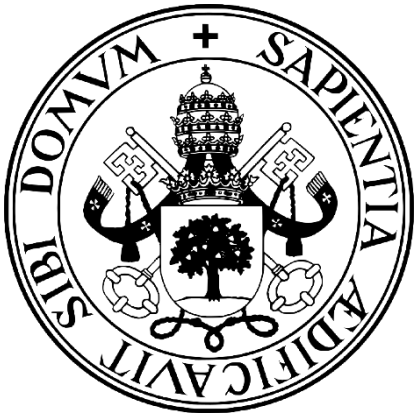


Universidad de Valladolid

ESCUELA DE INGENIERÍA INFORMÁTICA DE SEGOVIA

Grado en Ingeniería Informática

de Servicios y Aplicaciones

Análisis del estado del arte de la generación de texto con redes neuronales mediante modelos de Transformer

---

Alumno: Javier Sánchez Gozalo

Tutor: Fernando Díaz Gómez

Agradecimientos

Mi profundo agradecimiento a los compañeros y al personal universitario con los que he convivido, y en especial agradezco a los profesores que durante este periodo me han enseñado los fundamentos de ingería informática y a superarme durante todo el periodo lectivo.

"The way to succeed is to double your failure rate."

-Thomas J. Watson

"I really believe there are no borders for science."

-Fei-Fei Li

Análisis del estado del arte de la generación de texto con redes neuronales mediante modelos de Transformers

Javier Sánchez Gozalo

Resumen

El presenta Trabajo de Fin de Grado (TFG) pretende aplicar los conocimientos actuales sobre el Procesamiento del Lenguaje Natural (PLN) a el problema de "Responder a preguntas" (*Question Answering*) haciendo uso de la arquitectura *Transformer*.

Se alcanzará este objetivo presentando las características del PLN y su utilidad, se presentarán los modelos más relevantes para el modelo que se utilizará en la aplicación práctica, se realizará una aplicación práctica sobre este tema, cuya función será resolver el problema de "Responder a preguntas" dada una fuente donde buscar las respuestas, y se explicarán de manera analítica cual es la mejor forma de enfrentar esta técnica en un entorno real.

El proyecto incluye código que permitirá "Responder a preguntas" dado previamente un contexto y sus preguntas. El idioma utilizado será el inglés.

Palabras clave: *Transformer*, Procesamiento del Lenguaje Natural, Redes Neuronales Artificiales.

Abstract

The present Final Degree Project aims to apply current knowledge about Natural Language Processing (NLP) to the field of Question Answering using Transformer architecture.

This objective will be achieved by presenting: the characteristics of the PLN and its utility, the most relevant models for the model to be used in practical application will be presented, a practical application on this topic will be executed, which function will be to solve the problem of Question Answering, given a source where to look for the answers, and the best way to face this technique in a real environment will be explained in an analytical way.

The project includes code which will allow to answer questions, previously given a context and their questions. The language used is English.

Key words: *Transformer, Natural Language Processing, Artificial Neural Networks.*

Índice general

Índice de figuras

Capítulo 1

Introducción

En los últimos años han surgido nuevos campos de aplicación de las tecnologías de la información que exploran el tratamiento de la gran cantidad de datos digitales existentes y cómo transformarlos en conocimiento explícito.

En general, las tecnologías de la información están alcanzando cotas cada vez más altas en la vertiente de análisis automático de los documentos. El análisis del contenido documental (resumen e indización) ya se puede realizar de modo automático gracias al Procesamiento del Lenguaje Natural (PLN).

Las técnicas de Procesamiento del Lenguaje Natural son capaces de extraer información de los textos digitales, presentados en forma narrativa (en lenguaje natural). Además, las técnicas de *Machine Learning* clasifican instancias o ejemplos, en función de sus atributos, en distintas categorías, aprendiendo de otros previamente clasificados.

Todo esto es debido al impulso de la Inteligencia Artificial y del Big Data, que han dado origen a una nueva era digital, proporcionando una mejora en nuestras vidas.

Ha sido en las últimas décadas, gracias al aumento de la capacidad de procesamiento, cuando se han podido entrenar modelos con grandes capacidades, y con ello han podido surgir productos inspirados en el procesamiento del lenguaje natural como Alexa, un producto conversacional de Amazon, gracias al cual, una consulta se le pasa por medio de la voz y puede responder se por el mismo medio, es decir, la voz. Esta tecnología se puede utilizar para preguntar cualquier contenido, buscar cualquier cosa, para reproducir canciones o incluso para reservar un taxi.

Estas tecnologías han tenido una gran evolución desde su aparición y no paran de crecer a día de hoy, lo que motiva una gran demanda de profesionales especializados en esta rama de la ingeniería informática.

El campo de la Inteligencia Artificial tiene varios subcampos entre ellos el del Aprendizaje Automático (ML, *Machine Learning*) y el del Procesamiento del Lenguaje Natural (NLP, Natural Language Processing).

