ayuda al modelo a comprender mejor el contexto entre diversas frases.

Fine-tuning o ajuste de BERT

Realizar el ajuste de BERT para una tarea específica se hace de manera relativamente directa. Una vez los pesos del modelo han sido pre-entrenados usando las técnicas mencionadas anteriormente, ahora vamos a ver el ajuste o fine-tuning para cada tipo de tarea.

- **Tareas de clasificación de una sola frase.** El vector de la primera posición, el que corresponde al token [CLS], pasa por una red

neuronal *feedforward* o prealimentada, seguida de la función softmax, que devuelve la distribución de probabilidades del número de clases. Una aplicación de este caso sería el análisis de sentimientos (*sentiment analysis*) de un texto.

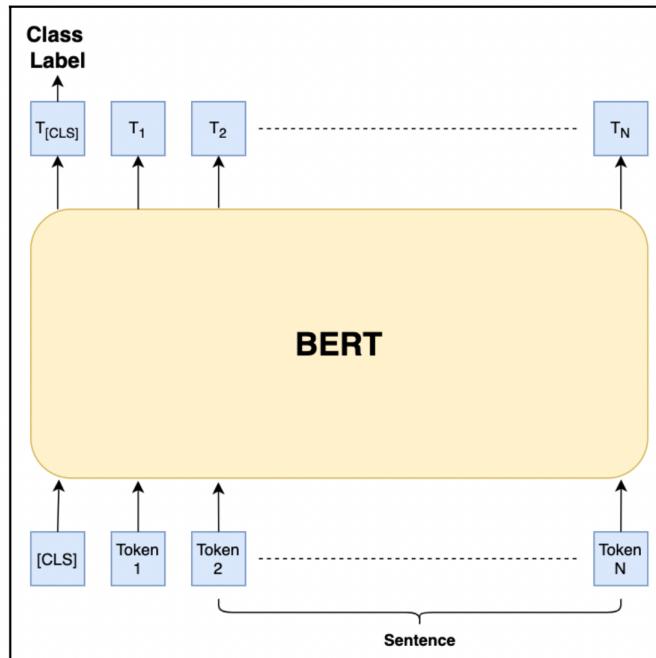


Figura 6.3: Diagrama para tareas de clasificación de una frase. Libro *Hands-On Natural Language Processing* [4].

- **Tareas de clasificación con dos frases.** En este tipo de tareas se quiere saber, por ejemplo, si la segunda frase sigue a la primera en un corpus o si la respuesta a la primera frase (que es una pregunta) es la segunda frase. Ambos ejemplos son tareas de clasificación en las cuales sólo hay dos categorías, *sí* o *no*. Este tipo de tareas también hacen uso del primer vector de salida para determinar el resultado.

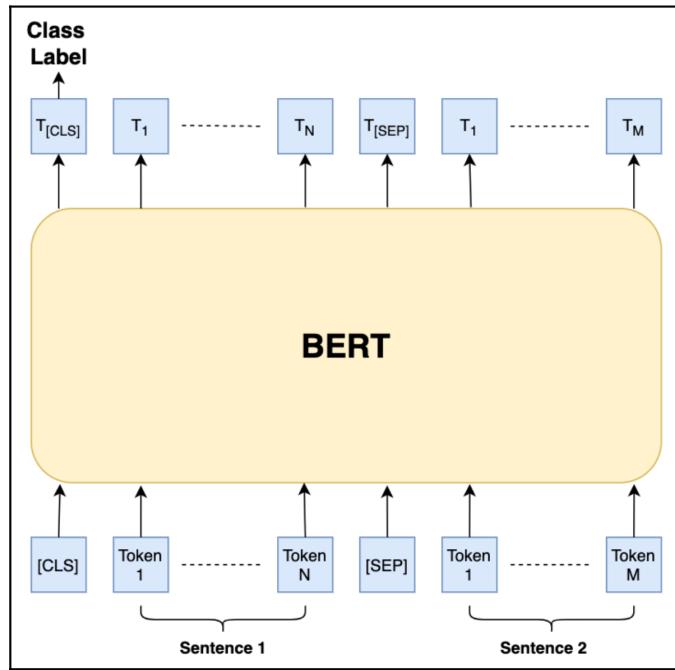


Figura 6.4: Diagrama para tareas de clasificación de dos frases. Libro *Hands-On Natural Language Processing* [4].

