Clasificación sobre Texto, Exploración de técnicas de Procesamiento de Lenguaje Natural y su impacto en el rendimiento sobre los Transformers

(Text Classification, Natural Language Processing techniques exploration, and their impact on Transformers's performance.)

Trabajo de Fin de Máster
para acceder al

Máster en Big Data. Tecnología y Analítica Avanzada

Autor: Revuelta San Emeterio, Fernando

Director: Tomás Herruzo, Elena

Junio – 2023

Resumen

A lo largo de este documento trataremos distintos modelos orientados al procesamiento de lenguaje natural, en especial los Transformers, propuestos en [12], así como diversas técnicas (Fine-Tune del Tokenizer y Data Augmentation) que permitan aumentar el rendimiento de estos y la medición del impacto que generan sobre los mismos.

Palabras clave: Transformer, Tokenizer Fine-Tune, Data Augmentation.

Abstract

Throughout this article we will study in detail differents natural language procesing models, specially Transformers, introduced in [12], diverse techniques (Tokenizer Fine-Tune and Data Augmentation) that will increases their performance as well as the impact generated on them.

Key words: Transformer, Tokenizer Fine-Tune, Data Augmentation.

Índice general

Índice de tablas

Índice de figuras

Capítulo 1

Introducción

El procesamiento de lenguaje natural, es una rama de la inteligencia artificial que se enfoca en la interacción entre los seres humanos y las máquinas a través del lenguaje humano. Su objetivo principal es permitir que las máquinas comprendan, interpreten y generen texto de manera similar a como lo haría un ser humano.

La historia del procesamiento de lenguaje natural inició con el objetivo de explorar cómo las máquinas podrían entender el lenguaje humano. Los primeros enfoques se basaban en reglas gramaticales. Sin embargo, estos métodos resultaron ser demasiado rígidos y difíciles de adaptar a la complejidad y variabilidad del lenguaje natural.

A medida que avanzaba la investigación, surgieron nuevos enfoques basados en el uso de la estadística y modelos probabilísticos. Estos modelos permitieron a las computadoras procesar grandes cantidades de texto y aprender patrones lingüísticos a partir de ellos.

En los últimos años, hemos sido testigos de una evolución impresionante en el campo del procesamiento de lenguaje natural gracias a llegada del aprendizaje profundo y las redes neuronales, especialmente los Transformers [12].

En este TFM, nos enfocaremos en comparar el rendimiento de los Transformers con modelos básicos, como el TF-IDF, para evaluar el impacto de estos avances recientes, así como en emplear técnicas para mejorar y explotar el rendimiento de los mismos.

Antes de adentrarnos en los Transformers, es importante comprender los modelos básicos que han sido fundamentales en el procesamiento del lenguaje natural en los años previos. Uno de ellos es el TF-IDF, una técnica ampliamente utilizada para representar y ponderar términos en un corpus de texto.
Aunque el TF-IDF ha sido efectivo en su momento y ha brindado resultados satisfactorios en algunas tareas, ha quedado demostrado que los Transformers pueden superar sus limitaciones y llevar el procesamiento del lenguaje natural a un nivel completamente nuevo.

La llegada de los Transformers ha sido una auténtica revolución en el proce-

samiento del lenguaje natural. Estos modelos, basados en la atención [1], han demostrado una capacidad excepcional para capturar dependencias a largo plazo y comprender la estructura y el contexto semántico de los textos.
Gracias a esta arquitectura innovadora, los Transformers han logrado superar las barreras de los modelos anteriores y han establecido nuevos estándares en una amplia gama de tareas de procesamiento del lenguaje natural.

En este proyecto, compararemos el rendimiento de los Transformers frente a modelos básicos; del mismo modo, trataremos de exprimir su rendimiento empleando diversas técnicas, todo ello sobre la tarea de clasificación de texto. Utilizaremos métricas de evaluación estándar para medir el rendimiento de los modelos, esto permitirá evaluar objetivamente cómo los Transformers superan a los modelos básicos y como logran proporcionar mejoras significativas en términos de rendimiento y comprensión contextual.

Una de las técnicas clave que exploraremos en este proyecto es el fine-tune del tokenizer en los Transformers. El tokenizer es fundamental para dividir el texto en unidades más pequeñas, como palabras o subpalabras. Al adaptar el tokenizer a un conjunto de datos específico, podemos mejorar la calidad de la tokenización y, por lo tanto, el rendimiento del modelo.
Mediante el fine-tune del tokenizer, buscamos capturar mejor las características y sutilezas específicas del lenguaje en el dominio de interés. Esto nos permitirá optimizar aún más la capacidad de los Transformers para comprender texto con mayor precisión y coherencia.

Además del fine-tune del tokenizer, también aplicaremos técnicas de data augmentation para mejorar el rendimiento de los Transformers. El concepto de data augmentation implica generar datos sintéticos adicionales mediante técnicas como la traducción, sustitución de sinónimos o inserción de ruido en los textos.
Al aumentar el tamaño y la diversidad de los datos de entrenamiento, podemos fortalecer la capacidad de generalización de los Transformers, permitiéndoles aprender patrones más robustos y mejorar su rendimiento en una variedad de tareas del procesamiento del lenguaje natural.

Es importante destacar que los modelos que exploraremos en este proyecto no existían hace unos pocos años. El campo del procesamiento del lenguaje natural ha experimentado un rápido crecimiento y una evolución significativa recientemente, esto se debe en parte al auge de Internet y las redes sociales, gracias a los cuales se generaron grandes cantidades de texto, lo que permitió entrenar modelos a gran escala y mejorar su desempeño.
Adicionalmente, grandes empresas, punteras en el ámbito tecnológico, han invertido considerablemente en investigación y desarrollo de NLP, lo que ha llevado a avances significativos en la comprensión y generación de lenguaje natural.
Los Transformers han desempeñado un papel fundamental en este progreso, demostrando su superioridad en diversas tareas y generando nuevas oportunidades para la aplicación del procesamiento del lenguaje natural en diversos campos.

En la actualidad, el procesamiento de lenguaje natural se aplica en una amplia

variedad de áreas y sectores. A medida que esta tecnología continúa avanzando, es probable que el NLP juegue un papel cada vez más importante en nuestra interacción diaria con las máquinas.

1.1. Objetivos y Motivación

Como ya hemos comentado, este proyecto tiene como objetivo, no solo comparar el rendimiento de los Transformers con modelos básicos, sino también explorar técnicas para mejorar aún más el rendimiento de los mismos, donde pequeños progresos pueden dar lugar a importantes beneficios económicos. El hecho de comprender el impacto de estos avances recientes en el procesamiento del lenguaje natural, permite abrir nuevas posibilidades para aplicaciones más avanzadas y precisas, siendo esta la motivación del mismo.
Con este proyecto, esperamos obtener resultados que respalden la superioridad de los Transformers, así como descubrir en qué medida las técnicas de mejora pueden potenciar aún más su rendimiento.

Capítulo 2

Estado del Arte

La clasificación de texto, dentro del ámbito del procesamiento de lenguaje natural, consiste en asignar automáticamente categorías predefinidas a un texto dado.
El objetivo es identificar y categorizar el contenido del texto en función de distintas características del mismo, como pueden ser su temática o intención entre otras, las cuales explotará el modelo.
Este proceso implica el uso de algoritmos y técnicas de aprendizaje automático para entrenar modelos que puedan reconocer patrones y características sobre el texto.

La clasificación de texto tiene una amplia gama de aplicaciones en diversos campos, el análisis de sentimientos, la detección de spam o la categorización de noticias son algunas de las mas conocidas.
En el contexto actual, la clasificación de texto desempeña un papel crucial en varias áreas. Una de las más importantes se trata del análisis de opiniones en redes sociales, donde debido al gran crecimiento que han experimentado en los últimos años, es posible emplear la clasificación de texto para comprender la opinión de los usuarios en relación con productos, servicios o temas específicos. Esto supone una gran ayuda en la toma de decisiones.

En cuanto a los métodos tradicionales, la mayoría se basan en la representación de diversas características sobre el texto para capturar la información relevante del mismo y posteriormente clasificarlo.
Estas características las podemos agrupar en 3 grandes niveles:

- Características léxicas: Basadas en el vocabulario y la frecuencia de las palabras en el texto. Se pueden utilizar enfoques como la frecuencia de términos.
- Características sintácticas: Basadas en la estructura gramatical del texto. Pueden incluir el análisis de la dependencia sintáctica o etiquetas gramaticales entre otros.
- Características basadas en conocimientos: Utilizan conocimientos específicos del dominio para extraer características relevantes. Uno de los posibles enfoques es el basado en palabras clave.

Una vez obtenidas las características, como ya hemos comentado anteriormente, el siguiente paso es explotar esta representación con algoritmos de clasificación. Algunos de los más empleados en este contexto son SVM o árboles de decisión.

Finalmente, hemos de comentar la existencia de algoritmos que no siguen el esquema anterior, como podría ser Naive Bayes, el cual se basa en el teorema de Bayes y asume independencia entre los sucesos.

Aunque los métodos tradicionales de clasificación de texto han sido ampliamente utilizados y han mostrado buenos resultados, también presentan limitaciones y desafíos.
Uno de los principales inconvenientes es la dificultad para capturar relaciones contextuales, esto limita su capacidad para comprender el significado y la intención detrás de las palabras.
Del mismo modo, algunos métodos tradicionales pueden tener dificultades para manejar grandes volúmenes de datos y volverse computacionalmente costosos, esto se debe en gran medida a la alta dimensión de algunas de las características obtenidas.

A pesar de contar con estos inconvenientes, en los últimos años han surgido técnicas y modelos que consiguen solucionar la gran mayoría de estos.

Uno de los sucesos que supuso un gran avance es la aparición de los denominados word embeddings (expuestos en 4.2), los cuales ofrecen representación para las palabras del vocabulario. Más concretamente, se tratan de representaciones vectoriales densas y de baja dimensionalidad que capturan el significado semántico y las relaciones contextuales entre las palabras. El uso de ellos, permite capturar mejor la similitud y el contexto de los textos.

Sobre estas representaciones, se empezaron a utilizar modelos basados en redes neuronales profundas, que habían ofrecido un rendimiento sobresaliente en una gran cantidad de tareas, entre ellas la clasificación de texto. Entre las arquitecturas empleadas encontramos Redes Neuronales Recurrentes (RNN) [9], las Redes Neuronales Convolucionales (CNN) y los Transformers [12].

Con el objetivo de explotar al máximo los modelos anteriores y reducir el consumo de tiempo y recursos surgió el denominado Transfer Learning (expuesto en 4.7). Esta técnica implica entrenar modelos en grandes volúmenes de texto y luego utilizar esos modelos pre-entrenados como base para tareas específicas, en nuestro caso, clasificar. Esto permite aprovechar el conocimiento lingüístico y las representaciones aprendidas previamente, lo que supone una mejora el rendimiento y la generalización del modelo.

Todas estas técnicas y modelos supusieron un gran avance en términos de rendimiento, lo cual se ve reflejado en los resultados obtenidos.
El objetivo de este proyecto es, por tanto, ratificar y cuantificar la mejora en el rendimiento que supusieron estas nuevas técnicas y modelos, así como aprovechar el máximo rendimiento de los mismos.

Capítulo 3

Modelo Básico (TF-IDF)

A lo largo de este capítulo, analizaremos y desarrollaremos el modelo que tomaremos como baseline, un TF-IDF.

3.1. Tf-Idf

El TF-IDF es una técnica estadística de procesamiento de lenguaje natural que se utiliza para evaluar la importancia relativa de una palabra en un documento en relación con un corpus de documentos. Las siglas TF-IDF provienen de “Term Frequency - Inverse Document Frequency".

Dicha técnica funciona en dos etapas, la primera es calcular la frecuencia de término (TF) de cada palabra en un documento. La segunda es calcular la frecuencia inversa de documento (IDF) de cada palabra en el corpus de documentos.

La frecuencia de término (TF), mide la frecuencia con la que aparece una palabra en un documento específico. Se calcula dividiendo el número de veces que aparece una palabra en un documento por el número total de palabras en el documento. Por ejemplo, si una palabra aparece 10 veces en un documento que tiene 1000 palabras en total, entonces la frecuencia de término de esa palabra sería 0.01 (es decir, 10/1000).