En la siguiente figura se muestran un diagrama de las relaciones entre estos campos:

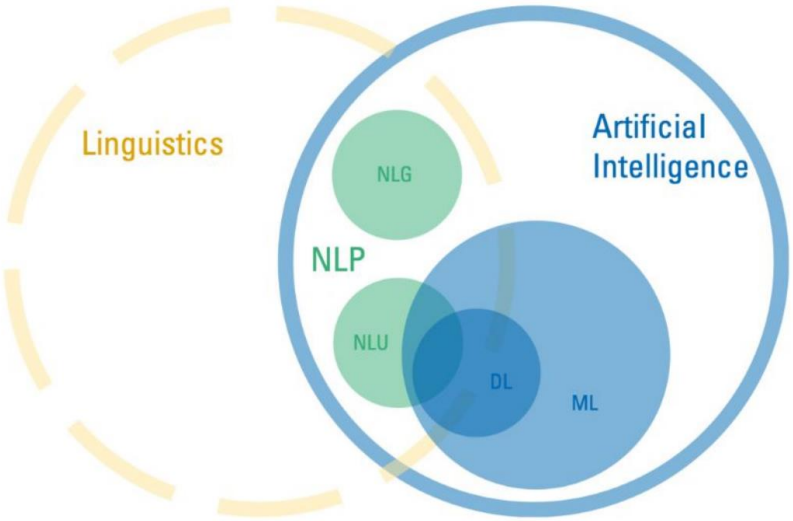


Figura 1. Disciplinas involucradas en el problema de Responder a preguntas

En la figura anterior se encuentran dos campos principales, el de la lingüística y el campo de la Inteligencia Artificial en cuya intersección se encuentra el dominio del *Natural Language Processing* (NLP o Procesamiento del Lenguaje Natural). A su vez dentro del PLN se pueden diferenciar otros dos subdominios:

-Generación de Lenguaje Natural (NLG*, Natural Language Generation*)

- Entendimiento de Lenguaje Natural (NLU, *Natural Language Understanding*)

El dominio de aplicabilidad que pretende entender la lengua está relacionado con la disciplina del *Machine Learning* (ML), y dentro de esta última se encuentra el Aprendizaje Profundo (DL*, Deep Learning*).

Dentro del campo del Procesamiento del Lenguaje Natural se encuentran algunas aplicaciones como son:

- Recuperación de información
- Extracción de información
- Traducción automática
- Simplificación de texto
- Análisis de sentimiento
- Resumen de texto
- Filtro de correo no deseado
- Predicción automática
- Autocorrección
- Reconocimiento de voz
- Responder a preguntas (*QA, Question Answering*)
- Generación de lenguaje natural a partir de otras fuentes

De todos estos problemas, este trabajo se centra en el de "Responder a preguntas", el cual trataremos en este documento.

Empezaremos entendiendo cuales son los principales campos de aplicación del PLN, y luego aplicaremos la técnica a un problema concreto de "Responder a preguntas".

1.1 Motivación

Los principales inconvenientes que se enfrentan estos días en el campo del PLN se relacionan con el hecho de que el lenguaje es muy complicado.

El proceso de comprensión y manipulación del lenguaje es extremadamente complejo, por lo que es común utilizar diferentes técnicas para manejar los diferentes desafíos que se plantean, para después, unirlo todo.

Los avances recientes en la investigación moderna del PLN han estado dominados por la combinación a gran escala de métodos de aprendizaje por transferencia con modelos de lenguaje del tipo *Transformer*.

Con ellos se produjo un cambio de paradigma en el campo del PLN, dejándose de crear modelos nuevos desde cero, pasándose a crear modelos que, partiendo de un modelo pre-entrenado de propósito general, se utilizaba este para entrenar un modelo específico.

Es aquí donde aparece la arquitectura *Transformer*, el cual sigue esta línea de modelos de propósito general.

Transformer ha ganado una gran atención entre las comunidades de investigadores y profesionales.

En este documento pretendemos acercarnos a esta arquitectura y utilizar un modelo inspirado en él, para aplicarlo a un problema de preguntas y respuestas.

Aun así, la creación de estos modelos de propósito general sigue siendo un proceso costoso y que requiere mucho tiempo, por lo que se restringe el uso de estos métodos a un pequeño subconjunto de la comunidad de PLN. Por lo que esta parte, la creación de modelos de propósito general, no se tratará en este TFG.

1.2 Objetivo

Se busca proporcionar el conocimiento para construir sistemas autónomos de aprendizaje en el campo del PLN. Para ello se describirán las partes fundamentales para lograr este objetivo, partiendo desde los fundamentos del campo del PLN, pasando por los modelos más relevantes, y así explicar el modelo que se utilizará en la aplicación práctica.

En concreto se proporcionará una aplicación al problema de "Responder a preguntas" o *(QA, Question Answering)* utilizando el idioma inglés. El modelo utilizado en la aplicación práctica se denomina Representación de Codificadores Bidireccionales de la arquitectura *Transformer* (*BERT*, *Bidirectional Encoder Representations from Transformer).*

Vamos a utilizar para ello el siguiente esquema.