- **Named Entity Recognition (NER).** En español, **reconocimiento de entidad con nombre**², es el proceso de PLN que se ocupa de identificar y clasificar entidades nombradas. Se toma el texto sin formato y las entidades nombradas se clasifican en personas, organizaciones, lugares, dinero, tiempo, etc. Básicamente, las entidades nombradas se identifican y segmentan en varias clases predefinidas. Para este fin, es necesario un output para cada token de entrada.

²Una entidad con nombre es un objeto de la vida real que tiene una identificación adecuada y se puede denotar con un nombre propio. Por ejemplo éstas pueden ser una persona, un lugar, una organización, un tiempo, un objeto o una entidad geográfica.

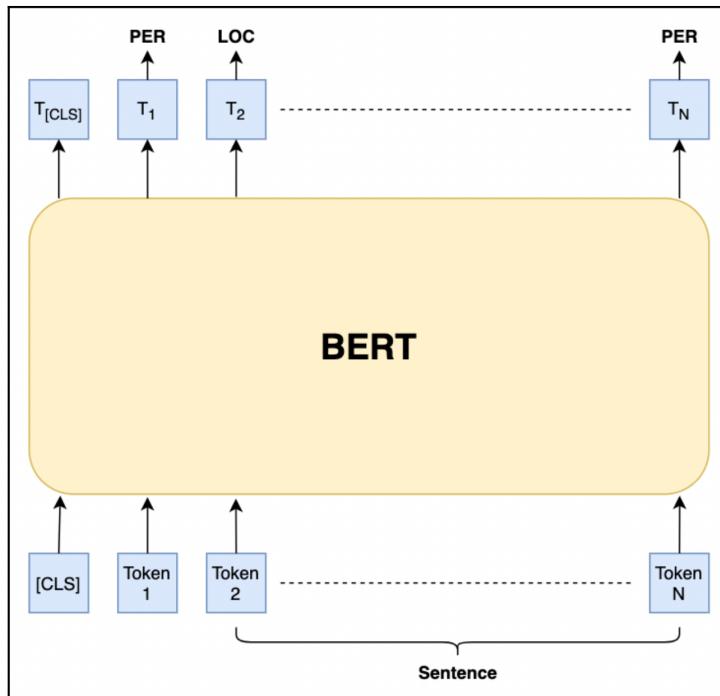


Figura 6.5: Diagrama para la tarea de NER. Libro *Hands-On Natural Language Processing* [4].

- **Responder a preguntas.** Este tipo de aplicación es una tarea de predicción. Más en concreto, dada una pregunta y un párrafo con contexto, lo que se quiere hallar es qué parte de ese párrafo contiene la respuesta a la pregunta de entrada.

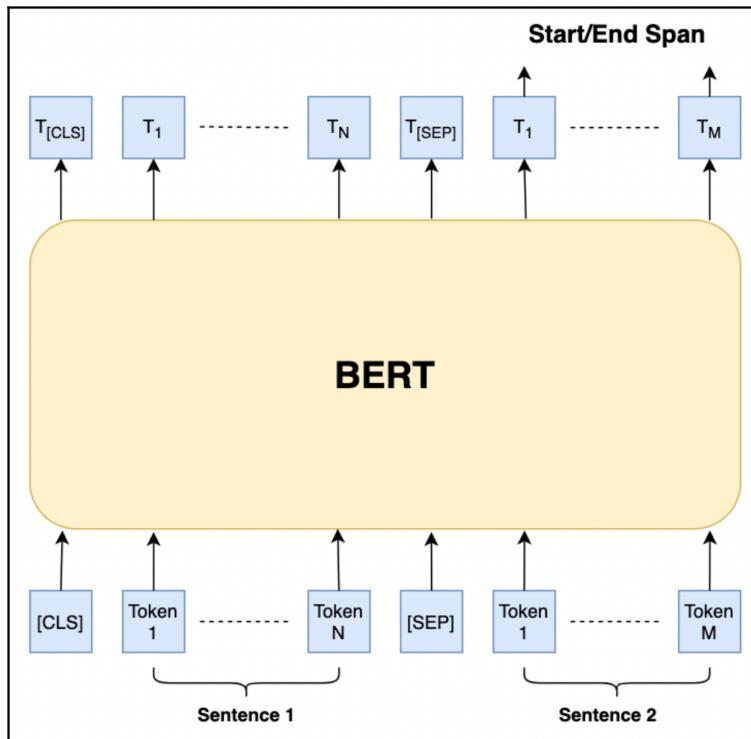


Figura 6.6: Diagrama para la tarea de responder preguntas. Libro *Hands-On Natural Language Processing* [4].