La frecuencia inversa de documento (IDF), mide la singularidad de una palabra en el corpus de documentos. Se calcula dividiendo el número total de documentos en el corpus por el número de documentos que contienen la palabra. En ocasiones, luego se toma el logaritmo natural del resultado para ajustarlo. En cualquier caso, se consigue que las palabras que aparecen en muchos documentos tengan un IDF más bajo que las palabras que aparecen en pocos documentos. Esto es útil para identificar palabras que son relevantes para un documento específico.

Finalmente, para calcular el TF-IDF de una palabra en un documento, se multiplica la frecuencia de término (TF) de la palabra en el documento por la frecuencia inversa de documento (IDF) de la palabra en el corpus de documentos. Esto produce un valor numérico que representa la importancia relativa de la palabra en el documento.

Formalmente, sea $v \in V$ una palabra del vocabulario y $d \in D$ un documento del conjunto de documentos con el que contamos, podemos definir:

$$\text{tf}(v, d) = \frac{N(v, d)}{\sum_{w \in V} N(w, d)} \tag{3.1}$$

$$\text{idf}(v, D) = \frac{|D|}{|\{d \in D : v \in d\}|} \tag{3.2}$$

$$\text{tf-idf}(v, d, D) = \text{tf}(v, d) \cdot \text{idf}(v, D) \tag{3.3}$$

Donde $N(v, d)$ representa el número de veces que aparece la palabra v en el documento d.

Como podemos ver, debido a su naturaleza, se trata de una técnica útil para la clasificación de documentos, la recuperación de información y otras tareas de procesamiento de lenguaje natural.

3.2. Preprocesado

El preprocesado de los datos jugará un papel fundamental tanto en los resultados obtenidos como en la interpretación de los mismos.

En primer lugar, se formatea el texto, para ello, eliminamos ciertos caracteres (por ejemplo *, +, =, ...) que no aportan información lingüística y solamente añadirían ruido al modelo. Tras ello, (según el caso de uso lo requiera) transformamos el texto a minúsculas y eliminamos caracteres especiales (unidecode). Finalmente, eliminamos espacios en blanco innecesarios. Ciertas transformaciones, como por ejemplo el paso a minúscula, son opcionales pues dependen del caso de uso (por ejemplo si necesitamos distinguir nombres propios).

Para realizar las transformaciones indicadas emplearemos paquetes de python como *re* [1] o *unidecode* [2].

Otro de los pasos que realizaremos durante el preprocesado es la definición de los documentos en lo que a TF-IDF se refiere. Como ya se ha comentado anteriormente esta elección afectara a la interpretación de los resultados. En este caso, cada documento estará formado por todos los registros que comparten target, por lo tanto, contaremos con tantos documentos como número de targets distintos existan.

3.3. Selección Tokens

El TF-IDF permite evaluar la importancia relativa de un término en un documento. Por ello, previo a realizar los cálculos hemos de definir con que términos trabajaremos.

[1] Módulo de Python que ofrece operaciones con expresiones regulares: *re*.
[2] Módulo de Python que permite representaciones ASCII de Unidecode: *unidecode*

Para comprender la elección realizada hemos de introducir dos conceptos: Lematización y Stop-Words.

La lematización en el procesamiento de lenguaje natural (NLP) se refiere a la acción de reducir las palabras a su forma base o raíz, también conocida como "lema". Un lema es una forma canónica de una palabra que representa su significado básico, sin tener en cuenta la forma específica que puede tomar en diferentes contextos.
La lematización se utiliza en NLP para normalizar las palabras y reducir la complejidad del texto, lo que facilita su procesamiento. En lugar de tratar cada forma de una palabra como una entidad separada, se reduce la palabra a su forma base, lo que ayuda a reducir el vocabulario y a identificar patrones de palabras más fácilmente.

Las stop-words, por otro lado, son palabras muy comunes en un idioma y que a menudo se eliminan del texto durante el procesamiento del lenguaje natural porque no aportan información semántica útil para la tarea que se está realizando. Son por tanto, palabras que se pueden ignorar sin perder el significado de la oración.

Finalmente, notemos que tanto la lematización como la identificación de stop-words depende del lenguaje, por ello, suponen un mayor trabajo en textos multilingües.

En este caso, se ha tomado la decisión de tomar como términos los lemas de aquellos elementos que contengan palabras y no sean stop-words. Para esta selección de tokens se han empleado módulos de python como *spacy* [3] y *langdetect* [4].

3.4. Entrenamiento e Inferencia

Una vez contamos con los documentos y el dato preprocesado, así como los términos de los mismos ya definidos, podemos ajustar el TF-IDF.

Para cada término en cada documento se calcula su frecuencia dentro del documento, así como su frecuencia inversa a lo largo de todos los documentos.
De este modo obtenemos finalmente una matriz con los documentos (texto asociado a un target) representados en cada fila y las palabras (vocabulario) en las columnas de forma que, en cada celda, encontramos el valor asociado por el TF-IDF. Notemos que se trata de una matriz *sparse* [5].

Veamos como se realiza la asignación de targets a la hora de inferir.
Una vez contamos con un nuevo texto, hemos de realizar los mismos pasos de preprocesado expuesto en la sección anterior. Es importante realizarlos, en otro caso estaríamos comparando elementos distintos.
Tras ello, seleccionamos los tokens de igual modo a lo descrito anteriormente. Importante que sea del mismo modo, en otro caso, de nuevo, estaríamos comparando elementos distintos.

[3] Módulo de Python para el procesamiento de lenguaje natural avanzado: *spacy*
[4] Módulo de Python para la detección de idiomas: *langdetect*
[5] Matriz en la que la mayoría de los elementos toman el valor nulo.

A continuación, obtenemos el valor TF-IDF asociado al nuevo registro, esta información se encuentra recogida en un vector. Notemos que el TF es obtenido del mismo registro, no obstante, el IDF empleado es el del dato de entrenamiento.

Finalmente, asignamos el target cuya representación en la matriz (fila) sea lo más parecida a la representación del nuevo registro. Para ello utilizaremos la *similitud del coseno* [6], que mide cuan parecidos son dos vectores tomando el coseno del ángulo entre ambos.

[6]Medida de similitud entre dos vectores calculando el coseno del ángulo que forman.

Capítulo 4

Red Neuronal (Transformer)

A lo largo de este capítulo detallaremos el funcionamiento y los pasos seguidos para ajustar una Red Neuronal, en este caso un Transformer [12], que permita resolver nuestra tarea.

4.1. Tokenizers

A lo largo del capítulo anterior, hemos podido observar como existen técnicas que permiten asociar a cada elemento del vocabulario una representación que pueda ser comprendida por las máquinas.

Hasta el momento, por simplicidad, hemos considerado que los distintos elementos que conforman nuestro vocabulario son las palabras, no obstante, veremos a continuación que podemos considerar un vocabulario formado por palabras y subpalabras. A estas unidades mínimas del vocabulario las denominaremos tokens. Al proceso de dividir un texto en unidades más pequeñas se le conoce como tokenizar.

Notemos que existen distintas maneras de tokenizar un mismo texto. Tokenizar un texto solamente separando en palabras puede resultar en un vocabulario de grandes dimensiones, dando lugar a un incremento de memoria y complejidad a la hora de asignarles una representación numérica. Del mismo modo, tokenizar un texto por caracteres, a pesar de que reduciría el vocabulario significativamente, no permitiría a dichas representaciones capturar el significado del texto.

Una de las técnicas mas empleadas para tokenizar un texto es conocida como subword tokenization. Esta técnica se basa en la idea de que las palabras que se emplean frecuentemente no han de separarse y aquellas que aparecen rara vez han de separarse en unidades que contengan más significado por si mismas. De esta forma, conseguimos que el tamaño del vocabulario no escale y garantizamos la comprensión de aquellas palabras que se emplean con frecuencia. Adicionalmente, al descomponer ciertas palabras, podemos comprender el significado de palabras nunca vistas antes.

Existen varios tokenizadores siguiendo este principio. Cada modelo de NLP emplea un tokenizador distinto. En función del modelo a emplear, algunos tokenizadores pueden incorporar de forma adicional algún preprocesado (lower case por ejemplo)

o tokens especiales (**[MASK]** por ejemplo, en el caso de Masked languaje modeling).

Veamos a continuación los algoritmos de subword tokenization más conocidos y su entrenamiento y funcionamiento en detalle.

4.1.1. BPE (Byte-Pair Encoding)

BPE [11] toma como base un tokenizador que separa el texto de entrenamiento en palabras. Tras ello, se obtiene un conjunto de palabras y la frecuencia de cada una de ellas.

A continuación, se descomponen dichas palabras en los símbolos que las componen. Estos pares compuestos por los símbolos que conforman una palabra y su frecuencia constituirán el conjunto de entrenamiento del modelo. Adicionalmente se crea un vocabulario con todos los símbolos encontrados. Con esta información, BPE aprende reglas de agregación que permiten ampliar el vocabulario hasta alcanzar el tamaño deseado de la siguiente manera:

1. Apoyándose en los pares definidos previamente para facilitar el cómputo, se obtiene la frecuencia de cada par de símbolos consecutivos en el texto de entrenamiento.

2. La pareja con mayor frecuencia constituirá la primera regla de agregación. Sea c_1, c_2 la pareja con mayor frecuencia, se agruparan en el símbolo c_1c_2.

3. Se añade c_1c_2 al vocabulario.

4. Se aplica dicha regla sobre el conjunto de entrenamiento (pares de símbolos y frecuencias). Esto permitirá en posteriores iteraciones crear tokens de longitud superior a 2.

Estos pasos se repetirán hasta obtener la longitud de vocabulario deseada. La longitud del vocabulario viene dad por: |Conjunto de símbolos iniciales|+|Conjunto de reglas|.

Para tokenizar un nuevo texto, basta descomponerlo en símbolos y aplicar las reglas de agregación aprendidas en el orden de creación. En caso de aparecer un símbolo no visto antes, este se remplazará por el token **[UNK]**.

Por último, considerar una variante de BPE denominada Byte-level BPE, el cual utiliza los bytes como vocabulario base.

4.1.2. WordPiece

WordPiece [10] se trata de un algoritmo muy parecido a BPE, en primer lugar inicializa un vocabulario con todos los símbolos presentes en el texto de entrenamiento y de forma iterativa aprende una secuencia de reglas de agregación que permiten ampliar el vocabulario.

La única diferencia la encontramos a la hora de escoger la pareja de símbolos que constituirá una nueva regla de agregación. En el caso de BPE, se escogía la pareja

más frecuente en el texto de entrenamiento, en este caso, se escoge la pareja que maximice la verosimilitud del texto de entrenamiento una vez ha sido añadido al vocabulario el nuevo elemento resultante, esto es:

$$(c_1^*, c_2^*) = argmax_{c_1,c_2} \left(\frac{P(c_1 c_2)}{P(c_1) \cdot P(c_2)} \right) \tag{4.1}$$

Donde, la probabilidad de un símbolo viene dada de la siguiente manera:

$$P(c) = \frac{N(c)}{N} \tag{4.2}$$

Siendo $N(c)$ el número de ocurrencias del símbolo en el texto de entrenamiento y N el número total de símbolos en el mismo.

Este será el tokenizador empleado en el modelo final.

4.1.3. Unigram

Continuamos con este algoritmo, que sigue una filosofía similar a los dos anteriores aunque con un enfoque distinto, Unigram [5] inicializa un vocabulario base con un gran número de símbolos e iterativamente elimina una selección de estos hasta obtener un vocabulario del tamaño deseado.

Para inicializar el vocabulario base existen distintas estrategias, por ejemplo, se podrían escoger las subpalabras más frecuentes.

Sea $x_1, x_2, ..., x_N$ las palabras que forman nuestro texto de entrenamiento, dado un vocabulario y sea $S(x_i)$ el conjunto de todas las tokenizaciones posibles para x_i en dicho vocabulario, definimos la siguiente función de perdida:

$$L = -\sum_{i=1}^{N} log \left(\sum_{x \in S(x_i)} P(x) \right) \tag{4.3}$$

Donde $P(x)$ para una tokenización $x = [s_1, s_2, ..., s_m]$ viene dada por:

$$P([s_1, s_2, ..., s_m]) = \prod_{i=1}^{m} P(s_i) = \prod_{i=1}^{m} freqRel(s_i) \tag{4.4}$$

Siendo $freqRel(s_i)$ la frecuencia relativa de dicho símbolo en el texto de entrenamiento.

Notemos que dicha expresión premia a un vocabulario que permita generar varias tokenizaciones probables para las distintas palabras del texto de entrenamiento.
Por su parte, una tokenización es más probable cuantos menos símbolos tenga y más probable sea cada uno de estos. Esta definición de probabilidad ofrece un balance en la longitud de los símbolos.