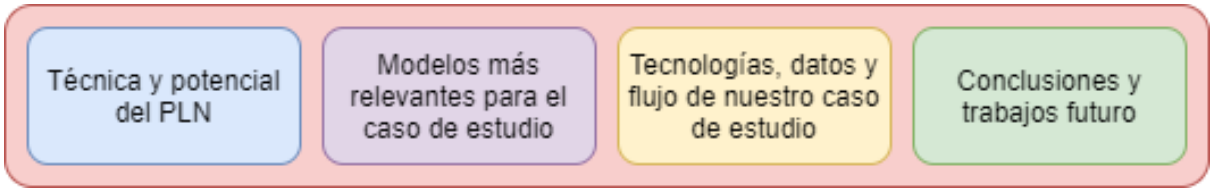


Figura 2 Guía general del trabajo

Como se aprecia en la imagen anterior, se empezará presentando las técnicas más usadas y la aplicabilidad del PLN, después, se mostrarán los modelos que han precedido al modelo que utilizaremos en el caso de estudio, el apartado referente al caso de estudio mostrará las tecnologías utilizadas, los datos de entrada, que se necesitan para inicializar el modelo, y se verá todo el proceso que re ha realizado para obtener la solución final. Finalmente se verán las conclusiones y el trabajo futuro.

Capítulo 2

Dominio del problema

Vamos a tratar el problema de "Responder a preguntas", este problema se basa en que, partiendo de una serie de entradas, las cuales son:

- **Contexto**: conjunto de sentencias, a partir de las cuales nuestro código obtendrá un conjunto de valores internos del problema.
- **Pregunta**: basado en una pregunta bien formulada, a partir de la cual nuestro código obtendrá un conjunto de valores internos del problema.

Partiendo de estos dos conjuntos de valores, el código obtendrá otro conjunto de valores que serán transformados en una sentencia que será nuestra salida o respuesta, respondiéndose así a la pregunta planteada sobre el contexto.

Figura 3 Problema general de QA

Debido a la complejidad que conlleva responder eficazmente a una respuesta, nuestro caso práctico no proporcionará una respuesta construida en función de la pregunta, sino que se limitará a proporcionar la parte del contexto que nos serviría para construir dicha respuesta.

Una de las formas más usuales de representar el flujo de trabajo característico de un algoritmo de PLN, es mediante el siguiente *pipeline*.

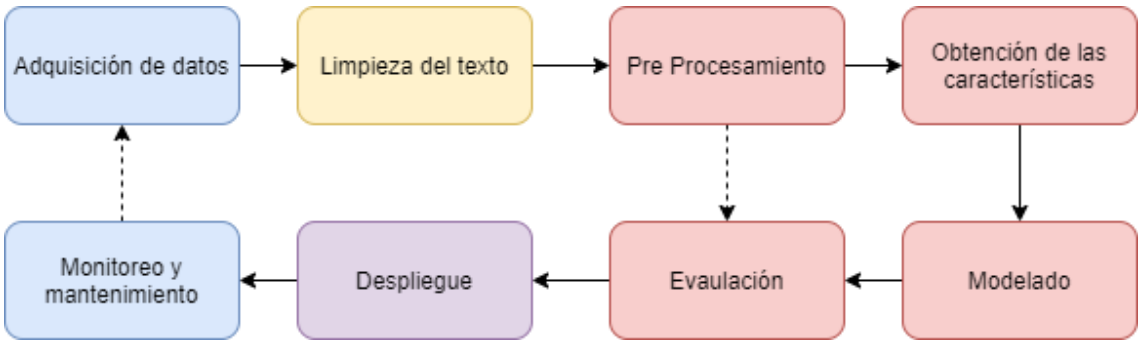


Figura 4 Características de una aplicación de PLN

En nuestro proyecto no realizaremos algunos pasos de los indicados en este *pipeline*.

Para la adquisición de datos y limpieza de texto, no requeriremos obtener datos de diferentes fuentes, agruparlos, seleccionarlos y luego verificar y corregir, sino que se irá a utilizar datos que son comúnmente utilizados en esta área, y que ya están previamente tratados.

Después pasaremos por la fase de preprocesamiento, la obtención de características, el modelado, y la evaluación.

Finalmente, se utilizará el modelo y se sugerirán mejoras.

2.1 Tratamiento de la información

En este apartado se pretenden proporcionar una breve explicación sobre cómo se debe tratar la información en el ámbito del PLN.

2.1.1 Ambigüedad del texto

Primeramente, hay que entender cuál es el significado de la información escrita. Sin embargo, éste puede ser más de uno, ya que para una misma sentencia puede haber ambigüedad. En PLN se tienen los siguientes tipos de ambigüedad:

- **Ambigüedad léxica**: ocurre cuando hay ambigüedad en una sola palabra. Por ejemplo, tratando la palabra "*silver*" como sustantivo, adjetivo o verbo.

- **Ambigüedad sintáctica**: ocurre cuando una oración se analiza de diferentes maneras. Por ejemplo, la frase: "*The man saw the girl with the telescope*". Es ambiguo si el hombre vio a la chica que llevaba un telescopio o la vio a través de su telescopio.