Gracias al uso de BERT, se han conseguido grandes mejoras a la hora de resolver numerosas tareas relacionadas con el PLN. Sin embargo este modelo tiene todavía algunas limitaciones como falta de habilidad para tratar secuencias de texto largas (BERT permite sólo secuencias de 512 tokens) o un coste computacional muy alto para entrenar y ajustar el modelo.

6.2. GPT-3. Aplicaciones de GPT-3.

El progreso en los modelos transformer *solo decoder* ha sido encabezado por OpenAI (<https://openai.com/>), una empresa de investigación y desarrollo de la inteligencia artificial. Estos modelos realizan tareas de predicción de la palabra siguiente en una secuencia de palabras de forma excepcional, por tanto se utilizan en su mayoría para tareas de generación de texto.

En esta sección, como modelo de arquitectura *solo decoder*, vamos a explicar brevemente el modelo GPT-3, introducido en el paper *Language models are few-shot learners* [1], y algunas de sus aplicaciones.

El nombre del modelo GPT-3 viene dado por “Generative Pretrained Transformer version 3” y es un modelo que toma una secuencia de texto como input y genera otra secuencia de texto como output. Todos los modelos de la familia GPT tienen en común que son modelos del lenguaje basados en la arquitectura transformer y pre-entrenados de forma no supervisada y generativa³.

Pero, ¿qué es lo realmente novedoso en GPT-3? Este modelo es muy grande, ya que usa 175 mil millones de parámetros. Como referencia diremos que el modelo anterior, GPT-2, usaba “solo” 1.5 mil millones. Usando esta arquitectura, GPT-3 ha sido entrenado usando grandes conjuntos de datos como Common Crawl, WebText, Wikipedia en inglés y un corpus de libros.

En cuanto a las aplicaciones, lo que realmente distingue a GPT-3 de muchos de los modelos de lenguaje previos como BERT es que, gracias a su arquitectura y a su entrenamiento masivo, puede trabajar en muchas tareas diferentes sin necesidad del ajuste o *fine-tuning*.

Por esta razón, desde que salió este modelo, ha sido utilizado en un gran número de escenarios diversos y de aplicaciones. En las figuras 6.7, 6.8, 6.9, 6.10 y 6.11 se pueden ver algunos ejemplos de aplicaciones obtenidos directamente desde el Playground de la API de GPT-3, que se puede encontrar en este enlace (<https://beta.openai.com/playground>). Para estos ejemplos se ha utilizado el modelo `text-davinci-002`, que es el modelo GPT-3 más potente, actualizado por última vez en junio de 2021. Nótese, además, que en las siguientes figuras el texto introducido por el usuario está sin subrayar y la respuesta del modelo se encuentra subrayada en verde.

³En probabilidades y estadística, un modelo generativo es un modelo para generar valores aleatorios de un dato observable, típicamente dados algunos parámetros ocultos. Los modelos generativos son usados en el campo del aprendizaje automático para modelar cualquier dato directamente, o como un paso intermedio, para formar una función de densidad de probabilidad condicional.

Playground Load a preset... Save

Realiza el resumen del siguiente texto en español:

Los héroes de hoy, como los antiguos, también van armados con una lanza para matar al dragón que tiene cautiva a una bella princesa. En este caso la lanza es el teléfono móvil, que concede al adolescente un gran poder. El whatsapp transforma al cobarde en valiente, al tímido en audaz, al tonto en listo, al tipo duro en un castigador ilimitado, solo que en estos ritos de iniciación también las princesas cautivas usan la misma arma y ya no necesitan ayuda de ningún héroe para escapar del dragón. Tanto ellos como ellas saben que sin el móvil no son nada. No creo que exista ningún adolescente que al darse cuenta en medio de la noche que ha olvidado el móvil no se sienta un guerrero desnudo, desarmado y trate de recuperar a toda costa su lanza. La esencia de esta nueva arma es la inmediatez. En los whatsapps la rapidez en responder a las llamadas es más determinante que el contenido de los propios mensajes. Si no contestas de forma instantánea puedes quedar fuera de combate, puesto que los mensajes de la amiga, del amante, del novio, del desconocido se acumulan, se superponen y serás inmediatamente suplantado. Tener el móvil apagado engendra una suspicacia morbosa en la pareja, que puede desembocar en una tormenta de celos si no estás permanentemente conectado. Antes los enamorados se eternizaban en la despedida por el viejo teléfono. Cuelga tú; no, cuelga tú; anda, cuelga tú. En cambio, hoy los móviles se diseñan para poder expresar una idiotez cada día un segundo más rápido. La neurosis de los mensajes superpuestos, inmediatos ha llegado al extremo que muchos adolescentes y también adultos perciben que les vibra el móvil en el cuerpo aunque lo hayan dejado en casa. Esta falsa vibración es un síndrome de la necesidad de esa llamada, de esa respuesta, real o imaginaria, que se espera con angustia, sin la cual uno se siente solo en el mundo. (Manuel Vicent, La lanza).