Una vez definida dicha función de pérdida, el algoritmo computa para cada símbolo el incremento que se produciría en la función de pérdida si este fuese eliminado

del vocabulario. Tras ello se eliminan del vocabulario el $p\,\%$ (tomando p un valor entorno al 10-20 %) de símbolos que producen un menor incremento en la función de pérdida, es decir, aquellos menos significativos dado el texto de entrenamiento.

En el vocabulario base de Unigram siempre encontramos los caracteres básicos, esto permite tokenizar cualquier palabra.

Finalmente, a la hora de tokenizar una palabra pueden existir varias maneras de hacerlo, en este caso no se siguen reglas de agregación y los símbolos que encontramos en el vocabulario final pueden dar lugar a estas situaciones. Para escoger una de las posibles tokenizaciones, Unigram almacena la probabilidad de cada símbolo. De este modo, es posible computar la probabilidad de cada una de las posibles tokenizaciones y escoger una en función de estos valores, ya sea escogiendo la más probable o muestreando de entre ellas.

4.1.4. SentencePiece

Por último, hemos de considerar que no todos los idiomas separan sus palabras empleando espacios, para resolver este problema surge SentencePiece [6].

Este, trata todo el texto de entrenamiento como una única cadena de caracteres, considerando el espacio como un símbolo más. Tras ello entrena un modelo de, o bien BPE, o bien Unigram, sobre dicha cadena.

Finalmente, notemos que debido a que no se consideran espacios, no es posible aprovechar la estrategia de separar el texto de entrenamiento en palabras para el cálculo de frecuencias (como se hacía en BPE).

4.2. Word Embeddings

En la sección anterior hemos observado como, a partir de un texto, este podía descomponerse en unidades. No obstante, para que una máquina pueda comprenderlo, hemos de asignar a cada unidad una representación. El objetivo de los word embeddings es exactamente ese, buscar una representación del vocabulario. Son estas las representaciones empleadas en diversos modelos de NLP.

Como ya hemos comentado, en primer lugar se ha de definir el vocabulario con el que se trabajará, formado por un conjunto de tokens. Cuando encontremos elementos que no pertenezcan a nuestro vocabulario, se empleará el token **[UNK]** para representarlo.

Consideremos a continuación, distintas técnicas existentes para su representación.

4.2.1. One-hot Vectors

Los One-hot Vectors se tratan de la primera y más sencilla aproximación. Dado un vocabulario V, cada palabra se encuentra representada por un vector de longitud

$|V|$ con todas sus coordenadas 0 salvo la posición i-ésima, distinta para cada elemento del vocabulario.

Dicha representación conlleva consigo distintos problemas. En primer lugar, la longitud de los vectores se corresponde con el tamaño del vocabulario, en la práctica este puede escalar rápidamente y resultar en vectores de gran longitud. El otro de los problemas principales se debe a que esta representación no consigue aprender el significado de las palabras, resultando así en que todas las palabras se encuentren a la misma distancia unas de otras, no obstante, sabemos que existen palabras más cercanas a otras en términos semánticos.

4.2.2. Count-Based Methods

Notemos que la noción de significado y semántica se puede obtener a partir del contexto en el que se emplean las palabras, las palabras que se usan en contextos similares tendrán por tanto un significado similar. Este es el procedimiento que emplean las personas intuitivamente y emplearán los métodos que veremos a continuación.

Con la premisa de capturar el contexto de las palabras en vectores, emplearemos la información que podemos inferir de diversos documentos. A dicha colección de documentos la denominaremos como corpus.

El procedimiento general en este tipo de métodos consiste en generar una matriz de palabras versus contextos. Dicha matriz puede ser reducida posteriormente, pues la mayoría de las palabras aparecen en un conjunto reducido de contextos, dando lugar así a una matriz con una gran cantidad de elementos sin información (*sparse*).

Para establecer la similitud entre contextos y palabras se pueden emplear medidas como, por ejemplo, la similitud del coseno.

Dadas estas nociones, para establecer un Count-Based method, hemos de definir los dos siguientes elementos:

- Contexto.
- Medida de similitud, esto dará lugar los elementos de la matriz ya mencionada.

En función de estas dos características podemos definir distintos modelos recogidos en la siguiente tabla:

Nombre	Contexto	Medida
Co-Ocurrence Counts	Ventana de longitud L	$N(v,c)$
PPMI	Ventana de longitud L	$PPMI(v,c)$
LSA	Documento	$tf-idf(v,d,D)$

Donde, $N(v,c)$ representa el número de veces que la palabra v aparece en el contexto c, $tf-idf(v,d,D)$ coincide con la definición de 3.3 y:

$$PPMI(v,c) = \text{máx}\left(0, \log\left(\frac{N(v,c)\cdot|(v,c)|}{N(v)\cdot N(c)}\right)\right) \tag{4.5}$$

Siendo $|(v, c)|$ el cardinal de todas las palabras y contextos que aparecen en el corpus.

4.2.3. Prediction-Based Methods (Word2Vec)

En este tipo de métodos el objetivo es el mismo, capturar la información de los contextos de cada palabra. Estos se aprenderán tratando de predecirlos.

Uno de los métodos más populares es Word2Vec [7]. Este cuenta con los vectores de palabras como parámetros, que serán aprendidos de forma iterativa tratando de optimizar una función objetivo la cual forzará a los vectores de palabras a aprender la información del contexto en el que aparecen.

Dado un corpus para entrenar el modelo, el proceso a alto nivel sería el siguiente:

1. Establecer una ventana deslizante (fijar longitud) y recorrer el corpus definiendo en cada iteración una palabra central y un contexto delimitado por dicha ventana.

2. Computar las probabilidades de que aparezcan las palabras del contexto, dada la palabra central.

3. Entrenar el modelo buscando maximizar dichas probabilidades.

Como ya hemos comentado al inicio de esta subsección, buscamos optimizar una función objetivo. Esta se trata del Likehood, donde se busca maximizar la probabilidad de que aparezca un elemento del contexto dada la palabra central. La expresión de dicha función presenta problemas numéricos, para solventaros se empleará una función estrictamente creciente como es el logaritmo. Además se considerará su negativa, de este modo buscamos minimizar en vez de maximizar la función. Esta nueva expresión se conoce como Negative Log-Likelihood funtion y para un corpus con $i \in \{1, ..., n\}$ posiciones y una ventana de tamaño l tiene la siguiente expresión:

$$NLL(w_1, .., w_n) = -\sum_{t=1}^{n} \sum_{j \in \{-l,...,l\}} log(P(w_{t+j}|w_t, \theta)) \tag{4.6}$$

Donde θ representa los parámetros del modelo, es decir, los vectores de palabras [1].
En cuanto a la definición de la probabilidad que aparece en 4.6, se empleará el producto vectorial entre las palabras (esto permite medir la similitud entre ambas) y se normalizará el resultado con la función softmax empleando el resto de palabras del vocabulario, esto es:

$$P(w|c) = \frac{e^{v_w^t v_c}}{\sum_{p \in V} e^{v_p^t v_c}} \tag{4.7}$$

En cuanto al entrenamiento del modelo, este se realizará empleando el descenso del gradiente palabra a palabra, es decir, para cada par palabra central y una de las palabras que aparecen en su contexto.
Observando 4.6 y 4.7, es claro que se consigue aumentar la probabilidad de que la

[1] Cada palabra se encuentra representada por dos vectores, en función de si forma parte del contexto o se trata de la palabra principal. Por simplicidad no tendremos en cuenta este detalle.

palabra del contexto aparezca junto con la central y disminuir las probabilidades de que el resto de palabras del vocabulario aparezcan junto a la central. A pesar de que puedas disminuir la probabilidad de que una palabra aparezca en el contexto de otra cuando es el caso, este efecto se compensa al realizar varias actualizaciones.
Por último, en cada iteración hemos de actualizar un número de vectores proporcional a la longitud del vocabulario. Para paliar este problema se han desarrollado técnicas como por ejemplo, Negative Sampling, donde solo se actualizan un subconjunto de vectores seleccionados aleatoriamente.

4.2.4. Global Vectors (GloVe)

El último de los métodos que comentaremos es GloVe [8]. Este combina los conceptos vistos en los Count-Based y Prediction-Based methods, es decir, obtiene la información de las estadísticas de un corpus (al igual que los Count-Based methods) y los vectores de palabras son obtenidos empleando el descenso del gradiente (al igual que en los Prediction-Based methods).

La función de pérdida es la siguiente:

$$\sum_{c,w \in V} f(N(c,w)) \cdot (v_w^t v_c + b_c + b_w - log(N(c,w)))^2 \tag{4.8}$$

Donde $N(c,w)$ indica el número de veces que la palabra w aparece en el contexto de la palabra c, los términos b_c y b_w se tratan de sesgos que han de ser aprendidos por el modelo también y f es definida de la siguiente forma:

$$f(x) = \begin{cases} (x/x_{max})^\alpha & x < x_{max} \\ 1 & \text{Otro caso} \end{cases} \tag{4.9}$$

Para unos valores de x_{max} y α establecidos previamente.
Notemos que el uso de f permite, por un lado penalizar a los eventos poco frecuentes y por otro, no sobredimensionar aquellos eventos muy frecuentes.

Al definir la función de pérdida de este modo, forzamos a que las palabras que aparecen en los mismos contextos se parezcan para no penalizar en la función.

4.3. Seq2Seq

En el campo del procesamiento de lenguaje natural se define una tarea denominada *seq2seq* (sequence-to-sequence), esta se refiere a la generación de una secuencia de salida a partir de una secuencia de entrada. En otras palabras, el objetivo es crear una relación entre dos secuencias las cuales pueden tener diferentes longitudes e incluso que pueden pertenecer a diferentes idiomas.

Formalmente, dada una secuencia $x = x_1, ..., x_n$, el objetivo es encontrar una secuencia $y^* = y_1, ..., y_m$ de forma que:

$$y^* = argmax_y P(y|x) \tag{4.10}$$

Por ejemplo, en la tarea de traducción automática, la secuencia de entrada es una oración en un idioma fuente, y la secuencia de salida es la oración correspondiente en un idioma objetivo. En este caso, la tarea de *seq2seq* consiste en aprender esta relación entre los idiomas para poder traducir automáticamente oraciones de un idioma a otro.

El entrenamiento de los modelos *seq2seq* se realiza maximizando la probabilidad de la secuencia de salida condicionada a la secuencia de entrada, es decir, se busca maximizar la probabilidad de generar la secuencia de salida correcta dada la secuencia de entrada, como se puede apreciar en 4.10. Para ello, se utiliza una función de pérdida, como la pérdida de entropía cruzada, que mide la discrepancia entre la distribución de probabilidad predicha por el modelo y la distribución de probabilidad real de la secuencia de salida.
Siguiendo la misma notación, dado un tiempo t, en el que se han predicho los $t-1$ elementos anteriores de la secuencia, definimos $P_t = (v|y_1, ..., y_{t-1}, x_1, ..., x_n)\ \forall v \in V$, donde V es el vocabulario. Del mismo modo, definimos $P_t^* = \text{one-hot}(y_t)$, entonces podemos establecer como función de pérdida la entropía cruzada:

$$\text{Loss}(P_t^*, P_t) = -\sum_{i=1}^{|V|} P_t^*(i) \cdot \log(P_t(i)) \tag{4.11}$$

4.4. Modelos de lenguaje

Un modelo de lenguaje en el campo del Procesamiento del Lenguaje Natural es un tipo de modelo estadístico que se utiliza para estimar la probabilidad de una secuencia de palabras dada. En términos simples, un modelo de lenguaje intenta predecir la probabilidad de que una determinada secuencia de palabras aparezca en el lenguaje natural, dado cierto contexto.

Debido a lo comentado anteriormente, es claro que los modelos de lenguaje están fuertemente relacionados con la tarea de *seq2seq*, pues permiten abarcarla.

El objetivo principal de un modelo de lenguaje es aprender la distribución de probabilidad de las secuencias de palabras. Esto se logra mediante el entrenamiento del modelo en grandes corpus de texto, que se utilizan para aprender la frecuencia y distribución de las secuencias y de las palabras que las conforman.

