Proyecto Fin de Carrera
Ingeniería de Telecomunicación

Uso del aprendizaje máquina para encontrar relaciones semánticas en el lenguaje natural

Autor: Diego Quintana Sánchez

Tutor: Francisco José Simois Tirado

Dpto. Teoría de la Señal y Comunicaciones
Escuela Técnica Superior de Ingeniería
Universidad de Sevilla

Sevilla, 2023

Proyecto Fin de Carrera
Ingeniería de Telecomunicación

Uso del aprendizaje máquina para encontrar relaciones semánticas en el lenguaje natural

Autor:
Diego Quintana Sánchez

Tutor:
Francisco José Simois Tirado
Profesor Contratado Doctor

Dpto. de Teoría de la Señal y Comunicaciones
Escuela Técnica Superior de Ingeniería
Universidad de Sevilla
Sevilla, 2023

Proyecto Fin de Carrera: Uso del aprendizaje máquina para encontrar relaciones semánticas en el lenguaje natural

Autor: Diego Quintana Sánchez

Tutor: Francisco José Simois Tirado

El tribunal nombrado para juzgar el Proyecto arriba indicado, compuesto por los siguientes miembros:

Presidente:

Vocales:

Secretario:

Acuerdan otorgarle la calificación de:

Sevilla, 2023

El Secretario del Tribunal

A mi familia

A mis maestros

AGRADECIMIENTOS

Primero que todo, a mis queridos padres, Avelino y Sonia. Les quiero agradecer su apoyo, tanto emocional como financiero, que ha sido el pilar que me ha permitido llegar hasta el punto en el que me encuentro ahora. Vuestra confianza en mí y vuestro amor incondicional han sido mi mayor motivación y fuente de fortaleza. No puedo agradecerles lo suficiente todo lo que han hecho por mí y principalmente, agradecerles haberme enseñado que las cosas no llueven del cielo, que todo hay que ganárselo con el esfuerzo de uno mismo, a quedarme con lo bueno y a aprender de lo malo. Gracias por todo.

Agradecer también a mi tutor Francisco José Simois por abrirme las puertas de un campo tan pionero y apasionante como son las redes neuronales y el machine learning, así como agradecerle la ayuda y guía a la hora de realizar el trabajo, sobre todo con la redacción de este documento.

También agradecerle su apoyo durante todos estos años a mi pareja Marina, por haber estado siempre a mi lado, por haberme acompañado durante los buenos momentos y sobre todo, por apoyarme en las dificultades. Finalmente agradecerle a mi amigo Martine el haberme acompañado durante todos estos años de carrera y de vida, por esos descansos entre clase de risas bajo el puente de los laboratorios y por esas tardes de estudio intensivo donde pensábamos que no había final, por estar siempre.

Muchísimas gracias por todo.

Diego Quintana Sánchez

Sevilla, 2023

RESUMEN

Desde hace algunos años, se ha podido observar una tendencia en alza de hacer uso de modelos de *IA* y *machine learning* para la resolución de numerosas tareas en campos tan complejos y difusos como podría ser el procesamiento de imágenes o lenguaje natural. En el caso del procesamiento del lenguaje natural, el *machine learning,* y concretamente el uso de las redes neuronales ha provocado una absoluta revolución en el campo, obteniendo resultados tan increíbles que en algunos casos llegan a superar la pericia de un ser humano real. En este trabajo se ha buscado implementar un modelo capaz de detectar, para una premisa y una hipótesis dada, la relación semántica que existe entre ellas (relacionada, neutral o contradictoria). El dataset utilizado para entrenar nuestro modelo es ofrecido por Kaggle en su competición "Contradictoy My Dear Watson" y está formado por numerosos ejemplos de hipótesis y premisas en 15 lenguajes diferentes, aunque nosotros nos centramos en el inglés. Durante el trabajo hemos desarrollado de cero un modelo imitando la arquitectura y el proceso de entrenamiento de BERT. El rendimiento de nuestro modelo solo alcanzo el 46% de exactitud, por ello se hizo uso de un modelo ya pre-entrenado mucho más potente, RoBERTa, para poder terminar de comprender la increíble capacidad que poseen este tipo de redes neuronales, consiguiendo un 84% de exactitud.

ABSTRACT

For several years, it has been observed an increase in the usage of AI and machine learning models to resolve many tasks in fields such as complex and diffuse as image processing or natural language processing. Referring to natural language processing, machine learning and more specifically neural networks has achieved an absolute revolution in the field, in some cases even surpassing the skills of a human being. The objective of this work is to implement a neural network model able to identify, between one premise and one hypothesis, the semantical relation they have (related, neutral and contradictory). The dataset used in this work is offered by Kaggle competition "Contradictory My Dear Watson" and it is made up of numerous examples of hypothesis and premises in 15 different languages, however we will only consider the English ones. During this work we have developed from scratch our own model imitating the architecture and training process of BERT. Our own model only achieved an accuracy of 46%, is that why a much more powerful already pre-trained model, RoBERTa, has been used to eventually comprehend the incredible capacity of those types on neural networks, reaching an 84% of accuracy.

ÍNDICE

ÍNDICE DE TABLAS

ÍNDICE DE FIGURAS

ÍNDICE DE ECUACIONES

ÍNDICE DE CÓDIGOS

1 INTRODUCCIÓN

1.1 Estructura de la Memoria

El documento va a estar dividido en los siguientes apartados:

- **Apartado 1. Introducción**: se corresponde con el actual apartado, se mostrará un resumen del contenido de cada apartado y se dará una breve descripción de la motivación y el objetivo de este trabajo.
- **Apartado 2. Inteligencia Artificial**: el objetivo es proporcionar un contexto acerca de la *IA* y el aprendizaje máquina, así como una breve introducción a las redes neuronales que nos permitan comprender como funcionan las mismas.
- **Apartado 3. Procesamiento del Lenguaje Natural**: se ofrecerá una breve descripción del campo del *NLP* y se introducirán las topologías de redes neuronales, así como las diferentes técnicas que se utilizan en esta disciplina. Adicionalmente se hará un breve repaso de algunos de los modelos de redes neuronales del estado del arte en *NLP*.
- **Apartado 4. Método Propuesto**: se propondrá un entorno de trabajo y una arquitectura de red neuronal que sea capaz de extraer relaciones semánticas del lenguaje y posteriormente hacer uso de dichas relaciones para clasificar dos pares de frases.
- **Apartado 5. Pre-entrenamiento y Fine-tuning**: se relatará como ha sido todo el proceso de pre-entrenamiento de nuestro modelo y se hará un análisis de los resultados de este. Adicionalmente se realizará un traspaso de conocimientos para el *fine-tuning* de nuestro modelo en la tarea propuesta por el desafío "Contradictory My Dear Watson" de Kaggle y se analizarán los resultados en dicha tarea. Finalmente realizaremos este mismo *fine-tuning* pero con el modelo pre-entrenado *RoBERTa* para comparar los resultados.
- **Apartado 6. Conclusiones y Futuros Pasos**: se extraerán conclusiones de todo el trabajo realizado y se propondrán futuras metas que conseguir y los posibles pasos que se necesitarán para conseguirlas.

1.2 Motivación

El humano es un ser social, lo que quiere decir que tenemos una necesidad inherente de interactuar y relacionarnos con otros individuos, además de depender unos de otros para satisfacer nuestras necesidades básicas y alcanzar metas comunes. En esta interrelación entre individuos tiene un papel fundamental la comunicación y es donde entra el juego el concepto de lenguaje natural. El lenguaje natural se refiere al sistema de comunicación utilizado por los seres humanos para expresar y transmitir información de manera verbal. Es el medio principal a través del cual nos comunicamos en nuestro día a día, tanto de forma oral como escrita.

Además, el lenguaje natural es la base de muchas tecnologías de procesamiento del lenguaje natural (*NLP*, por sus siglas en inglés), que buscan desarrollar sistemas informáticos capaces de comprender, interpretar y generar lenguaje humano. Esto tiene un gran número de aplicaciones, como la creación de asistentes virtuales, sistemas de traducción automática, análisis de sentimientos, procesamiento de texto y muchas otras.

Sin embargo, el *NLP* es un campo de investigación extremadamente extenso y complejo, que se puede incluso subdividir en otros subcampos, como podría ser el análisis, la traducción e incluso la generación de textos. Dado la increíble complejidad que abarca la comprensión del lenguaje natural para una máquina, surgen diversas ramas que aplicaran diferentes algoritmos y métodos para lograr estos objetivos e incluso cooperarán entre ellas, como podría ser la estadística y el modelado probabilístico, la ingeniería de características o el *machine learning*.

Teniendo esto en cuenta, es interesante profundizar en la forma que usan las personas para comunicarse con las

máquinas y facilitar esta interacción desarrollando métodos que faciliten que las máquinas comprendan la manera predilecta del ser humano para comunicarse, el lenguaje natural.

1.3 Objetivo

El objetivo será desarrollar un sistema capaz de encontrar las relaciones semánticas del lenguaje natural entre dos frases. Para ello se va a profundizar en una de las disciplinas que más relevancia tiene hoy día, la Inteligencia Artificial y el *Machine Learning*, y las aplicaciones que tiene dicha tecnología en el procesamiento del lenguaje natural.

Finalmente se desarrollará e implementará un modelo capaz de lograr dicho objetivo. Dado que este es nuestro primer acercamiento al aprendizaje máquina y las redes neuronales, que dentro de este campo (el *NLP*), es una de las disciplinas más complejas y que disponemos de recursos computacionales bastante limitados, no vamos a buscar la excelencia en los resultados de nuestro modelo, sino más bien que sirva como un acercamiento a ambas disciplinas y un punto de partida para continuar profundizando en un campo de investigación tan complejo e interesante como incipiente.

2 INTELIGENCIA ARTIFICIAL

I propose to consider the question, 'Can machines think?'

- Alan Turing-

2.1 Inteligencia Artificial

La inteligencia artificial o *IA* hace referencia a la capacidad de las máquinas u ordenadores de llevar a cabo tareas que requieren de inteligencia a nivel humano para realizarse. Este término es comúnmente utilizado para referirnos a sistemas capaces de resolver problemas tales como aprender a base de ejemplos, razonar, entender el lenguaje humano o incluso tomar decisiones.

El concepto fue introducido por primera vez en 1950 de mano de los visionarios en ciencia de la computación McCarthy et al. [1] (1955), al proponer la idea de construir máquinas capaces de realizar tareas complejas que solo un ser humano podía llevar a cabo. En un comienzo se pensaba que sería posible lograr este objetivo aplicando reglas explícitas codificadas a mano en los sistemas, lo que se conoce como *IA* simbólicas, y alcanzó su pico de popularidad durante los años 80. Sin embargo, las *IA* simbólicas, aunque eficaces en la resolución de problemas lógicos, fallan de manera inexorable a la hora de resolver problemas más complejos y difusos, como la clasificación de textos o imágenes. Aunque más tarde entraría en juego un nuevo paradigma que sustituyó a las *IA* simbólicas, el aprendizaje máquina. (Russell et al. [2] (2010)).

2.2 Aprendizaje Maquina

En la década de 1950 el pionero de la *IA* Alan Turing publicó su manuscrito, (Turing [3] (1955)), donde introdujo el conocido Test de Turing, así como diversos conceptos clave que ayudarían al desarrollo del campo de la *IA*. En este documento, Turing analizaba la posibilidad de que las máquinas fuesen capaces de desarrollar originalidad, es decir, si serían capaces de ir más allá de simplemente realizar tareas que nosotros le hemos enseñado a cómo llevar a cabo y ser capaces de aprender a realizar estas tareas por sí mismos, sin la necesidad de un conjunto de reglas explícitas que le digan qué hacer en cada momento.

De esta idea nace el aprendizaje máquina, un paradigma que rompe con la computación clásica, en la que los humanos proporcionamos a las máquinas información para procesar, así como las reglas que debe aplicar a la hora de procesar dicha información para obtener el resultado deseado. Con el aprendizaje máquina, las personas proporcionan información a procesar y el resultado deseado, y es la propia máquina la que se encarga de determinar las reglas necesarias para tratar la información y conseguir dichos resultados. Un sistema de aprendizaje máquina, más que programado (como las *IA* simbólicas), es entrenado con un gran número de ejemplos relevantes para la tarea objetivo y es el propio sistema el que se encarga de encontrar una estructura estadística apta para la resolución de tal tarea haciendo uso de diversos algoritmos. (Goodfellow et al. [4] (2016)).

En la Figura 2-1 podemos visualizar una comparativa.

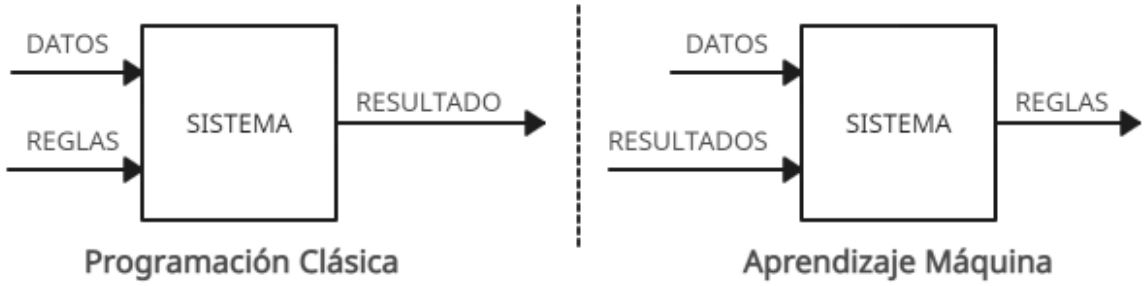

Figura 2-1. Comparativa Programación Clásica vs Machine Learning

2.3 Redes Neuronales

Las redes neuronales se modelaron imitando la estructura y funcionamiento del cerebro humano. Consisten en un número determinado de nodos interconectados entre sí que reciben el nombre de neurona artificial, (Rosenblatt [5] (1958)), y que son utilizados para el reconocimiento de patrones y la toma de decisiones basándose en los datos que reciben. Entre las interconexiones de estas neuronas artificiales se almacenan valores paramétricos que determinarán el resultado de las operaciones realizadas sobre los datos recibidos, podría definirse como el estado de la neurona, estos valores reciben el nombre de parámetros o pesos y podría considerarse como el conocimiento de la red neuronal. (Bishop [6] (1995)).

Aunque la idea surge también a principio de los 50, esta tardó bastante más en desarrollarse. La principal barrera estaba en la dificultad de entrenar las redes neuronales más grandes, es decir, optimizar el valor de todos estos pesos para obtener el resultado deseado. Esto cambiaría con la aparición del algoritmo de *backpropagation* (Linnainmaa [7] (1970)), que hace uso de la optimización por descenso de gradiente tomando como referencia una única función de pérdida determinada, que mide la diferencia entre los datos originados y los deseados, y actualiza convenientemente el valor de los pesos de la red neuronal. Además de un increíble aumento del poder computacional y de los datos disponibles para entrenar a las redes neuronales en los últimos años, que han terminado de impulsar las redes neuronales y convertirlas en el algoritmo de aprendizaje máquina predominante hoy día.

2.3.1 Shallow y Deep Learning

Este avance ofrece una gran ventaja frente al resto de disciplinas del *machine learning*. En los anteriores métodos los datos sufren un número reducido de transformaciones, lo que provoca que los datos que reciben deban de tener representaciones cuyo valor informativo sea lo más grande posible, es lo que se conoce como *shallow learning*. Es decir, los parámetros del modelo se optimizan según las características de los ejemplos con los que lo entrenamos, lo que obliga al ser humano a refinar al máximo estos ejemplos. Útil para tareas en las que los datos no necesiten un gran nivel de abstracción, pero ineficiente a la hora de atajar tareas más complejas, dado que es el propio ser humano el que debe optimizar las transformaciones y extraer las características más relevantes de los datos.

Sin embargo, con el algoritmo de *backpropagation* la tarea de *feature engineering*, es decir de optimizar estos datos de entrada, está automatizada ya que es el propio modelo el que se encarga de hacerlo. Con el algoritmo de *backpropagation* es posible apilar un gran número de transformaciones sucesivas y optimizarlas todas simultáneamente (al contrario que en el *shallow learning* que se optimizan de una en una) y que sea el propio modelo el que extraiga las características más óptimas de los datos. En la Figura 2-2 se puede observar la diferencia entre un modelo de *shallow* y otro de *deep learning*. Esto ayuda al modelo a elevar los datos a niveles de abstracción superiores, dando lugar a representaciones más complejas en espacios intermedios conocidos

como *layers*, cada una de estas *layers* aplica una sola transformación sencilla a los datos haciendo cada representación más abstracta y compleja de manera incremental. De tal forma que todas estas *layers* están relacionadas entre sí y se optimizan para suplir las necesidades de las representaciones del resto de *layers*, esto es lo que se conoce como *deep learning*. (Raschka et al. [8] (2019)).

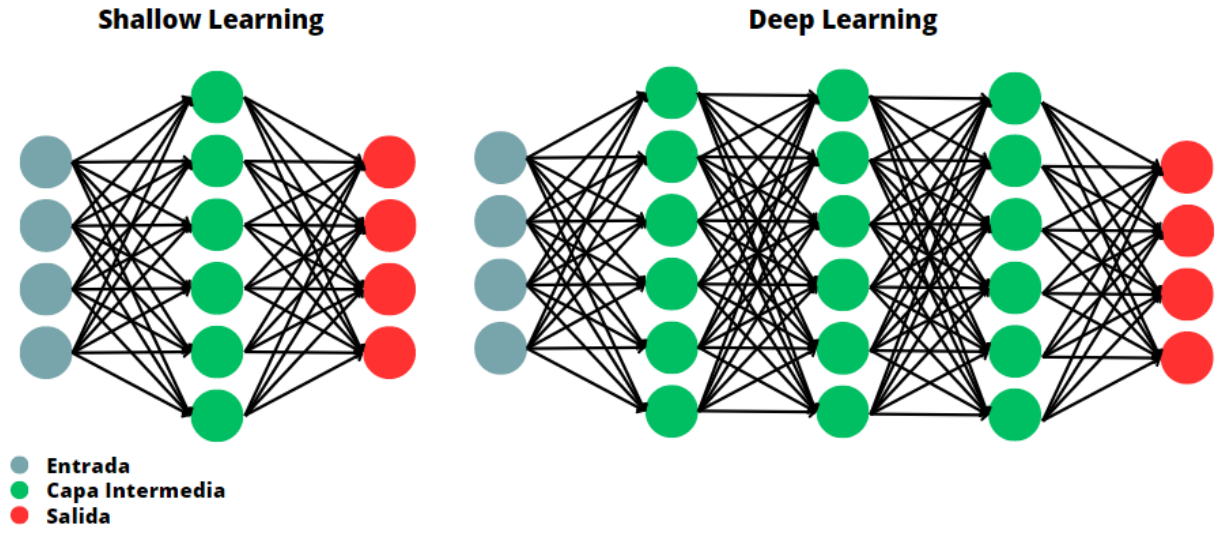

Figura 2-2. Comparativa Shallow vs Deep Learning

2.3.2 Estructura

Una red neuronal está formada por capas, y estas capas a su vez están formadas por neuronas. Cada una de las neuronas de una capa está interconectada directa o indirectamente con las neuronas del resto de capas que forman la red. En la Figura 2-3 tenemos un ejemplo visual.

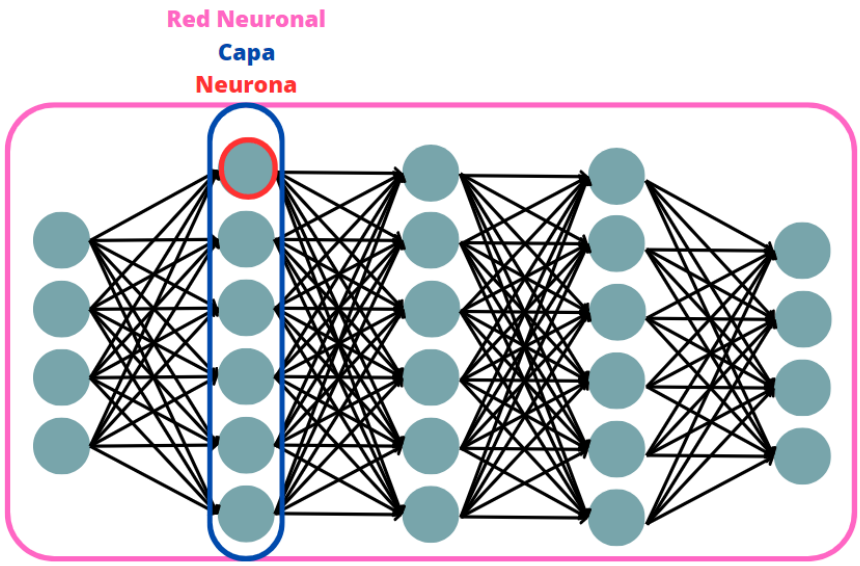

Figura 2-3. Esquema de la arquitectura de una red neuronal

Cada neurona de una capa recibe un vector numérico que contiene información ya sea de las características que

definen los datos o el resultado del procesamiento de dichos datos por parte de otras neuronas de la red. Entre las interconexiones de las neuronas se almacena una serie de valores paramétricos que determinan la relevancia que se le debe de dar a la información recibida, esto se conoce como pesos (por su traducción del inglés *weights*).

La neurona básica hace una suma ponderada de la información recibida con estos pesos y a esta suma ponderada le suma otro valor paramétrico adicional llamado *bias*. El problema es que, realmente dado que la transformación sobre los datos es lineal, todas las neuronas de las capas sucesivas que apilamos estarían siempre trabajando con la misma información ya que solo podrían extraer las relaciones lineales. Es por eso por lo que al resultado anterior se le pasa por lo que se conoce como una función de activación, una función no lineal determinada que permitirá al modelo extraer información más compleja de los datos y que será la encargada de introducir un carácter no lineal al mapear el resultado de las neuronas de una capa. En la Figura 2-4 podemos observar un esquema del funcionamiento. Del tipo de función de activación que utilizamos es de lo que dependerá la transformación realizada sobre la información de entrada a las neuronas de una capa. (Chollet [9] (2018)).

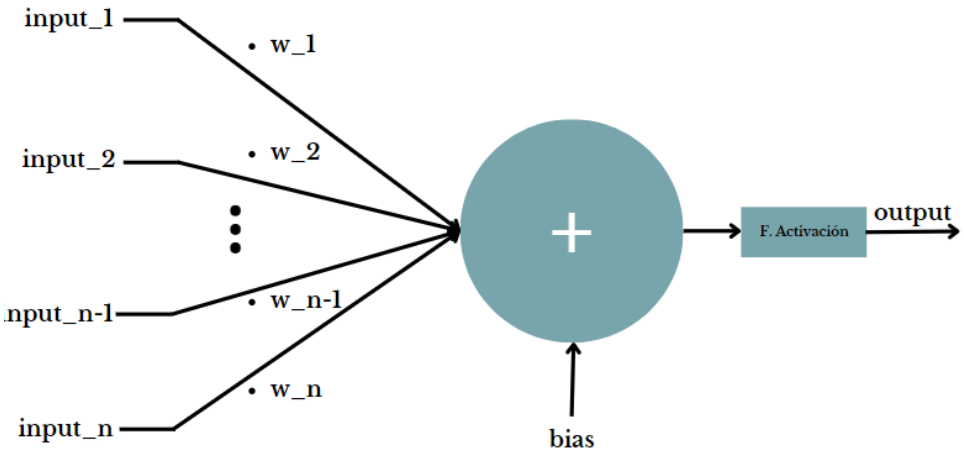

Figura 2-4. Funcionamiento de la neurona artificial

$$output = f_{activacion}\left(\left(\sum_{i=1}^{N} input_i * w_i\right) + b\right)$$

Ecuación 2-1. Salida de una neurona artificial

Siendo $\boldsymbol{N}$ el número de elementos de entrada, $\boldsymbol{w}$ los pesos, $\boldsymbol{b}$ el *bias* y $\boldsymbol{f_{activacion}}$ la función de activación utilizada. A continuación, en la figura 2-5, algunas de las funciones de activación más usadas.

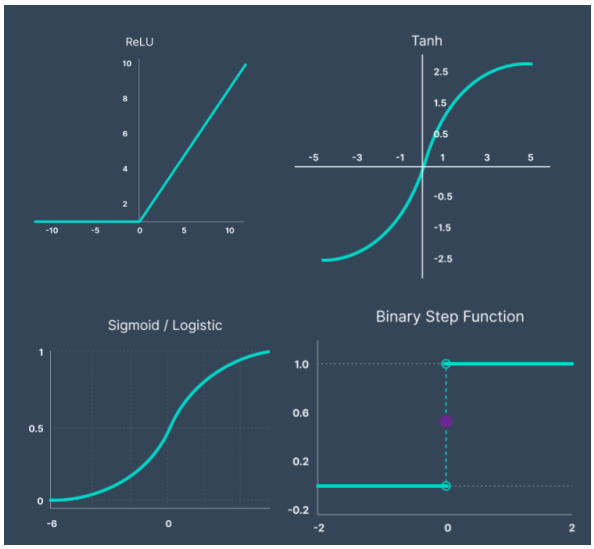

Figura 2-5. Funciones de activación más comunes

Estos pesos junto con el *bias,* son los valores que optimizamos a través del algoritmo de *backpropagation*. Se conocen como los parámetros del modelo, aunque frecuentemente se utilice el término pesos en su lugar, que como hemos mencionado anteriormente es el conocimiento de la red neuronal. De tal forma que una capa es un conjunto de neuronas que transforman los datos de entrada en otros datos de salida con mayor nivel de información y abstracción. Podríamos definir entonces una capa como una función que realiza una operación matricial y una red neuronal como una función que a su vez está formada por funciones anidadas.

$$f_{layer} = f_{activacion}\big((input * W) + B\big)$$

Ecuación 2-2. Salida de una capa de una red neuronal

Siendo $\boldsymbol{input}$ un vector de tamaño [1, nºentradas], $\boldsymbol{W}$ una matriz de tamaño [nºentradas, nºneuronas], $\boldsymbol{B}$ un vector de tamaño [1, nºneuronas], $\boldsymbol{f_{activacion}}$ la función de activación de las neuronas de la capa y $\boldsymbol{f_{layer}}$ un vector de tamaño [1, nºneuronas].