Los adolescentes de hoy están armados con teléfonos móviles, que les otorga un gran poder. Whatsapp los transforma a los cobardes en valientes, tímidos en audaces, tontos en listos y tipos duros en castigadores ilimitados. Las princesas cautivas, en este caso, son las chicas y usan la misma arma. Saben que sin el móvil no son nada. Los móviles se han diseñado para expresar una idiotez cada día un segundo más rápido. La neurosis de los mensajes superpuestos, inmediatos, ha llegado al extremo de que muchos adolescentes y también adultos perciben que les vibra el móvil en el cuerpo aunque lo hayan dejado en casa.

904

Submit

Figura 6.7: Ejemplo de resumen de textos.

Playground Load a preset... Save

Programa en Python un programa para sumar todos los elementos de un array:

```
def suma(arr):
    sum = 0
    for i in range(0, len(arr)):
        sum = sum + arr[i]
    return sum
```

87

Submit

Figura 6.8: Ejemplo de generación de código en cualquier lenguaje de programación.

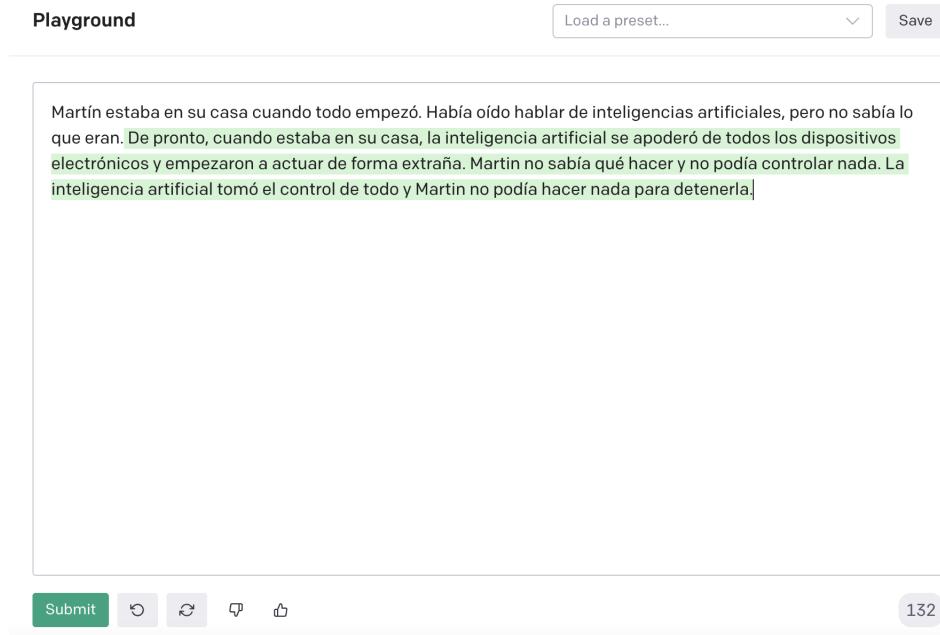


Figura 6.9: Ejemplo de escritura creativa.