Existen varios tipos de modelos de lenguaje, que varían en su complejidad y capacidad para modelar el lenguaje natural. Los modelos más simples, como los modelos de *n-grams*, utilizan una ventana de n palabras para predecir la siguiente palabra en una secuencia empleando métodos count-based. Estos modelos son relativamente simples y rápidos de entrenar, pero tienen limitaciones en su capacidad para modelar secuencias de palabras más complejas y largas.
Los modelos de lenguaje más avanzados, como los modelos basados en redes neuronales, utilizan una arquitectura más compleja para modelar el lenguaje natural. Estos modelos suelen estar compuestos por redes neuronales de varias capas, que procesan la entrada para aprender patrones en las secuencias de palabras.

4.5. Encoder-Decoder y Atención

Una de las arquitecturas más utilizadas en el procesamiento del lenguaje natural, es la conocida como encoder-decoder, empleada comúnmente para resolver tareas *seq2seq* debido a su estructura y naturaleza. En el contexto del NLP, el encoder toma una secuencia de entrada y la transforma en una representación vectorial de alta dimensión que contiene información semántica y sintáctica sobre la misma. El decoder, por su parte, se encarga de generar una nueva secuencia a partir de la representación ofrecida por el encoder.

Las redes neuronales recurrentes (RNN) [9] se utilizan comúnmente como encoder y decoder para el procesamiento del lenguaje natural. De este modo, el encoder toma una secuencia y la transforma en una representación vectorial de alta dimensión, que se utiliza como entrada para el decoder.

En el contexto de las RNN, el encoder consta una capa de neuronas, que procesan la entrada de manera recursiva. Cada neurona toma la entrada actual y la salida de la neurona anterior, utilizando esta información para actualizar su estado interno. En otras palabras, cada neurona aprende a partir de la información anterior y la entrada actual, lo que permite que la RNN capture las relaciones y significados en la secuencia de entrada.
Una vez que se ha procesado toda la secuencia de entrada, la última salida del encoder se utiliza como representación vectorial de la entrada. Esta representación contiene información semántica y sintáctica sobre la secuencia inicial, se utilizará como entrada para el decoder.
El decoder también es una RNN, que toma la representación vectorial generada por el encoder como entrada y genera la salida deseada.

Esta estrategia tiene asociada un problema, el último estado puede olvidar la información de los elementos previos de la secuencia.
Como solución a este problema, podemos usar dos RNN, una que lee la entrada de izquierda a derecha y otra que lee la entrada de derecha a izquierda. Luego, podemos usar los estados finales de ambos modelos, uno recordará mejor la parte final de un texto y el otro el comienzo.
Adicionalmente, para obtener una mejor representación del texto, se pueden apilar varias capas. En este caso, las entradas para las RNN superiores son las representaciones provenientes de la capa anterior.
La idea es que, al contar con varias capas, las inferiores capturarán fenómenos locales (por ejemplo, frases), mientras que las capas superiores podrán aprender cosas más generales (por ejemplo, temas).

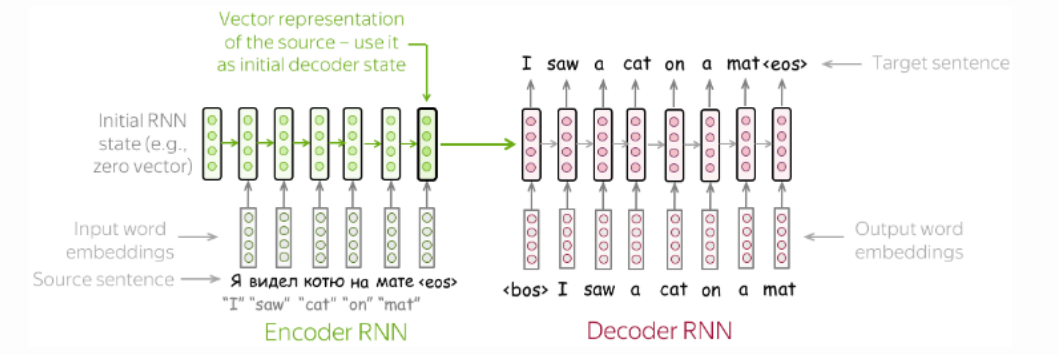


Figura 4.1: Arquitectura de una RNN simple [13]

Finalmente, introducimos el mecanismo de atención, propuesto en [1].

Desde un punto de vista de alto nivel, el mecanismo de atención permite que el modelo se centre en cada paso en diferentes partes de la secuencia de entrada.

Un mecanismo de atención es una parte de una red neuronal. En cada paso del decoder, decide qué partes de la secuencia original son más importantes. En este caso, el encoder necesita proporcionar representaciones para todos los elementos de la secuencia original (por ejemplo, los distintos estados de la RNN).

En cada paso del decoder, la atención funciona de la siguiente manera:

1. Se reciben los inputs necesarios para el mecanismo de atención: estado del decoder en el instante t, h_t, así como todos los estados del encoder $s_1, ..., s_n$
2. Se computan los scores de atención para cada estado del encoder, el objetivo es asignar a cada uno de ellos la importancia de los mismos con respecto el estado del decoder en este instante.
3. Se define una distribución de probabilidad sobre los estados del encoder aplicando softmax a sus respectivos scores de atención.
4. Se obtiene el output de la atención como la suma de los estados del encoder ponderada por la distribución calculada en el paso anterior.

En cuanto a como calcular los scores de atención, las técnicas más distendidas son:

- Producto escalar: $h_t^T \cdot s_i$
- Bilineal: $h_t^T \cdot W \cdot s_i$ para una matriz W dada.
- MLP: $w_2 \cdot \tanh(W_1[h_t, s_i])$ para una matriz W_1 y un vector w_2 dados, donde $[h_t, s_i]$ representa la concatenación de ambos vectores.

El resultado de la atención se puede incluir en el modelo de diversas formas, ya sea tomando la atención obtenida en $t-1$ como input del paso t, o bien combinando la atención obtenida en el paso t junto con su representación del decoder, h_t, para

obtener el siguiente elemento de la secuencia generada.

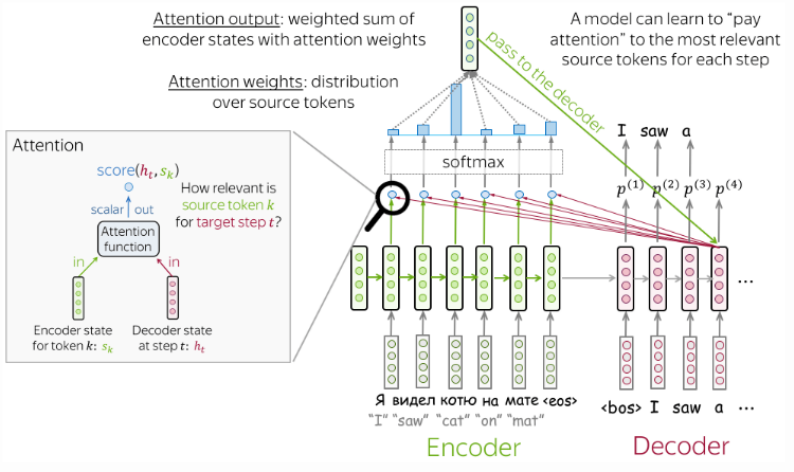


Figura 4.2: Mecanismo de atención sobre una RNN [13]

La idea principal es que la red puede aprender qué partes de la entrada son más importantes en cada paso. Como todo aquí es diferenciable (función de atención, softmax y el resto de operaciones), se puede entrenar un modelo con atención end to end, de este modo, el modelo mismo aprenderá a seleccionar la información importante.

4.6. Attention is All You Need

Los Transformer son un modelo presentado en el paper "Attention is All You Need"[12] en 2017, estos siguen la estructura encoder-decoder introducida en la sección anterior. Los Transformers se basan únicamente en mecanismos de atención como los presentados previamente. Adicionalmente, gracias a su estructura, el modelo es más rápido de entrenar hasta en un orden de magnitud, esto lo veremos más adelante.

Actualmente, los Transformers son modelos en el estado del arte, principalmente en tareas *seq2seq* como por ejemplo el modelado del lenguaje, la traducción automática o la generación de respuestas a preguntas entre otros.

Los Transformers introdujeron un nuevo paradigma, en contraste con los modelos anteriores donde el procesamiento dentro del encoder y decoder se realizaba empleando recurrencia o convolución, los Transformers opera solo con atención.

Cuando se codifica una secuencia, las RNN no entenderán el significado de todos sus elementos hasta que lean toda la oración, esto puede suponer mucho tiempo para secuencias largas, así como una perdida de información. Por otro lado, en el encoder del Transformer, los tokens interactúan entre sí en cada instante.

Intuitivamente, se puede pensar en el encoder de un Transformer como una secuencia de pasos de razonamiento (capas). En cada paso, los tokens se miran entre sí (aquí es donde se aplicara la denominada self-attention), intercambian información e intentan entenderse mejor en el contexto de toda la secuencia. Este proceso sucede en cada capa o paso de razonamiento.

Por otro lado, en cada capa del decoder, los tokens interactúan entre sí aunque cada token solamente puede interactuar con los previos, de esta forma evitamos ver el futuro, esto sucede mediante un mecanismo denominado masked self-attention. Adicionalmente, miran los estados finales del encoder empleando nuevamente mecanismos de atención.

4.6.1. Self-Attention

El mecanismo de Self-Attention es uno de los componentes clave de los Transformers. La diferencia entre el mecanismo de atención y el de self-attention es que este último opera entre elementos de la misma secuencia, es decir, todos los estados del encoder en alguna capa.

Este mecanismo es la parte del modelo que permite que los tokens interactúen entre sí. Cada token observa al resto de tokens en la secuencia, obtiene información del contexto y actualiza la representación anterior de sí mismo.

En la práctica, gracias a la notación matricial que veremos a continuación, esto sucede en paralelo.

Formalmente, este paradigma se implementa con una atención de Query-Key-Value. Cada token de entrada posee tres representaciones, cada una de ellas se corresponde con el rol que desempeña:

- Query: Consultar información.
- Key: Indicar si posee información.
- Value: Proporcionar información.

Estas representaciones se obtienen empleando trasformaciones lineales a partir de la representación original (Una matriz por representación).
La representación de Query se utiliza cuando un token mira al resto buscando información para entenderse mejor a sí mismo. La representación de Key se emplea para responder a la solicitud de la consulta, permite calcular los pesos de atención. Finalmente, la representación de Value se utiliza para calcular la salida de atención, da información a los tokens que la solicitan.

Formalmente, dada una secuencia de longitud l, sea Q $(l \times d_k)$ la matriz que contiene los vectores de Query por fila, sea K $(l \times d_k)$ la matriz que contiene los vectores de Key por fila y sea V $(l \times d_v)$ la matriz que contiene los vectores de Value por fila también.

El resultado de la atención para dicha secuencia viene dado por:

$$\text{Attention}(Q, K, V) = \text{softmax}\left(\frac{QK^T}{\sqrt{d_k}}\right)V \tag{4.12}$$

De este modo cada fila de la matriz obtenida contiene el resultado de la atención para cada elemento de la secuencia.
Notemos que la función softmax se aplica de forma independiente para cada fila.

4.6.2. Masked Self-Attention

En el decoder, también encontramos un mecanismo de self-attention, este se encarga de mirar los tokens anteriores, el resto no se pueden mirar, en otro caso contaríamos con información de los elementos a predecir.

Por ello, en el decoder este algoritmo es ligeramente diferente. Mientras que en el encoder se reciben todos los tokens a la vez y estos pueden mirar al resto de tokens de la secuencia de entrada, en el decoder no es posible, ya que a la hora de inferir generamos un token a la vez, no sabemos qué tokens generaremos en el futuro. No obstante, durante el entrenamiento, contamos con las secuencias objetivo completas, lo cual puede ser explotado para acelerarlo.

Notemos que durante el entrenamiento, en caso de emplear self-attention en el decoder, sería alimentado con toda la secuencia objetivo y los tokens verían el futuro, esto no es lo que queremos, pues como hemos comentado, al inferir no contamos con esta información.
Para evitar que durante el entrenamiento el decoder mire hacia adelante, el modelo utiliza masked self-attention.

Formalmente, dada la matriz resultante de la siguiente operación:

$$\frac{QK^T}{\sqrt{d_k}} \tag{4.13}$$

Basta con establecer a $-\infty$ los elementos de de la diagonal superior derecha, al aplicar softmax, dichos pesos que se corresponden con los elementos posteriores de la secuencia, quedarán establecidos a 0, es decir, no se obtendrá información de los mismos. La diagonal no es necesario enmascararla, pues la secuencia objetivo se encuentra desplazada a las derecha un elemento (como elemento inicial se toma un token especial, **[BOS]** , que representa el inicio de una secuencia) y se trata por tanto del último elemento de cuya información disponemos para predecir el siguiente.