Por lo que una red neuronal con 3 capas intermedias tendría la siguiente función:

$$f_{red_{neuronal}}(\text{input}) = f_{layer\_3}\left(f_{layer\_2}\left(f_{layer\_1}(input)\right)\right)$$

Ecuación 2-3. Función de una red neuronal

2.3.3 Entrenamiento

Entrenar a una red neuronal no es más que optimizar sus parámetros para realizar una tarea determinada. Hemos visto que una red neuronal se puede expresar como una función, en la que los pesos son las variables, y sabemos que es posible hallar los puntos críticos de una función diferenciable hallando los valores en los que su derivada es 0, sin embargo, esta tarea en una función con cientos, miles e incluso millones de variables es una tarea prácticamente imposible, por lo que para encontrar este valor se usa la optimización por descenso de gradiente.

La optimización por descenso de gradiente, (Nocedal et al. [10] (2006)), consiste en modificar el valor de los parámetros de una función en la dirección opuesta a su gradiente en un punto determinado (generalmente se parten de unos valores generados aleatoriamente, en ese caso se conoce como "descenso estocástico de gradiente"), multiplicado por un factor de reducción llamado *learning rate* o *step* (ya que el gradiente de una

función es fiable entorno al punto en el que lo calculamos) que nos dará como resultado un punto en el que el valor de la función es ligeramente menor que el anterior, así sucesivamente hasta lograr encontrar ese punto en el que la función tiene el valor mínimo.

$$W_{t+1} = W_t - \alpha * \nabla C(W_t)$$

Ecuación 2-4. Optimización por descenso de gradiente

Siendo $\boldsymbol{W_t}$ el valor de los pesos del modelo en el instante $\boldsymbol{t}$, $\boldsymbol{\alpha}$ es el *learning rate* y $\boldsymbol{\nabla C(W_t)}$ el gradiente de la función en $\boldsymbol{W_t}$. $\boldsymbol{W_{t+1}}$ es el nuevo valor de los pesos calculados para el que la función es menor.

La función utilizada para optimizar el valor de estos parámetros se conoce como función de pérdida. La función de pérdida es una función que calcula la diferencia entre el resultado deseado y el resultado obtenido por el modelo para unos datos determinados. Dado que el resultado del modelo depende únicamente del valor de sus pesos, ya que tanto los datos como el resultado de ejemplo que se quiere obtener son constantes, es posible mapear esta función a otra que dependa únicamente de los pesos del modelo.

$$C(y, input, W) = loss\left(f_{red_{neuronal}}(input, W), y\right) \equiv C(W)$$

Ecuación 2-5. Mapeado de la función de coste a sus parámetros

Siendo $\boldsymbol{x}$ la salida de la red neuronal (resultado real) e $\boldsymbol{y}$ el resultado esperado (etiqueta), ***loss*** hace referencia a la función de pérdida utilizada para optimizar los parámetros de la red y $\boldsymbol{C}$ la función de coste final.

Como ya hemos dicho, un modelo lo podemos ver como una función anidada compuesta de otras funciones diferenciables, y no solo eso, sino que son funciones diferenciables cuyas derivadas son conocidas. Por lo que gracias a la regla de la cadena podemos encontrar el gradiente de la función de toda la red neuronal. Aquí es donde entra en juego el algoritmo de *backpropagation*.

El algoritmo de *backpropagation*, (Rumelhart et al. [11] (1986)), calcula los gradientes de la red neuronal haciendo uso de la regla de la cadena, calculando las derivadas parciales de las capas inferiores de la red y va ascendiendo a las capas superiores utilizando las derivadas parciales calculadas con anterioridad. Posteriormente se hace uso de este gradiente de la red neuronal para optimizar todos sus parámetros de manera conjunta. A la hora de calcular este gradiente el algoritmo tiene en cuenta la importancia de cada uno de los parámetros sobre la función de pérdida y se actualizarán cada uno de manera correspondiente. Así es como toda la red se optimiza para realizar la tarea para la que ha sido entrenada, es decir, así es como aprende la red neuronal. Existen y se utilizan diversas versiones mejoradas del algoritmo de optimización por descenso de gradiente que reciben el nombre de *optimizers*, es decir, el algoritmo que optimiza los pesos del modelo (Ruder [12] (2016)).

Para una red *FF* de 3 capas el algoritmo de *backpropagation* calcularía las derivadas parciales de la red neuronal haciendo uso de la regla de la cadena de la siguiente forma. (Mazur [13] (2015)).

1º. Derivada parcial de la función de pérdida respecto de la salida de la red. Donde $\boldsymbol{A_x}$ hace referencia a la función de activación en la capa correspondiente.

$$\frac{\partial C}{\partial Output} = \frac{\partial Loss}{\partial A_{output}} * \frac{\partial A_{output}}{\partial Output}$$

Ecuación 2-6. Derivada parcial de la función de coste respecto a la salida de la red.

2º. Derivada parcial de la función de pérdida respecto de la salida de la segunda capa de la red.

$$\frac{\partial C}{\partial Layer_2} = \frac{\partial C}{\partial Output} * \frac{\partial Output}{\partial A_{Layer_2}} * \frac{\partial A_{Layer_2}}{\partial Layer_2}$$

Ecuación 2-7. Derivada parcial de la salida de la red respecto a la salida de la capa 2

3º. Derivada parcial de la función de pérdida respecto de la salida de la primera capa de la red.

$$\frac{\partial C}{\partial Layer_1} = \frac{\partial C}{\partial Layer_2} * \frac{\partial Layer_2}{\partial A_{Layer_1}} * \frac{\partial A_{Layer_1}}{\partial Layer_1}$$

Ecuación 2-8. Derivada parcial de la salida de la red respecto a la salida de la capa 1

4º. Finalmente ya podemos calcular el gradiente de la función de pérdida respecto de $\boldsymbol{W}$, siendo $\boldsymbol{W}$ el conjunto total de los parámetros o pesos que conforman la red.

$$\nabla C(W) = (\frac{\partial C}{\partial Output}, \frac{\partial C}{\partial Layer_2}, \frac{\partial C}{\partial Layer_1})$$

Ecuación 2-9. Gradiente de la función de coste

Para calcular el valor de los parámetros de cada capa haríamos lo siguiente:

$$\frac{\partial C}{\partial W_i} = \frac{\partial C}{\partial Layer_i} * \frac{\partial Layer_i}{\partial W_i} \text{ y } \frac{\partial C}{\partial B_i} = \frac{\partial C}{\partial Layer_i} * \frac{\partial Layer_i}{\partial B_i}$$

Ecuación 2-10. Derivada parcial de la salida de una capa respecto de sus parámetros

Donde $\boldsymbol{W_i}$ y $\boldsymbol{B_i}$ hacen referencia a los pesos y al *bias*.

Sin embargo, por la forma de funcionamiento del algoritmo de *backpropagation* encontramos dos problemas de las redes neuronales (cuanto más grande es la red más tiende a sufrir estos problemas) que es necesario solucionar, el desvanecimiento y la explosión de gradiente. De hecho, son 2 problemas que provocan que el modelo aprenda de manera incorrecta o no aprenda, pero su origen es el contrario.

El desvanecimiento de gradiente ocurre cuando el gradiente que calculamos es tan pequeño que no es capaz de modificar lo suficiente los parámetros del modelo para que aprenda de manera efectiva, o siquiera que aprenda. Esto ocurre como consecuencia de la forma en el que el algoritmo de *backpropagation* calcula los gradientes (con la regla de la cadena empezando por las capas inferiores), por lo tanto, si los gradientes de las capas previas tienen valores pequeños, los valores calculados a partir de estos valores van decreciendo de manera exponencial según subimos por la topología de la red neuronal. Es decir, provoca que las capas más superiores no sean capaces o que aprendan muy lento.

La explosión de gradiente ocurre cuando el algoritmo de *backpropagation* calcula los gradientes del modelo, y estos gradientes alcanzan valores muy elevados, lo que da lugar a que los pesos del modelo se actualicen de manera muy brusca, lo que provoca un aprendizaje inestable. El origen de estos valores tan elevados es el mismo que en el desvanecimiento de gradiente. Si los gradientes calculados previamente tienen valores elevados estos valores se van acumulando hasta dar lugar a valores tan elevados que causen *overflow* e imposibiliten o que inestabilizan el aprendizaje de la red.

Ambos problemas se solucionan o mitigan utilizando funciones de activación apropiadas en el caso del desvanecimiento de gradiente, y de la normalización L1 y L2 o el *gradient clipping* en el caso de la explosión de gradiente. (Géron [14] (2019)).

2.3.4 **Función de Pérdida**

La función de pérdida es la encargada de proporcionar al modelo una referencia de su desempeño al realizar la tarea para la que está siendo entrenado. Esta función es escogida por el ser humano y debe ser precisa a la hora de indicar al modelo el rendimiento de este sobre la tarea realizada. Una mala función de pérdida provocaría que nuestro modelo sea incapaz de aprender a realizar dicha tarea o que el modelo realice esta tarea, pero con un efecto secundario no deseado por falta de especificidad en la misma.

El ser humano es el que debe de definir una función de pérdida lo más precisa posible para enseñar al modelo de forma óptima, por lo que para cada tarea sería necesario una función de pérdida concreta, que sería inservible para enseñar al modelo a realizar cualquier otra tarea de diferente naturaleza. Por ejemplo, para clasificación se utiliza una función de pérdida y para regresión otra distinta, incluso dependiendo de la naturaleza del problema de clasificación puede ser necesario utilizar diferentes funciones de pérdida. (Brownlee [15] (2021))

2.3.5 **Comportamiento**

Durante el proceso de entrenamiento, nuestro modelo o red neuronal lo que hará es adaptarse a los ejemplos optimizando sus parámetros al efecto. Existen múltiples elementos que influyen en el resultado del entrenamiento, como son la función de pérdida elegida, la topología de la red o los ejemplos con los que enseñamos a nuestro modelo.

Mientras entrenamos a nuestro modelo y una vez este ha sido entrenado, será necesario evaluar el resultado de su aprendizaje. Cuando entrenamos a nuestro modelo podemos encontrar varios tipos de comportamiento dependiendo del rendimiento del modelo sobre los datos de entrenamiento y sobre nuevos datos nunca vistos.

Puede darse el caso que nuestro modelo sea incapaz de predecir correctamente los resultados deseados con los datos de entrenamiento, esto es debido a que el modelo es incapaz de encontrar las relaciones de estos datos de entrada con los resultados deseados. Este fenómeno se conoce como *underfitting* y puede deberse a que nuestra red es muy simple para la tarea, la función de pérdida no es adecuada o los datos no son lo suficientemente complejos o informativos respecto a la tarea en cuestión.

También puede ocurrir que se aprenda demasiado de estos datos de entrenamiento, de tal manera que su rendimiento a la hora de trabajar con datos que ya ha visto antes sería excepcional, pero a la hora de procesar nuevos datos el modelo rinde pobremente. Esto ocurre porque el modelo solo puede optimizarse/aprender en base a los datos de ejemplo o training set, puede darse el caso que el modelo encuentre el estado óptimo para tratar estos ejemplos y sea muy preciso con dichos ejemplos, pero incapaz de procesar correctamente nueva información, pues se ha especializado demasiado o en otras palabras no tiene generalización. Este fenómeno se conoce como *overfitting* y puede ocurrir con un número de ejemplos reducidos pero muy caracterizados o por que el modelo es muy complejo para la tarea.

Por último, y este es el objetivo de entrenar a nuestro modelo, puede darse el caso que nuestro modelo sea capaz de aprender perfectamente de los datos de entrenamiento y que también logre realizar la tarea para la que fue entrenado con nuevos datos nunca antes vistos. En la Figura 2-6 tenemos un ejemplo de los distintos comportamientos posibles.

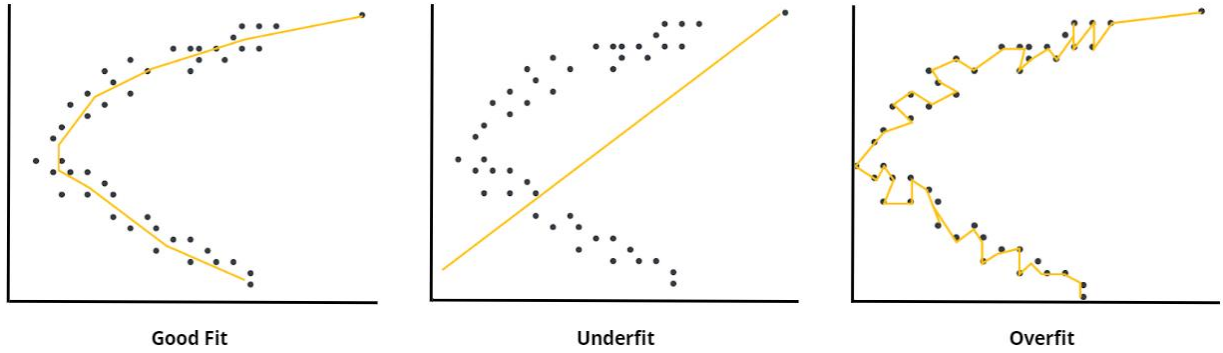

Figura 2-6. Posibles comportamientos de una red neuronal

Evitar el *underfitting* es relativamente sencillo y lo podemos lograr aumentando la complejidad de nuestro modelo o realizando un mejor *feature engineering* sobre los datos de entrada. Para evitar el *overfitting* también podemos reducir la complejidad del modelo, obtener más datos de ejemplo o reducir la dimensión de sus representaciones, pero también surgen técnicas de regularización, como la *L1* y *L2 regularization* (modificar la función de pérdida añadiendo un nuevo parámetro que penaliza la complejidad del modelo, Friedman et al. [16] (2010)) o el *dropout* (establecer a 0 de manera aleatoria algunos valores del vector resultado de un *layer*, Srivastava et al. [17] (2014)).

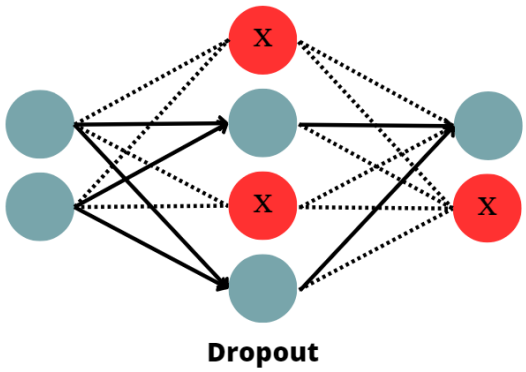

Figura 2-7. Funcionamiento del dropout

$$\boldsymbol{L1\ Reg.} \equiv \ \boldsymbol{loss}(\boldsymbol{x},\boldsymbol{y}) + \ \boldsymbol{\alpha} * \sum |\boldsymbol{W}|$$

Ecuación 2-11. Regularización L1

$$\boldsymbol{L2\ Reg.} \equiv \ \boldsymbol{loss}(\boldsymbol{x},\boldsymbol{y}) + \ \boldsymbol{\alpha} * \sum |\boldsymbol{W}|^{\boldsymbol{2}}$$

Ecuación 2-12. Regularización L2

Siendo $\boldsymbol{x}$ el resultado, $\boldsymbol{y}$ el resultado deseado, ***loss*** la función de pérdida, $\boldsymbol{\alpha}$ un factor de reducción y $\boldsymbol{W}$ todos los pesos de la red neuronal.

2.3.6 Evaluación del Modelo

El modelo optimiza su rendimiento atendiendo a la función de pérdida y el ser humano evalúa el rendimiento de un modelo basándose en 2 elementos fundamentales, un conjunto de datos nunca visto por el modelo, también llamado *testset* y un valor o función que nos permita evaluar su rendimiento sobre estos nuevos datos, también

llamado *metric* o métrica. Es decir, conocer si ha alcanzado un buen nivel de generalización.

Existen un gran número de métricas que podemos usar y que se adaptan a la tarea que queremos que nuestro modelo realice. Al igual que con la función de pérdida, la métrica va a depender de la naturaleza de la tarea, para la regresión por ejemplo nos basta con medir la diferencia entre el valor obtenido y el valor esperado para evaluar el rendimiento, con la tarea de clasificación la cosa es más complicada, y es donde entran en juego el concepto de la matriz de confusión, (Narkhede [18] (2021)), que permite calcular y obtener métricas muy importantes para evaluar la capacidad de nuestro modelo para clasificar ejemplos. En la Figura 2-8 podemos observar la matriz de confusión de un clasificador binario. Donde TP y TN hacen referencia a los verdaderos positivos y verdaderos negativos respectivamente (el modelo ha predicho correctamente la etiqueta) y FP y FN hacen referencia a los falsos positivos y falsos negativos (el modelo ha predicho la etiqueta opuesta).

Matriz Confusión Clasificador Binario

Pred. / Etiq.	0	1
0	TN	FP
1	FN	TP

Figura 2-8. Matriz de confusión clasificador binario

Algunas de las métricas más utilizadas y extraídas de la matriz de confusión son:

- Precisión: mide la proporción de positivos reales acertados entre el total de positivos, útil en tareas en las que un falso positivo es un problema grande, como en los sistemas de detección de intrusión en redes o IDS.

$$\boldsymbol{Precisión} = \frac{TP}{TP + FP}$$

Ecuación 2-13. Precisión en clasificador binario

- Recall: mide la proporción de positivos reales que son correctamente identificados, se utiliza en tareas en las que un falso negativo puede acarrear consecuencias graves como en la detección de enfermedades.

$$\boldsymbol{Recall} = \frac{TP}{TP + FN}$$

Ecuación 2-14. Recall en clasificador binario

- Exactitud: como su propio nombre indica mide la exactitud del modelo, es decir el número de veces que acierta. Se ha de destacar que es una métrica útil en el caso en el que las clases estén balanceadas,

es decir que el número de ejemplo sea similar entre las clases. En el caso de que no sea así puede dar lugar a una falsa sensación de éxito.

$$\boldsymbol{Exactitud} = \frac{TP + TN}{TP + FP + TN + FN}$$

Ecuación 2-15. Exactitud en clasificador binario

- F1: es una métrica que combina la precisión y el recall.

$$\boldsymbol{F1} = \frac{2 * Precision * Recall}{Precision + Recall}$$

Ecuación 2-16. F1

Se debe tener en cuenta que utilizar una sola de estas métricas para medir el rendimiento del modelo en una tarea con cierto nivel de complejidad puede dar lugar a una falsa sensación de éxito en los resultados, por ello generalmente se utilizan varias de estas métricas simultáneamente y en algunos casos incluso combinaciones de estas para obtener unos resultados menos sesgados.

3 PROCESAMIENTO DEL LENGUAJE NATURAL

Natural language processing is about understanding the meaning of human language, not just the form.

- Dan Jurafsky -

Dentro del *deep learning* conviven un gran número de campos de investigación como el análisis de audio o el procesado de imágenes. Uno de los que más está dando que hablar hoy en día debido a los increíbles avances que ha sufrido en los últimos años es el procesamiento del lenguaje natural o *NLP* (por sus siglas en inglés *Natural Language Processing*).

El procesamiento del lenguaje natural abarca el desarrollo de algoritmos y modelos capaces de dar capacidad de entender, interpretar y generar lenguaje humano a las máquinas como si de un verdadero ser humano se tratase. Dentro del *NLP* participan y convergen diferentes disciplinas además del *machine learning* y las redes neuronales, como serían la estadística y el modelado probabilístico, la ingeniería de características o la teoría de la información. El *NLP* tiene un gran número de aplicaciones como podría ser la traducción entre idiomas, clasificación de textos, *chatbots* (máquinas especializadas en conversar) o incluso motores de búsqueda. El *NLP* ha revolucionado la forma en la que nos comunicamos con las máquinas haciendo su uso mucho más cómodo y accesible.

3.1 Herramientas para NLP

Como podemos llegar a imaginar es una tarea extremadamente compleja que necesita hacer uso de herramientas y técnicas adicionales al *deep learning* para conseguir lograr estos objetivos. Dado que las redes neuronales trabajan con vectores numéricos y no con secuencias de palabras, en el procesamiento del lenguaje natural es fundamental tratar los textos para que sea más cómodo procesarlos a las redes neuronales. Para realizar esta tarea se hace uso del *tokenizer*, el elemento encargado de transformar estas secuencias de palabras para que el modelo pueda trabajar con ellas

3.1.1 Tokenizer

El *tokenizer*, (Manning et al. [19] (1999)), es la herramienta encargada de fragmentar el lenguaje natural en unidades más pequeñas que reciben el nombre de *tokens*. Estos *tokens* son generalmente palabras o fragmentos de palabras que se utilizan como entrada para las tareas de *NLP*. La *tokenización* del lenguaje natural es un paso fundamental ya que estandariza y normaliza el texto y hace más fácil al modelo trabajar con este tipo de información.

Al igual que una red neuronal, estos *tokenizers* tienen su propia forma de aprender a partir de textos de ejemplo, de los que se encargan de extraer estos *tokens* atendiendo a un algoritmo concreto o unas reglas predefinidas y que dependen del tipo de *tokenizer* y de la tarea en la que vayamos a entrenar al modelo. Generalmente estos

textos de los que se extraen los *tokens* son los mismos que usaremos para entrenar nuestro modelo de lenguaje.

Una vez entrenamos un *tokenizer* y entrenamos un modelo con los *tokens* extraídos, es necesario utilizar siempre este *tokenizer* para *tokenizar* los textos que usemos para entrenar a nuestro modelo. El *tokenizer* asocia a cada palabra o fragmento de palabra un identificador numérico, atendiendo a diversos criterios como la frecuencia en la que aparece la palabra en los textos con lo que lo entrenamos. Este número va a representar unívocamente una palabra, es lo que conocemos como *token*. Si cambiamos el *tokenizer* con el que *tokenizamos* los textos de entrenamiento a mitad del proceso, puede ocurrir que dos palabras diferentes tengan el mismo *token* o que la misma palabra tenga asignada un *token* diferente, lo que confundiría al modelo. Existen un gran número de diferentes algoritmos de *tokenizado* cada uno con sus propios procesos, aunque la mayoría tienen en común algunas fases. (Huggingface [20] (2021)).

La primera fase es el pre-procesamiento o normalización, el objetivo de esta frase es homogeneizar y normalizar el texto para facilitar su conversión en *tokens*, algunas transformaciones usadas frecuentemente es el *lower casing* (todas las letras en minúsculas), la eliminación de caracteres especiales y la eliminación de acentos. Tenemos un ejemplo en la Figura 3-1.

Figura 3-1. Ejemplo de pre-procesamiento de una frase

La segunda fase es el *tokenizado*, en esta fase es en la que se fragmenta el lenguaje natural en *tokens*, esta fragmentación se puede realizar atendiendo a un gran número de criterios que dependen del tipo de *tokenizado* que queramos y las exigencias de la tarea para la que entrenaremos a nuestro modelo. Algunos criterios podrían ser separar y *tokenizar* el texto por espacios y signos de puntuación y otros por patrones de letras o hasta incluso por caracteres. En la Figura 3-2 podemos observar un ejemplo.

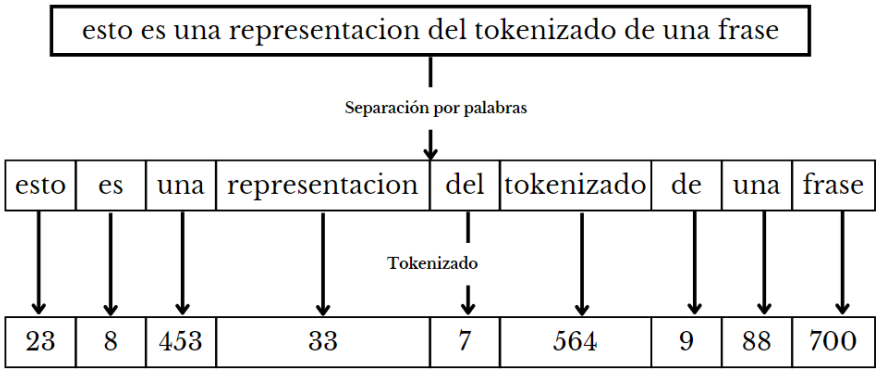

Figura 3-2. Tokenizado de una frase

Finalmente, el postprocesado, una vez el texto ha sido *tokenizado*, aún podemos realizar alguna transformación más para refinar aún más los ejemplos antes de entrenar a nuestro modelo. Algunas herramientas muy utilizadas en esta fase son el *padding* o el truncado, que consiste en añadir relleno (*tokens* especiales sin valor informativo) o eliminar valores de la secuencia de *tokens* para alargarlas o recortarlas, respectivamente, y normalizar la longitud de las mismas. También es común incluir *tokens* especiales que ayuden al modelo a diferenciar cada frase, también llamado *sentence segmentation*, como el *token* de inicio de frase o el de fin de frase. En la Figura

3-3 encontramos un ejemplo.

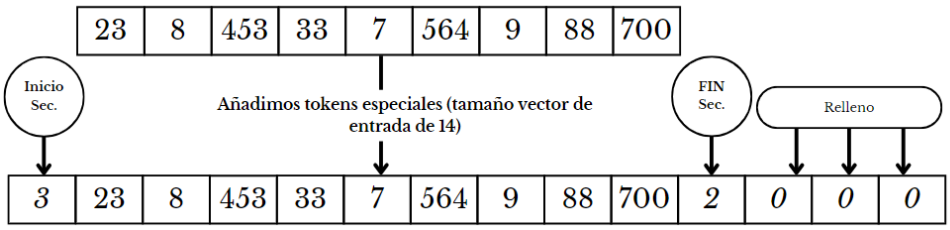

Figura 3-3. Ejemplo de postprocesamiento de una frase

En resumen, el *tokenizado* es fundamental en el procesamiento del lenguaje natural ya que permite a las redes neuronales procesar y extraer información de los datos de ejemplo que le ayuden a identificar patrones y relaciones en el lenguaje natural relevantes para la tarea en la que va a ser entrenado. A pesar de que el *tokenizer* ayuda enormemente a el modelo a procesar el lenguaje natural, aún tiene que enfrentarse a problemas derivados de la necesidad de tratar este tipo de información, algunos de los más comunes serían la ambigüedad (palabras con diferente significado, pero misma morfología), palabras fuera del vocabulario o la contextualización.

3.1.2 Encoding

Esta secuencia de *tokens*, aunque ya son valores numéricos que una red neuronal puede procesar, son muy poco informativos. Es necesario darle una representación más refinada a estos *tokens* que le permita al modelo extraer relaciones complejas, es decir necesitamos caracterizar estos *tokens* para darles un conjunto de atributos que en cierto modo los describan. (Jurafsky et al. [21] (2019)).