- **Ambigüedad semántica**: Este tipo de ambigüedad ocurre cuando el significado de las palabras mismas puede ser malinterpretado. En otras palabras, la ambigüedad semántica ocurre cuando una oración contiene un palabra o frase ambigua. Por ejemplo, la oración "*The car collided with the pole while in motion*" tiene ambigüedad semántica porque las interpretaciones pueden ser las siguientes, "El coche mientras se movía, golpeó el poste" y "El coche golpeó el poste mientras el poste se movía".

- **Ambigüedad pragmática**: se representa cuando el contexto de una frase le da múltiples interpretaciones o la declaración no es específica. Por ejemplo, la oración "*Maria is with Eva. I will ask her*" puede tener múltiples interpretaciones porque no deja claro a quién se va a preguntar.

Para tratar de resolver estos problemas, se realizan los siguientes pasos:

- **Procesamiento morfológico:** su propósito es trocear el texto de entrada en fragmentos más pequeños con cierto significado, tanto a nivel de párrafos, oraciones y palabras. Por ejemplo, una palabra como "*dealing*" se pueden dividir en dos *tokens* significativos como son "*deal*" + "*ing*".

- **Análisis de sintáctico:** su propósito es doble: comprobar si una oración está bien formada o no, y dividirla en una estructura que muestre la relaciones entre las diferentes palabras. Por ejemplo, una oración como "*School goes to the child*" debería rechazarse, por el analizador sintáctico.

- **Análisis semántico:** su propósito es deducir el significado exacto. Por ejemplo, el analizador semántico rechazaría una oración como "*Hot ice cream*.".

- **Análisis pragmático**: su propósito es ajustarse a los eventos reales, que existen en un contexto, en términos de los eventos reales obtenidos durante la última fase (análisis semántico). Por ejemplo, la oración "*Put the banana in the basket on the shelf*". Puede tener dos interpretaciones semánticas y el analizador pragmático debería elegir entre estas dos posibilidades.

De forma gráfica, normalmente se siguen los siguientes pasos para analizar el contenido del texto:

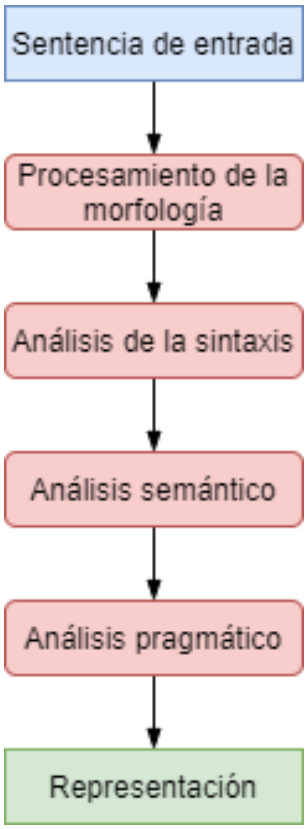


Figura 5 Flujo para procesar la ambigüedad

El único objetivo de los sistemas de PLN es mantener la ambigüedad al mínimo mientras se procesa la estructura de una sentencia del lenguaje natural. Intentar resolver el problema completamente y para todos los casos posibles, conlleva a intentar resolver un problema computacionalmente inabarcable con los modelos de cálculo actualmente disponibles (NP hard).

Por tanto, se busca una aproximación a la solución del problema y no una solución exacta.

2.1.2 *Corpus*

Una vez presentado cómo entender texto, se necesitará introducir los datos que se van a utilizar. Estos datos denominaremos *corpus*.

Un *corpus* es un conjunto grande y estructurado de textos legibles para una máquina, que se han producido en un entorno comunicativo natural. Pueden construirse de diferentes formas a partir de texto que originalmente era electrónico, transcripciones de lenguaje hablado, transcripciones de lenguaje óptico reconocimiento de caracteres, etc.

Dado un *corpus*, se necesitará muestrear e incluir proporcionalmente una amplia gama de tipos de texto para garantizar un buen diseño del mismo. Uno de los elementos más importantes para el diseño de *corpus* es su representatividad. Las siguientes definiciones de dos grandes investigadores, Leech y Biber, nos ayudarán a comprender la representatividad de un *corpus*:

- Según Leech (1991), "Se cree que un *corpus* es representativo de la variedad de lenguaje que se supone que representa, si los hallazgos basados en su contenido se pueden generalizar a dicha variedad lingüística".
- Según Biber (1993), "La representatividad se refiere al grado en que una la muestra incluye el rango completo de variabilidad en una población".

De esta forma, podemos concluir que la representatividad de un *corpus* está determinada por los siguientes dos factores:

- Equilibrio: el rango del tipo de contenido incluido en un *corpus*.
- Muestreo: cómo se seleccionan los fragmentos de cada género. Dentro del muestreo debemos considerar lo siguiente:

    - Unidad de muestreo: unidad que requiere una muestra. Por ejemplo, para texto escrito, una unidad de muestreo puede ser un periódico, un diario o un libro.

    - Marco de muestreo: la lista de todas las unidades de muestreo se denomina marco de muestreo.

    - Población: conjunto de todas las unidades de muestreo. Está definido en términos de producción del lenguaje, recepción del lenguaje o lenguaje como producto para esa población.