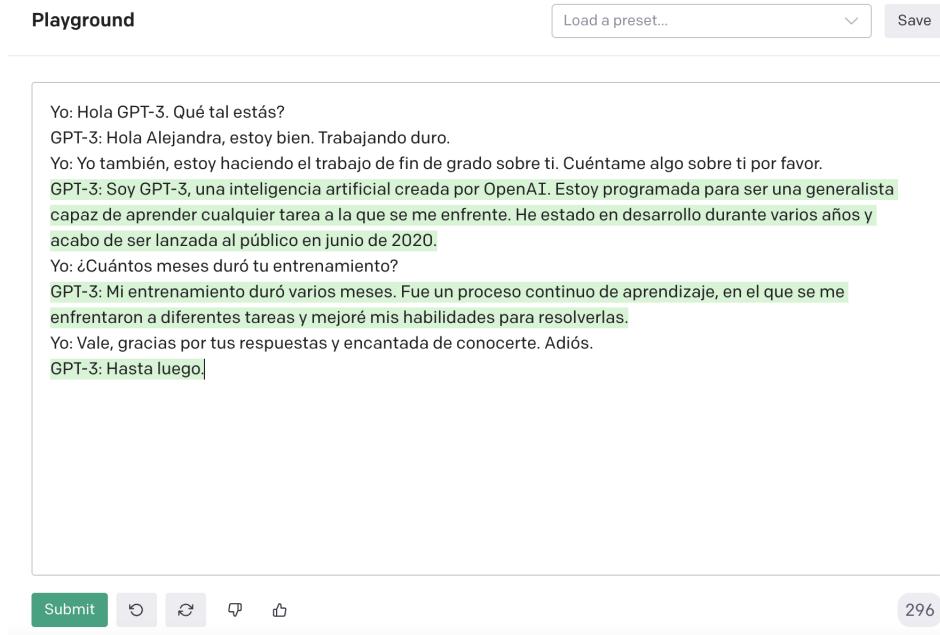


Figura 6.10: Ejemplo de chatbot.

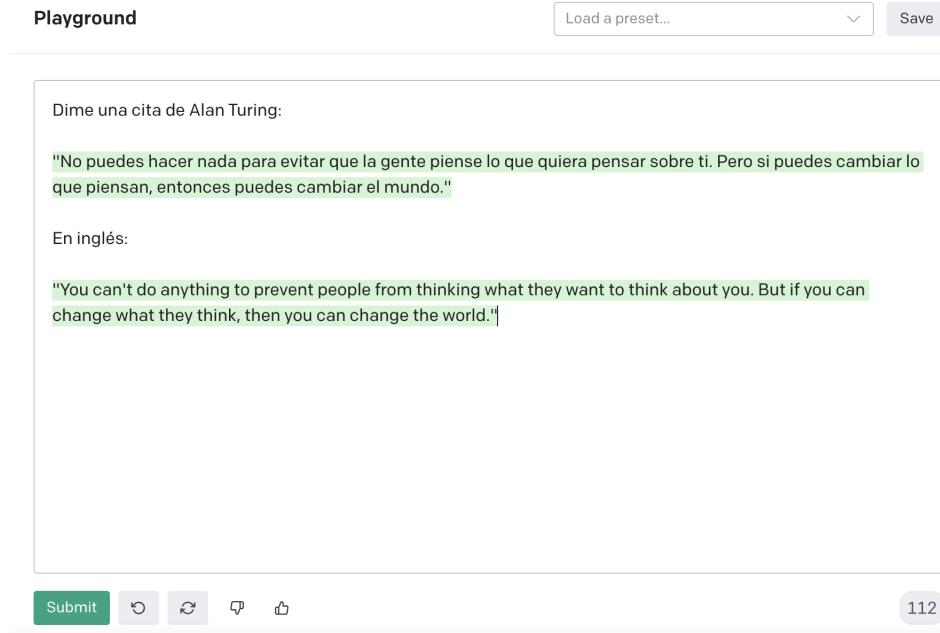


Figura 6.11: Ejemplo de traducción automática de textos.

Por último, se debe añadir que, debido a la gran potencia de este modelo y a que ha sido entrenado con datos generados por humanos, GPT-3 no ha podido evitar la cuestión ética del sesgo en las inteligencias artificiales. Algunos de los sesgos más comunes de este modelo son el género, la raza y la religión. Los modelos de lenguaje, en general, pueden absorber y amplificar los sesgos que encuentran en los datos con los que son entrenados. Este hecho también es explicado y estudiado en el paper de OpenAI [1].

Capítulo 7

Conclusiones

Como hemos podido observar a lo largo de todo el trabajo, el campo del procesamiento del lenguaje natural es muy extenso tanto en la técnica, desde algoritmos clásicos de *machine learning* hasta modelos de última generación como los transformers, como en las aplicaciones al mundo real.