Esta caracterización puede hacerse a mano, es decir, es el ser humano el que se encarga de extraer características representativas de las palabras, por ejemplo, si es negativa o positiva, femenina o masculina, singular o plural… de tal forma que el modelo pueda utilizar estas representaciones más complejas para darle un "significado" a los *tokens* que le ayude a procesar mejor la información. Sin embargo, esta tarea de *feature engineering* es bastante compleja, por lo que se prefiere y se intenta hacer uso del *deep learning* para que sea el propio modelo el que convierta estos *tokens* en representaciones mucho más complejas y que extraiga las características óptimas que le permitan describir y dar significado a cada *token*.

Para construir estas representaciones más complejas de los *tokens* los convertimos en vectores, como se observa en la Figura 3-4, de tal manera que cada palabra o *token* es un punto representado en un espacio vectorial. Esta forma de representar las palabras para darles un significado se conoce como *vector semantics*.

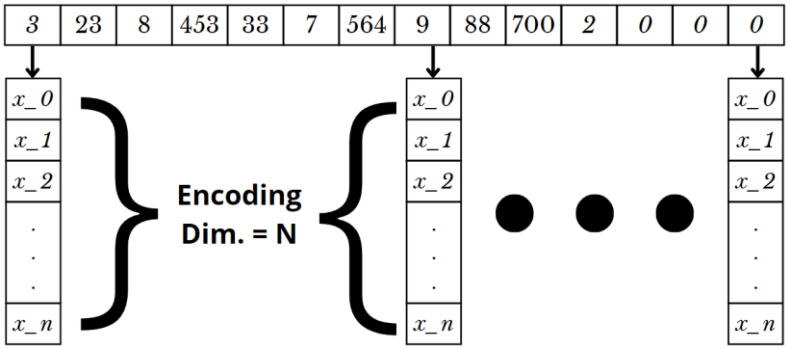

Figura 3-4. Vectorización de los tokens que conforman una frase

Existen 2 tipos de *vector semantics*, los *sparse vectors* o los *dense vectors*. Los *sparse vector* son aquellos cuya dimensión es igual o mayor que el tamaño del vocabulario, como el *one-hot encoding* o las *term-term matrix*.

Mientras que los *dense vectors*, también llamados *embeddings*, tienen una dimensión mucho más reducida (en torno a 50-2000), aunque estas dimensiones no tienen una interpretación directa. Podemos ver una comparativa en la Figura 3-5. Existen diversos algoritmos para calcular *embeddings*, que también los diferenciamos en 2 tipos: estáticos o contextuales. Los *embedding* estáticos utilizan métodos que asignan un vector fijo a cada palabra, como word2vec (Mikolov et al. [22] (2013)), y los *embedding* contextuales asignan un vector diferente a cada palabra dependiendo del contexto, como ocurre con *BERT*, que veremos más adelante.

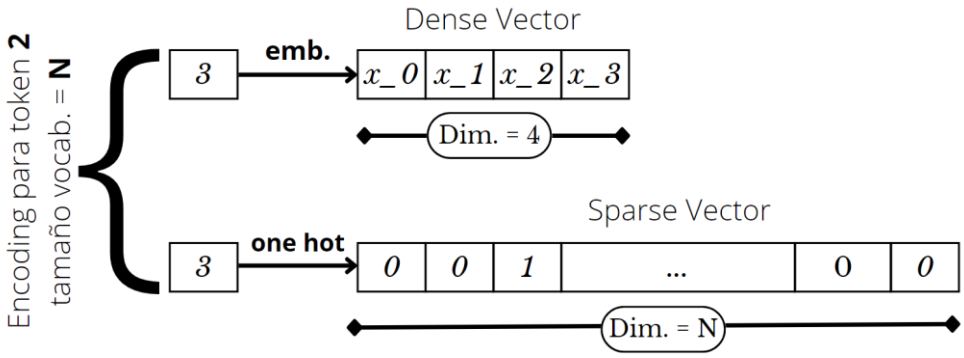

Figura 3-5. Comparativa dense vs sparse vector

3.2 Pre-entreno, Traspaso de Conocimientos y Fine-tuning

La caracterización de un texto es algo extremadamente complejo, que puede variar enormemente dependiendo del contexto o del lenguaje del texto que vamos a tratar y además un paso fundamental en el *NLP* ya que de esta caracterización dependerá la facilidad que tendrá nuestro modelo para hallar relaciones y patrones en los datos de entrada y resultados que le ayuden a aprender.

El pre-entreno, (Dai et al. [23] (2015)), de los modelos de lenguaje natural es una técnica que se basa en entrenar a un modelo sobre una increíblemente grande cantidad de datos no clasificados (mucho más cómodo y baratos de conseguir) para que sea el modelo el encargado de extraer relaciones y características generales del lenguaje de estos textos que le serán útiles posteriormente para realizar diferentes tareas más específicas dentro del *NLP*. Esto se conoce como pre-entreno.

Una vez el modelo ha extraído estas relaciones y características generales durante el proceso de pre-entreno es necesario transmitir esta información, es decir hacer uso de este conocimiento adquirido para aplicarlo a una tarea más precisa como podría ser la clasificación de sentimientos. Para ellos se extraerán los pesos del modelo pre-entrenado y se reutilizarán en un nuevo modelo incorporándolos a su arquitectura. A esto se le conoce como traspaso de conocimientos.

Este nuevo modelo se usará posteriormente para una de estas tareas, optimizando los parámetros para el objetivo y los ejemplos específicos de dicha tarea, pero manteniendo aun así esos conocimientos generales adquiridos durante el pre-entreno. A esta última fase se le conoce como *fine-tuning,* que hace referencia a este segundo entrenamiento con un carácter más específico. Gracias al pre-entreno este *fine-tuning* se puede realizar sobre un *dataset* clasificado mucho más reducido.

Estas técnicas permiten al modelo aprender de grandes cantidades de datos no clasificados y posteriormente aplicar estos conocimientos para una tarea mucho más específica. Está demostrado que estas técnicas mejoran enormemente el rendimiento de los modelos *NLP* y han permitido alcanzar resultados excepcionales en un gran número de tareas y está presente en todos los modelos de estado del arte de la actualidad. (Liu et al. [24] (2015)).

3.3 Redes Neuronales para NLP

Existen múltiples arquitecturas de redes neuronales óptimas para el procesamiento de textos, a continuación, vamos a detallar algunas y cuál es su funcionamiento.

3.3.1 RNN

Las redes neuronales recurrentes (o *RNN* por sus siglas en inglés *Recurrent Neural Networks*) es una topología de red neuronal que trata de tener memoria en el procesamiento de secuencias, de tal manera que almacena la información de las entradas de la secuencia previamente procesadas. Esta característica es muy importante a la hora de procesar texto, pues un texto no es más que una secuencia de palabras en la que el significado de las palabras está sujeto al contexto, es decir a las palabras que la preceden y anteceden. Esto se logra haciendo un bucle interno en el *layer* manteniendo cierta información de los elementos de la secuencia procesados con anterioridad.

Existen diferentes tipos de *RNN*, siendo las *LSTM* (por sus siglas en inglés *Long-Short Term Memory*), (Hochreiter et al. [25] (1997)), de las más utilizadas para *NLP* ya que se caracterizan por tener una memoria mucho más longeva que el resto. Las capas *LSTM* hacen uso de 2 vectores adicionales a la entrada para calcular la salida de cada elemento de la secuencia, cada uno con su propia matriz de pesos. El primer vector es el resultado de procesar el anterior elemento de la secuencia, este valor lo conocemos como el *state*. El segundo es un vector llamado *carrier*, que se calcula haciendo uso del elemento actual y del *state*, utilizando 3 pares de matrices de *weights* diferentes que dan lugar a 3 vectores independientes, que reciben el nombre de *gates* y que usaremos para actualizar el valor del *carrier* para usarlo en el siguiente elemento. Es decir, el *carrier* que usamos para calcular la salida de un determinado elemento de la secuencia ha sido calculado haciendo uso del anterior elemento y el *state* del anterior elemento. A continuación, un esquema grafico de la celda *LSTM* en la Figura 3-6.

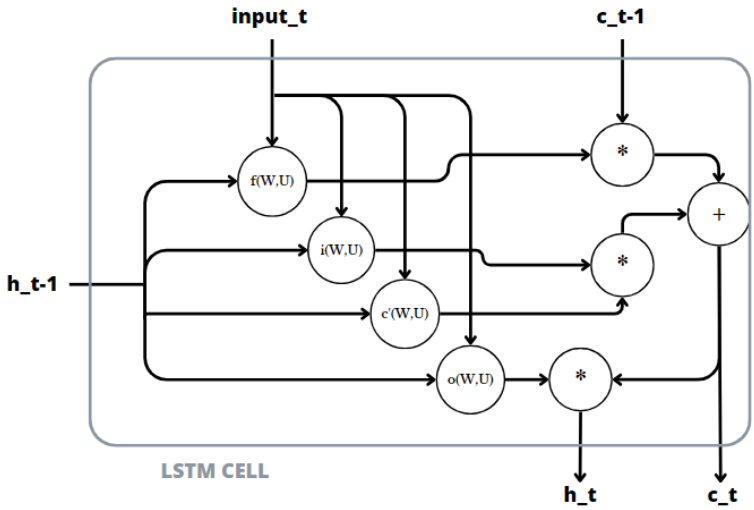

Figura 3-6. Celda LSTM

$$state_t = h_{t-1}$$

$$o_t = f_{activacion}(input_t * W_o + state_t * U_o + B_o)$$

$$i_t = sigmoid(input_{t*} * W_i + state_t * U_i + B_i)$$

$$f_t = sigmoid(input_{t*} * W_f + state_t * U_f + B_f)$$

$$c'_t = sigmoid(input_{t*} * W_{c'} + state_{c'} * U_i + B_{c'})$$

$$c_t = i_t * c'_t + c_{t-1} * f_t$$

$$h_t = o_t * tanh(c_t)$$

Ecuación 3-1. Ecuaciones de celda LSTM

Donde $\boldsymbol{input_t}$ es la entrada en el instante actual y una matriz de tamaño [nºtokens, dimensión embedding], $\boldsymbol{state_t}$ es la salida del anterior elemento de la secuencia, una matriz de tamaño [nºtokens, nºunidades ocultas], $\boldsymbol{W_x}$ y $\boldsymbol{U_x}$ son la matriz de pesos de cada gate, para el elemento de entrada y el state actual, de tamaño [dimensión embedding, nºunidades ocultas], $\boldsymbol{c_t}$ es el valor del carrier calculado para este instante, una matriz de tamaño [nºtokens, nºunidades ocultas] y finalmente $\boldsymbol{h_t}$ es la salida de la red para el instante o elemento de la secuencia número ***t***, una matriz de tamaño [nºtokens, nºunidades ocultas].

Estas operaciones adicionales permiten a la capa tener cierta memoria de los elementos procesados anteriormente y darle relevancia a la hora de calcular el elemento actual y consecuentemente toda la secuencia. Generalmente se le suele dar un significado a cada uno de estos *gates*, la función activación de los vectores gates que hemos llamado o_t, i_t y f_t, es una función *sigmoid*, de tal forma que van a ser vectores cuyos valores van a estar entre 0 y 1, su supuesta función es modular la salida, modular la entrada de información y olvidar información irrelevante, respectivamente. Aunque realmente se le suele dar ese significado a cada una de las operaciones, lo cierto es que es muy complicado asignar uno u otro objetivo a cada operación pues en última instancia todo va a depender del valor de la matriz de pesos y esto lo va a determinar el algoritmo de *backpropagation*. (Saxena [26] (2021)).

3.3.2 **CNN**

Las redes neuronales convolucionales o *CNN* (por sus siglas en inglés *Convolutional Neural Networks*) es una topología de red neuronal que es generalmente utilizada para el procesamiento de imágenes, sin embargo, también encuentra una aplicación en el campo del procesamiento del lenguaje natural, (Kim [27] (2014)). Las *CNN* son especialmente hábiles en la extracción de patrones espaciales en el caso de procesar imágenes o temporales en el caso del procesamiento de texto.

Al igual que para el procesamiento de imágenes, las *CNN* incluyen un *layer* convolucional que será el encargado de extraer características del texto de entrada. El *layer* convolucional cuenta con un filtro de tamaño determinado, que determinará cuántos valores de entrada serán incluidos en cada iteración, para recorrer la secuencia de palabras y aplicar a cada bloque de datos una operación convolucional. Esto permite extraer patrones en el texto que contengan o representen características importantes. Este filtro es también llamado *kernel* y no es más que una matriz de pesos. A diferencia de en el procesamiento de imágenes, en el procesamiento de texto la operación convolucional ocurre en una sola dimensión, así como el desplazamiento del *kernel* a través de la secuencia, el *kernel* recorre la secuencia dando saltos de un tamaño establecido de izquierda a derecha, a este valor se le llama *stride*. A continuación, un ejemplo grafico en la Figura 3-7. En la figura cada elemento ***x*** y ***w*** representa un vector, y ***x*w*** representa la multiplicación elemento a elemento de ambos vectores, siendo el resultado un vector de mismo tamaño.

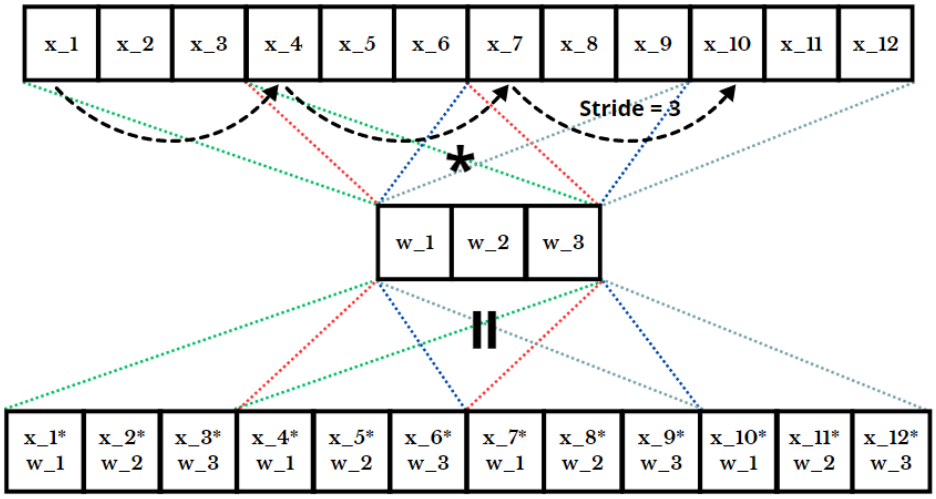

Figura 3-7. Operación de convolución

Después del *layer* convolucional se suele añadir una capa de *pooling* que reduce la dimensionalidad de la salida de la capa convolucional y por lo tanto el tamaño de las características extraídas. Se puede hacer *pooling* atendiendo a diferentes criterios, como el valor máximo o el valor medio en el bloque de datos sobre el que haremos *pool*. En la Figura 3-8 tenemos un ejemplo.

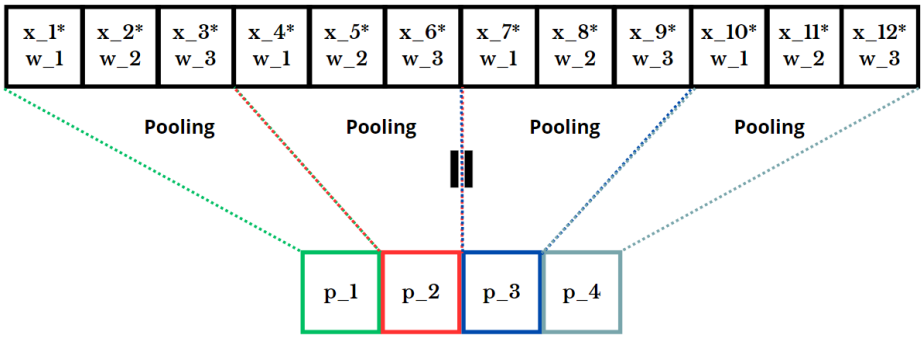

Figura 3-8. Operación de pooling

Aunque no es un tipo de capa especialmente usada en *NLP*, sí que la solemos encontrar en modelos cuyo *tokenizer* actúa a nivel de caracteres. Se suele hacer uso de varios filtros de diferente tamaño para extraer diferentes características de distinto ámbito, como podría ser a nivel de palabra o de conjunto de palabras. Un modelo bastante conocido que hace uso de este tipo de capas es *ELMo* (Peters et al. [28] (2018)).

3.3.3 Transformers

Es una de las topologías de red neuronal más reciente y ha causado un gran revuelo en el mundo del *NLP* ya que ha roto récords posicionándose como la topología de *state of the art* de la actualidad. Vamos a profundizar en su funcionamiento. (Vaswani et al. [29] (2017)).

La arquitectura del *transformer* se divide en 2 bloques, el *encoder* y el *decoder*. El *encoder* es el encargado de extraer las características de la información de entrada y proporcionar una representación abstracta y compleja que contenga el significado de la frase al completo y el *decoder* es el encargado de recibir dicha información y generar nueva información en base a ella. Nosotros vamos a profundizar en la estructura del *encoder*, que es muy similar a la del *decoder* menos ciertos aspectos específicos, pero dado que nuestro trabajo no se va a enfocar en un modelo de lenguaje generativo, vamos a saltarnos esa concreción. En la Figura 3-9 tenemos una imagen

de la estructura del *transformer*. Como podemos observar, tenemos a la izquierda el bloque del *encoder*, que es el que recibe las entradas y se encarga de procesarlas y a la derecha el bloque del *decoder* que es el que genera nueva información a partir de la información procesada por el *encoder* y los *tokens* previos al *token* actual que el mismo ha predicho, estos *tokens* previos atraviesan una capa de *masked multi-head attention* que se encarga de solo tener en cuenta los *tokens* a la izquierda del *token* actual, es decir los *tokens* previos en la secuencia. Posteriormente esta información se utiliza como entrada a una capa de *multi-head attention* así como la información procesada por el *encoder* que se utilizara finalmente para predecir cuál de los *tokens* es el más probable de ocupar la siguiente posición.

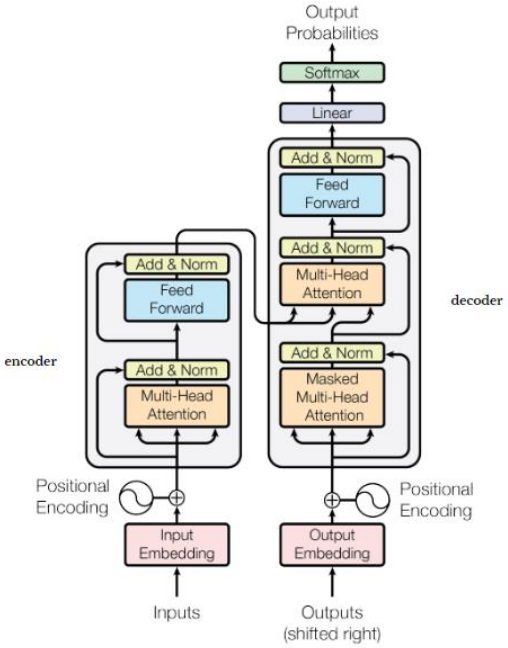

Figura 3-9. Esquema Transformer [29]

Antes de continuar con la arquitectura del *encoder*, vamos a explicar la herramienta fundamental del *transformer*, el a*ttention mechanism*, (Bahdanau et al. [30] (2014)). El a*ttention mechanism* es un mecanismo que permite al modelo diferenciar y valorar la importancia de cada parte de la secuencia de información de entrada según la tarea para la que está siendo entrenado. La idea principal es calcular un conjunto de *attention weights* que le indiquen al modelo cómo de relevante es cada parte de la secuencia y que son optimizados por el modelo durante el proceso de entrenamiento. Gracias a los *attention weights* el modelo es capaz procesar qué partes de la secuencia son más importantes para prestarles más atención a dichas partes y menos a las partes menos relevantes.

Existen múltiples tipos de a*ttention mechanism*, pero generalmente todos incluyen realizar una comparación entre un vector llamado *query*, y uno o varios vectores de atención, también llamado *keys*, para posteriormente utilizar estas relaciones para calcular los *attention weights*. Estos *attention weights* se multiplican con un vector adicional denominado *values*, de tal forma que se destaquen los elementos del vector *values* que se consideren más relevantes. A la capa que realiza esta operación la llamaremos *attention layer*.

$$W_{Attention} = \frac{Q * K}{\sqrt{d_k}}$$

Ecuación 3-2. Cálculo de attention weights

Donde $\boldsymbol{Q}$ es una matriz de tamaño [nºelementos query, d_k], $\boldsymbol{K}$ una matriz de tamaño [nºelementos keys, d_k] y

$\boldsymbol{d_k}$ hace referencia a la dimensión de cada elemento de *query* y *keys*, que tiene el mismo valor. La función de dividir entre la raíz cuadrada de la dimensión de $\boldsymbol{d_k}$ es la de normalizar el resultado.

$$Attention(W_{Attention}, V) = softmax(W_{Attention}) * V$$

Ecuación 3-3. Cálculo de la matriz de atención

Donde $\boldsymbol{W_{attention}}$ es una matriz de tamaño [nºelementos query, nºelementos key] y $\boldsymbol{V}$ una matriz de tamaño [nºelementos value, d_v], siendo d_v la dimensión de cada elemento del vector *value*.

El *transformer* hace uso de un tipo concreto de *attention mechanism*, el *multi-head attention*. Este mecanismo proyecta los 3 vectores, *query*, *key* y *values* en múltiples representaciones lineales de menor tamaño y realiza la función de atención descrita anteriormente de manera independiente para cada proyección, posteriormente las concatena y transforma este vector en otra proyección lineal. El objetivo de esto es aumentar la capacidad de representación ya que cada *attention layer* extrae información diferente. A continuación, un ejemplo de una *attention-head* normal y una *multi-head attention* en la Figura 3-10.

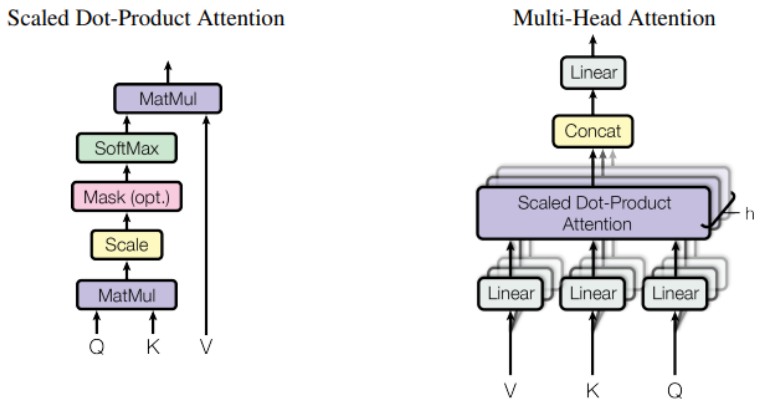

Figura 3-10. Scaled Dot-Product Attention y Multi-Head Attention [29]

Ahora que ya conocemos el mecanismo *multi-head attention* vamos a detallar la estructura del *encoder*.

La información de entrada, además de estar formada por una secuencia de *embeddings* (los *tokens* del texto extraídos por el *tokenizer* y transformados en *dense vectors*), va a incluir un vector de posición o *positional encoding*. El objetivo de este vector es proporcionar una referencia de la posición relativa y absoluta de cada *token* al modelo, ya que la *multi-head attention* al no realizar operaciones ni convolucionales ni recurrentes carece de información del orden de la secuencia. La manera de proporcionar esta información es con la suma de estos vectores de posición a los *embeddings* de los *tokens*. Esto se podría conseguir añadiendo un *embedding* fijo aprendido que dependa de la posición de cada *token*, o como en este caso, haciendo uso de la función coseno y seno para lograr el mismo objetivo.

$$PE(pos, i) = \begin{cases} \sin\left(pos \Big/ 10000^{2i/d}\right), & i\%2 = 0 \\ \cos\left(pos \Big/ 10000^{2i/d}\right), & i\%2 \neq 0 \end{cases}$$

Ecuación 3-4. Vector de posición

Siendo ***pos*** la posición del *embedding* (es decir la posición que ocupaba la palabra o fragmento en la secuencia) y siendo ***i*** el elemento número *i* del vector de *embedding* que representa a la palabra que está en el elemento

número ***pos*** de la secuencia.

Este vector será la entrada de la capa *multi-head attention*, va a ser el mismo vector para los 3 elementos de entrada, es decir el vector *query*, *value* y *key*. Este tipo de *attention mechanism* se conoce como *self-attention* y se utiliza para extraer las relaciones e importancia que tienen el resto de las palabras del texto o frase sobre cada palabra, de tal manera que el modelo conozca qué *tokens* tienen mayor o menor relación con el resto de los *tokens* de la secuencia. La ventaja de este método es que conseguimos relaciones de alcance muy elevado de tal forma que es posible computar la relación entre cada una de las palabras sin verse especialmente afectada por la separación entre ellas dentro de la secuencia.

Este vector va a ser procesado por una capa *FF* que transformará lineal e independientemente cada una de las posiciones con el mismo tipo de transformación. Se realizan dos transformaciones lineales con una función de activación *RELU* entre medias. Finalmente vamos a obtener unas representaciones complejas con un alto nivel informativo acerca del contexto y el significado de cada palabra.

3.4 Estado Del Arte

Una vez tenemos una visión general, vamos a destacar algunos de los modelos que representan el Estado del Arte en el procesamiento del lenguaje natural hoy día, y vamos a profundizar en los diferentes *layers* y técnicas de procesamiento del lenguaje natural del que hacen uso para alcanzar un rendimiento excepcional en dichas tareas.

3.4.1 Tareas NLP

Existen diferentes tareas comúnmente utilizadas para evaluar el rendimiento de un modelo *NLP*. La tarea específica usada dependerá del tipo de modelo *NLP* y la aplicación que le queramos dar, sin embargo, estas son algunas de las tareas *NLP* más utilizadas:

- Modelado de Lenguaje: consiste en predecir la similitud de una secuencia de palabras con otra dada. Se utiliza normalmente para el pre-entreno del modelo para otras tareas de *NLP* como clasificación de textos, análisis de sentimiento o traducción.
- Clasificación de textos: consiste en asignar una categoría o etiqueta a un texto o fragmento de texto. Los ejemplos más comunes son análisis de sentimiento, clasificación por temas o detección de spam.
- *NER* (por sus siglas en inglés *Named Entity Recognition*): consiste en la identificación de entidades como personas, organizaciones o sitios y lugares dentro del texto.
- Etiquetado *POS* (por sus siglas en inglés *Part-Of-Speech*): consiste en asignar una categoría gramatical (adjetivo, verbo, sustantivo, pronombre…) a cada elemento del texto.
- Traducción Máquina: como su propio nombre indica consiste en traducir textos de un idioma a otro.
- Respuesta a Preguntas: consiste en responder a preguntas formuladas en lenguaje natural basándose normalmente en un contexto o temática.
- Resumen de textos: consiste en resumir un texto de mayor tamaño manteniendo la intencionalidad y mensaje original de la forma más compacta posible.
- Generación de lenguaje: consiste en generar textos en lenguaje natural. Es el motor de sistemas como los *chatbots* y se usa también para completar textos e incluso para generar textos completos según una instrucción o *promt*.