2.2 Aplicabilidad

En un mundo en el que se generan 2,5 trillones de bytes de datos todos los días, el análisis de texto se ha convertido en una herramienta clave para estructurar los datos y obtener información útil.

El aprovechamiento de los datos de manera eficaz permite a las empresas, no solo controlar los costos y riesgos, sino también competir de manera más efectiva e impulsar la rentabilidad al atender a sus clientes finales de manera eficiente.

Aquí aparecen campos de la Inteligencia Artificial como la Automatización de Procesos Robóticos *(RPA, Robotic Process Automation),* entre otros. Describiremos brevemente este campo para dar un contexto para el PLN.

El campo del RPA permite a las organizaciones ahorrar una fracción del costo y del tiempo que aportaban anteriormente. Tiene una naturaleza no intrusiva y aprovechan la infraestructura existente sin causar interrupciones en los sistemas subyacentes que, por otro lado, serían difíciles y costosos de reemplazar.

Una las funciones del PLN en el ámbito del RPA es analizar documentos estructurados, no estructurados o "semiestructurados" para identificar, extraer y estructurar datos dentro de ellos para su posterior análisis.

Según un estudio sobre automatización, llevado por la consultora Mckinsey, existen tres principales actividades que tienen el potencial de automatizarse, tal y como se muestra en la siguiente figura:

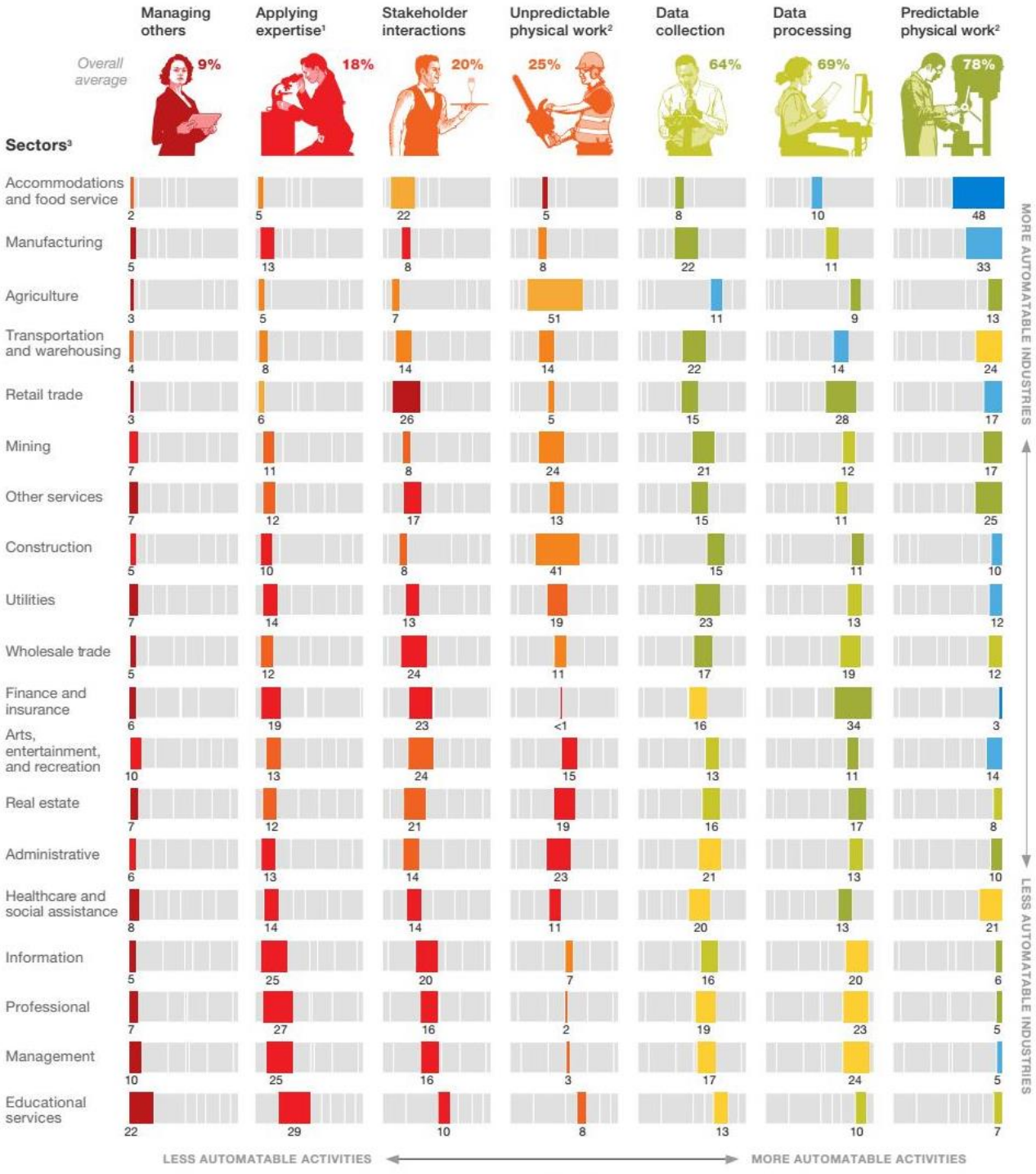


Figura 6 Actividades susceptibles de automatización

En la figura de arriba vemos que tenemos grupos de actividades (en horizontal) y de industrias (en vertical), ordenados ambos de mayor a menor automatización. Dentro de cada industria tenemos la misma barra gris semicontinua repetida para cada actividad. En diferentes colores está el porcentaje de actividades automatizadas, en esa industria en concreto.