4.6.3. Multi-Head Attention

Por lo general, comprender el papel de un elemento en una secuencia requiere comprender cómo está relacionado con diferentes partes de la misma.
Esto es importante, no solo en el procesamiento de la secuencia, sino también en la generación de la secuencia objetivo. En el Castellano, por ejemplo, el sujeto de una oración se relaciona con los artículos determinantes, verbos o adjetivos. Cada elemento en una secuencia forma parte de múltiples relaciones relaciones dentro de la misma.

El hecho de permitir que el modelo se centre en diferentes relaciones es la motivación detrás de multi-head attention.
En este mecanismo, la atención tiene varias cabezas, cada una de ellas trabaja de forma independiente.
Esto se implementa como varios mecanismos de atención cuyos resultados se combinan.

Formalmente, consideramos las matrices Q, K y V previamente definidas. En este caso, la dimensión de todas ellas será $(l \times d_{model})$, donde l es la longitud de la secuencia a procesar y d_{model} un entero que establece la longitud de los vectores pertenecientes a las matrices anteriores, este será denominado como la dimensión del modelo. El objetivo es transformar los vectores de Q, K y V para mandarlos a espacios con d_k, d_k y d_v dimensiones respectivamente, para ello, emplearemos 3 proyecciones, una por matriz. Esto se realizará h veces, se necesitarán por tanto $3 \cdot h$ proyecciones, de esta forma, cada tripleta de proyecciones asociadas a una de las h transformaciones permitirá definir una cabeza.
Basta aplicar la función de atención definida en 4.12 para cada transformación, obteniendo h matrices $l \times d_v$, cada una de estas define una cabeza.

$$head_i = \text{Attention}(QW_i^Q, KW_i^K, VW_i^V), \quad i \in \{1, ..., h\} \tag{4.14}$$

Por último, se concatena cada vector con los correspondientes en las distintas matrices obtenidas y se proyectan, pasando de las $h \cdot d_v$ dimensiones de los vectores concatenados a d_{model}.

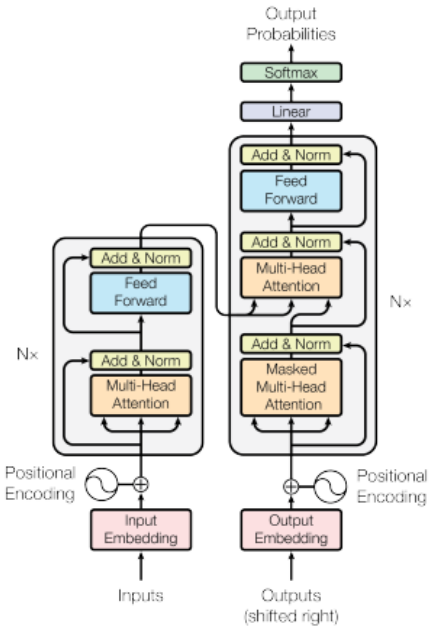
$$\text{MultiHead}(Q, K, V) = \text{Concat}(head_1, ..., head_h)W^O \tag{4.15}$$

En cuanto a las proyecciones utilizadas, estas son matrices de parámetros $W_i^Q \in \mathbb{R}^{d_{model} \times d_k}$, $W_i^K \in \mathbb{R}^{d_{model} \times d_k}$, $W_i^V \in \mathbb{R}^{d_{model} \times d_v}$ y $W^O \in \mathbb{R}^{h \cdot d_v \times d_{model}}$.

Notemos que, gracias a W^O, los modelos con una sola cabeza de atención o varias de ellas tienen el mismo tamaño, por ello, la atención de varias cabezas no aumenta el tamaño del modelo.

4.6.4. Arquitectura

Una vez entendemos los componentes principales del modelo y sus respectivas funciones, echemos un vistazo al modelo completo.



Desde un punto de vista de alto nivel, el modelo hace exactamente lo ya comentado, en el encoder los tokens se comunican entre sí y actualizan sus representaciones, en el decoder cada token de la secuencia objetivo primero mira los tokens previamente generados, luego a los tokens de la secuencia origen y finalmente actualiza su representación. Esto se reproduce tantas veces como capas se hayan establecido en cada componente.

Veamos con más detalle el resto de componentes del modelo.

Bloques Feed-Forward:

Además de los mecanismos de atención, cada capa tiene un bloque de feed-forward (FFN). Estos se componen de dos capas lineales aplicando ReLU entre ellas.

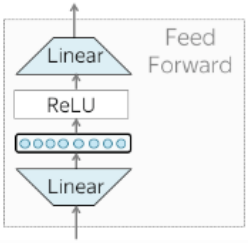


Figura 4.3: Arquitectura de un bloque Feed-Forward [13]

Los modelo usan estos bloques para procesar la nueva información, permiten aprender representaciones no lineales de los datos.

Residual connections y Normalization

Las conexiones residuales se tratan de una estrategia distendida y útil, facilitan el flujo de gradiente a través de una red, evitando problemas asociados al gradiente (gradient vanishing y gradient exploding). Adicionalmente, permiten ver información de capas previas, especialmente útil cuando se trabaja con redes muy profundas. Estas se basan en añadir la entrada de un bloque a su salida, de esta forma la información fluye a través de la capa sin ser modificada.

Formalmente se define de la siguiente manera:

$$\text{Layer}(x) = x + \text{Sublayer}(x) \tag{4.16}$$

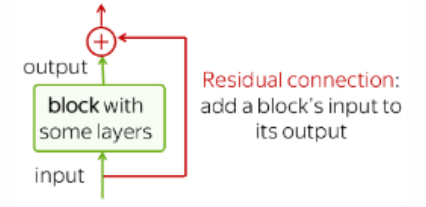


Figura 4.4: Arquitectura de las conexiones residuales [13]

La parte de Normalization se realiza de manera independiente para cada elemento de una secuencia (Layer Normalization).

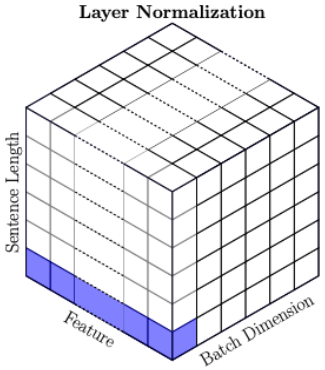


Figura 4.5: Elementos sobre los cuales actúa la normalización.

Consiste en normalizar las representaciones vectoriales, esto mejora la estabilidad de la convergencia y en ocasiones, la calidad de las soluciones.
En el modelo se normaliza la representación vectorial de cada elemento de la secuencia. Además, existen varios parámetros entrenables, *scale* y *bias*, utilizados después de la normalización para reescalar las salidas obtenidas.

Formalmente, sea $x = [v_1, ..., v_n]$ donde v_i se corresponde con la representación vectorial del elemento i-ésimo de una secuencia, definimos:

$$\text{norm}(v) = \frac{1}{\sigma_k} \cdot (v - \mu_k) \tag{4.17}$$

$$\text{rescale}(v) = v \cdot scale + bias \tag{4.18}$$

$$\text{LayerNorm}(x) = [\text{rescale}(\text{norm}(v_1)), ..., \text{rescale}(\text{norm}(v_n))] \tag{4.19}$$

Donde μ_k y σk son la media y varianza de los valores de v respectivamente. Notemos que los parámetros *scale* y *bias* pertenecen a la capa en cuestión, no dependen de la secuencia ni los tokens.

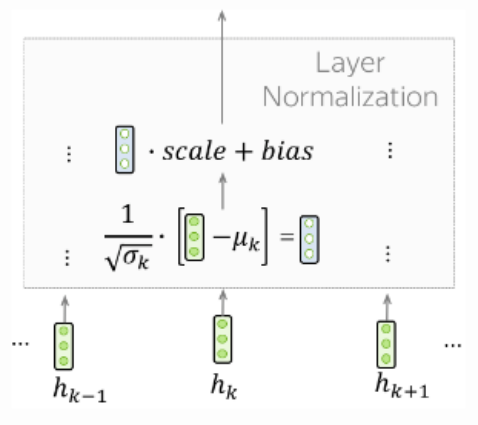


Figura 4.6: Representación del proceso seguido en la Layer Normalization.

En la arquitectura final, observamos que se utilizan después de cada bloque de atención y FFN (Add & Norm).

$$\text{Add \& Norm}(x) = \text{LayerNorm}(x + \text{Sublayer}(x)) \tag{4.20}$$

Positional encoding

Notemos que este modelo no contiene información del orden de los tokens de entrada, pues no posee recurrencia o cualquier estructura que le permita inferirla. Por lo tanto, tenemos que hacer de manera explícita que conozca las posiciones de los mismos. Para ello, añadiremos a los embeddings originales unos nuevos, embeddings posicionales, que codifiquen las posiciones. La representación final de un token será por tanto la suma de ambos.

Las codificaciones posicionales empleadas en el modelo son:

$$PE_{pos,2i} = sin\left(\frac{pos}{10000^{\frac{2i}{d_{model}}}}\right) \tag{4.21}$$

$$PE_{pos,2i+1} = cos\left(\frac{pos}{10000^{\frac{2i}{d_{model}}}}\right) \tag{4.22}$$

Donde *pos* es la posición en la secuencia e i se trata de un índice que permite identificar la posición en el vector. La elección de estas funciones permite a la red

diferenciar claramente entre posiciones en la secuencia, ya que cada una tiene un patrón de onda diferente. Además, al usar una progresión geométrica, la red puede generalizar a secuencias más largas.

4.6.5. Complejidad del entrenamiento

Una vez conocemos la arquitectura del modelo, hemos de comentar una de las ventajas fundamentales de la misma, la velocidad del entrenamiento. El Transformer no tiene recursividad, por ello es más eficiente computacionalmente.

En el encoder se emplea Multi-Head Self-Attention, esto implica obtener las representaciones Q, K, V, realizar para cada cabeza las proyecciones y aplicar la función de atención, es decir, productos de matrices y aplicar la función softmax. Como cada cabeza trabaja de forma independiente, el proceso se puede paralelizar para finalmente concatenar los resultados y proyectarlos como se ve en 4.15. Este proceso se realizará tantas veces como número de capas en el encoder se hayan establecido, el cual es un número finito independiente de la secuencia de entrada.

En cuanto al decoder, en este aplicamos Multi-Head Masked Self-Attention, esto requiere de seguir los mismos pasos mencionados en el encoder, enmascarando el resultado antes de aplicar softmax. Adicionalmente, en el decoder se aplica Multi-Head Attention entre el encoder y el decoder, como ya hemos visto requiere de las mismas operaciones. Estos mecanismos se aplicarán tantas veces como capas en el decoder se hayan establecido, nuevamente un número finito independiente de la secuencia de entrada.

Finalmente, hemos de tener en cuenta el resto de componentes definidas en la sección anterior, las cuales se basan en realizar productos de matrices, aplicar ReLU, computar medias y varianzas, escalar vectores y crear y sumar los embeddings posicionales.

Por todo ello, y teniendo en cuenta que para una secuencia de longitud l se realizan l predicciones durante una iteración del entrenamiento, es claro que el coste computacional del modelo es bajo, siendo este un proceso altamente paralelizable y dependiendo únicamente de funciones básicas como las ya expuestas.

4.7. Transfer Learning

Transfer learning es una técnica de aprendizaje automático en la que se utiliza el conocimiento adquirido por un modelo sobre una tarea para mejorar el rendimiento en otra tarea relacionada. En lugar de entrenar un modelo desde cero para cada tarea, transfer learning utiliza el conocimiento previo adquirido por el modelo en una tarea anterior para mejorar el rendimiento en otra tarea relacionada.

Un ejemplo de transfer learning sobre texto es el uso de modelos pre-entrenados de lenguaje natural, como BERT [3] o GPT-3 [2]. Estos modelos se entrenan en

grandes conjuntos de datos de texto, como Wikipedia [14] o la *World Wide Web* [2], y aprenden a representar el lenguaje capturando relaciones semánticas y sintácticas.

Como ya expusimos en 4.4, un modelo de lenguaje es un modelo estadístico que asigna una probabilidad a una secuencia de palabras dado un histórico de palabras previo.
El entrenamiento de modelos de lenguaje implica alimentar al modelo con una gran cantidad de texto. De esta forma, son entrenados para predecir la probabilidad del siguiente token en una secuencia, dados los tokens previos. Esto permite al modelo aprender las relaciones entre las palabras y las estructuras gramaticales del lenguaje, lo que a su vez le permite generar texto coherente y natural.