Adicionalmente a estas tareas existen numerosas métricas de evaluación usadas frecuentemente para medir el rendimiento de los modelos *NLP*. Además de las métricas más clásicas como precisión, exactitud o *F1*, algunos ejemplos serían la *BLEU score*, *ROUGE score* y métricas basadas en la perplejidad. (Kumar et al. [31] (2019)).

Junto con estas métricas y tareas surgen un conjunto de pruebas diseñadas para probar la eficacia de los modelos en las diferentes tareas mencionadas y proporcionar un resultado que permite medir el rendimiento general del

modelo, esto es lo que se conoce como los *benchmarks*. El objetivo es la posibilidad de comparar el rendimiento de los diferentes modelos en las distintas tareas y proporcionar un indicador del estado actual del campo del *NLP*.

Algunos de los *benchmarks* más utilizados y lo que se encargan de medir:

- *GLUE* (*General Language Understanding Evaluation*): como su nombre indica es un *benchmark* muy general en el que se mide el rendimiento en tareas como el análisis de sentimientos, la clasificación de textos o las respuestas a preguntas (Wang et al. [32] (2018)). Existe la versión extendida conocida como *superGLUE* (Wang et al. [33] (2019)).
- *SQuAD* (*Stanford Question Answering Dataset*): evalúa el rendimiento de los modelos en tareas de comprensión lectora. Es un conjunto de preguntas basadas en artículos de Wikipedia (Rajpurkar et al. [34] (2016)).
- *CoNLL*: mide el rendimiento de los modelos en tareas de *NER*.
- *MNLI* (*Multi-Genre Natural Language Inference*): evalúa el rendimiento del modelo en tareas de inferencia en el lenguaje natural, como por ejemplo determinar la relación entre dos frases (Williams et al. [35] (2018)).

3.4.2 Modelos State of the Art

3.4.2.1 BERT

BERT o *Bidirectional Encoder Representations from Transformers*, (Devlin et al. [36] (2019)), es un modelo de lenguaje pre-entrenado desarrollado por Google en 2018 capaz de crear complejas representaciones para diversas tareas del lenguaje natural como las introducidas en el apartado anterior y que hace uso de la estructura del *transformer*, en este caso solo hace uso del *encoder*, ya que no busca la generación de lenguaje sino el procesamiento y la extracción de características.

Ha sido entrenado en una amplísima cantidad de información textual, como libros, artículos… haciendo uso de un tipo de aprendizaje no supervisado conocido como *language modeling* o modelado de lenguaje. Durante el entrenamiento, una determinada cantidad de los *tokens* que forman los textos van a ser sustituidos por un *token* especial conocido como *mask*, el objetivo será que el modelo prediga que *token* había originalmente en dicha posición haciendo uso del contexto otorgado por el resto de los *tokens* del texto o secuencia, esta técnica se conoce como *masked language* modeling (*MLM*) o modelado de lenguaje enmascarado. Tenemos un ejemplo gráfico en la Figura 3-11.

Figura 3-11. Masking de una frase

Adicionalmente es entrenado con otro tipo de aprendizaje semi-supervisado conocido como *next sentence prediction (NSP)* o predicción de siguiente frase, que consiste básicamente en predecir en un par de frases si la segunda es continuación o no de la primera. El objetivo es conseguir proporcionar al modelo un conocimiento general de las relaciones y significados de las palabras y estructuras que conforman el lenguaje natural. Como se puede observar en la Figura 3-12.

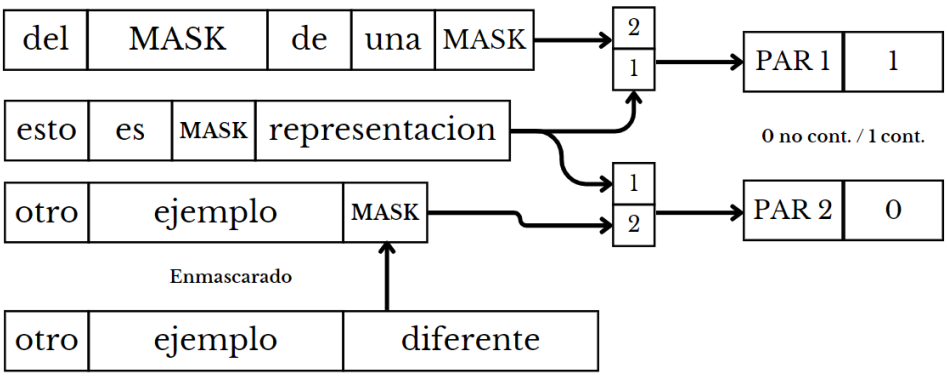

Figura 3-12. Creación de pares para NSP

Una de las características más destacadas de *BERT* es que es un modelo de lenguaje bidireccional, lo que significa que a la hora de predecir los *tokens* tiene en cuenta tantos los *tokens* a la izquierda como a la derecha del *token* enmascarado, es decir, que hace uso de la secuencia completa. Esto ayuda al modelo a capturar el contexto y significado del texto o secuencia más eficientemente.

BERT ha conseguido resultados de *state of the art* en numerosos campos del *NLP* y ha sido ampliamente adoptado tanto en la industria comercial como en el campo de la investigación. El objetivo fundamental de *BERT* es el procesamiento de textos para la extracción de características y posteriormente usar esta información para las diferentes tareas que existen en el campo del *NLP*. Actualmente existen un gran número de variaciones y modelos *fine-tuned* de *BERT* en la mayoría de las tareas, como podrían ser *RoBERTa* (Liu et al. [37] (2019)) o *DeBERTa* (He et al. [38] (2020)).

3.4.2.2 **GPT**

GTP o *Generative Pre-trained Transformer*, (Radford et al. [39] (2018)), es un modelo de lenguaje natural desarrollado por OPEN AI capaz de generar información textual de alta calidad en un gran número de contextos y campos dentro del *NLP*. También hace uso de la estructura del *transformer*, en este caso tanto del *encoder* como del *decoder*, ya que, si es un modelo generativo, siendo uno de los más grandes que existen actualmente con un inmenso número de parámetros, lo que lo convierte en uno de los modelos *NLP* más potentes de la actualidad.

Ha sido también entrenado en una inmensa cantidad de textos en este caso haciendo uso de otra técnica de aprendizaje no supervisado que consiste en predecir la siguiente palabra para una secuencia dada, lo que es útil para ayudar al modelo a comprender la estructura y funcionamiento del lenguaje natural. Hay que destacar que *GTP* es unidireccional y que solo tiene en cuenta los elementos previos de la secuencia a la hora de calcular la probabilidad del siguiente *token*. Como se muestra en la Figura 3-13.

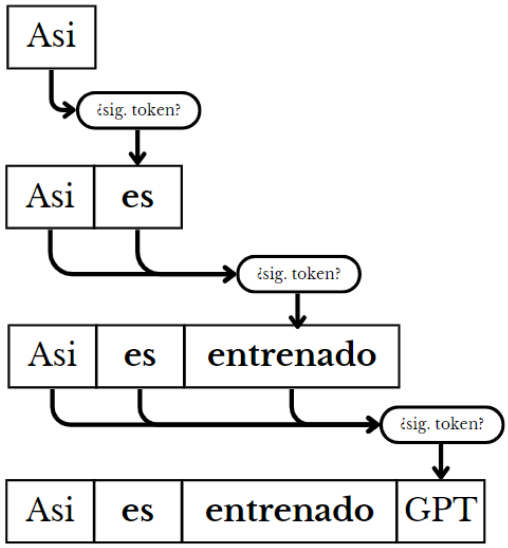

Figura 3-13. Predicción de siguiente token

Una de las características más destacadas de *GPT* es su capacidad de llevar a cabo lo que se conoce como *zero-shot* o *few-show learning,* (Brown et al. [40] (2020)), esto consiste en la capacidad de generar información textual precisa en tareas o contextos en los que no ha sido o ha sido muy poco entrenado, convirtiéndolo en uno de los modelos *NLP* generativos más versátiles y siendo además capaz de lograr rendimientos excepcionales en la mayoría de las tareas de *NLP*.

El objetivo fundamental de *GPT* es la generación de lenguaje natural coherente y lo más similar a un humano que sea posible para una petición o *prompt* dado o un contexto determinado. Actualmente está en su versión mejorada *GPT* 4.0 que además es multimodal, es decir que aceptara otros tipos de información de entrada diferente al texto, como podrían ser imágenes o videos.

3.4.2.3 **ERNIE**

ERNIE o *Enhanced Representation through Knowledge Integration*, (Sun et al. [41] (2019)), es un conjunto de modelos de lenguaje pre-entrenados desarrollados por Baidu Research diseñados para integrar conocimientos de diferentes dominios y fuentes para mejorar su capacidad de entender y generar lenguaje natural.

Al igual que *GPT* o *BERT* han sido entrenados en una inmensa cantidad de textos usando técnicas de aprendizaje no supervisados para posteriormente hacer *fine-tuning* en tareas más específicas del *NLP*. Lo que marca la diferencia con el resto de los modelos es su capacidad de aprender de diferentes fuentes de información como grafos o textos no estructurados. Esto permite que *ERNIE* sea capaz de capturar más eficientemente el contexto y significado del lenguaje y generar respuestas más precisas y relevantes.

4 MÉTODO PROPUESTO

Tras esta introducción teórica acerca de la *IA*, el *machine learning*, las redes neuronales y la aplicación de estas en el campo del procesamiento del lenguaje natural, así como un breve vistazo a los modelos del estado del arte durante los dos anteriores capítulos, vamos a proponer un modelo de red neuronal que sea capaz de extraer las relaciones semánticas entre dos frases (una premisa y su consiguiente hipótesis) y determinar si la hipótesis es correcta, contradictoria o neutral.

4.1 Recursos Utilizados

A continuación, se hará una breve descripción del material hardware y software utilizado para diseñar y entrenar a nuestro modelo.

4.1.1 Hardware

Como ya hemos visto, entrenar un modelo de *machine learning* es una tarea que dependiendo del tamaño y la complejidad del modelo puede requerir una gran capacidad de cálculo computacional. En nuestro caso, vamos a necesitar un modelo bastante complejo para la tarea en la que lo vamos a entrenar, la extracción de relaciones semánticas entre una hipótesis y una premisa, que no solo es una tarea complicada de por sí para una máquina, sino que puede llegar a ser complejo incluso para un ser humano real.

A pesar de que un modelo de *machine learning* puede ser entrenado con una *CPU* común, esta no es especialmente eficiente pues no requerimos de cálculos complejos si no de un enorme número de operaciones sencillas, tarea en la que estas no rinden bien. Aquí es donde entran en juego las *GPUs*, hardware específicamente diseñado para resolver de manera simultánea un número muy elevado de operaciones, que, aunque no fueron diseñadas exactamente para entrenar redes neuronales si no más enfocadas a ejecución de gráficos en videojuego, diseño gráfico o animación 3D, se adaptan especialmente bien a la naturaleza del *deep learning* y que hasta los últimos años ha sido la herramienta más usada a la hora de entrenar los modelos.

Dado que no disponemos de ningún equipo con una *GPU* lo suficientemente potente como para entrenar en un tiempo razonable, vamos a optar por la solución del hacer uso de un servicio alojado, en este caso de Google Colab. Google Colab es el servicio alojado de Google que permite la ejecución de código Python en el navegador sobre una máquina virtual dedicada, que da acceso a recursos como *CPUs* o *GPUs* y está basado en Jupyter. Cabe destacar que dado el reciente auge del *machine learning* y las redes neuronales, Google ha diseñado hardware específicamente optimizado para el entrenamiento de redes neuronales al que ha llamado *TPUs* (*Tensor Procesor Units*), que superan en rendimiento a las *GPUs* incluso con una menor cantidad de recursos. Dado que nos ofrecen tanto *GPUs* como *TPUs* para entrenar a nuestro modelo, nosotros lo entrenaremos con una *TPU*. Una ventaja adicional de entrenar a nuestro modelo sobre un servicio alojado es la disponibilidad de equipos con altas cantidades de memoria RAM, que también es muy útil a la hora de trabajar con mucha información de manera rápida.

Puesto que el entrenamiento no se va a realizar sobre un equipo local y que las máquinas virtuales alojadas de Google Colab no tienen disponibilidad ilimitada, para almacenar nuestro modelo durante el proceso de entrenamiento vamos a utilizar Google Drive que es fácilmente integrable con Google Colab, también almacenaremos en Google Drive nuestro propio *tokenizer*.

En resumen, el entrenamiento de nuestro modelo se va a realizar sobre una maquina alojada en los servidores de Google haciendo uso de su servicio Google Colab.

4.1.2 Lenguaje de Programación

Existe un gran número de lenguajes de programación que pueden ser utilizados para el *deep learning*, como C, Java, C++, JS, Matlab… pero los que más destacan tanto en el *machine learning* como en el *data science*, son Python y R. A continuación, un gráfico con los lenguajes más utilizados según la encuesta realizada en 2022 por

la famosa plataforma de *deep learning* Kaggle en la Figura 4-1.

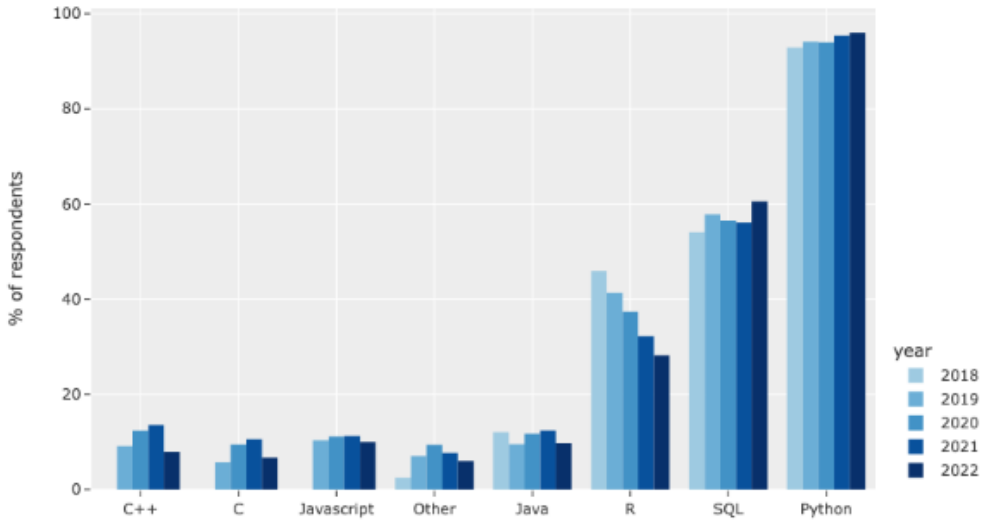

Figura 4-1. Lenguajes de programación más utilizados para ML. Kaggle 2022.

Como podemos ver Python se posiciona como el lenguaje preferido para el *machine learning*, compitiendo con otros como SQL o R. Dado que el entorno sobre el que vamos a entrenar a nuestro modelo está diseñado para trabajar con Python, no tenemos mucha capacidad de elección en ese sentido por lo que toda la programación se hará en un cuaderno de Google Colab que utiliza Python como lenguaje de programación.

4.1.3 Framework

Hay diversos *frameworks* disponibles para diseñar y entrenar modelos de redes neuronales, vamos a ver una comparativa de los *frameworks* más utilizados para *machine learning* y *data science*. A continuación, un gráfico en la Figura 4-2.

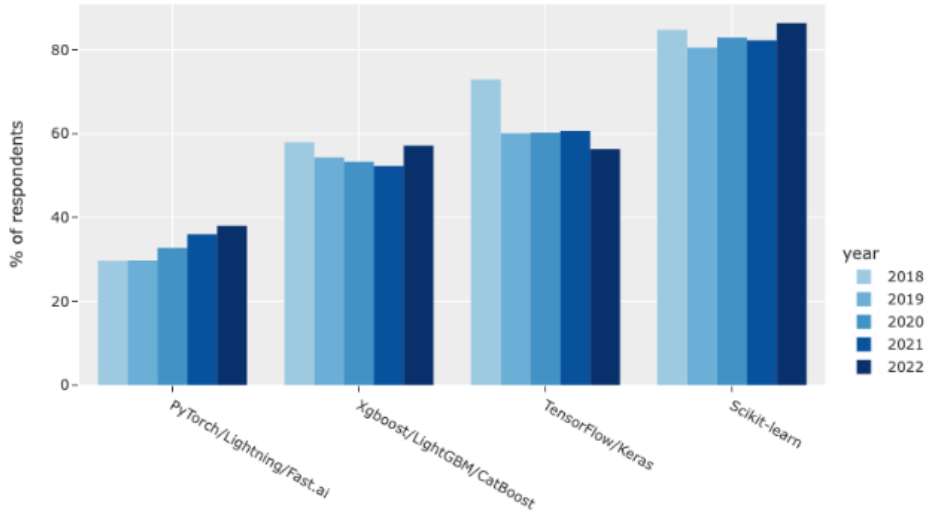

Figura 4-2. Frameworks más utilizados para ML. Kaggle 2022.

Como podemos ver, existen un gran número de alternativas entre las que podemos elegir, nosotros vamos a escoger Keras, que es un *framework* de código libre que hace uso de la librería y *framework* de bajo nivel TensorFlow. Fue creada por el ingeniero de Google François Chollet y está escrita en Python. La principal ventaja de Keras es la facilidad que ofrece a la hora de trabajar con vectores y operaciones matriciales, así como

para diseñar y entrenar los modelos de *machine learning* gracias a su API de alto nivel, también he de destacar que la *TPU* que vamos a usar para entrenar a nuestro modelo esta específicamente optimizada para utilizar Keras, por lo que es la opción más óptima para diseñar y entrenar nuestro modelo.

4.1.4 Librerías

Adicionalmente a Keras como *framework* (que hace uso de la librería Tensorflow) vamos a hacer uso de algunas librerías adicionales más:

- Pandas: librería para la manipulación y tratamiento de datos. Vamos a utilizarla únicamente para cargar uno de los *datasets* de los que haremos uso a la hora de entrenar a nuestro modelo.
- Numpy: librería para el manejo de vectores y matrices. La vamos a utilizar múltiples veces para el manejo de matrices a lo largo del pre-procesamiento de los datos.
- Tokenizers: librería específica para la creación y entrenamiento de los *tokenizers*. La utilizaremos para configurar y entrenar nuestro propio *tokenizer* desde 0.
- Datasets: librería que facilita el acceso y la manipulación de los *datasets* con los que se entrenan a los modelos de *machine learning*. La utilizaremos para descargar otro *dataset* que necesitamos.

Haciendo uso de estas herramientas junto con Keras, vamos a procesar la información de los *datasets* y a prepararla para que nuestro modelo puede trabajar con ella y aprender, así como a diseñar y a entrenar a nuestro modelo.

4.2 Datasets

Para entrenar a nuestro modelo vamos a hacer uso de 2 *datasets*. El primer *dataset* lo vamos a utilizar para realizar el pre-entreno de nuestro modelo, es decir, va a ser un *dataset* no etiquetado para el aprendizaje no supervisado. El *dataset* que vamos a utilizar es Wikipedia-20220301 que consiste en textos en claro de todos los artículos de Wikipedia, en nuestro caso haremos uso únicamente de los escritos en inglés. El objetivo de esto es proporcionar al modelo un conocimiento general del lenguaje que posteriormente trasladaremos a una tarea más específica.

Cada uno de los ejemplos contara con los siguientes campos:

- ID: el identificador único del ejemplo
- URL: dirección URL del artículo en cuestión
- Title: título del articulo
- Text: texto que compone el artículo.

Nosotros vamos a hacer uso fundamentalmente de los textos que componen los artículos ya que no necesitamos darle gran importancia al título de este para la tarea en la que lo vamos a entrenar. En total van a ser 6.458.670 ejemplos o artículos, de los cuales algunos serán más útiles que otros que quizá sean demasiado cortos o carecen de información relevante para que nuestro modelo extraiga relaciones del lenguaje, que es el objetivo del pre-entreno con este *dataset*. Dado que es un *dataset* no etiquetado vamos a tener que hacer un pre-procesamiento de la información para adaptarla al método de entrenamiento que utilizaremos para pre-entrenar a nuestro modelo, pero esto lo detallaremos más adelante.

Hemos elegido este *dataset* en inglés para el pre-entreno ya que es extremadamente extenso y contiene una gran cantidad de información que es mucho mayor al de resto de lenguajes disponibles, y que tras un poco de pre-procesamiento podremos usar para que nuestro modelo comprenda mejor las características y relaciones del lenguaje natural en inglés.

El segundo *dataset,* que utilizaremos para el *fine-tuning* de nuestro modelo en establecer la relación semántica

entre 2 frases, será el ofrecido por la plataforma Kaggle en su desafío “Contradictory, My Dear Watson”. Este desafío ofrece un *dataset* con un conjunto de 12.120 ejemplos clasificados de premisas e hipótesis en 15 idiomas y 3 clases diferentes: relacionadas, contradictorias o neutrales. Los ejemplos están formados por los siguientes campos:

- ID: el identificador único del ejemplo
- Premise: texto que compone la premisa
- Hypothesis: texto que compone la hipótesis
- Language: idioma de la premisa y de la hipótesis
- Lang_abv: la abreviación del idioma
- Label: clase en la que se incluye el ejemplo

Vamos a profundizar un poco en el contenido del *dataset*. Comencemos por el número de ejemplos de cada idioma. A continuación, una gráfica comparativa en la figura 4-3.

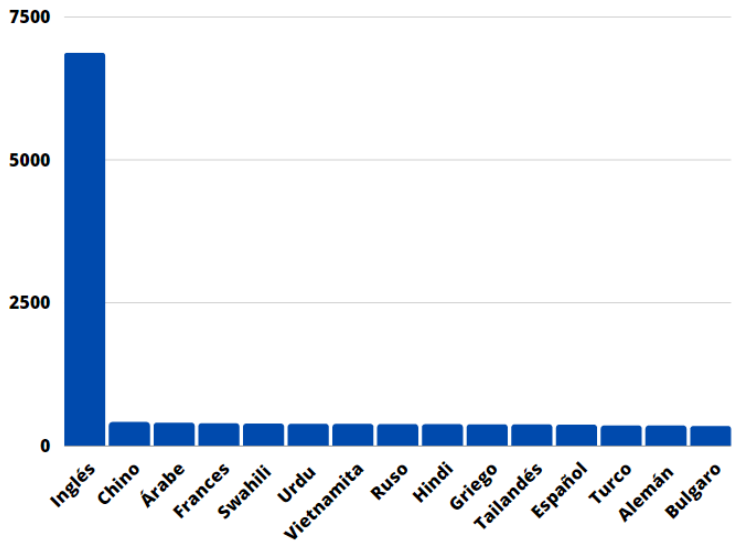

Figura 4-3. Comparativa de idiomas en el dataset para el fine-tuning

Como podemos observar, el número de ejemplos en inglés es la mayor parte del *dataset*, siendo 6870 ejemplos en total. Es por ello por lo que nosotros nos vamos a enfocar solo en los ejemplos en inglés, ya que el pre-entreno de nuestro modelo se ha realizado en textos en inglés y puesto que componen la mayor parte del *dataset* vamos a descartar el resto de los idiomas para entrenar a nuestro modelo.

Las clases en las que podemos clasificar cada ejemplo son las siguientes:

Identificador	Clase
0	Relacionada
1	Neutral
2	Contradictoria

Tabla 4-1. Etiquetas de clases dataset fine-tuning

Ahora, observemos el número de ejemplos de cada clase, para determinar si nos encontramos ante un *dataset* cuyas clases están balanceadas o no. Veamos cómo se han clasificado cada par de hipótesis y premisas de entre todos los ejemplos en inglés que componen el *dataset*. Ejemplo grafico en la Figura 4-4.

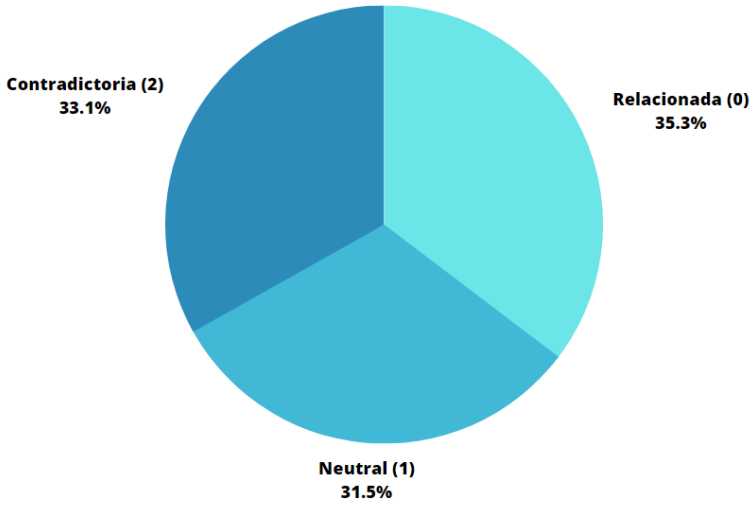

Figura 4-4. Clasificación de los distintos ejemplos en ingles en el dataset para el fine-tuning

Como podemos observar, si es un problema balanceado ya que aproximadamente tenemos el mismo número de ejemplos de cada clase. Ya tenemos una visión general de la información que vamos a usar en ambas fases del entrenamiento de nuestro modelo de lenguaje natural.

4.3 Diseño del Modelo

A continuación, vamos a detallar el diseño que propondremos para la resolución de nuestra tarea objetivo y la forma que tendremos de implementarlo. Adicionalmente vamos a configurar e implementar nuestro propio *tokenizer,* que usaremos posteriormente para extraer los *tokens* con los que entrenaremos a nuestro modelo.