El color resaltado en cada franja representa el porcentaje de tiempo real que a esa industria le cuesta invertir en ese tipo de trabajo. La franja de colores sigue la siguiente escala:

Así por ejemplo, como ejemplo para la industria del comercio mayorista, la cual tiene un 24% para la actividad de relaciones con los clientes y un 19% para el tratamiento de datos, sería más eficaz automatizar el tratamiento de datos que las relaciones con los clientes.

Como se observa, dos de estas actividades son el procesamiento de datos y la recopilación de datos, actividades normalmente ligadas al PLN.

El mismo estudio también muestra la siguiente imagen:

Figura 7 Dominios de utilidad de las tecnologías de Aprendizaje automático

En esta figura se muestra la capacidad (*Capability*) que tienen los diferentes campos de Inteligencia Artificial de producir un cambio positivo en la sociedad.

La figura está dividida por sectores (*Domain*), tal y como se muestra, el PLN sobresale en la mayoría de sectores de nuestra sociedad.

Por tanto, el PLN tiene un gran potencial, por lo que intentar resolver alguno de sus problemas más generales, como son el de "Responder a preguntas", también lo tendrá.

Capítulo 3

Contexto de la generación de texto

En este capítulo se va a comenzar explicando los principales modelos que han contribuido al desarrollo del modelo BERT (*Bidirectional Encoder Representations*), el cual es el que utilizaremos en el caso de estudio.

Algunos de los conceptos empleados en este apartado, están explicados en el Anexo de esta memoria.

3.1 Glosario de términos

En este apartado se resumen conceptos que se utilizan dentro del área del PLN y que serán necesarios para entender los siguientes apartados de este capítulo. Algunos serán desarrollados más adelante.

- **Arquitectura:** describe un enfoque general de un problema de IA (Inteligencia Artificial) y la parametrización de ese enfoque. Por ejemplo, una arquitectura de red neuronal definiría el número y tamaño de diferentes capas, el tipo de cada capa, etc.

- ***Embedding*:** representación de palabras en términos de vectores numéricos consecutivos, por lo que obtendremos matrices por cada sentencia. Una sentencia (formada por varias palabras) se representará mediante una matriz.

- ***Fine tunning*:** es la acción de utilizar un modelo de lenguaje para resolver un problema específico.

- **kernel_initializer** término utilizado para la distribución estadística o función que se utiliza para inicializar los pesos. En caso de distribución estadística, generará números a partir de esa distribución estadística y los utilizará como pesos iniciales.

- **Lingüística**: Ciencia que estudia el lenguaje humano y las lenguas.

- **Modelo:** una instancia específica de una arquitectura dada, entrenada en un conjunto de datos dado. Para el ejemplo de las redes neuronales, el modelo consistiría principalmente en los pesos entrenados de cada nodo de la arquitectura.

- **Modelo del lenguaje:** es un tipo de modelo que analizan conjuntos de datos de texto para proporcionar una base para predecir palabras.

- **Red neuronal densamente conectada**: es una red neural completamente conectadas entre sí. Cada neurona en una capa recibe una entrada de todas las neuronas presentes en la capa anterior. Su utilidad es la de proporcionar características de aprendizaje de todas las combinaciones de las características de la capa anterior (no en campos repetitivos).

- ***Softmax***: es un tipo de función de activación de una neurona utilizada para convertir su entrada en valores dentro del conjunto [0,1], de modo que la suma de todos los valores de las neuronas de la capa de salida sea la unidad.

- **Sentencia u oración:** conjunto de palabras estructuradas en función de un modelo de lenguaje.

- **Sistema de predicción de respuestas**: es un tipo de sistema que, mediante sentencias, predice palabras como respuesta dada con una pregunta dada. La predicción se realiza porque el sistema ha aprendido previamente cómo predecir una pregunta dada que entender la oración, esto lo puede hacer de dos formas:

  - **Unidireccional**: solo entiende los significados que están relacionados de izquierda a derecha o viceversa, pero no en ambos sentidos a la vez.

  - **Bidireccional (Bidirectional)**: entiende los significados que están relacionados en ambos sentidos tanto de izquierda a derecha como de derecha a izquierda.

- ***Token:*** unidad utilizada en una tarea específica del PLN, la "tokenización" (*tokenitation*) es el paso previo al paso de *embedding*. Su objetivo es dividir el significado para que el modelo sea más eficaz.

3.2 Modelo Secuencia a secuencia

El modelo "Secuencia a Secuencia" o Seq2Seq es una familia de enfoques de aprendizaje automático, y han alcanzado mucho éxito en tareas como traducción automática, resumen de texto y subtítulos de imágenes. Google Translate comenzó a utilizar este modelo, en producción, a finales de 2016.