Luego, estos modelos pre-entrenados se pueden ajustar a tareas más específicas de procesamiento de lenguaje natural, como la clasificación de sentimientos o la traducción de idiomas, entre otros. El fine-tune de estos modelos pre-entrenados para tareas específicas, se realiza mediante el entrenamiento en un conjunto de datos más pequeño y específico para la tarea en cuestión.

Observamos por tanto, que el proceso de transferencia de conocimiento en NLP generalmente implica dos etapas: pre-entrenamiento y afinamiento.
En la primera etapa, se entrena un modelo de lenguaje general en un gran conjunto de datos no supervisado, como la Wikipedia o un corpus de texto similar.
Durante el pre-entrenamiento, el modelo aprende representaciones del lenguaje de una forma más general y abierta, permitiendo que este generalice mejor para otras tareas.
En la segunda etapa, se ajusta el modelo pre-entrenado en un conjunto de datos más pequeño y específico para la tarea deseada. En este paso, se descongelan y se ajustan algunos parámetros del modelo pre-entrenado. Esto permite que el modelo aprenda representaciones de lenguaje más específicas para la tarea en cuestión, lo que conduce a un mejor rendimiento en la misma.

Notemos que la estrategia de Transfer learning trae consigo varios beneficios, especialmente en el procesamiento de lenguaje natural, donde podemos destacar:

- Mejora del rendimiento: El transfer learning ha demostrado ser muy efectivo mejorando el rendimiento en una amplia variedad de tareas de procesamiento de lenguaje natural, incluyendo la clasificación y generación de texto, así como la traducción. Los modelos pre-entrenados aprenden representaciones del lenguaje generales que pueden aplicarse a una amplia variedad de tareas, esto da como resultado mejoras significativas en el rendimiento en tareas específicas.

- Reutilización de modelos: Los modelos pre-entrenados pueden ser reutilizados para muchas tareas diferentes, lo que aumenta su valor y utilidad. Esto significa que no es necesario entrenarlos completamente para cada tarea específica. Esto da como resultado el último de los beneficios.

- Reducción del tiempo y los recursos necesarios para entrenar modelos: Al utilizar modelos pre-entrenados como punto de partida, el tiempo y los recursos

[2] Sistema interconectado de páginas web públicas accesibles a través de Internet.

necesarios para entrenarlos en tareas especificas se reducen significativamente. Esto se debe a que el modelo pre-entrenado ya ha aprendido características y patrones del lenguaje en un conjunto de datos masivo. Este hecho permite comenzar el entrenamiento desde una posición más avanzada.

En nuestra situación particular, emplearemos como modelo base 'bert-base-multilingual-cased', propuesto en [3]. Modelo pre-entrenado sobre 104 idiomas con Masked Languaje Modeling como objetivo, esto le permite tener un conocimiento profundo de la semántica y estructura a través de los distintos lenguajes.

4.8. Preprocesado

En cuanto al preprocesado de los datos, este está relacionado principalmente con el modelo y tokenizador usado. Algunos de ellos solo trabajan con minúsculas y en general, solo funcionan en un idioma. Hemos de adaptar por tanto, el texto a los requisitos del modelo, para que este pueda comprender la información.

En cuanto a las técnicas a emplear, encontramos el paso a minúsculas (modelos uncased) o la eliminación de caracteres especiales (unidecode). Para emplear ambas transformaciones hemos de tener distintos aspectos en cuenta. En el caso del unidecode, por ejemplo, hemos de tener en cuenta el idioma, ya que puede que cambiemos el significado de las oraciones (por ejemplo al eliminar tildes en el Castellano), resultando en un mal rendimiento por parte del modelo.

En el modelo final, debido al idioma y al modelo base seleccionado, no se empleará ninguna de las técnicas mencionadas.

4.9. Fine-Tune Tokenizer

Una vez tenemos definidos los datos y el modelo con el que vamos a trabajar, podemos aplicar diversas técnicas para mejorar el rendimiento de los Transformers, una de ellas es el Fine-Tune del tokenizer.

En ocasiones, el corpus con el que trabajamos cuenta con palabras que, debido a lo técnicas que son en relación al caso de uso, el tokenizador las segmenta, esto da lugar a una perdida de información. El objetivo es por tanto, identificar aquellas palabras importantes y específicas para el caso de uso, añadirlas al tokenizador y dotarlas de significado. De esta forma, conseguimos que el modelo acceda a nueva información más específica con el objetivo de incrementar su rendimiento.

El primer paso sería, como ya hemos comentado, identificar estas nuevas palabras.
Una de las alternativas consiste en obtenerlas empleando el conocimiento de aquellas personas expertas en el campo.
Otra de las alternativas consiste en identificarlas empleando un TF-IDF, algoritmo ya descrito en 3. El procedimiento sería el siguiente.
En primer lugar, realizamos bins en función de los targets (en los problemas de clasificación, cada target define un bin) y agrupamos el texto de cada uno, de esta forma

el TF-IDF capta aquellas palabras que mejor identifican a cada bin. A continuación, preprocesamos el texto, esto incluye transformar a minúsculas, eliminar caracteres especiales, etc (estas transformaciones se han de escoger en función del modelo y tokenizador base). Tras ello, para cada secuencia, obtenemos los elementos que contengan palabras, eliminando stop-words. Las stop-words dependen del idioma, como en nuestro caso el texto es multilingüe, hemos de detectarlo previamente.
Por último, aplicaremos el TF-IDF a los documentos resultantes y obtenemos, para cada bin, las palabras que mejor les identifica (escogiendo las n más importantes o a partir de cierto percentil o valor en la matriz resultante).

Una vez contamos con las palabras que resultan interesantes de cara al caso de uso, hemos de conseguir que sean interpretadas por el modelo, hemos de dotarlas de significado.
Para este propósito, entrenaremos un word2vec (definido en 4.2.3) sobre el corpus de entrenamiento. De esta forma, obtendremos representaciones de las palabras que capturan el significado contextual de las mismas.
Estas representaciones servirán como punto inicial al modelo.

Finalmente, hemos de añadir las palabras seleccionadas al tokenizador y ampliar la matriz de embeddings del modelo con las nuevas representaciones obtenidas.

Una vez realizada esta secuencia de pasos, hemos conseguido realizar un Fine-Tune del tokenizador, dando lugar a una adquisición de conocimiento específico del caso de uso.

4.10. Data Augmentation

Otra de las técnicas que emplearemos para mejorar el rendimiento del modelo es Data Augmentation.

En el aprendizaje automático, Data Augmentation se trata de una técnica cuyo propósito es aumentar el tamaño y variedad de un conjunto de datos de entrenamiento existente. Para ello, se aplican diferentes transformaciones a los datos originales, creando así nuevos registros de entrenamiento.
La idea es que estas transformaciones no alteren la salida esperada, pero sí agreguen variaciones de los datos que puedan ayudar al modelo a ser más robusto y generalizable a datos nuevos.

En general, la técnica de Data Augmentation es útil en el aprendizaje automático, especialmente cuando los conjuntos de datos son reducidos, ya que permite aprovechar al máximo los datos de entrenamiento disponibles y mejorar la capacidad de generalización del modelo.

En nuestro caso, sobre el conjunto de entrenamiento aplicaremos dos técnicas de Data Augmentation distintas, Easy Data Augmentation y Back Translation.

En cuanto a la técnica de Easy Data Augmentation (EDA), esta permite aumentar los datos de forma simple y efectiva en el ámbito del procesamiento del lenguaje

natural. EDA aplica, con cierta probabilidad, cuatro operaciones de transformación distintas a los datos de entrenamiento para generar nuevos, aumentando así la cantidad de estos.

Las cuatro operaciones utilizadas por EDA son:

- Sinónimos aleatorios: Esta operación reemplaza algunas palabras con sinónimos elegidos al azar.
- Inserción aleatoria: Esta operación inserta nuevas palabras al azar. Las palabras se seleccionan de una lista de palabras comunes o de un diccionario.
- Reemplazo aleatorio: Esta operación reemplaza algunas palabras con otras seleccionadas al azar. Nuevamente, las palabras se seleccionan de una lista de palabras comunes o de un diccionario.
- Eliminación aleatoria: Esta operación elimina palabras al azar de una secuencia.

Al combinar la aplicación de estas operaciones, se generan nuevas oraciones similares en significado a las originales pero con ciertas variaciones. Notemos que no garantizan la conservación de la salida, a cambio, ofrecen velocidad en la generación.

Por otro lado encontramos Back Translation. Técnica de Data Augmentation que consiste en traducir el texto de origen a otro idioma para luego volver a traducir el texto resultante al idioma original. De esta manera, se genera una nueva versión del texto original que puede tener una estructura y palabras ligeramente diferentes, pero que sigue manteniendo el mismo significado general.

El proceso de Back Translation ofrece flexibilidad en los idiomas seleccionados, así como en el número de estos por los que pasa la secuencia original. En este caso, a pesar de no ofrecer una gran velocidad, se garantiza la conservación de la salida.

Para realizar las traducciones se barajaron distintas opciones. Una de ellas basada en emplear el paquete *googletrans* [3] que ofrece python, este realiza llamadas a la API de Google Translate. Debido al tiempo requerido así como al límite de llamadas a la API, se descartó la opción.
La alternativa empleada consistió en usar un nuevo Transformer, el modelo m2m100 [4] propuesto por Facebook. Este se trata de un modelo entrenado para la tarea de traducción multilingüe.

Una vez presentados los dos métodos de Data Augmentation, estos pueden combinarse, ya sea aplicando ambos a cada nueva secuencia, o empleando cada uno en cierta proporción, de esta forma se aprovecha la velocidad de EDA y la calidad de los resultados obtenidos por Back Translation. En función del tiempo disponible y la calidad del nuevo dato requerida, se pueden emplear las distintas alternativas.

En el caso del modelo final, se optó por la primera alternativa.

[3] Módulo de python que disponibiliza la API de Google Translate: *googletrans*

4.11. Entrenamiento

El objetivo de esta sección es entrenar un Transformer con el propósito de clasificar texto.

Para ello, debido a los beneficios comentados en 4.7, se empleará Transfer Learning, es decir, emplearemos como punto inicial los pesos de un modelo de lenguaje previamente entrenado en tareas como Casual Language Modeling o Masked Language modeling, de esta forma recoge el significado semántico y sintáctico del lenguaje en términos generales.

Otro de los aspectos importantes a la hora de entrenar un modelo es su validación, para ello realizaremos un Train-Test Split sobre el dato con el que contamos. Esta división del dato será estratificada, es decir, cada target se encontrará en la proporción indicada tanto en el dato de entrenamiento como en el de test, de esta forma nos aseguramos que el modelo pueda ver tanto dato como sea posible de cada target para el entrenamiento.

Algunos de los parámetros configurables en cada entrenamiento son los siguiente:

- Uncase: Si es necesario o no el paso a minúsculas. En general, pasar a minúsculas simplifica el dato de entrenamiento, sin embargo, en ocasiones y dependiendo del caso de uso puede introducir ambigüedad (Por ejemplo algunos nombres propios donde las mayúsculas marcan el significado).
- Unidecode: Indica si escapar caracteres especiales o no convirtiéndoles en su equivalente ASCII. De nuevo, su comportamiento es similar al parámetro Uncase, simplifica el dato de entrenamiento aunque en ocasiones introduce ambigüedad (Por ejemplo, en el Castellano, aquellas palabras donde la existencia de acentos marca el significado).
- Modelo Base: Uno de los puntos críticos a la hora de realizar este tipo de entrenamiento, especialmente debido a la tarea sobre la que se ha entrenado o al idioma pues, si no es el adecuado, el modelo no podrá interpretar correctamente la información.
- Número de epochs: Representa el número de veces que se visita el dato completo, al realizar Transfer Learning, el número de estas no deberá ser muy elevado, dando como resultado un ahorro computacionalmente hablando.
- Batch Size: Número de muestras con la que se conforma un gradiente con el objetivo de ajustar la red y reducir el error (Gradient Descent). Un Batch Size mayor puede dar lugar a gradientes más generales y estables así como a una mayor velocidad en los tiempos de entrenamiento, no obstante requiere también de un mayor espacio en memoria. En este parámetro se buscara por tanto un compromiso entre la calidad y estabilidad de los gradientes obtenidos así como los recursos disponibles.
- Learning Rate: Hiperparámetro crítico en los algoritmos de aprendizaje automático que afecta el rendimiento y la convergencia del modelo durante el

proceso de entrenamiento. Permite controlar la magnitud de los ajustes realizados en los pesos del modelo durante el descenso del gradiente. La elección adecuada del Learning Rate es crucial para garantizar un buen rendimiento del modelo, si el Learning Rate es demasiado pequeño, los ajustes en los pesos serán muy pequeños y el modelo puede tardar mucho en converger o incluso quedarse atascado en mínimos locales subóptimos, por otro lado, si el Learning Rate es demasiado grande, los ajustes en los pesos pueden ser demasiado grandes y el modelo puede oscilar o incluso divergir, lo que resulta en un rendimiento deficiente.