El modelo que vamos a proponer va a seguir la arquitectura propuesta por Devlin et al. [36], aunque vamos a implementar un modelo bastante simplificado, ya que la complejidad de *BERT* es excesiva, y entrenar un modelo de esas características con los recursos de los que disponemos sería totalmente ineficiente ya que necesitaríamos demasiado tiempo. Es por ello por lo que nosotros vamos a simplificar su arquitectura haciendo uso de un único *encoder* (en vez de apilar 12 o 24 como en el modelo *BERT* original).

El procedimiento que se ha seguido en la implementación del modelo es el siguiente:

1. Configuración y entrenamiento del *tokenizer* para la extracción de *tokens* con los que procesaremos los *datasets* para entrenar posteriormente a nuestro modelo.
2. Diseño de un modelo simplificado de la arquitectura propuesta por Devlin et al. [36].
3. Pre-entrenamiento del modelo para obtener representaciones contextuales de los *tokens,* este pre-entreno consistirá en dos tareas: para la primera el modelo tendrá que adivinar que palabra ocupaba originalmente en la frase el lugar del token de *mask* y para la segunda el modelo tendrá que predecir, para un par de frases, si las frases son o no continuación una de la otra.
4. *Fine-tuning* del modelo, a partir de estas representaciones contextuales extraídas durante el pre-entreno, en la tarea objetivo propuesta por la competición “Contradictory My Dear Watson” de Kaggle.

4.3.1 Instalación de Librerías y Obtención de los Datasets

El primer paso que tomaremos será la instalación de las librerías que necesitaremos para procesar los datos, diseñar, entrenar y evaluar a nuestro modelo (Código 0-1).

Adicionalmente, creamos una función para importar nuestro *tokenizer*, *datasets* y modelos entrenados o en proceso de entrenamiento desde Drive. Cargamos 3 carpetas, donde almacenaremos nuestros modelos, nuestros *tokenizers* y nuestro *dataset.* (Código 0-2).

Descargamos el *dataset* para el pre-entreno haciendo uso de la librería Datasets. Después obtenemos los valores del *dataset* y comprobamos el número de ejemplos que lo forman. Que como se ha mencionado anteriormente son 6.458.670 artículos de Wikipedia en inglés (Código 0-3).

Para el *dataset* del *fine-tuning* lo que haremos es descargarlo de la página de Kaggle, y cargarlo en nuestro Drive, para posteriormente subirlo en nuestro entorno virtual de Google Colab. Lo cargamos en memoria con el Código 0-4.

4.3.2 Configuración y Entrenamiento del Tokenizer

Ahora es momento de configurar y entrenar nuestro propio *tokenizer* que nos ayude a procesar la información textual para que sea más manejable por nuestro modelo. Lo primero que haremos es definir una función que procese y filtre los textos para entrenar a nuestro *tokenizer* con ellos. (Código 0-5). Lo que hacemos con esta función es dividir el texto en párrafos y comprobar que al menos cada párrafo tiene 30 palabras. Finalmente generamos una lista de párrafos, que será con los que entrenaremos nuestro *tokenizer*. Este filtrado del *dataset* se utilizará exclusivamente a la hora de entrenar el tokenizer para la extracción de tokens, el *tokenizer* posteriormente se encargará de *tokenizar* las frases con la que entrenaremos el modelo, no párrafos completos.

Ahora vamos a configurar y entrenar nuestro *tokenizer* desde cero. Primero importamos el tipo de *tokenizer* que utilizaremos, en este caso va a ser un *WordPiece tokenizer* (Huggingface [42] (2021)). Después definimos los *tokens* especiales de los que haremos uso, en este caso será un *token* desconocido (el *tokenizer* lo usará si aparece una palabra que no se encuentra en el vocabulario), también definimos el *token* de *mask* para el pre-entreno, el *token* de *padding* y por último los *tokens* de *sentence segmentation*, el de inicio de frase o *BOS* y el de fin de frase o *EOS*, finalmente obtenemos el índice asociado a dichos *tokens* para usarlos más adelante.

La manera de extraer *tokens* del *WordPiece tokenizer* consiste en dividir el texto en unidades más pequeñas basándose en la identificación de los patrones de caracteres más comunes. Comienza con caracteres individuales y poco a poco los unifica en secuencias de caracteres según los patrones que detecta más a menudo, que dan lugar finalmente a fragmentos de palabras y palabras completas.

Definimos un *trainer* para entrenar nuestro *tokenizer*, vamos a utilizar un vocabulario de 25.000, es decir, el *tokenizer* identificara como máximo 25.000 *tokens* y extraemos una parte de nuestro *dataset* para el pre-entreno (no utilizamos todo el *dataset* porque tomaría demasiado tiempo y a partir de un determinado número de artículos tampoco va a detectar nuevo vocabulario). Antes de comenzar el entrenamiento, definimos el pre-procesamiento, es decir, de que forma el *tokenizer* limpiara los textos antes de procesarlos (en este caso todo a minúscula, y eliminara los acentos y caracteres especiales) y finalmente entrenamos a nuestro *tokenizer*.

Una vez finalizado el entrenamiento del *tokenizer*, definimos el postprocesado de cara al entrenamiento del modelo, cuando el *tokenizer* tenga que *tokenizar* los pares de frases (la primera frase comenzara con el *token BOS* y ambas finalizaran con el *token EOS,* adicionalmente, a los *tokens* correspondientes a la primera frase se les asociara el valor 1 y a los de la segunda el valor 2 para la creación del vector de frase). Establecemos el tamaño para el conjunto de las dos frases a 256 *tokens*, tanto para añadir *padding* como para truncar el par de frases. Finalmente guardamos nuestro *tokenizer* (Código 4-1).

```
 from tokenizers import
(decoders,models,normalizers,pre_tokenizers,processors,trainers,Toke
nizer,)

 !mkdir ./data
 !mkdir ./data/tokenizer/

 tokenizer_folder = '/content/drive/MyDrive/GITT/TFG/tokenizer/'

 tokenizer = Tokenizer(models.WordPiece(unk_token="<unk>"))

 special_tokens = ["<pad>","<s>","</s>","<mask>","<unk>"]

 trainer = trainers.WordPieceTrainer(vocab_size=VOCAB_SIZE,
special_tokens=special_tokens, min_frequency=4)

 tokenizer_ds = ClearDS(ml_train_dataset[:2500000])

 tokenizer.normalizer = normalizers.Sequence([normalizers.NFD(),
normalizers.Lowercase(), normalizers.StripAccents()])

 tokenizer.pre_tokenizer =
pre_tokenizers.Sequence([pre_tokenizers.WhitespaceSplit(),
pre_tokenizers.Punctuation()])

 tokenizer.train_from_iterator(tokenizer_ds, trainer=trainer)

 pad_token = tokenizer.token_to_id("<pad>")
 bos_token = tokenizer.token_to_id("<s>")
 eos_token = tokenizer.token_to_id("</s>")
 mask_token = tokenizer.token_to_id("<mask>")
 unk_token = tokenizer.token_to_id("<unk>")

 tokenizer.post_processor =
processors.TemplateProcessing(single=f"<s>:1 $A:1 </s>:1",
pair=f"<s>:1 $A:1 </s>:1 $B:2 </s>:2", special_tokens=[("<s>",
bos_token), ("</s>", eos_token)],)

 tokenizer.enable_truncation(max_length=MAX_LEN)
 tokenizer.enable_padding(length=MAX_LEN, pad_token='<pad>', pad_id
= tokenizer.token_to_id("<pad>"))

 tokenizer.save(tokenizer_folder+"wp_tokenizer.json")
```

Código 4-1. Creación y entrenamiento del tokenizer

4.3.3 Diseño y Creación de Nuestro Modelo

Ahora que ya tenemos un *tokenizer* apto para procesar la información que entreguemos a nuestro modelo, vamos a diseñar e implementar nuestro propio modelo de lenguaje siguiendo la estructura de *BERT*. Primero necesitamos crear una función que sobrecargue la clase *keras.Model* (Código 0-9) porque vamos a necesitar un proceso de entrenamiento especial, ya que haremos uso de dos funciones de pérdida diferentes para las dos tareas de pre-entreno, en los detalles de este pre-entreno como las funciones de pérdida o las métricas entraremos más adelante en detalle. Vamos también a definir una función para la creación del vector de posición (Código 0-7),

así como una función auxiliar que cree el *encoder* del *transformer* (Código 0-8).

Por último, creamos una función para construir el modelo completo haciendo uso de las 2 funciones nombradas anteriormente, así como de la clase sobrecargada BertLikeModel (Código 0-9). Nuestro modelo va a recibir 2 vectores de entrada, el vector de frase y el vector del par de frases enmascaradas, ambos van a pasarse por una capa de *embedding,* al igual que los vectores de posición que generamos sobre la marcha. Estos valores, matrices de tamaño [256,512], siendo 512 la dimensionalidad que hemos asignado a los *embeddings*, se suman y se usan de entrada para el *encoder* del *transfomer*. Ahora definimos las ramas del modelo para las dos tareas en las que será pre-entrenado.

Para la primera tarea, en la salida de esta capa, se extrae el *embedding* contextual del primer *token*, un vector de tamaño 512 (también llamado *CLS* y que en el modelo de *BERT* original se usa para clasificación) y es lo que usaremos para determinar si la segunda frase es continuación o no de la primera, usándolo de entrada a una capa *FF* (función de activación *tanh*) y finalmente la salida de la rama de *NSP* que es un capa *FF* que llamaremos clasificadora, que dará como resultado un valor entre 0 y 1 (función de activación *sigmoid*) y determinara si es o no la segunda frase continuación de la primera.

Para la segunda tarea hacemos uso de todos los *embeddings* contextuales de la frase al completo, para utilizarlo en el aprendizaje enmascarado, en el que el modelo intentara predecir el *token* que ocupa el lugar de los *tokens* de *mask*, usándolos de entrada para una capa *FF* clasificadora (función de activación *softmax*) del tamaño del vocabulario, la salida es una matriz de tamaño [256, 25000], en la que para cada una de las 256 posiciones se determinaran cuál de los 25.000 posibles *tokens* es el más probable de ocuparla (la suma de la puntuación de los 25.000 *tokens* posibles para una posición dará 1). (Código 4-2).

```
def create_bl_model():

  inputs = layers.Input((2,MAX_LEN), dtype=tf.int64)

  word_embeddings =
layers.Embedding(VOCAB_SIZE,EMB_DIM)(inputs[:,0])

  position_embeddings =
layers.Embedding(input_dim=MAX_LEN,output_dim=EMB_DIM,weights=[get_p
os_encoding_matrix(MAX_LEN, EMB_DIM)],)(tf.range(start=0,
limit=MAX_LEN, delta=1))

  type_embeddings = layers.Embedding(3,EMB_DIM)(inputs[:,1])

  embeddings = word_embeddings + position_embeddings +
type_embeddings

  encoder_output = encode_sublayer(embeddings,True)

  ml_output =
layers.Dense(VOCAB_SIZE,activation="softmax")(encoder_output)
ns_act = layers.Dense(512,
activation="tanh")(encoder_output[:,0,:])

  ns_output = layers.Dense(1, activation="sigmoid")(ns_act)

  bl_model = BertLikeModel(inputs, [ml_output, ns_output],
name="bert_like_model")

  return bl_model
```

Código 4-2. Creación de nuestro modelo

En la Figura 4-5 podemos observar un esquema de la estructura de nuestra red neuronal.

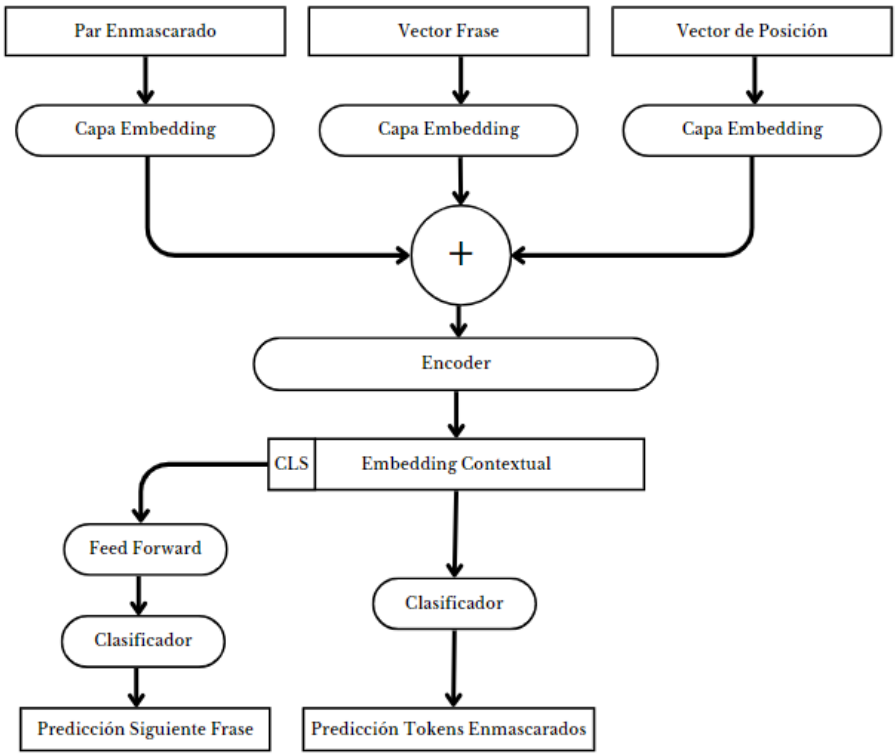

Figura 4-5. Esquema de nuestro modelo

También podemos visualizar a continuación, en la Figura 4-6 el resultado de compilar el modelo, donde podremos observar los parámetros en cada capa y el número total de parámetros entrenables de nuestro modelo.

```
Model: "bert_like_model"
__________________________________________________________________________________________________
 Layer (type)                   Output Shape         Param #     Connected to
==================================================================================================
 input_2 (InputLayer)           [(None, 2, 256)]     0           []

 tf.__operators__.getitem_3 (Sl  (None, 256)         0           ['input_2[0][0]']
 icingOpLambda)

 embedding_3 (Embedding)        (None, 256, 512)     12800000    ['tf.__operators__.getitem_3[0][0
                                                                 ]']

 tf.__operators__.getitem_4 (Sl  (None, 256)         0           ['input_2[0][0]']
 icingOpLambda)

 tf.__operators__.add_4 (TFOpLa  (None, 256, 512)    0           ['embedding_3[0][0]']
 mbda)

 embedding_5 (Embedding)        (None, 256, 512)     1536        ['tf.__operators__.getitem_4[0][0
                                                                 ]']

 tf.__operators__.add_5 (TFOpLa  (None, 256, 512)    0           ['tf.__operators__.add_4[0][0]',
 mbda)                                                            'embedding_5[0][0]']

 multi_head_attention_1 (MultiH  (None, 256, 512)    1050624     ['tf.__operators__.add_5[0][0]',
 eadAttention)                                                    'tf.__operators__.add_5[0][0]',
                                                                  'tf.__operators__.add_5[0][0]']

 dropout_2 (Dropout)            (None, 256, 512)     0           ['multi_head_attention_1[0][0]']

 tf.__operators__.add_6 (TFOpLa  (None, 256, 512)    0           ['tf.__operators__.add_5[0][0]',
 mbda)                                                            'dropout_2[0][0]']

 layer_normalization_1 (LayerNo  (None, 256, 512)    1024        ['tf.__operators__.add_6[0][0]']
 rmalization)

 dense_5 (Dense)                (None, 256, 2048)    1050624     ['layer_normalization_1[0][0]']

 dense_6 (Dense)                (None, 256, 512)     1049088     ['dense_5[0][0]']

 dropout_3 (Dropout)            (None, 256, 512)     0           ['dense_6[0][0]']

 tf.__operators__.add_7 (TFOpLa  (None, 256, 512)    0           ['layer_normalization_1[0][0]',
 mbda)                                                            'dropout_3[0][0]']

 bl_output (LayerNormalization)  (None, 256, 512)    1024        ['tf.__operators__.add_7[0][0]']

 tf.__operators__.getitem_5 (Sl  (None, 512)         0           ['bl_output[0][0]']
 icingOpLambda)

 dense_8 (Dense)                (None, 512)          262656      ['tf.__operators__.getitem_5[0][0
                                                                 ]']

 dense_7 (Dense)                (None, 256, 25000)   12825000    ['bl_output[0][0]']

 dense_9 (Dense)                (None, 1)            513         ['dense_8[0][0]']

==================================================================================================
Total params: 29,042,089
Trainable params: 29,042,089
Non-trainable params: 0
```

Figura 4-6. Resumen de capas y parámetros del modelo

5 PRE-ENTRENO Y FINE-TUNING

Una vez definida la arquitectura de nuestro modelo y diseñadas las funciones que nos ayudaran a procesar la información con la que lo entrenaremos, vamos a comenzar con el pre-entrenamiento y haremos una evaluación de los resultados del mismo. Posteriormente procederemos con en *fine-tuning* en la tarea de clasificación de los pares de hipótesis y premisas en 3 posibles clases, como ya se ha mencionado. Por último, llevaremos a cabo una comparativa del rendimiento de nuestro modelo con un modelo *SOTA*, en este caso *RoBERTa*, haciendo uso de la misma estructura para el modelo clasificador, pero con *RoBERTa* en lugar de nuestro modelo pre-entrenado.

5.1 Pre-entrenamiento del Modelo

Una vez hemos configurado y entrenado nuestro *tokenizer* y diseñado la arquitectura de nuestro modelo, vamos a proceder con el pre-procesamiento del *dataset* para el pre-entreno.

5.1.1 Pre-procesamiento del Dataset para el Pre-entreno

En nuestro caso vamos a procesar todos los artículos de Wikipedia para obtener pares de frases y vamos a enmascarar el 15% de los *tokens* que forman las frases, adicionalmente vamos a obtener el vector de frase que ayudará al modelo a diferenciar cada una de las frases (los *tokens* de *sentence segmentation* también ayudan al modelo, pero esto es un apoyo adicional). Para lograr esto vamos a definir la clase *CustomDataset*, que convertirá el *dataset* Wikipedia en un *dataset* formado por pares de frases *tokenizadas* y enmascaradas. Esta clase (Código 0-6), va a recibir como parámetro el *tokenizer*, el *dataset* a procesar y un valor booleano que determinará si enmascaramos o no los pares de frases cuando los procesemos (con enmascarar nos referimos a sustituir aleatoriamente un determinado número de los *tokens* que conforman las frases por el *token* de *mask*).

Primero vamos a filtrar el *dataset* original, para ello obtendremos todos los artículos que componen el *dataset* y los dividiremos por párrafos, posteriormente dividimos cada párrafo en frases y solo seleccionaremos las frases con más de 5 palabras (nos ayuda a librarnos de las cabeceras y títulos de los artículos). Una vez filtrado cada párrafo, comprobamos que estén formados al menos más de 2 frases y por último comprobamos que la lista de párrafos filtrados no este vacía. Una vez tenemos todos los textos filtrados y organizados por párrafos y frases, vamos a crear los pares de frases y a enmascararlos y finalmente obtenemos los vectores de frase.

Para la creación de los pares de frases hacemos uso de la lista de textos filtrados y de la función auxiliar *create_pairs*. Esta función va a recorrer el listado de textos filtrados y por cada párrafo va a recorrer la totalidad de frases que lo forman, con cada una de estas frases vamos a crear pares de frases, en la que la segunda puede o no ser la continuación de la primera. Habrá un 50% de posibilidades de crear un par de frases de cada tipo. En el caso de que la segunda no deba ser continuación de la primera, se tomara como el segundo elemento del par una frase aleatoria de un texto diferente. Si es la continuación de la primera le asignamos la etiqueta IsNext (1) y al contrario la etiqueta IsNotNext (0).

Una vez tenemos el listado de los pares de frases, vamos a enmascarar el 15% de los *tokens* (dentro de este 15% el 10% se sustituirá por un *token* aleatorio en vez del *token* de *mask* y un 10% se dejará como estaba, el objetivo de esto es reducir el impacto de dejar de ver el *token* de *mask* durante el *fine-tuning*). Para ello creamos un vector de *masking* con longitud igual a la carga útil de la frase (sin contar el *padding*) en el que el 15% de los valores serán 1, dichos índices cuyo valor es 1 son los que sustituiremos en la frase original, e ignoramos los *tokens* especiales a la hora de sustituir el *token*. Esto se logrará haciendo uso de la función auxiliar *mask_sequence*.

Finalmente obtenemos el vector de frase, un vector de igual tamaño que el vector de pares de frases enmascarados en el que los índices de la primera frase tienen valor 1, los de la segunda frase valor 2 y el *padding* valor 0.

Tras este procesamiento, obtendremos un *dataset* listo para pre-entrenar con él a nuestro modelo y formado por los pares de frases *tokenizados* y enmascarados, sus vectores de frase asociados, el vector de los pares de frases *tokenizados* sin enmascarar (resultado esperado para el aprendizaje de lenguaje enmascarado) y la etiqueta que indica si la segunda frase es continuación o no de la primera. Todos los vectores tendrán una longitud de 256 (256 *tokens* entre ambas frases que forman el par), y la etiqueta de la siguiente frase un escalar, 0 indica no es continuación y 1 que si lo es.

5.1.2 Implementación del modelo

Ahora que ya hemos diseñado nuestro modelo y creado la función para procesar el *dataset*, queda el momento de crearlo y compilarlo para comenzar con el pre-entreno. Para ello nos conectamos con la TPU de Google (Malapaka [43] 2020) y creamos y compilamos en ella nuestro modelo. Escogemos las métricas, así como las funciones de pérdida a utilizar durante el pre-entreno y el *optimizer* que utilizaremos (Código 5-1).

```
  tpu = tf.distribute.cluster_resolver.TPUClusterResolver()

  print('Running on TPU ', tpu.cluster_spec().as_dict()['worker'])

  tf.config.experimental_connect_to_cluster(tpu)
  tf.tpu.experimental.initialize_tpu_system(tpu)
  tpu_strategy = tf.distribute.TPUStrategy(tpu)

  with tpu_strategy.scope():

      ns_loss_fn =
tf.keras.losses.BinaryCrossentropy(reduction=tf.keras.losses.Reducti
on.NONE)

      ml_loss_fn =
tf.keras.losses.SparseCategoricalCrossentropy(reduction=tf.keras.los
ses.Reduction.NONE)

      ml_metric=tf.keras.metrics.SparseCategoricalAccuracy()

      ns_metric=tf.keras.metrics.BinaryAccuracy()

      optimizer = keras.optimizers.Adam()

      bl_model = create_bl_model()

      bl_model.compile(optimizer=optimizer)

      bl_model.summary()
```

Código 5-1. Creamos y compilamos el modelo en la TPU

Ahora vamos a entrar en detalle en las métricas y funciones de pérdida utilizadas durante el pre-entreno. Como hemos mencionado anteriormente, hemos sobrecargado el método de entrenamiento de la clase *keras.Model* para diseñar un entrenamiento que tenga dos tipos de aprendizaje con sus respectivas métricas y funciones de pérdidas. En el Código 5-2 podemos observar como el modelo recibe como entrada un *dataset* en el cual sus ejemplos están formados por 4 vectores, 2 de estos vectores, como ya se ha mencionado, serán las etiquetas para el aprendizaje en las dos tareas en las que vamos a pre-entrenar a nuestro modelo y los otros los pares de frase enmascarados y el vector de frase asociado. Separamos los vectores de las etiquetas y los juntamos para dar lugar a la entrada que recibirá nuestro modelo, esta entrada se utilizará para realizar las predicciones.

Posteriormente con dichas predicciones calculamos el valor de la función de pérdida para cada tipo de aprendizaje, el lenguaje enmascarado y la predicción de continuación de frase, calculamos la pérdida total que en nuestro caso será la suma de ambas pérdidas y finalmente calculamos los gradientes de todas las variables entrenables del modelo, aplicamos los gradientes y actualizamos e imprimimos por pantalla las métricas (Código 0-9).

Vamos a detallar las métricas y funciones de pérdidas elegidas para ambos tipos de aprendizaje. Para el aprendizaje por lenguaje enmascarado elegimos como función de pérdida la *Sparse Categorical Crossentropy*, esta función calcula la entropía cruzada entre la predicción y el resultado esperado para ejemplos en el que existen más de dos clases posibles y cada entidad solo puede tener una clase asignada (como es el caso del lenguaje enmascarado en el que hay tantas clases como palabras en el vocabulario) y la *Binary Crossentropy*, para la predicción de continuación de frase, que es igual pero cuando solamente existen dos clases posibles (en este caso, continuación o no de la primera frase). (Gómez [44] 2018).

$$Cross\ Entropy = -\sum_{i}^{C} l_i * \log(p_i)$$

Ecuación 5-1. Función de entropía cruzada para C clases

Donde l_i hace referencia a la etiqueta y p_i hace referencia a la predicción para cada clase *i* dentro de *C.* La clasificación binaria es un caso concreto en el que *C*=2.

Con respecto a las métricas para la monitorización del aprendizaje hacemos uso de la exactitud en ambos casos, la *Binary Accuracy* para la predicción de continuación de frase y la *Sparse Categorical Accuracy* para el modelado de lenguaje enmascarado. Una vez definidas tanto las funciones de pérdida como las métricas es momento de comenzar con el pre-entreno.