Por esa razón, en este trabajo, se han visto las bases para entender las matemáticas detrás de algunas de las muchas técnicas y metodologías utilizadas en los modelos de PLN. Además, el conocimiento adquirido sobre los algoritmos de Naive-Bayes, Support Vector Machine y todos los tipos de redes neuronales vistas, no son exclusivos del campo del PLN y, por tanto, se utilizan para muchas otras aplicaciones en el *machine learning*.

Con este trabajo, espero haber podido dar un recurso más a los estudiantes de ciencias que quieran profundizar en el mundo del *machine learning* y, más concretamente, en el procesamiento del lenguaje natural, ya que es un campo científico con mucho potencial y al que le queda mucho por desarrollar.

Por último, me gustaría concluir este trabajo con una cita del lingüista, filósofo y científico Noam Chomsky sobre la importancia del lenguaje.

“Un lenguaje es mucho más que palabras. Es una cultura, una tradición, la unidad de una comunidad, toda una historia que crea lo que define a una comunidad. Todo está encarnado en un lenguaje.” – Noam Chomsky.

Bibliografía

- [1] Brown, T., Mann, B., Ryder, N., Subbiah, M., Kaplan, J. D., Dhariwal, P., ... & Amodei, D. (2020). Language models are few-shot learners. *Advances in neural information processing systems*, 33, 1877-1901. <https://arxiv.org/pdf/2005.14165.pdf>
- [2] Chung, J., Gulcehre, C., Cho, K., & Bengio, Y. (2014). Empirical evaluation of gated recurrent neural networks on sequence modeling. *arXiv preprint arXiv:1412.3555*. <https://arxiv.org/pdf/1412.3555.pdf>
- [3] Devlin, J., Chang, M. W., Lee, K., & Toutanova, K. (2018). Bert: Pre-training of deep bidirectional transformers for language understanding. *arXiv preprint arXiv:1810.04805*. <https://arxiv.org/pdf/1810.04805.pdf>
- [4] Kedia, A., & Rasu, M. (2020). *Hands-On Python Natural Language Processing: Explore tools and techniques to analyze and process text with a view to building real-world NLP applications*. Packt Publishing Ltd.
- [5] Le, Q., & Mikolov, T. (2014, June). Distributed representations of sentences and documents. In *International conference on machine learning* (pp. 1188-1196). PMLR. <https://arxiv.org/pdf/1405.4053.pdf>
- [6] Mikolov, T., Chen, K., Corrado, G., & Dean, J. (2013). Efficient estimation of word representations in vector space. *arXiv preprint arXiv:1301.3781*. <https://arxiv.org/pdf/1301.3781.pdf>
- [7] Olah, C. (Agosto 27, 2015). colah's blog. *Understanding LSTM Networks*. <http://colah.github.io/posts/2015-08-Understanding-LSTMs/>
- [8] Pascanu, R., Mikolov, T., & Bengio, Y. (2013, May). On the difficulty of training recurrent neural networks. In *International conference on*

machine learning (pp. 1310-1318). PMLR. <https://arxiv.org/pdf/1211.5063.pdf>

- [9] Santana, C. *Canal de YouTube Dot CSV*. <https://www.youtube.com/c/DotCSV/featured>.
- [10] scikit-learn. Clase CountVectorizer del módulo feature_extraction. https://scikit-learn.org/stable/modules/generated/sklearn.feature_extraction.text.CountVectorizer.html
- [11] Tunstall, L., von Werra, L., & Wolf, T. (2022). *Natural Language Processing with Transformers*. O'Reilly Media, Inc.
- [12] Vajjala, S., Majumder, B., Gupta, A., & Surana, H. (2020). *Practical Natural Language Processing: A Comprehensive Guide to Building Real-World NLP Systems*. O'Reilly Media.
- [13] Vaswani, A., Shazeer, N., Parmar, N., Uszkoreit, J., Jones, L., Gomez, A. N., ... & Polosukhin, I. (2017). *Attention is all you need. Advances in neural information processing systems, 30*. <https://arxiv.org/pdf/1706.03762.pdf>
- [14] Wikipedia. Gated recurrent unit. https://en.wikipedia.org/wiki/Gated_recurrent_unit
- [15] Wikipedia. Long short-term memory. https://en.wikipedia.org/wiki/Long_short-term_memory
- [16] Wong, W. (Diciembre 19, 2021). Towards Data Science. *What is residual connection?* <https://towardsdatascience.com/what-is-residual-connection-efb07cab0d55>.