- Warmup Steps: Se trata de un parámetro que controla la velocidad de aprendizaje durante el entrenamiento, permite ajustar gradualmente el Learning Rate (en este caso de forma lineal) durante el número de pasos que especifica, desde un valor bajo (o incluso cero) hasta su valor máximo. En el contexto de los Transformers, es beneficioso a la hora de inicializar los mecanismos de atención. Si se establece un valor demasiado pequeño, el Learning Rate aumentará rápidamente y el modelo puede experimentar oscilaciones o inestabilidad al principio del entrenamiento. Por otro lado, si se establece un valor demasiado grande, el modelo puede tomar mucho tiempo en alcanzar el valor máximo del Learning Rate, esto puede resultar en un entrenamiento más lento y prolongado.

- Weight Decay: Término de regularización comúnmente utilizado en algoritmos de aprendizaje automático para evitar el overfitting, penalizando los valores grandes en los pesos del modelo. El Weight Decay afecta directamente cómo se actualizan los pesos del modelo durante el proceso del descenso del gradiente. La elección adecuada del Weight Decay es importante para obtener un buen equilibrio entre el rendimiento durante el entrenamiento y la capacidad de generalización del modelo. Si el Weight Decay es demasiado grande, los pesos se volverán demasiado pequeños y el modelo no puede aprender información. Por otro lado, si el Weight Decay es demasiado pequeño o nulo, el modelo puede producir overfitting, resultando en dificultades para generalizar a nuevos datos.

- Metric for Best Model: A lo largo del proceso de entrenamiento, se generan varios modelos, uno por cada actualización que sufren los pesos. Este parámetro permite identificar cual es, de todos los modelos generados, el que produce un mayor rendimiento según la métrica establecida.

- Data Augmentation: Indica si realizar data augmentation o no, así como la estrategia empleada.

- Times Data Augmented: Permite establecer en que proporción se aumenta el tamaño del dato (Por ejemplo, $\times 1$ mantiene el tamaño original).

- Percentage Translate: En caso de que la alternativa empleada para realizar Data Augmentation consista en emplear las dos técnicas ya mencionadas en 4.10 de forma independiente, este parámetro permite controlar la carga asociada a cada uno. Como ya comentamos, Back Translation se trata de un método más costoso, computacionalmente hablando, aunque ofrece mejores resultados, por otro lado, EDA no ofrece soluciones de tan buena calidad aunque es más

eficiente y por lo tanto ofrece mejores tiempos. Este parámetro nos va a permitir encontrar un balance entre la calidad de Back Translation y la velocidad de EDA.

- Max Sequence Length: Permite establecer el máximo número de tokens con los que el modelo trabajará, truncando el resto. Un valor excesivamente bajo dará lugar a una perdida de información, por otro lado, un valor excesivamente alto (sin llegar al límite del modelo) tendrá como resultado un mayor coste en términos temporales y computacionales.

- Tokenizer Fine-Tune Threshold: Se trata del parámetro ya descrito en 4.9. En este caso, se incluirán al tokenizador aquellas palabras cuyo valor supere cierto umbral, más concretamente, aquellas palabras cuya importancia relativa en algún documento supere dicho valor. Un valor alto permite introducir solamente aquellas palabras que cuenten con una gran importancia en los documentos, corriendo el riesgo de que el número de estas no sea suficientemente grande para que al modelo le resulte de utilidad. Del mismo modo, un valor bajo permite introducir varias palabras, en este caso, contamos con el riesgo de incluir demasiadas y que el modelo no pueda identificar el significado de cada una de ellas, resultando en ruido.

Utilizando esta estructura, entrenaremos 4 modelos, a continuación, resumimos la configuración empleada en cada uno de ellos, así como el Train/Eval Loss (Cross Entropy) a lo largo de las epochs:

- Transformer Básico:

Parámetro	Valor
Uncase	False
Unidecode	False
Modelo Base	bert-base-multilingual-cased
Número de epochs	15
Batch Size	64
Learning Rate	$5e^{-6}$
Warmup Steps	500
Weight Decay	0
Metric for Best Model	Accuracy
Data Augmentation	None
Times Data Augmented	-
Percentage Translate	-
Max Sequence Length	150
Tokenizer Fine-Tune Threshold	-

Figura 4.7: Train (Naranja)/Eval (Rojo) loss a lo largo de las epochs para un Transformer básico.

- Transformer + Fine-Tune Tokenizer:

Parámetro	Valor
Uncase	False
Unidecode	False
Modelo Base	bert-base-multilingual-cased
Número de epochs	15
Batch Size	64
Learning Rate	$5e^{-6}$
Warmup Steps	500
Weight Decay	0
Metric for Best Model	Accuracy
Data Augmentation	None
Times Data Augmented	-
Percentage Translate	-
Max Sequence Length	150
Tokenizer Fine-Tune Threshold	0.01

Figura 4.8: Train (Naranja)/Eval (Rojo) loss a lo largo de las epochs para un Transformer con Fine-Tune sobre el Tokenizer.

- Transformer + Data Augmentation:

Parámetro	Valor
Uncase	False
Unidecode	False
Modelo Base	bert-base-multilingual-cased
Número de epochs	15
Batch Size	64
Learning Rate	$5e^{-6}$
Warmup Steps	500
Weight Decay	0
Metric for Best Model	Accuracy
Data Augmentation	Back Translation ∧ EDA
Times Data Augmented	2
Percentage Translate	-
Max Sequence Length	150
Tokenizer Fine-Tune Threshold	-

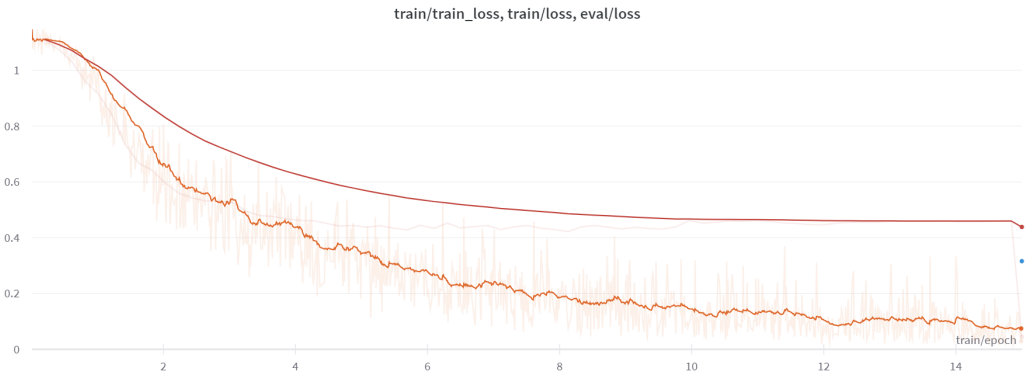


Figura 4.9: Train (Naranja)/Eval (Rojo) loss a lo largo de las epochs para un Transformer donde previamente se ha aplicado Data Augmentation.

- Transformer + Fine-Tune Tokenizer + Data Augmentation:

Parámetro	Valor
Uncase	False
Unidecode	False
Modelo Base	bert-base-multilingual-cased
Número de epochs	15
Batch Size	64
Learning Rate	$5e^{-6}$
Warmup Steps	500
Weight Decay	0
Metric for Best Model	Accuracy
Data Augmentation	Back Translation $\wedge$ EDA
Times Data Augmented	2
Percentage Translate	-
Max Sequence Length	150
Tokenizer Fine-Tune Threshold	0.01

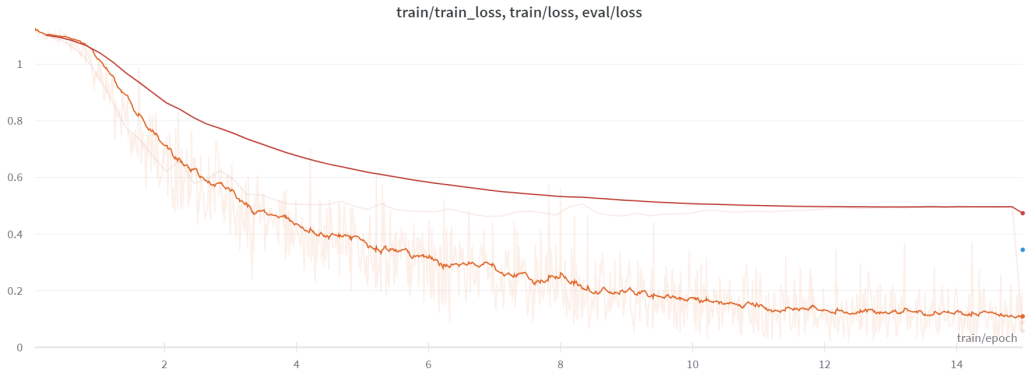


Figura 4.10: Train (Naranja)/Eval (Rojo) loss a lo largo de las epochs para un Transformer con Fine-Tune sobre el Tokenizer donde previamente se ha aplicado Data Augmentation.

Finalmente, una vez probadas distintas configuraciones, se optó por las descritas en las tablas anteriores, dando lugar a las evoluciones de entrenamiento observadas en las figuras que las acompañan respectivamente. Observamos que las 4 configuraciones se comportan de manera similar, llegando a una meseta donde no se pueden realizar mejoras sobre el dato de validación. Notemos que, a medida que avanza el entrenamiento, el dato de entrenamiento es aprendido a mayor velocidad, dando lugar a una divergencia entre las funciones de pérdida de ambos conjuntos, no obstante, observamos que el dato de validación se va aprendiendo también, por lo tanto, no se considera overfitting. En cualquier caso, se realizaron pruebas con parámetros que permitiesen introducir un efecto regularizador, dando lugar a peores resultados en entrenamiento y validación.

Capítulo 5

Análisis y Comparación de resultados

A lo largo de este capítulo, nos centraremos en el análisis y comparación de los resultados obtenidos con los diversos modelos. No obstante, previamente hemos de describir el dataset empleado y definir las métricas que nos permitirán evaluar los modelos objetivamente.

5.1. Dataset

El dataset escogido juega un papel fundamental tanto en los resultados como en la interpretación de los mismos, por ello es de gran importancia su comprensión.

Como ya hemos comentado, la tarea a realizar es la clasificación sobre texto. Este dataset contiene noticias, tanto en castellano como en inglés, con su correspondiente valoración atribuida en forma de clase: Negativo, Neutral y Positivo. Se trata por tanto de un análisis de sentimientos sobre noticias.

En cuanto a la distribución de las clases sobre el dataset, contamos con 12296 registros repartidos de la siguiente manera:

- Negativo: 1731 (14 %)
- Neutro: 6952 (57 %)
- Positivo: 3613 (29 %)

Como observamos, la clase Neutro es la mayoritaria, cayendo más de la mitad del dato sobre ella. El resto del dato se divide entre las otras 2 clases, contando la Positiva con más del doble de registros que la Negativa.

Finalmente y con el objetivo de conocer algo más el dato con el que trabajamos, mostraremos algunas de las palabras seleccionadas durante el Fine-Tune del Tokenizer (seleccionadas por un TF-IDF). Entre las palabras escogidas encontramos: *Agenda, Anuncio, Mercantil, Subdirección, Subsecretaría, adjudicador, operativa, presupuestariamente* o *ibex* entre otras.
Gracias a estas, podemos tener una idea del contenido del dataset.

El resto de información acerca del mismo se reserva debido a la política de confidencialidad de la empresa.