Dado que nuestro *dataset* de pre-entreno es tan amplio, vamos a necesitar fragmentarlo antes de generar el *dataset* con los pares de frases para evitar agotar la memoria RAM de nuestro entorno virtual. Para ello dividimos el *dataset* para el pre-entreno en *pools* de 20.000 ejemplos. A continuación, creamos un bucle para recorrer el *dataset* al completo. Por cada *pool* de datos generamos un *dataset* con pares de frase haciendo uso de la clase CustomDataset, obtenemos cada vector del *dataset* por separado y creamos un *dataset* apto para entregarlo a nuestro modelo y que comience su aprendizaje. Cada 5 veces el *pool* de datos creamos un punto de guardado del modelo, ya que el tiempo de entrenamiento es tan elevado que seremos desconectados múltiples veces y podríamos perder el progreso. Adicionalmente llevamos un pequeño control del estado del entrenamiento, llevando la cuenta del total de frases sobre el que hemos entrenado y el porcentaje de cada clase durante el pre-entreno. (Código 5-2). También definimos un *callback* para comprobar como el modelo predice los *tokens* enmascarados (Código 0-10).

```
  batch_size = 20000
  test_examples_num = 40000
  train_examples = ml_ds_tam-test_examples_num
  ml_train_dataset = ml_train_dataset.shuffle(12345)

  total = 0
  nonext = 0
  next = 0

  for i in range(batch_size, train_examples, batch_size):

      prev_i = i-batch_size

      partial_dataset =
CustomDataset(ml_train_dataset[prev_i:i]['text'],tokenizer)

      train_inputs = np.array([x.get('input_ids') for x in
partial_dataset])

      train_sentence_vectors = np.array([x.get('sentence_vector')
for x in partial_dataset])

      train_ml_labels = np.array([x.get('ml_label') for x in
partial_dataset])

      train_ns_labels = np.array([x.get('ns_label') for x in
partial_dataset])

      for x in train_ns_labels:
         if x==0:
            nonext+=1
         else:
            next+=1
         total+=1

      feed_dataset =
tf.data.Dataset.from_tensor_slices((train_inputs,train_sentence_vect
ors,train_ml_labels,train_ns_labels))

      print("NoNext: "+str((nonext/total)*100)+ "% out of
"+str(total))
      print("IsNext: "+str((next/total)*100)+ "% out of
"+str(total))
      print("Progress: "+str(i)+"/"+str(train_examples))

      feed_dataset = feed_dataset.batch(64,drop_remainder=True)

      bl_model.fit(feed_dataset,callbacks=[generator_callback],
epochs=1, batch_size=64)

      if i%(batch_size*5) == 0:
        bl_model.save("/content/drive/MyDrive/GITT/TFG/models/"+"bl_
model_ult.v3.h5")
```

Código 5-2. Pre-entrenamiento del modelo

5.1.3 Proceso de Pre-entrenamiento

En la Tabla 5-1, podemos observar un resumen de los parámetros generales que hemos usado para pre-entrenar a nuestro modelo.

Pool de datos	20.000
Función Pérdida Masked Language	Sparse Categorical Crossentropy
Función de Pérdida NSP	Binary Crossentropy
Métrica Masked Language	Sparse Categorical Accuracy
Métrica NSP	Binary Accuracy
Optimizer	Adam
Learning Rate	0,001
Épocas por pool	1
Batch	64

Tabla 5-1. Parámetros del pre-entrenamiento

Una vez comenzado el pre-entrenamiento vamos a analizar más en detalle el proceso. Primero he de mencionar que por cada *pool* de datos obtenemos aproximadamente 180.000 pares de frases enmascarados y el tiempo que toma a nuestro modelo realizar una iteración de entrenamiento para cada *pool* es aproximadamente de 4 minutos. Dado que el *dataset* original posee un total de 6.458.670 ejemplos (sin contar los 40.000 últimos que reservaremos para evaluar los resultados del pre-entreno), hemos tardado aproximadamente 34 horas de entrenamiento (sin contar interrupciones) en pre-entrenar a nuestro modelo sobre el *dataset*, que serían aproximadamente 90M de pares de frases tras su pre-procesamiento.

En la Figura 5-1 podemos observar cómo sería el proceso de entrenamiento y como lo monitorizamos, así como los pares de frases que se van generando para asegurarnos que los ejemplos están balanceados.

```
NoNext: 49.956459392517274% out of 191775
IsNext: 50.043540607482726% out of 191775
Progress: 5800000/6418670
   6/2996 [..............................] - ETA: 4:16 - ml_metric: 0.9778 - ns_metric: 0.8737 - ml_loss: 0.1504 - ns_loss: 0.2232
1/1 [==============================] - 3s 3s/step
Masked Sentence: She <mask> born <mask> 1950 .<mask> is <mask> famous <mask>.
Predicted Sentence: she was born in 1950 . it is a famous school .
2996/2996 [==============================] - 239s 74ms/step - ml_metric: 0.9769 - ns_metric: 0.8657 - ml_loss: 0.1668 - ns_loss: 0.3186
```

Figura 5-1. Proceso de pre-entrenamiento

5.1.4 Análisis de Resultados del Pre-entreno

Una vez hemos finalizado el pre-entreno, vamos a evaluar si el modelo ha conseguido o no ese conocimiento general del lenguaje que posteriormente le permita realizar diversas tareas de procesamiento del lenguaje natural, que es nuestro objetivo. Nuestro modelo ha finalizado el pre-entrenamiento logrando aproximadamente un 84% de exactitud en la tarea de lenguaje enmascarado (acertamos el 12,67/15% de los *tokens* enmascarados, los no enmascarados los acierta siempre) y aproximadamente un 85% en *NSP*. Adicionalmente vamos a crear las

siguientes funciones, la primera para obtener la matriz de confusión en la tarea de *NSP* (Código 0-11) y la segunda para comprobar la capacidad de nuestro modelo para predecir correctamente el *token* enmascarado (Código 0-12).

En las Figura 5-2 podemos observar los resultados de ejecutar el Código 0-11 y la representación de estos valores en la matriz de confusión.

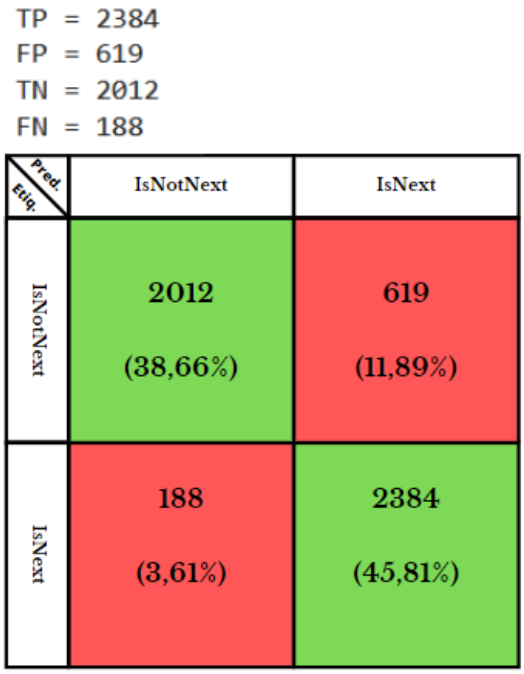

Figura 5-2. Matriz de confusión NSP

Como se puede observar, hemos obtenido una exactitud del 84,47%, también podemos comprobar que el modelo es mucho mejor prediciendo cuando la frase es continuación de la primera que cuando no lo es.

En la Figura 5-3 podemos ver el resultado de ejecutar el Código 0-12 con diferentes frases enmascaradas. Como se puede observar, el modelo es bastante bueno a la hora de predecir *tokens* que podrían ocupar dichas posiciones enmascaradas, es decir, para la primera frase del ejemplo el modelo predice que en la posición de la primera máscara debe de haber un verbo, en la segunda una preposición, en la tercera de nuevo un verbo y por último un sustantivo. Los *tokens* predichos que se muestran son únicamente los que mejor puntuación han obtenido para cada posición. Podemos decir que el modelo genera representaciones informativas acerca del contexto que le permiten predecir que *token* debería ocupar la posición del *token* de *mask* para mantener cierto sentido.

```
Masked Sentence: This <mask> an example <mask> prediction.It is <mask> to predict the <mask>.
1/1 [==============================] - 1s 528ms/step
Predicted Sentence: this is an example of prediction . it is able to predict the problem .
Masked Sentence: he <mask> born <mask> 1950.<mask> was <mask> famous <mask>.
1/1 [==============================] - 1s 532ms/step
Predicted Sentence: he was born in 1950 . he was a famous actor .
```

Figura 5-3. Predicción de tokens enmascarados del modelo pre-entrenado.

Con esta información, podemos concluir que el modelo sí que ha conseguido, en cierto modo, adquirir ese conocimiento general del lenguaje natural y extraer estas representaciones contextuales que posteriormente le permitirán enfocarse en una sola tarea para la que no necesitará ni tantos datos de ejemplos, ni tanto tiempo de entrenamiento.

5.2 Fine-tuning

Una vez disponemos de nuestro modelo ya pre-entrenado, vamos a utilizar estos *embeddings* contextuales, extraídos por nuestro modelo durante el pre-entreno para la tarea de clasificación de los pares de hipótesis y premisas en 3 posibles categorías: relacionadas, neutrales y contradictorias.

5.2.1 Pre-procesamiento del Dataset para el Fine-tuning

Lo primero que haremos será definir el código que utilizaremos para procesar el *dataset* para el *fine-tuning*. Comenzamos extrayendo las hipótesis, premisas y etiquetas de la parte del *dataset* que está en inglés, ya que como nuestro modelo ha sido pre-entrenado con textos en inglés vamos a coger solo este tipo de dato. Posteriormente realizamos un recuento de la cantidad de ejemplos de cada clase dentro de las 3 posibles. Finalmente creamos las secuencias de hipótesis y premisas haciendo uso del *tokenizer* y obtenemos también sus vectores de frase asociados. Ya tenemos tanto los datos de entrada como las etiquetas listas para entrenar a nuestro modelo clasificador. Los primeros 5.000 ejemplos los utilizaremos para el entrenamiento, el resto de los ejemplos, un total del 1870, se usarán para le evaluación del modelo y posteriormente para calcular la matriz de confusión del modelo (Código 5-3).

```
  sc_english_premises =
list(sc_dataset[sc_dataset['language']=="English"].premise)

  sc_english_hypothesis =
list(sc_dataset[sc_dataset['language']=="English"].hypothesis)

  sc_english_labels=list(sc_dataset[sc_dataset['language']=="English
"].label)

  val_0 = 0
  val_1 = 0
  val_2 = 0
  total = 0

  for label in sc_english_labels:
    total+=1
    if label==0:
      val_0+=1
    elif label==1:
      val_1+=1
    else:
      val_2+=1

  print('Type 0 represents the %s percentage of the population' %
(100*val_0/total))

  print('Type 1 represents the %s percentage of the population' %
(100*val_1/total))

  print('Type 2 represents the %s percentage of the population' %
(100*val_2/total))


  data_sequences =
np.array([[tokenizer.encode(sc_english_premises[t],sc_english_hypoth
esis[t]).ids,tokenizer.encode(sc_english_premises[t],sc_english_hypo
thesis[t]).type_ids] for t in range(0,len(sc_english_hypothesis))])

  label_sequences = np.array(sc_english_labels)

  train_sequences = data_sequences[:5000]
  train_labels = label_sequences[:5000]
  test_sequences = data_sequences[5000:]
  test_labels = label_sequences[5000:]
```

Código 5-3. Pre-procesamiento del dataset para el fine-tuning

5.2.2 Traspaso de Conocimientos

Ahora es momento de comenzar con el traspaso de conocimientos de nuestro modelo pre-entrenado a nuestro nuevo modelo clasificador que usaremos para el *fine-tuning*. Para ello definimos una función que cree el modelo clasificador haciendo uso de los *embedings* contextuales generados por nuestro modelo pre-entrenado. Dichos *embeddings* serán utilizados de entrada para una capa de *pooling*, que extraerá los valores medios. Estos valores

serán finalmente la entrada de la capa clasificadora de nuestro modelo, con función de activación *softmax*, que se encargara de determinar a qué clase pertenecen los ejemplos, dentro de las 3 clases posibles. (Código 5-4).

```
  def create_classifier_bl_model(pretrained_bl_model):

      input = layers.Input(shape=(2,MAX_LEN,), dtype='int32')

      pretrained_bl_output = pretrained_bl_model(input)

      pooled_output =
layers.GlobalAveragePooling1D()(pretrained_bl_output)

      result = layers.Dense(3, activation="softmax")(pooled_output)

      classifer_model = keras.Model(input, result)

      return classifer_model
```

Código 5-4. Función para crear modelo clasificador

Ahora podemos definir la función que se encargue de crear y compilar finalmente nuestro modelo clasificador, haciendo uso de nuestro modelo pre-entrenado y la función definida anteriormente. Elegimos también el *optimizer* que usaremos, la función de pérdida y la métrica para el *fine-tuning*. (Código 5-5).

```
  model_folder = "/content/data/models/"
  model_name = "bl_model_ult.v2.h5"

  tpu = tf.distribute.cluster_resolver.TPUClusterResolver()

  print('Running on TPU ', tpu.cluster_spec().as_dict()['worker'])

  tf.config.experimental_connect_to_cluster(tpu)

  tf.tpu.experimental.initialize_tpu_system(tpu)

  tpu_strategy = tf.distribute.TPUStrategy(tpu)

  with tpu_strategy.scope():

      pretrained_bl_model =
tf.keras.models.load_model(model_folder+model_name,custom_objects={'
BertLikeModel':BertLikeModel})

      pretrained_bl_model = tf.keras.Model(pretrained_bl_model.input,
pretrained_bl_model.get_layer("bl_output").output)

      classifier_model =
create_classifier_bl_model(pretrained_bl_model)

      optimizer = tf.keras.optimizers.Adam(learning_rate=5*1e-4)

      classifier_model.compile(optimizer=optimizer,
loss="sparse_categorical_crossentropy",
metrics=["sparse_categorical_accuracy"])

      classifier_model.summary()
```

Código 5-5. Creamos y compilamos el modelo clasificador en la TPU

En la Figura 5-4 se puede observar un esquema de la estructura de nuestro modelo clasificador. Se han probado diferentes arquitecturas siendo esta la que mejores resultados ha obtenido. Hemos probado las dos siguientes topologías adicionales: la primera consistía en añadir una capa FF intermedia con 512 unidades y capa de activación *relu* entre la salida de la capa de *pooling* y la capa clasificadora, en la segunda opción descartada se añadían 2 capas intermedias FF en el mismo lugar que en la opción anterior, la primera con función de activación *relu* y 512 unidades que se usaba de entrada para la segunda con función de activación *tanh* y 256 unidades. Sin embargo, más allá de mejorar el rendimiento del modelo, estas capas adicionales provocaban que hiciese *overfit* más rápido sin aportar ninguna mejora ni en la perdida ni en la exactitud. Por ello se optó por no utilizar ninguna capa intermedia entre la capa de *pooling* y la capa clasificadora.

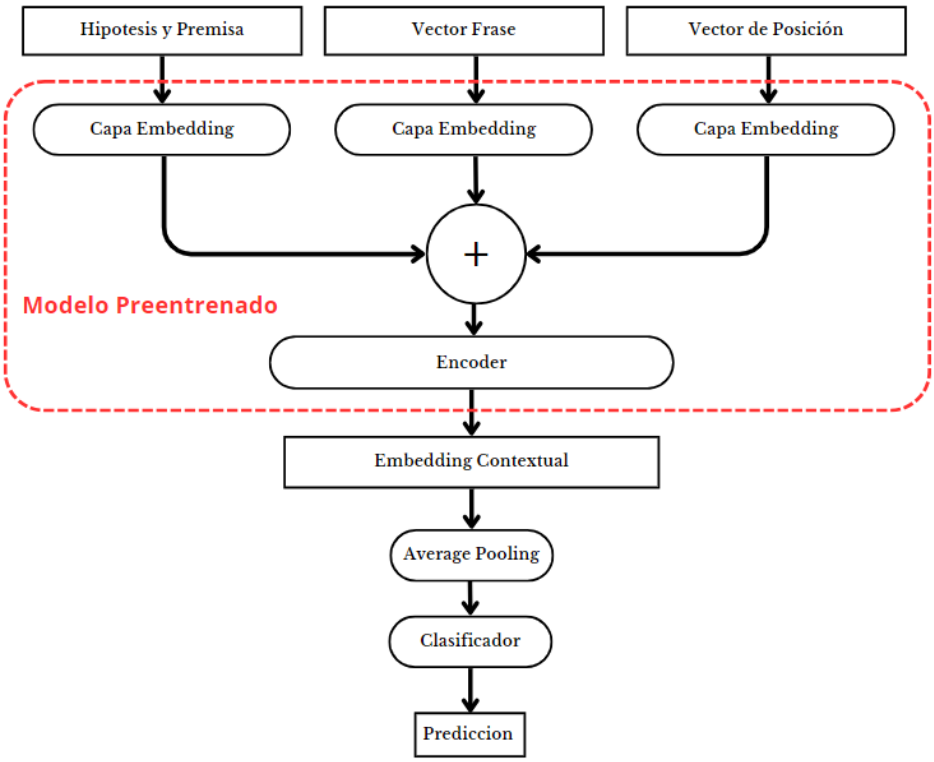

Figura 5-4. Modelo clasificador

Adicionalmente, en la Figura 5-5 tenemos un resumen de las capas que forman el modelo clasificador, así como los parámetros y el número de pesos de cada capa.

```
Model: "model_1"
_________________________________________________________________
 Layer (type)                Output Shape              Param #
=================================================================
 input_1 (InputLayer)        [(None, 2, 256)]          0

 model (Functional)          (None, 256, 512)          15953920

 global_average_pooling1d (G  (None, 512)              0
 lobalAveragePooling1D)

 dense (Dense)               (None, 3)                 1539

=================================================================
Total params: 15,955,459
Trainable params: 15,955,459
Non-trainable params: 0
```

Figura 5-5. Resumen de capas y parámetros del modelo clasificador

5.2.3 Proceso de Fine-tuning

Una vez creado el modelo de clasificación y procesado el *dataset*, podemos proceder con el *fine-tuning* del modelo. A continuación, en la Tabla 5-2 podemos observar los parámetros generales con los que realizaremos el *fine-tuning* de nuestro modelo. Estos parámetros han sido escogidos a base de prueba y error y han resultado ser los que mejores resultados han obtenido. Las primeras pruebas se realizaron con un *learning rate* de 0,0001,

pero este era demasiado pequeño y el modelo no aprendía a un ritmo razonable, la otra opción fue un *learning rate* de 0,001 pero el resultado fue el opuesto, el modelo hacia *overfit* demasiado pronto. Por ello se escogió 0,0005 que es un punto intermedio y se obtuvieron mejorías notables tanto en la perdida como en la exactitud. El *optimizer* así como el número de épocas ha sido el mismo en todas las pruebas. Adicionalmente se probó utilizando un tamaño de *batch* de 8, pero no se obtuvo ninguna mejoría, simplemente aumentaba el tiempo de *fine-tuning*, por lo que se decidió utilizar finalmente 16.

Función Pérdida	Sparse Categorical Crossentropy
Métrica	Sparse Categorical Accuracy
Optimizer	Adam
Learning Rate	0,0005
Épocas	15
Batch	16

Tabla 5-2. Resumen parámetros generales fine-tuning

En el Código 5-6 podemos observar cómo hacemos este *fine-tuning*.

```
   callbacks_list = [
    keras.callbacks.EarlyStopping(
      monitor='val_loss',
      patience=5,
    ),
    keras.callbacks.ReduceLROnPlateau(
      monitor='val_loss',
      factor=0.1,
      patience=5,
    ),
    keras.callbacks.ModelCheckpoint(
      filepath=model_folder+model_name,
      monitor = "val_loss",
      save_best_only = True,
      save_freq = "epoch",
    ),
    ]

   history = classifier_model.fit(train_sequences, train_labels,
epochs=15,
batch_size=16,callbacks=[callbacks_list],validation_data=[test_seque
nces, test_labels])
```

Código 5-6. Fine-tuning del modelo

También definimos los *callbacks* que nos ayudaran a controlar el proceso de *fine-tuning*. Los *callbacks* serían: *EarlyStoping* (tras un determinado número de épocas se finaliza el entrenamiento si no mejora algún parámetro), *ReduceLROnPlateu* (tras un determinado número de épocas se reduce el *learning rate* si no mejora algún

parámetro) y *ModelCheckpoint* (crea un punto de guardado del estado del modelo que mejor rinde). En la Figura 5-6 podemos observar el proceso de *fine-tuning*.

```
Epoch 1/15
313/313 [==============================] - 26s 50ms/step - loss: 1.1024 - sparse_categorical_accuracy: 0.3512 - val_loss: 1.0850 - val_sparse_categorical_accuracy: 0.4011 - lr: 5.0000e-04
Epoch 2/15
313/313 [==============================] - 14s 44ms/step - loss: 1.0693 - sparse_categorical_accuracy: 0.4056 - val_loss: 1.0477 - val_sparse_categorical_accuracy: 0.4524 - lr: 5.0000e-04
Epoch 3/15
313/313 [==============================] - 12s 37ms/step - loss: 1.0352 - sparse_categorical_accuracy: 0.4548 - val_loss: 1.0567 - val_sparse_categorical_accuracy: 0.4262 - lr: 5.0000e-04
Epoch 4/15
313/313 [==============================] - 13s 40ms/step - loss: 1.0140 - sparse_categorical_accuracy: 0.4722 - val_loss: 1.0489 - val_sparse_categorical_accuracy: 0.4636 - lr: 5.0000e-04
Epoch 5/15
313/313 [==============================] - 11s 36ms/step - loss: 0.9835 - sparse_categorical_accuracy: 0.5068 - val_loss: 1.0629 - val_sparse_categorical_accuracy: 0.4545 - lr: 5.0000e-04
Epoch 6/15
313/313 [==============================] - 13s 41ms/step - loss: 0.9453 - sparse_categorical_accuracy: 0.5402 - val_loss: 1.0932 - val_sparse_categorical_accuracy: 0.4428 - lr: 5.0000e-04
Epoch 7/15
313/313 [==============================] - 13s 42ms/step - loss: 0.9018 - sparse_categorical_accuracy: 0.5742 - val_loss: 1.1346 - val_sparse_categorical_accuracy: 0.4246 - lr: 5.0000e-04
```

Figura 5-6. Proceso de fine-tuning

A continuación, podemos observar la gráfica de rendimiento del entrenamiento (73% de los ejemplos) frente a la validación (27% de los ejemplos) por época, tanto para la métrica como la pérdida, que podemos observar en la Figura 5-7 y 5-8. (Código 0-13).

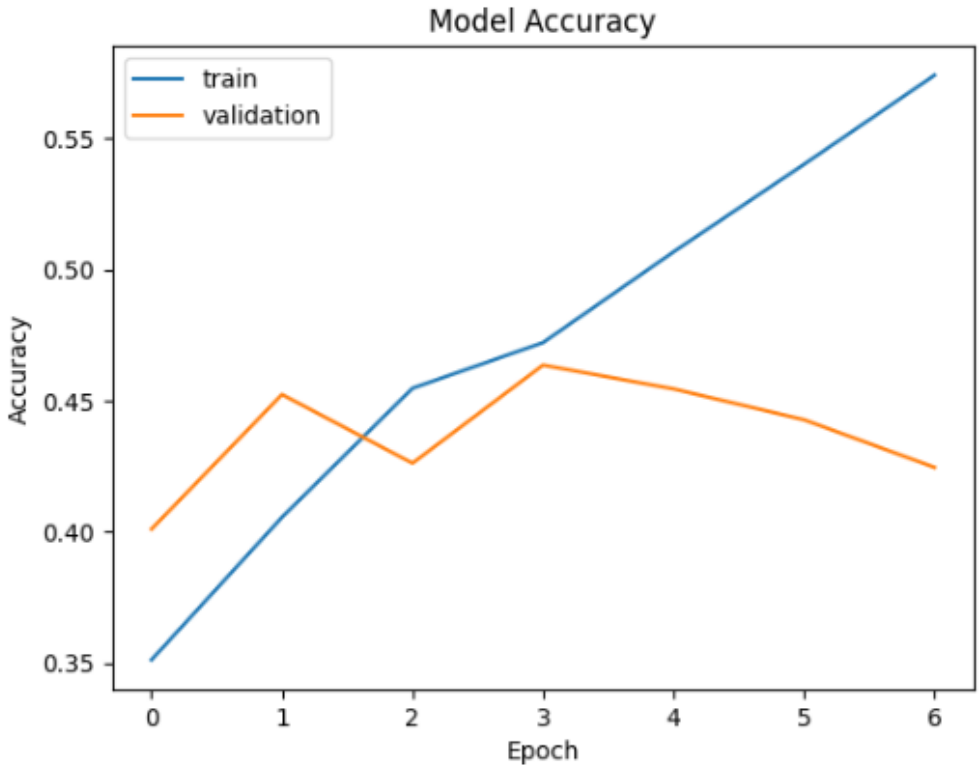

Figura 5-7. Grafica comparativa de la exactitud

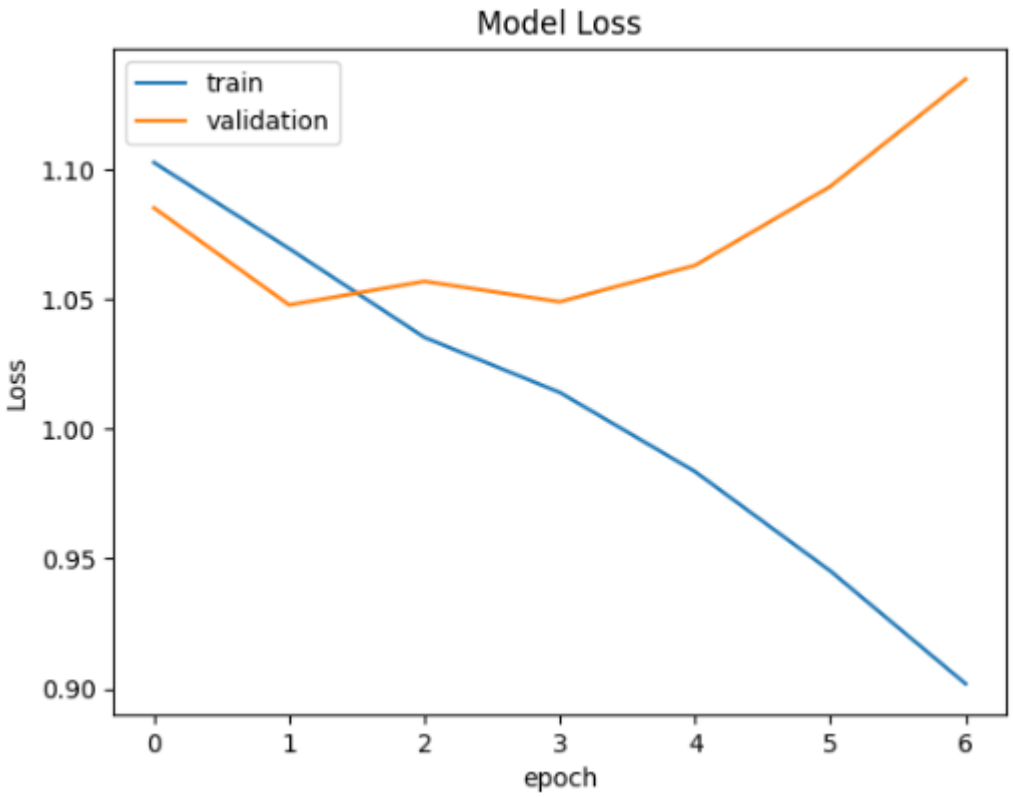

Figura 5-8. Grafica comparativa de la pérdida

5.2.4 **Análisis de Resultados del Fine-Tuning**

Una vez finalizado el *fine-tuning*, vamos a comprobar los resultados del entrenamiento para ver el desempeño de nuestro modelo en la tarea de clasificación de las premisas e hipótesis. Durante el entrenamiento hemos obtenido como máximo una precisión aproximadamente del 46% para la validación. Adicionalmente hacemos uso de la matriz de confusión generada por nuestro modelo clasificador para el *testset*, que nos permitirá comprender en mayor profundidad como procesa y clasifica nuestro modelo esta información (Código 0-14), que podemos ver en la Figura 5-9.

```
Predicted: 0 Label:0: 317
Predicted: 0 Label:1: 226
Predicted: 0 Label:2: 198
Predicted: 1 Label:0: 233
Predicted: 1 Label:1: 228
Predicted: 1 Label:2: 145
Predicted: 2 Label:0: 113
Predicted: 2 Label:1: 109
Predicted: 2 Label:2: 301
```

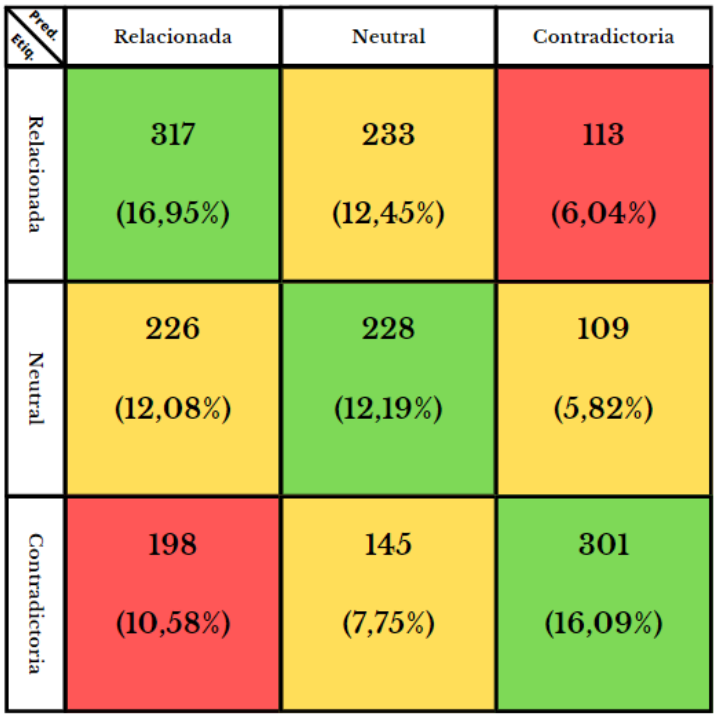

Figura 5-9. Matriz de confusión fine-tuning

Como podemos observar, nuestro modelo diferencia bien cuando una frase es relacionada o es contradictoria, pero encuentra bastantes problemas a la hora de clasificar las hipótesis y premisas si estas son neutrales, lo que tiene cierta lógica ya que semánticamente las frases contradictorias y relacionadas son más diferentes que entre las neutrales.

Un 45,23% de exactitud, que es lo que observamos en la matriz de confusión, no son resultados especialmente satisfactorios, de hecho, se podría considerar que son malos resultados. Uno de los motivos por los que nuestro modelo no rinde correctamente puede ser que su complejidad no es suficiente para realizar esta tarea.

5.3 Fine-tuning de RoBERTa

Para profundizar en las causas vamos a realizar el *fine-tuning* de un modelo *SOTA*, haciendo uso de la misma estructura para el clasificador que hacemos con nuestro modelo, pero con dicho modelo pre-entrenado *SOTA*. En este caso, vamos a utilizar *RoBERTa* para este *fine-tuning*.

Primero lo que haremos es descargar RoBERTa y su *tokenizer* asociado, para ello hacemos uso de las funciones *AutoTokenizer* y *TFAutoModel* de la librería Transformers, para obtener el *tokenizer* y el modelo, respectivamente. Después lo cargamos y compilamos en la TPU para comenzar con el *fine-tuning*. (Código 5-7).

```
  from transformers import TFAutoModel,AutoTokenizer

  tpu = tf.distribute.cluster_resolver.TPUClusterResolver()

  print('Running on TPU ', tpu.cluster_spec().as_dict()['worker'])

  tf.config.experimental_connect_to_cluster(tpu)

  tf.tpu.experimental.initialize_tpu_system(tpu)

  tpu_strategy = tf.distribute.TPUStrategy(tpu)

  roberta_tokenizer = AutoTokenizer.from_pretrained('roberta-large')

  with tpu_strategy.scope():

      roberta = TFAutoModel.from_pretrained('roberta-large')

      input_ids = tf.keras.Input(shape = (MAX_LEN,), dtype =
tf.int32,name='input_ids')

      input_mask=tf.keras.Input(shape=(MAX_LEN,),dtype=tf.int32,name
='input_mask')

      roberta = roberta([input_ids,input_mask])

      roberta_output = roberta[0]

      output =
tf.keras.layers.GlobalAveragePooling1D()(roberta_output)

      hidden_layer =
      output = tf.keras.layers.Dense(3, activation =
'softmax')(output)

      model = tf.keras.Model(inputs = [input_ids,input_mask],
outputs = output)

      model.compile(optimizer =
tf.keras.optimizers.Adam(learning_rate = 1e-5),
                    loss = 'sparse_categorical_crossentropy',
                    metrics = ['accuracy'])

      model.summary()
```

Código 5-7. Descargamos RoBerta, cargamos y compilamos en la TPU

En la Figura 5-10 podemos observar un esquema de la estructura de este otro modelo clasificador que hace uso del modelo pre-entrenado *RoBERTa*. Como se puede observar, salvo por hacer uso de *RoBERTa* en vez de utilizar mi modelo pre-entrenado, la estructura del modelo clasificador es la misma.

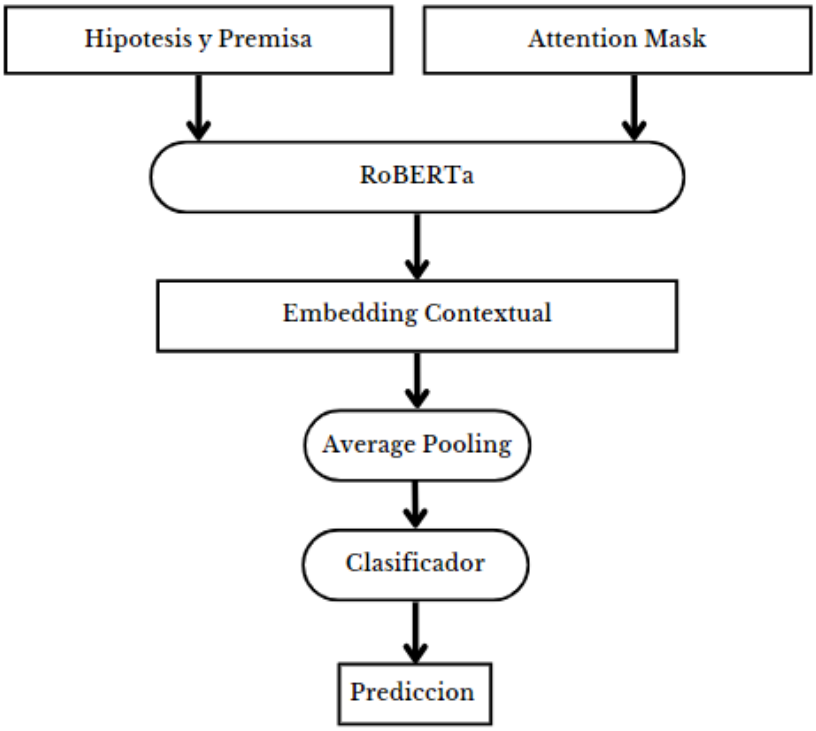

Figura 5-10. Estructura del modelo clasificador con RoBERTa

En la Figura 5-11 podemos observar el resultado de compilar este modelo, así como el número de parámetros que lo conforman por capas y en su totalidad.

```
Model: "model_8"
__________________________________________________________________________________________________
 Layer (type)                   Output Shape         Param #     Connected to
==================================================================================================
 input_ids (InputLayer)         [(None, 256)]        0           []

 input_mask (InputLayer)        [(None, 256)]        0           []

 tf_roberta_model_5 (TFRobertaM  TFBaseModelOutputWi  355359744  ['input_ids[0][0]',
 odel)                          thPoolingAndCrossAt               'input_mask[0][0]']
                                tentions(last_hidde
                                n_state=(None, 256,
                                 1024),
                                 pooler_output=(Non
                                e, 1024),
                                 past_key_values=No
                                ne, hidden_states=N
                                one, attentions=Non
                                e, cross_attentions
                                =None)

 global_average_pooling1d_8 (Gl  (None, 1024)        0           ['tf_roberta_model_5[0][0]']
 obalAveragePooling1D)

 dense_8 (Dense)                (None, 3)            3075        ['global_average_pooling1d_8[0][0
                                                                 ]']

==================================================================================================
Total params: 355,362,819
Trainable params: 355,362,819
Non-trainable params: 0
__________________________________________________________________________________________________
```

Figura 5-11. Resumen parámetros del modelo clasificador con RoBERTa

A simple vista, podemos observar que este modelo es muchísimo más amplio y complejo que el modelo clasificador creado con nuestro propio modelo pre-entrenado, 355M frente a 16M de parámetros. *RoBERTa* (en este caso *RoBERTa large*) hace uso de 24 *encoders* en cascada (nuestro modelo solo usa un *encoder*) y su proceso de pre-entreno se ha realizado con mucha más información textual (además que *RoBERTa* solo se pre-entrena en la tarea de lenguaje enmascarado, no en *NSP*). La otra diferencia con nuestro modelo es que *RoBERTa* usa un *tokenizer BPE*, no *WordPiece* como en el caso de nuestro modelo o de *BERT*. En resumen, este modelo clasificador con *RoBERTa* es 22 veces más amplio que nuestro anterior modelo y bastante más complejo, además de haber sido pre-entrenado con mucha más información.

5.3.1 Proceso Fine-Tuning Modelo Clasificador RoBERTa

Ahora comenzamos el proceso de fine-tuning. En la Tabla 5-3 podemos observar los parámetros para el *fine-tuning*.

Función Pérdida	Sparse Categorical Crossentropy
Métrica	Sparse Categorical Accuracy
Optimizer	Adam
Learning Rate	0,00001
Épocas	5
Batch	16

Tabla 5-3. Parámetros fine-tuning modelo clasificador RoBERTa

En el Código 5-8 podemos observar cómo entrenamos el modelo clasificador *RoBERTa* para el *fine-tuning*. Hacemos uso de estos *callbacks* que para el anterior modelo clasificador. También modificamos los vectores de entrada, en el caso de *RoBERTa* no usa el vector de frase si no el vector de *attention mask* (indica con un valor 1 la posición en el que hay *tokens* con información y 0 el *padding*) y *tokenizamos* de nuevo los pares de premisas e hipótesis con el *tokenizer* de *RoBERTa*.

```
  data_sequences_roberta =
np.array([roberta_tokenizer.encode_plus([sc_english_premises[t],sc_e
nglish_hypothesis[t]],padding='max_length',max_length=MAX_LEN,trunca
tion=True,return_attention_mask=True).input_ids for t in
range(0,len(sc_english_hypothesis))])

  data_attention_roberta =
np.array([roberta_tokenizer.encode_plus(sc_english_premises[t],sc_en
glish_hypothesis[t],padding='max_length',max_length=MAX_LEN,truncati
on=True,return_attention_mask=True).attention_mask for t in
range(0,len(sc_english_hypothesis))])
  label_sequences_roberta = np.array(sc_english_labels)

  train_data_sequences = data_sequences_roberta[:5000]
  train_attention_sequences = data_masking_roberta[:5000]
  train_labels = label_sequences_roberta[:5000]
  test_data_sequences = data_sequences_roberta[5000:]
  test_attention_sequences = data_masking_roberta[5000:]
  test_labels = label_sequences_roberta[5000:]

  history =
model.fit([train_data_sequences,train_attention_sequences],
train_labels, epochs=5,
batch_size=16,callbacks=[callbacks_list],validation_data=[[test_data
_sequences,test_attention_sequences], test_labels])
```

Código 5-8. Fine-tuning de modelo clasificador RoBERTa

En la Figura 5-12 podemos observar el proceso de *fine-tuning.* Y en las Figuras 5-13 y 5-14 observamos una gráfica comparativa de la validación frente al entrenamiento, tanto para la métrica como la pérdida del modelo (Código 0-13).

```
313/313 [==============================] - 241s 276ms/step - loss: 0.8638 - accuracy: 0.5880 - val_loss: 0.5511 - val_accuracy: 0.8048 - lr: 1.0000e-05
Epoch 2/5
313/313 [==============================] - 67s 214ms/step - loss: 0.4374 - accuracy: 0.8372 - val_loss: 0.4688 - val_accuracy: 0.8364 - lr: 1.0000e-05
Epoch 3/5
313/313 [==============================] - 67s 214ms/step - loss: 0.2538 - accuracy: 0.9140 - val_loss: 0.4530 - val_accuracy: 0.8369 - lr: 1.0000e-05
Epoch 4/5
313/313 [==============================] - 96s 307ms/step - loss: 0.1386 - accuracy: 0.9526 - val_loss: 0.4423 - val_accuracy: 0.8358 - lr: 1.0000e-05
Epoch 5/5
313/313 [==============================] - 67s 215ms/step - loss: 0.0936 - accuracy: 0.9718 - val_loss: 0.5174 - val_accuracy: 0.8439 - lr: 1.0000e-05
```

Figura 5-12. Proceso de entrenamiento para el fine-tuning de RoBERTa

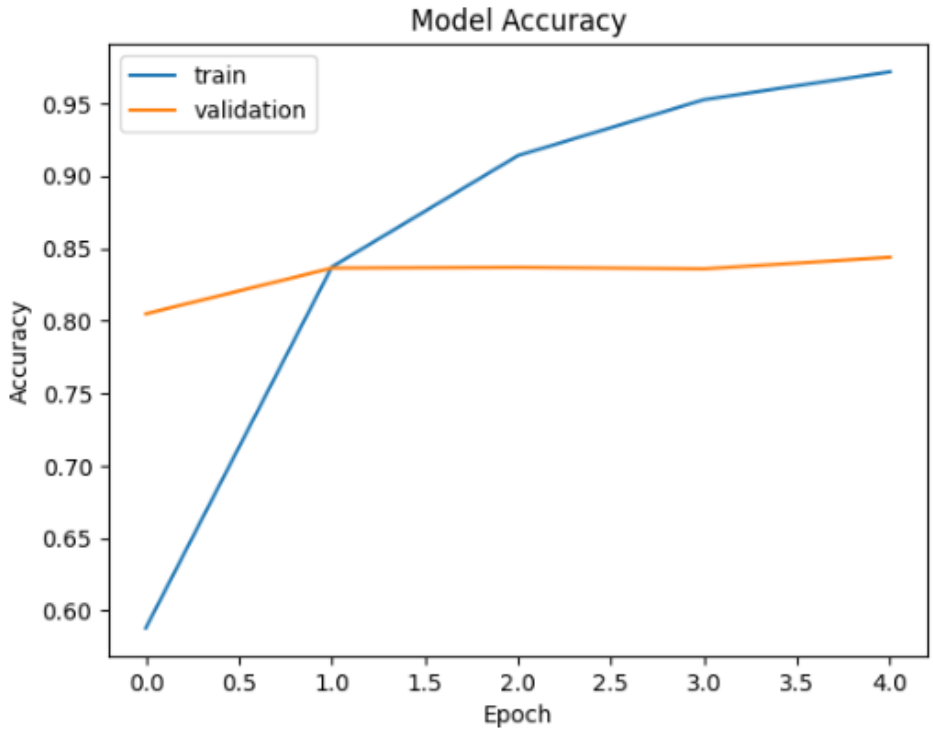

Figura 5-13. Grafica comparativa exactitud en validación frente a entrenamiento de RoBERTa

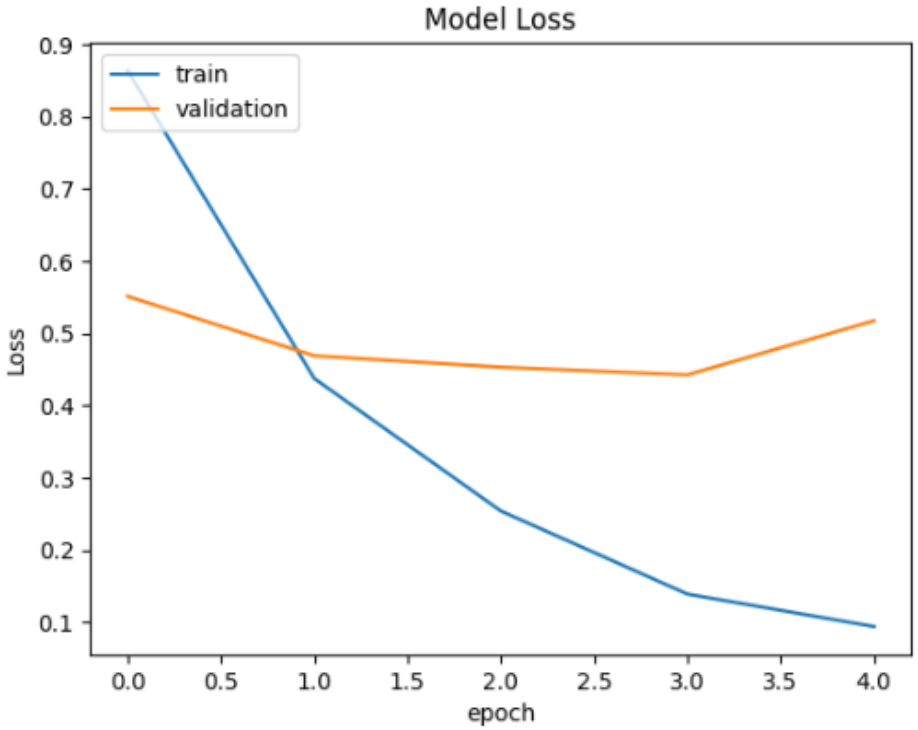

Figura 5-14. Grafica comparativa perdida en validación frente a entrenamiento de RoBERTa

5.3.2 **Análisis de Resultados Fine-tuning de RoBERTa**

Hemos conseguido que nuestro modelo clasificador con *RoBERTa* obtenga un 84,39% de exactitud, lo que es un resultado mucho más óptimo que el que obtuvimos con el modelo pre-entrenado por nosotros mismo.

Finalmente, vamos a mostrar la matriz de confusión generada por las predicciones de *RoBERTa* para el *test-set* y analizar estos resultados. (Código 0-15).

En la Figura 5-15 podemos observar dicha matriz de confusión.

```
Predicted: 0 Label:0: 569
Predicted: 0 Label:1: 67
Predicted: 0 Label:2: 36
Predicted: 1 Label:0: 66
Predicted: 1 Label:1: 425
Predicted: 1 Label:2: 62
Predicted: 2 Label:0: 28
Predicted: 2 Label:1: 71
Predicted: 2 Label:2: 546
```

Pred. / Etiq.	Relacionada	Neutral	Contradictoria
Relacionada	569 (30,42%)	66 (3,52%)	28 (1,49%)
Neutral	67 (3,58%)	425 (22,72%)	71 (3,79%)
Contradictoria	36 (1,19%)	62 (3,31%)	546 (29,19%)

Figura 5-15. Matriz de confusión fine-tuning RoBERTa

Ateniendo a estos resultados, podemos determinar que no solo este modelo clasificador con *RoBERTa* es más grande y complejo, si no que sus resultados son mucho mejores que los que obtuvimos durante el *fine-tuning* de nuestro modelo. Dado que estructuralmente, menos por el número de *encoders* que forma el modelo pre-entrenado u otros hiperparámetros como el número de *attention heads* y la dimensionalidad del *embedding*, *RoBERTa* es muy similar a nuestro modelo pre-entrenado, podemos determinar que esta estructura propuesta, así como este proceso de pre-entreno, si le añadimos el suficiente tamaño y lo entrenamos sobre la suficiente cantidad de información, podemos obtener resultados excepcionales.

Por último, para este análisis de resultados final encuentro interesante ahondar en las propuestas de los mejores competidores de "Contradictory My Dear Watson" de Kaggle. Por ejemplo, el usuario AL TURUTIN [45] (2020) propone un modelo muy similar al mío, el modelo pre-entrenado *XLM-RoBERTa* seguido de una capa de *max pooling* (yo uso *average pooling)* y una capa clasificadora *softmax*, *XLM-RoBERTa* es una modificación de *RoBERTa* desarrollada por A. Conneau et al. [46] (2019). Obtiene el sorprendente porcentaje de un 97,67% de exactitud. Aunque el objetivo de este usuario presentando estos increíbles resultados no es más que demostrar que existe un problema de inferencia ya que el *dataset* de la competición está incluido en los *datasets* usados para entrenar *XLM-RoBERTa*, por lo que no es que este teniendo un increíble rendimiento, sino que está

haciendo *overfiting.* Todos los competidores que han obtenidos resultados tan buenos han realizado el mismo error de utilizar *XLM-RoBERTa* o alguna variación de este, por lo que sus resultados están totalmente sesgados.

Para encontrar un usuario que no haya caído en este sesgo, tenemos que irnos al usuario MARCELLO SUSANTO [47] (2020), que hace uso de *DistilBERT*, una versión compacta del modelo *BERT* (es un 40% más pequeño y un 60% más rápido, manteniendo un 97% de su capacidad, según sus creadores Sanh et al. [48] (2020)) y obtuvo un 78% de exactitud. Otro usuario que no ha caído en el sesgo y ha obtenido buenos resultados es DI [49] (2021) haciendo uso de la versión *baseline* de *BERT* y obteniendo un 76% de exactitud. Ambos utilizan el *token CLS* de los modelo pre-entrenados como entrada para la capa *softmax* clasificadora.

Dejando de lado todos los resultados sesgados, se puede concluir que la mayoría de los competidores han hecho uso de alguna variación de *BERT*, algunos utilizan una capa de *pooling* al igual que yo y otros hacen uso del vector *CLS* de entrada para la capa *softmax* clasificadora.

6 CONCLUSIONES Y FUTUROS PASOS

Durante el desarrollo de este trabajo, hemos aprendido a diseñar e implementar una red neuronal desde cero haciendo uso del *framework* Keras. Partiendo del punto de inicio que proporciona la competición "Contradictory My Dear Watson" ofrecida por Kaggle para la clasificación de pares de frases en tres posibles clases, hemos desarrollado una estructura de red neuronal imitando la arquitectura de BERT y hemos implementado un proceso de pre-entrenamiento muy similar, posteriormente hemos realizado el *fine-tuning* con este modelo en el *dataset* de la competición de Kaggle. Adicionalmente hemos hecho uso de un modelo *SOTA* ya pre-entrenado, como es *RoBERTa*, para realizar este mismo *fine-tuning* y posteriormente poder hacer una comparativa de los resultados con los de nuestro propio modelo.

6.1 Conclusiones

Al comenzar con la investigación teórica para la contextualización de la *IA* y el Aprendizaje Maquina, así como al profundizar en el funcionamiento de las redes neuronales, nos hemos dado cuenta de que nos encontramos con uno de los campos de investigación dentro de la ciencia de la computación más versátiles y potentes, así como más misteriosos y difusos. Un campo que, a diferencia de la mayoría de las disciplinas científica, no sigue a pie de letra un método científico estricto, sino que está influenciado por un carácter más experimental en el que el proceso de prueba y error juega un papel fundamental. Podría considerarse que tiene un cierto carácter artístico.

Aunque se haya demostrado la increíble capacidad que tienen las redes neuronales, estas siguen encontrando grandes barreras a las que nos enfrentamos hoy día. Uno de los mayores inconvenientes que tiene esta disciplina es de la enorme cantidad de potencia de cálculo que necesita, lo que la limita enormemente. Esta necesidad de cómputo, como hemos podido observar durante el desarrollo de nuestro trabajo, juega un papel fundamental a la hora de diseñar las arquitecturas, en nuestro caso, no podíamos entrenar un modelo lo suficiente complejo como para realizar la tarea para la que lo queríamos entrenar, y tuvimos que hacer uso de un modelo ya pre-entrenado por un grupo de investigadores, que, si contaban con esta potencia de cómputo necesaria, para conseguir buenos resultados. El otro gran inconveniente es la información disponible, que, aunque haya aumentado enormemente en los últimos años, no toda esta información es útil o significante para todas las tareas, lo que puede hacerla no apta para entrenar a nuestro modelo dependiendo de su naturaleza. Aquí entra en juego el otro carácter fundamental de las redes neuronales, que es el pre-procesamiento de la información. Este pre-procesamiento incluye numerosas técnicas, como la normalización, el *tokenizado* o el *data augmentation* entre muchas otras. Con una arquitectura apta y datos informativos para la tarea, podemos entrenar un modelo neuronal en casi cualquier tarea, la complejidad está en hallar dicha arquitectura y procesar correctamente estos datos.

Sabemos que el pre-procesamiento de la información con la que entrenaremos a nuestros modelos es igual de importante que la arquitectura de este, pero más importante es aún la función de pérdida que definiremos para entrenar a nuestro modelo. La función de perdida es el elemento más importante, es la referencia que tendrá el modelo de su rendimiento en la tarea que tiene que desempeñar, por lo que sin importar lo muy complejo o apto que sea el modelo o lo amplios e informativos que sean los datos con los que lo entrenamos, una mala función de perdida haría que nuestro modelo sea incapaz de realizar la tarea que queremos.

En relación con nuestro trabajo, estamos muy satisfechos con los resultados obtenidos. Hemos profundizado en las arquitecturas más aptas para el procesamiento del lenguaje natural y hemos reproducido el proceso de entrenamiento que han sufrido casi todos los modelos del estado del arte de la actualizad en el *NLP*. Aunque los resultados obtenidos por nuestro modelo no hayan sido excepcionales, nos ha servido para comprender mejor todo este proceso y finalmente hemos podido hacer *fine-tuning* en un modelo *SOTA* con una arquitectura muy similar (aunque mucho más compleja) a la de nuestro modelo, con el que si hemos conseguido obtener buenos resultados. Adicionalmente hemos mejorado nuestros conocimientos de Python, que antes de comenzar este trabajo eran bastante limitados.