5.2. Métricas

Para evaluar los modelos, tendremos en consideración diversas métricas. Como ya sabemos, en un problema de clasificación la precisión indica cuan seguros estamos en la predicción de una categoría, del mismo modo, el recall indica cuan bien detectamos dicha categoría. El F1 score no es más que una combinación de las dos métricas anteriores.
Formalmente, en un problema de Clasificación Multi-Clase, dada la categoría i podemos definir:

$$precision(i) = \frac{TP_i}{TP_i + FP_i} \tag{5.1}$$

$$recall(i) = \frac{TP_i}{TP_i + FN_i} \tag{5.2}$$

$$F1(i) = 2 \cdot \frac{precision(i) \cdot recall(i)}{precision(i) + recall(i)} \tag{5.3}$$

Donde:

- TP_i: Indica los elementos correctamente clasificados de la clase i.
- FP_i: Indica los elementos clasificados como de la clase i, perteneciendo en realidad a otra.
- FN_i: Indica los elementos clasificados como otra clase, perteneciendo en realidad a la clase i.

Una vez contamos con estas métricas para cada clase, existen diversas formas de combinarlas para plasmar el rendimiento general del modelo en función de las capacidades que tiene sobre las distintas clases.

En primer lugar encontramos las Macro-Metrics. Estas calculan el promedio de las métricas de cada clase por separado. Primero, se calcula la métrica para cada clase. Luego, se promedian los resultados de todas las clases para obtener una única métrica macro. Esto significa que todas las clases se tratan de manera equitativa, sin importar su distribución o tamaño.

Por otro lado encontramos las Micro-Metrics. Estas calculan las métricas agregadas sobre todas las clases. En lugar de calcular la métrica para cada clase por separado, las Micro-Metrics consideran todas las predicciones y clases como un conjunto único (Debido a ello las Micro-Metrics empleadas coindicen con el Accuracy del modelo).

La elección entre usar métricas Macro o Micro depende del contexto y el objetivo del problema de clasificación. Si todas las clases son igualmente importantes o si el

objetivo es evaluar el rendimiento equitativo para cada clase, las métricas Macro son más adecuadas. Por otro lado, si se busca evaluar el rendimiento global del modelo independientemente de las clases, las métricas Micro pueden proporcionar una visión más representativa.

5.3. Comparación errores

Por último, una vez definidas las métricas a estudiar, realizaremos un estudio sobre los errores obtenidos con cada modelo. Como suele ser estándar, definiremos un modelo básico que tomaremos como baseline, en este caso, no es otro que aquel que predice todos los elementos como pertenecientes a la clase mayoritaria según el dato de entrenamiento.

Mostramos a continuación las siguientes tablas que recogen las métricas establecidas para los distintos modelos:

	precision(1)	*recall*(1)	*F*1(1)	*precision*(2)	*recall*(2)	*F*1(2)	*precision*(3)	*recall*(3)	*F*1(3)
Basic model	0.0	0.0	0.0	0.5654	1.0	0.7224	0.0	0.0	0.0
TF-IDF	0.4194	0.4740	0.4450	0.7088	0.8152	0.7583	0.5778	0.3748	0.4547
Transformer	0.6871	0.7746	0.7283	0.8301	0.8361	0.8331	0.7489	0.6929	0.7198
Transformer + Tokenizer Fine-Tune	0.6602	0.7919	0.7201	0.8454	0.8411	0.8432	0.7625	0.6971	0.7283
Transformer + Data Augmentation	0.7833	0.6792	0.7276	0.8231	0.8864	0.8536	0.7840	0.7178	0.7495
Transformer + Data Augmentation + Tokenizer Fine-Tune	0.7681	0.6127	0.6817	0.7947	0.8850	0.8374	0.7701	0.6763	0.7202

Tabla 5.1: Métricas por clase de cada modelo.

	macro-precision	macro-recall	macro-F1	micro-precision	micro-recall	micro-F1
Basic model	0.1885	0.3333	0.2408	0.5654	0.5654	0.5654
TF-IDF	0.5687	0.5547	0.5527	0.6378	0.6378	0.6378
Transformer	0.7554	0.7679	0.7604	0.7854	0.7854	0.7854
Transformer + Tokenizer Fine-Tune	0.7560	0.7767	0.7639	0.7919	0.7919	0.7919
Transformer + Data Augmentation	0.7968	0.7611	0.7769	0.8077	0.8077	0.8077
Transformer + Data Augmentation + Tokenizer Fine-Tune	0.7776	0.7247	0.7464	0.7854	0.7854	0.7854

Tabla 5.2: Macro y Micro métricas de cada modelo.

Una vez obtenidos los resultados de las pruebas, procedemos a su análisis.

En cuanto a la tabla 5.1, esta muestra la precision, el recall y el F1 en cada categoría (ordenadas según su presentación en la sección 5.1) para cada modelo. Esto permite ver con detalle cuan seguro está cada modelo a la hora de inferir una clase (precision) así como la capacidad de los mismos para detectarlas (recall). La combinación de estas (F1), permite medir objetivamente el rendimiento de los modelos en cada clase.

Es claro que los modelos que emplean la arquitectura de los Transformers sobrepasan claramente aquellos que no, siendo el modelo básico el que peores resultados obtiene en todas las métricas, claro está, a excepción del recall de la clase mayoritaria (2).

En cuanto a la primera categoría, los mejores resultados tanto en precision como en recall son obtenidos por los Transformers con técnicas adicionales, siendo el Transformer básico el que consigue un mejor balance entre estas y por tanto consiguiendo un mejor F1.
En la segunda categoría se observa un comportamiento similar a la primera donde los Transformers que emplean técnicas adicionales consiguen los mejores resultados, con la excepción del recall, donde el modelo básico lo hace y siempre lo hará de forma perfecta debido a su naturaleza. Finalmente, es el Transformer donde previamente se ha aplicado Data Augmentantion aquel que consigue un mejor equilibrio, consiguiendo el F1 más alto.
Por último, en la tercera categoría, el modelo básico nuevamente obtiene los peores resultados. Por otro lado, el Transformer donde previamente se ha realizado Data Augmentation es aquel que obtiene las mejores métricas tanto en precision como en recall y por tanto, también en F1.

Dado que el F1 representa un equilibrio entre precision y recall, podríamos decir en base a los resultados obtenidos que en la primera categoría el Transformer básico es el que mejor rendimiento ofrece, seguido de cerca por los Transformers donde se han realizado técnicas adicionales. Del mismo modo, en la segunda y tercera categoría es la combinación de un Transformer con Data Augmentation el modelo que mejor rendimiento ofrece.

Como ya hemos comentado, esta primera tabla nos permite ver el rendimiento de los modelos en cada categoría, no obstante, si queremos ver el rendimiento general del modelo de una forma transversal a las categorías, hemos de combinar los resultados, esto es precisamente lo que hacen las Macro y Micro métricas, que se encuentran en la tabla 5.2.

En esta segunda tabla se combinan las métricas obtenidas en cada categoría, de esta forma podemos observar la precision, el recall y el F1 de los distintos modelos sobre todas ellas. La principal diferencia entre las Macro-metrics y las Micro-metrics radica en la forma de agregación, como ya se mencionó en 5.2.

En cuanto a la precisión, tanto en las Macro-metrics como en las Micro-metrics

se observa el mismo comportamiento. Por un lado, el modelo básico es el que ofrece los peores resultados, por otro, es nuevamente la combinación entre un Transformer y Data Augmentation el modelo que mas certeza tiene a la hora de de inferir.
En cuanto al recall, nuevamente el modelo básico es aquel que peores resultados ofrece. No obstante, son los Transformer con fine-tune sobre el tokenizador y con Data Augmentation los que mejores resultados ofrecen en las Macro y Micro-metrics respectivamente. El hecho de que el Transformer con fine-tune sobre el tokenizador ofrezca la mejor Macro-recall se puede deber a las nuevas palabras con las que cuenta en su vocabulario, las cuales permiten identificar más nítidamente alguna de las categorías, esto resulta en un mayor promedio y por tanto mayor Macro-recall.
Finalmente, en relación al F1, se observa el comportamiento esperado dado el análisis anterior. Si en precisión y recall tanto en las Macro-metrics como en las Micro-metrics era el modelo básico el que peores resultados generaba y el Transformer con Data Augmentation el que dominaba al resto, al ser el F1 una combinación de estas, se repite el comportamiento.

5.4. Conclusiones

Una vez analizados y comparados los resultados obtenidos, podemos obtener una serie de conclusiones para finalizar el proyecto:

- Los modelos que emplean la arquitectura de los Transformers sobrepasan a aquellos que no. Algo que que esperábamos y pretendíamos respaldar.
- Para este dato particular, la técnica de Data Augmentation produce un mayor incremento en el rendimiento que un fine-tune del tokenizador. Esto en parte era también esperado, pues el dato empleado no es extremadamente técnico.

Adicionalmente, hemos de comentar que los incrementos en el rendimiento observados con cada técnica dependen altamente del dato con el que se trabaje.
Los casos en los que el volumen de dato no sea suficiente, es donde Data Augmentation producirá los mayores incrementos en los resultados.
Del mismo modo, será en los casos donde el texto sea excesivamente técnico aquellos en los que el fine-tune del tokenizador realizará los mayores aportes.

Por último, hemos de mencionar el hecho de que el Transformer al cual se le han aplicado ambas técnicas no sea aquel que produzca los mejores resultados. Esto se puede deber a distintos factores, uno de ellos es el tiempo de entrenamiento, al contar con nuevas palabras en el vocabulario, el modelo requiere de más tiempo para aprenderlas, el hecho de que además haya dato nuevo amplifica este efecto.

5.5. Líneas Futuras

En cuanto al trabajo a realizar próximamente para continuar con el desarrollo del proyecto, este se centrará principalmente en testear los modelos y obtener las comparaciones anteriores bajo distintas circunstancias.

En sintonía con lo comentado en la sección anterior (5.4), las técnicas para potenciar los Transformers dependen en gran medida del dato con el que se trabaje, por ello sería de utilidad observar el comportamiento de los diversos modelos sobre datasets extremadamente técnicos, así como sobre aquellos en los que contamos con poco dato.

Finalmente, otra de las tareas pendientes es observar el comportamiento del Transformer con Fine-Tune sobre el tokenizer donde previamente se ha realizado Data Augmentation. Esto con el objetivo de comprender por qué no se cumple la hipótesis que sosteníamos inicialmente, donde esperábamos que esté modelo aportase los mejores resultados.

Bibliografía

[1] D. Bahdanau, K. Cho, and Y. Bengio. Neural machine translation by jointly learning to align and translate. *arXiv preprint arXiv:1409.0473*, 2014.

[2] T. Brown, B. Mann, N. Ryder, M. Subbiah, J. D. Kaplan, P. Dhariwal, A. Neelakantan, P. Shyam, G. Sastry, A. Askell, et al. Language models are few-shot learners. *Advances in neural information processing systems*, 33:1877–1901, 2020.

[3] J. Devlin, M.-W. Chang, K. Lee, and K. Toutanova. Bert: Pre-training of deep bidirectional transformers for language understanding. *arXiv preprint arXiv:1810.04805*, 2018.

[4] A. Fan, S. Bhosale, H. Schwenk, Z. Ma, A. El-Kishky, S. Goyal, M. Baines, O. Celebi, G. Wenzek, V. Chaudhary, et al. Beyond english-centric multilingual machine translation. *The Journal of Machine Learning Research*, 22(1):4839–4886, 2021.

[5] T. Kudo. Subword regularization: Improving neural network translation models with multiple subword candidates. *arXiv preprint arXiv:1804.10959*, 2018.

[6] T. Kudo and J. Richardson. Sentencepiece: A simple and language independent subword tokenizer and detokenizer for neural text processing. *arXiv preprint arXiv:1808.06226*, 2018.

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

[8] J. Pennington, R. Socher, and C. D. Manning. Glove: Global vectors for word representation. In *Proceedings of the 2014 conference on empirical methods in natural language processing (EMNLP)*, pages 1532–1543, 2014.

[9] D. E. Rumelhart, G. E. Hinton, and R. J. Williams. Learning internal representations by error propagation. Technical report, California Univ San Diego La Jolla Inst for Cognitive Science, 1985.

[10] M. Schuster and K. Nakajima. Japanese and korean voice search. In *2012 IEEE international conference on acoustics, speech and signal processing (ICASSP)*, pages 5149–5152. IEEE, 2012.

[11] R. Sennrich, B. Haddow, and A. Birch. Neural machine translation of rare words with subword units. *arXiv preprint arXiv:1508.07909*, 2015.

[12] A. Vaswani, N. Shazeer, N. Parmar, J. Uszkoreit, L. Jones, A. N. Gomez, Ł. Kaiser, and I. Polosukhin. Attention is all you need. *Advances in neural information processing systems*, 30, 2017.

[13] L. Voita. Nlp course for you. `https://lena-voita.github.io/nlp_course.html`.

[14] Wikipedia. `https://wikipedia.org`.