En resumen, este trabajo nos ha servido de toma de contacto al mundo de las redes neuronales y del NLP, dejándonos apreciar el amplísimo y complejo campo que es la IA y el Aprendizaje Maquina, que ya desde hace tiempo, y se acentuara durante los próximos años, revoluciona el mundo de la automatización, el procesado y análisis de la información y la manera en la que los seres humanos nos relacionamos con las maquinas.

6.2 Futuros Pasos

A continuación, vamos a proponer otras dos líneas de investigación.

La primera línea seria hacer uso de diferentes modelos pre-entrenados *SOTA* y realizar una comparativa de cómo estos rinden ante alguna o varias de las múltiples tareas que forman parte del campo del *NLP*. Un ejemplo sería utilizar *BERT, GPT-3, RoBERTa* y/o *DeBERTa* para el *dataset* de Kaggle usado en nuestro trabajo, pero haciendo uso del *dataset* al completo, no solo de las muestras en inglés, es decir usando todos los lenguajes que conforman el *dataset*. También es interesante enfrentar estos modelos a los diferentes *benchmarks* disponibles y comparar sus resultados.

La otra línea de investigación seria profundizar en los modelos de lenguaje generativos, en cómo estos generan nueva información y en las técnicas y métodos que se utilizan para esta rama del *NLP*. Ya que nuestro trabajo se ha centrado exclusivamente en procesar información textual y no en generarla, y que la generación de lenguaje se corresponde con una parte fundamental del *NLP* ya que tiene muchísimas aplicaciones prácticas, esta es la línea de investigación que más interesante me parece para continuar.

REFERENCIAS

[1] J. McCarthy, M. Minsky and C. Shannon, "A proposal for the Dartmouth summer research project on artificial intelligence," Dartmouth College, 1955.

[2] S. J. Russell and P. Norvig, "Artificial intelligence: a modern approach." Pearson Education, 2010.

[3] A. Turing, "Computing Machinery and Intelligence." Mind, vol. 59, no. 236, pp. 433-460, 1950.

[4] I. Goodfellow, Y. Bengio, and A. Courville, "Deep learning." MIT press, 2016

[5] F. Rosenblatt, "The perceptron: A probabilistic model for information storage and organization in the brain." Psychological review, 65(6), 386-408, 1958.

[6] C.M. Bishop, "Neural Networks for Pattern Recognition." Oxford University Press, 1995.

[7] S. Linnainmaa, "The representation of the cumulative rounding error of an algorithm as a Taylor expansion of the local rounding errors." Master's thesis, University of Helsinki, 1970.

[8] S. Raschka and V. Mirjalili, "The Hundred-Page Machine Learning Book." Andrii Shvachko, 2019.

[9] F. Chollet, "Deep Learning with Python," Manning Publications, 2018.

[10] J. Nocedal and S. J. Wright, "Numerical optimization" (2nd ed.). Springer Science & Business Media, 2006.

[11] D.E Rumelhart, G.E. Hinton and R.J. Williams, "Learning representations by back-propagating errors." Nature, 323(6088), 533-536, 1986.

[12] S. Ruder, "An overview of gradient descent optimization algorithms." arXiv preprint arXiv:1609.04747, 2016.

[13] M. Mazur, "A Step by Step Backpropagation Example", 2015. https://mattmazur.com/2015/03/17/a-step-by-step-backpropagation-example/.

[14] A. Géron, "Hands-On Machine Learning with Scikit-Learn, Keras, and TensorFlow: Concepts, Tools, and Techniques to Build Intelligent Systems." O'Reilly Media, 2019.

[15] J. Brownlee, "A Comprehensive Guide to Loss Functions in Machine Learning" Machine Learning Mastery, 2021. https://machinelearningmastery.com/how-to-choose-loss-functions-when-training-deep-learning-neural-networks/.

[16] J. Friedman, T. Hastie and R. Tibshirani, "Regularization Paths for Generalized Linear Models via Coordinate Descent", Journal of Statistical Software, Vol. 33, Issue 1, pp. 1-22, 2010.

[17] N. Srivastava, G. Hinton, A. Krizhevsky, I. Sutskever, and R. Salakhutdinov, “Dropout: A Simple Way to Prevent Neural Networks from Overfittin,” J. Mach. Learn. Res., no. 15, pp. 1929–1958, 2014.

[18] S. Narkhede, "Understanding Confusion Matrix" Medium, 2018. https://towardsdatascience.com/understanding-confusion-matrix-a9ad42dcfd62

[19] C. D. Manning and H. Schütze, "Foundations of statistical natural language processing." MIT press, 1999.

[20] Huggingface, "Building a tokenizer, block by block", 2021. https://huggingface.co/learn/nlp-course/chapter6/8?fw=pt

[21] D. Jurafsky and J. H. Martin, "Speech and language processing." (3rd ed.). Pearson, 2019.

[22] T. Mikolov, K. Chen, G. Corrado and J. Dean, "Efficient estimation of word representations in vector space." arXiv preprint arXiv:1301.3781, 2013.

[23] A.M. Dai and Q.V. Le, "Semi-supervised Sequence Learning." Advances in Neural Information Processing Systems, 2015.

[24] X. Liu, P. He, W. Chen and J. Gao, "A Survey on Pre-training of Deep Learning Models" arXiv preprint arXiv:1904.02289, 2019

[25] S. Hochreiter and J. Schmidhuber, "Long Short-Term Memory" Neural Computation, 1997.

[26] S. Saxena, "Learn About Long Short-Term Memory (LSTM) Algorithms." Analytics Vidhya, 2021.

[27] Y. Kim, "Convolutional neural networks for sentence classification." In Proceedings of the 2014 Conference on Empirical Methods in Natural Language Processing (EMNLP) (pp. 1746-1751), 2014.

[28] M.E. Peters, M. Neumann, M. Iyyer, M. Gardner, C. Clark, K. Lee and L. Zettlemoyer, "Deep contextualized word representations" Proceedings of the 2018 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies, 2018.

[29] A. Vaswani, N. Shazeer, N. Parmar, J. Uszkoreit, L. Jones, A. N. Gomez, Ł. Kaiser and I. Polosukhin, "Attention Is All You Need" Proceedings of the 31st Conference on Neural Information Processing Systems, 2017.

[30] D. Bahdanau, K. Cho and Y. Bengio, "Neural Machine Translation by Jointly Learning to Align and Translate" arXiv:1409.0473, 2015.

[31] G. Kumar and A. Srivastava, "Understanding Metrics for Natural Language Generation" Towardsdatascience, 2019.

[32] A. Wang, A. Singh, J. Michael, F. Hill, O. Levy and S.R. Bowman, "GLUE: A multi-task benchmark and analysis platform for natural language understanding." arXiv preprint arXiv:1804.07461., 2018.

[33] A. Wang, Y. Pruksachatkun, N. Nangia, A. Singh, J. Michael, F. Hill and S. Bowman, "Superglue: A stickier benchmark for general-purpose language understanding systems." arXiv preprint arXiv:1911.11763, 2019.

[34] P. Rajpurkar, J. Zhang, K. Lopyrev, and P. Liang, "SQuAD: 100,000+ Questions for Machine Comprehension of Text." arXiv preprint arXiv:1606.05250, 2016.

[35] A. Williams, N. Nangia, S.R. Bowman, "A Broad-Coverage Challenge Corpus for Sentence Understanding through Inference" arXiv:1704.05426, 2018.

[36] J. Devlin, M.W. Chang, K. Lee and K. Toutanova, "BERT: Pre-training of Deep Bidirectional Transformers for Language Understanding." In Proceedings of the 2019 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies (NAACL-HLT) (pp. 4171-4186), 2019.

[37] Y. Liu, M. Ott, N. Goyal, J. Du, M. Joshi, D. Chen, O. Levy, M. Lewis, L. Zettlemoyer and V. Stoyanov, " RoBERTa: A Robustly Optimized BERT Pretraining Approach" arXiv, 2019.

[38] P. He, X. Liu, J. Gao, W. Chen, "DeBERTa: Decoding-enhanced BERT with Disentangled Attention" arXiv, 2020.

[39] A. Radford, K. Narasimhan, T. Salimans and I. Sutskever, "GPT: Improving Language Understanding by Generative Pre-Training.", 2018.

[40] T. B. Brown, B. Mann, N. Ryder, M. Subbiah, J. Kaplan, P. Dhariwal and D. Amodei, "Language models are few-shot learners." arXiv preprint arXiv:2005.14165, 2020

[41] Y. Sun, S. Wang, Y. Li, S. Feng, X. Chen, H. Zhang, X. Tian, D. Zhu, H. Tian and H. Wu, "ERNIE: Enhanced Language Representation with Informative Entities" arXiv, 2019.

[42] Huggingface, "WordPiece tokenization", 2021. https://huggingface.co/learn/nlp-course/chapter6/6?fw=pt

[43] U. Malapaka, "Using TPUs on Google Colab with Keras", Towardsdatascience, 2020. https://towardsdatascience.com/using-tpus-on-google-colab-966239d24573

[44] R. Gómez, "Understanding Categorical Cross-Entropy Loss, Binary Cross-Entropy Loss, Softmax Loss, Logistic Loss, Focal Loss and all those confusing names" Raúl Gómez Blog, 2018. https://gombru.github.io/2018/05/23/cross_entropy_loss/

[45] AL TURUTIN "Watson :: XLM-R & NLI :: inference", Kaggle, 2020. https://www.kaggle.com/code/alturutin/watson-xlm-r-nli-inference

[46] A. Conneau, K. Khandelwal, N. Goyal, V. Chaudhary, G. Wenzek, F. Guzmán, E. Grave, M. Ott, L. Zettlemoyer and Veselin Stoyanov, "Unsupervised Cross-lingual Representation Learning at Scale", arXiv:1911.02116, 2019.

[47] MARCELLO SUSANTO, "Input configuration benchmark using DistilBERT", Kaggle, 2020. https://www.kaggle.com/code/marcellosusanto/input-configuration-benchmark-using-distilbert

[48] V. Sanh, L. Debut, J. Chaumond and T. Wolf , "DistilBERT, a distilled version of BERT: smaller, faster, cheaper and lighter", arXiv:1910.01108, 2020.

[49] DI, "NLI Beginner: EDA & Bert Baseline", Kaggle, 2021. https://www.kaggle.com/code/jswxhd/nli-beginner-eda-bert-baseline

ANEXO A. CÓDIGOS AUXILIARES

En este anexo se mostrarán todos los códigos auxiliares utilizados durante el desarrollo de nuestro trabajo, y que hemos referenciado en sus apartados correspondientes.

```
!pip install tokenizers
!pip install datasets
!pip install apache_beam
!pip install transformers

import tokenizers
import numpy as np
import pandas as pd
import tensorflow as tf
import io
import torch
import keras

from google.colab import files
from datasets import load_dataset
from torch.utils.data import Dataset
from google.colab import drive
from matplotlib import pyplot as plt
```

Código 0-1. Descargamos e importamos librerías

```
drive.mount('/content/drive/',force_remount=True)
!mkdir data/
!mkdir data/models
!mkdir data/tokenizer
!mkdir data/datasets
!cp -r /content/drive/MyDrive/GITT/TFG/models/ ./data/
!cp -r /content/drive/MyDrive/GITT/TFG/tokenizer/ ./data/
!cp -r /content/drive/MyDrive/GITT/TFG/datasets/ ./data/
```

Código 0-2. Montamos Drive en nuestro entorno Colab

```
ml_dataset = load_dataset("wikipedia", "20220301.en")

ml_train_dataset = ml_dataset['train']

ml_ds_tam = len(ml_train_dataset)

print("Dataset contains: "+str(ml_ds_tam)+" examples")
```

Código 0-3. Obtenemos el dataset para el pre-entreno

```
dataset_folder = 'data/datasets/'

sc_dataset = pd.read_csv(dataset_folder+"train.csv")
```

Código 0-4. Cargamos dataset fine-tuning

```
import re

def ClearDS (dataset):
    paragraphs_list = []
    for example in dataset['text']:
            for paragraph in example.split("\n"):
              if paragraph.isascii():
                 if len(paragraph.split(" ")) > 30:
                   paragraphs_list.append(paragraph.lower())

    return paragraphs_list
```

Código 0-5. Filtrado dataset para entrenar tokenizer

```
class CustomDataset(Dataset):
    def __init__(self, dataset, tokenizer, masking=True):
          self.masking = masking
          self.tokenizer = tokenizer
          self.examples = []
          cleared_txts = []
          if dataset:
            for text in dataset:
              if len(text.split(" ")) > 300:
                paragraphs = []
                for paragraph in text.split("\n"):
                  sentences = []
                  for sentence in paragraph.split(". "):
                    if len(sentence.split(" "))>5:
                      if sentence != paragraph.split(". ")[-1]:
                        sentences.append(sentence+".")
                      else:
                        sentences.append(sentence)
                  if len(sentences)>2:
                    paragraphs.append(sentences)
                if len(paragraphs)>0:
                  cleared_txts.append(paragraphs)
            pairs_list = self.create_pairs(cleared_txts)
            for pair in pairs_list:
              if pair:
                tokenized_sequence =
self.tokenizer.encode(pair.get("A"),pair.get("B"))
                ns_label = pair.get("label")
                example = self.mask_sequence(tokenized_sequence)
                example["sentence_vector"] =
tokenized_sequence.type_ids
                example["ns_label"] = ns_label
                self.examples.append(example)
```

```
      def mask_sequence(self,tokenized_sequence):
        att_mask = np.array(tokenized_sequence.attention_mask)
        example = {}
        example["ml_label"] = np.array(tokenized_sequence.ids,
dtype=int)
        if self.masking:
          input_ids = tokenized_sequence.ids
          if np.where(att_mask==1)[0][-1] == MAX_LEN-1:
              data_len = MAX_LEN
          else:
              data_len = np.where(att_mask==0)[0][0]
          rand = np.random.rand(data_len)
          mask_arr = (rand < .15)
          masked_indexes =mask_arr.nonzero()[0]
           for index in masked_indexes:
            if input_ids[index] != bos_token and input_ids[index] !=
eos_token:
                action = np.random.rand()
                if action < 0.8:
                  input_ids[index] = mask_token
                elif action > 0.8 and action < 0.9:
                  random_token = np.random.randint(5,VOCAB_SIZE-1)
                  input_ids[index] = random_token
                else:
                    pass
          example['input_ids'] = np.array(input_ids, dtype=int)
        else:
          example['input_ids'] = np.array(tokenized_sequence.ids,
dtype=int)
        return example

      def create_pairs(self,texts):
        pairs_list = []
        for i in range(len(texts)):
          text = texts[i]
          for j in range(len(texts[i])):
            sentences = text[j]
            for k in range(len(sentences)-1):
                pair = {}
                action = np.random.rand()
                if action > 0.5:
                    pair["A"] = sentences[k]
                    pair["B"] = sentences[k+1]
                    pair["label"] = 1
                    pairs_list.append(pair)
```

```
                else:
                    random_text_index = i # Texto aleatorio
                    while random_text_index == i:
                      random_text_index =
np.random.randint(len(texts))
                    random_text = texts[random_text_index]
                    random_paragraph_index =
np.random.randint(len(random_text))
                    random_paragraph =
random_text[random_paragraph_index]
                    random_sentece_index =
np.random.randint(len(random_paragraph))
                    random_sentence =
random_paragraph[random_sentece_index]
                    pair["A"] = sentences[k]
                    pair["B"] = random_sentence
                    pair["label"] = 0 # IsNotNext
                    pairs_list.append(pair)
        return pairs_list

    def __len__(self):
        return len(self.examples)

    def __getitem__(self, i):
        return self.examples[i]
```

Código 0-6. Clase CustomDataset

```
  def get_pos_encoding_matrix(max_len, d_emb):

    pos_enc = np.array([[pos / np.power(10000, 2 * (j // 2) / d_emb)
for j in range(d_emb)] if pos != 0 else np.zeros(d_emb) for pos in
range(max_len)])

    pos_enc[1:, 0::2] = np.sin(pos_enc[1:, 0::2])

    pos_enc[1:, 1::2] = np.cos(pos_enc[1:, 1::2])

    return pos_enc
```

Código 0-7. Función para generar vector de posición

```
  def encode_sublayer(input,final=False):

      attention_output =
layers.MultiHeadAttention(num_heads=NUM_HEADS,key_dim=EMB_DIM//NUM_H
EADS)(input, input, input)
      attention_output = layers.Dropout(0.1)(attention_output)
      attention_output = layers.LayerNormalization(epsilon=1e-
6)(input + attention_output)

      ffn = layers.Dense(FF_DIM,
activation="relu")(attention_output)
      ffn = layers.Dense(EMB_DIM)(ffn)
      ffn_output = layers.Dropout(0.1)(ffn)
      if not final:
        output = layers.LayerNormalization(epsilon=1e-
6)(attention_output + ffn_output)
      else:
        output = layers.LayerNormalization(epsilon=1e-
6,name="bl_output")(attention_output + ffn_output)

      return output
```

Código 0-8. Función para crear encoder del transformer

```
  class BertLikeModel(tf.keras.Model):

      def train_step(self, inputs):
          if len(inputs) == 4:
              input_ids, sentence_vector, ml_labels, ns_labels =
inputs

          final_input =
tf.stack([input_ids,sentence_vector],axis=1)

          with tf.GradientTape() as tape:
              predictions = self(final_input, training=True)
              ml_loss = ml_loss_fn(ml_labels, predictions[0], None)
              ns_loss = ns_loss_fn(ns_labels, predictions[1], None)
              total_loss = ns_loss + ml_loss

          trainable_vars = self.trainable_variables
          gradients = tape.gradient(total_loss, trainable_vars)

          self.optimizer.apply_gradients(zip(gradients,
trainable_vars))

          ml_metric.update_state(ml_labels, predictions[0])
          ns_metric.update_state(ns_labels, predictions[1])

          return {'ml_metric': ml_metric.result(), 'ns_metric':
ns_metric.result(), 'ml_loss': ml_loss, 'ns_loss': ns_loss}
```

Código 0-9. Clase BertLikeModel

```
  class MaskedTextGenerator(keras.callbacks.Callback):
      def __init__(self, tokenizer,model,top_k=3):
          self.tokenizer = tokenizer
          self.k = top_k
          self.model=model

      def show_prediction(self,prediction):
          result = np.zeros(256).astype(int)
          i=0
          for pos in prediction:
            top_index = pos.argsort()
            predicted_token = top_index[-1]
            result[i] = int(predicted_token)
            i+=1
          return result

      def on_epoch_end(self, epoch, logs=None):
          first_sentence = "She <mask> born <mask> 1950 ."
          second_sentence = "<mask> is <mask> famous <mask>."
          masked_sentence =
tokenizer.encode(first_sentence,second_sentence)
          printable_sentence = first_sentence+second_sentence
          sentence_vector = masked_sentence.type_ids
          example=np.array([[masked_sentence.ids,sentence_vector]])
          prediction = bl_model.predict(example)[0][0]
          print("Masked Sentence: "+printable_sentence)
          final_prediction = self.show_prediction(prediction)
          print("Predicted Sentence:
"+tokenizer.decode(np.array(final_prediction)).replace(" ##",""))

  generator_callback = MaskedTextGenerator(tokenizer,bl_model)
```

Código 0-10. Callback evaluación predicción de tokens pre-entreno

```
  ml_train_dataset_sfed = ml_train_dataset.shuffle(12345)
  test_dataset = CustomDataset(ml_train_dataset[-
1000:]['text'],tokenizer,False)

  tp=0
  tn=0
  fp=0
  fn=0
  total=0

  for pair in test_dataset:
    example =
np.array([[pair.get('input_ids'),pair.get('sentence_vector')]])
    prediction = bl_model.predict(example)
    total+=1
    prediction = np.round(prediction[1],decimals=0).astype(np.int32)
    if prediction == 0 and pair.get('ns_label') == 0:
      tn+=1
    elif prediction == 1 and pair.get('ns_label') == 1:
      tp+=1
    elif prediction == 0 and pair.get('ns_label') == 1:
      fn+=1
    else:
      fp+=1

  print("TP = "+str(tp))
  print("FP = "+str(fp))
  print("TN = "+str(tn))
  print("FN = "+str(fn))
```

Código 0-11. Función para crear matriz de confusión tarea NSP

```
  def display_prediction(prediction):
      result = np.zeros(256).astype(int)
      i=0
      for pos in prediction:
         top_index = pos.argsort()
         predicted_token = top_index[-1]
         result[i] = int(predicted_token)
         i+=1
      return result

  def test_model(sentence1=None, sentence2=None):
    if sentence1:
      first_sentence=sentence1
    else:
      first_sentence = "he <mask> born <mask> 1950."
    if sentence2:
      second_sentence=sentence2
    else:
      second_sentence = "<mask> was <mask> famous <mask>."
    masked_sentence =
tokenizer.encode(first_sentence,second_sentence)
    printable_sentence = first_sentence+second_sentence
    sentence_vector = masked_sentence.type_ids
    example=np.array([[masked_sentence.ids,sentence_vector]])
    print("Masked Sentence: "+printable_sentence)
    prediction = bl_model.predict(example)[0][0]
    final_prediction = display_prediction(prediction)
    print("Predicted Sentence:
"+tokenizer.decode(np.array(final_prediction)).replace(" ##",""))

  test_model("This <mask> an example <mask> prediction.","It is
<mask> to predict the <mask>.")
  test_model()
```

Código 0-12. Función para evaluar predicciones tokens enmascarados

```
  plt.plot(history.history['sparse_categorical_accuracy'])
  plt.plot(history.history['val_sparse_categorical_accuracy'])
  plt.title('Model Accuracy')
  plt.ylabel('Accuracy')
  plt.xlabel('Epoch')
  plt.legend(['train', 'validation'], loc='upper left')
  plt.show()

  plt.plot(history.history['loss'])
  plt.plot(history.history['val_loss'])
  plt.title('Model Loss')
  plt.ylabel('Loss')
  plt.xlabel('epoch')
  plt.legend(['train', 'validation'], loc='upper left')
  plt.show()
```

Código 0-13. Función para mostrar graficas comparativas

```
label_00=0
label_01=0
label_02=0
label_10=0
label_11=0
label_12=0
label_20=0
label_21=0
label_22=0

total=0

predictions = classifier_model.predict(np.array(test_sequences))

for index in range(0,len(predictions)):
  prediction = predictions[index]
  prediction = prediction.argmax()
  total+=1

  if prediction == 0 and test_labels[index] == 0:
    label_00+=1
  elif prediction == 0 and test_labels[index] == 1:
    label_01+=1
  elif prediction == 0 and test_labels[index] == 2:
    label_02+=1
  elif prediction == 1 and test_labels[index] == 0:
    label_10+=1
  elif prediction == 1 and test_labels[index] == 1:
    label_11+=1
  elif prediction == 1 and test_labels[index] == 2:
    label_12+=1
  elif prediction == 2 and test_labels[index] == 0:
    label_20+=1
  elif prediction == 2 and test_labels[index] == 1:
    label_21+=1
  elif prediction == 2 and test_labels[index] == 2:
    label_22+=1

print("Predicted: 0 Label:0: "+str(label_00))
print("Predicted: 0 Label:1: "+str(label_01))
print("Predicted: 0 Label:2: "+str(label_02))
print("Predicted: 1 Label:0: "+str(label_10))
print("Predicted: 1 Label:1: "+str(label_11))
print("Predicted: 1 Label:2: "+str(label_12))
print("Predicted: 2 Label:0: "+str(label_20))
print("Predicted: 2 Label:1: "+str(label_21))
print("Predicted: 2 Label:2: "+str(label_22))
```

Código 0-14. Función para crear matriz confusión fine-tuning

```
  label_00=0
  label_01=0
  label_02=0
  label_10=0
  label_11=0
  label_12=0
  label_20=0
  label_21=0
  label_22=0

  test_data_sequences = data_sequences_roberta[5000:]
  test_masking_sequences = data_attention_roberta[5000:]
  test_labels = label_sequences_roberta[5000:]
  total=0
  predictions =
model.predict([test_data_sequences,test_masking_sequences])

  for index in range(0,len(predictions)):
    prediction = predictions[index]
    prediction = prediction.argmax()
    total+=1

    if prediction == 0 and test_labels[index] == 0:
      label_00+=1
    elif prediction == 0 and test_labels[index] == 1:
      label_01+=1
    elif prediction == 0 and test_labels[index] == 2:
      label_02+=1
    elif prediction == 1 and test_labels[index] == 0:
      label_10+=1
    elif prediction == 1 and test_labels[index] == 1:
      label_11+=1
    elif prediction == 1 and test_labels[index] == 2:
      label_12+=1
    elif prediction == 2 and test_labels[index] == 0:
      label_20+=1
    elif prediction == 2 and test_labels[index] == 1:
      label_21+=1
    elif prediction == 2 and test_labels[index] == 2:
      label_22+=1

  print("Predicted: 0 Label:0: "+str(label_00))
  print("Predicted: 0 Label:1: "+str(label_01))
  print("Predicted: 0 Label:2: "+str(label_02))
  print("Predicted: 1 Label:0: "+str(label_10))
  print("Predicted: 1 Label:1: "+str(label_11))
  print("Predicted: 1 Label:2: "+str(label_12))
  print("Predicted: 2 Label:0: "+str(label_20))
  print("Predicted: 2 Label:1: "+str(label_21))
  print("Predicted: 2 Label:2: "+str(label_22))
```

Código 0-15. Función para crear matriz de confusión fine-tuning RoBERTa

GLOSARIO

IA: Inteligencia Artificial

NLP: Natural Language Processing

TP: True Positive

TN: True Negative

FP: False Positive

FN: False Negative

FF: Feed Forward

RNN: Recurrent Neural Network

LSTM: Long-Short Term Memory

CNN: Convolutional Neural Network

NER: Named Entity Recognition

POS: Part of Speech

GLUE: General Language Understanding Evaluation

SQUAD: Stanford Question Answering Dataset

MNLI: Multi-Genre Natural Language Inference

BERT: Bidirectional Encoder Representations from Transformers

MLM: Masked Language Model

NSP: Next Sentence Prediction

RoBERTa: Robustly Optimized BERT Pre-training Approach

DeBERTa: Decoding-enhanced BERT with Disentangled Attention

GPT: Generative Pre-trained Transformer

ERNIE: Enhanced Representation through Knowledge Integration

CPU: Central Processing Unit

GPU: Graphics Processing Unit

TPU: Tensor Processing Unit

BPE: Byte-Pair Encoding

BOS: Beginning of Sentence

EOS: End of Sentence

XLM: Cross-lingual Language Model

XNLI: Cross-lingual Natural Language Inference

SOTA: State of the Art