Este modelo consigue que cada parte de sus salidas, dependan de sus anteriores entradas y salidas, pero no de las futuras y esto en el procesamiento de una frase en lenguaje natural es importante, para entender todo el contexto, antes de obtener un resultado definitivo.

Por tanto, son útiles cuando no se requiere conocer en gran detalle el significado completo de la secuencia. Sin embargo, para lograr un mayor entendimiento se requerirán otro tipo de modelos, pero describiremos primero el modelo Seq2Seq.

El modelo está compuesto por un codificador (*Encoder*) y descodificador (*Decoder*). El codificador captura la información de la secuencia de entrada en forma de un vector de estado oculto o contexto y lo envía al decodificador, que luego produce la secuencia de salida.

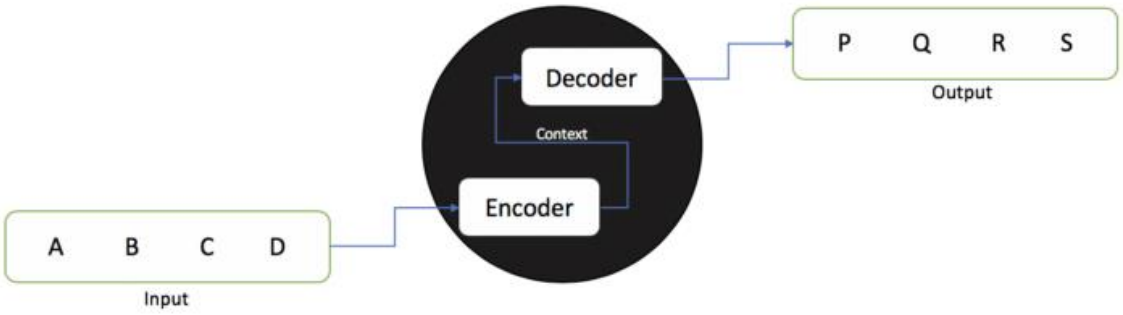


Figura 8 Esquema general del modelo Seq2Seq

Como se muestra en la figura anterior, un modelo Seq2Seq es un modelo que toma una secuencia de elementos (palabras, letras, series de tiempo, etc.) y genera otra secuencia de elementos.

A continuación, se explicará el funcionamiento del modelo Seq2Seq.

El codificador y el decodificador utilizan normalmente redes neuronales recurrentes. En la siguiente figura se detalla la arquitectura interna del modelo Seq2Seq:

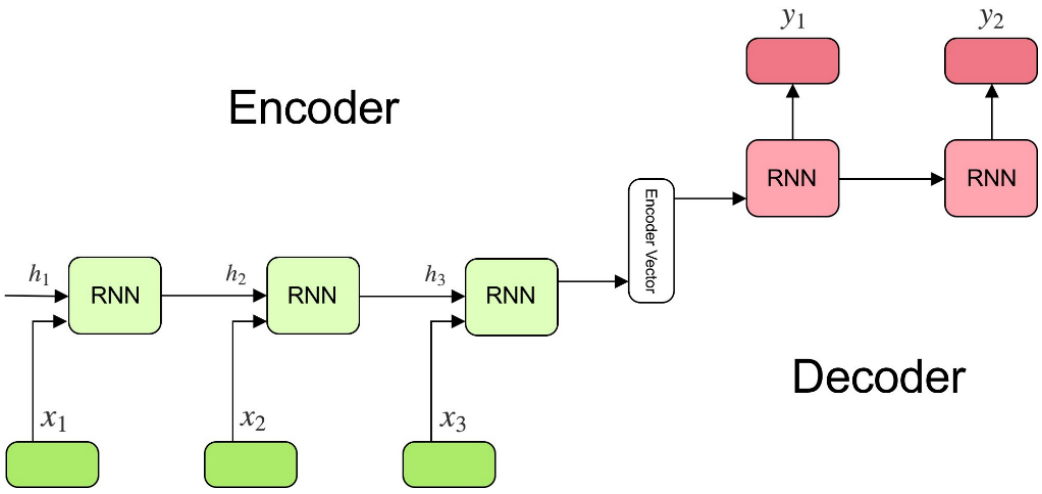


Figura 9 Arquitectura interna del modelo Seq2Seq

Como se muestra en la figura de arriba, vemos que cada parte está compuesta por celdas. Estas celdas son idénticas y hacen exactamente el mismo trabajo. Para generalizar, llamaremos celdas a cada red neuronal, tal y como se muestra en la figura siguiente:

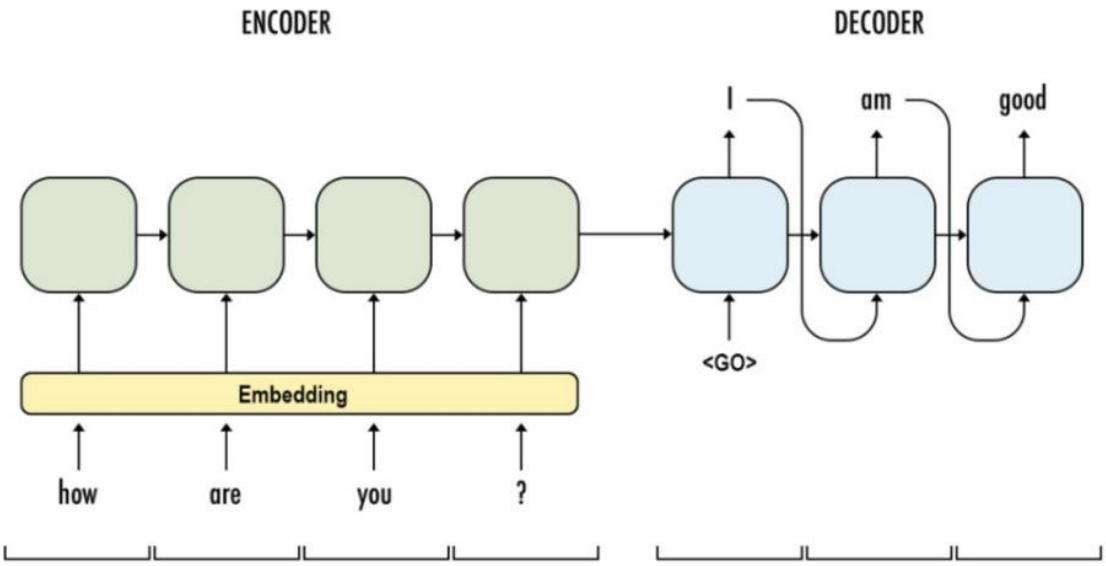


Figura 10 Modelo de caja negra del codificador y del decodificador

La figura anterior muestra qué entradas y salidas nos proporciona cada celda del modelo Seq2Seq.

En el lado del codificador, la primera celda solo recibe su correspondiente palabra de entrada. Las siguientes celdas reciben como entradas la salida de la anterior celda y la entrada correspondiente. Finalmente, la última celda proporciona la entrada al decodificador, y esta entrada es donde aparece el contexto completo del codificador.

En general, el codificador pretende resumir toda la información útil capturada a partir del texto de entrada y el decodificador utilizará esta información para crear nuestra salida correcta.

La primera celda del decodificador utiliza como entradas, la salida del codificador y un *token* de inicio. A diferencia de las celdas del codificador, las celdas del decodificador tienen dos salidas, una que permitirá utilizarse como salida de esa celda del decodificador y otra interna. Ambas serán entradas de la celda siguiente. El conjunto de salidas individuales asociadas a cada celda del decodificador describirá la salida final del modelo.

Como se ha mencionado previamente lo habitual es que cada celda se implemente mediante una Red Neuronal Recurrente (RNN). Brevemente, la forma más común de hacerlo es usando redes neuronales recurrentes. La siguiente figura muestra las entradas que recibe una RNN, así como las salidas que proporciona.

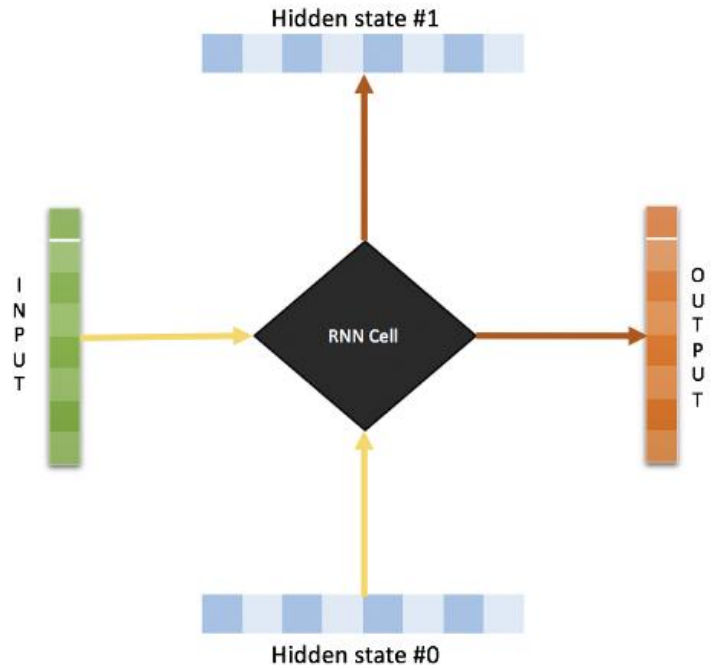


Figura 11 Funcionamiento interno de una RNN

Las RNN, por diseño, admiten dos entradas, una entrada podrá ser un *token* de entrada o la representación de la celda anterior. Así mismo, cada celda tiene como salidas, su salida final de la celda o su representación interna de su estado. Por lo tanto, la salida en el paso de tiempo t depende de la entrada actual, así como de la entrada en el tiempo t-1. Así pues, en este tipo de modelos la información secuencial, se conserva en un estado oculto de la red y se utiliza en las siguientes instancias. Esta es la razón por la que este modelo se utiliza para Sistemas de Predicción Unidireccionales.

La siguiente figura muestra cómo se reparten los estados ocultos (*Hidden states)*:

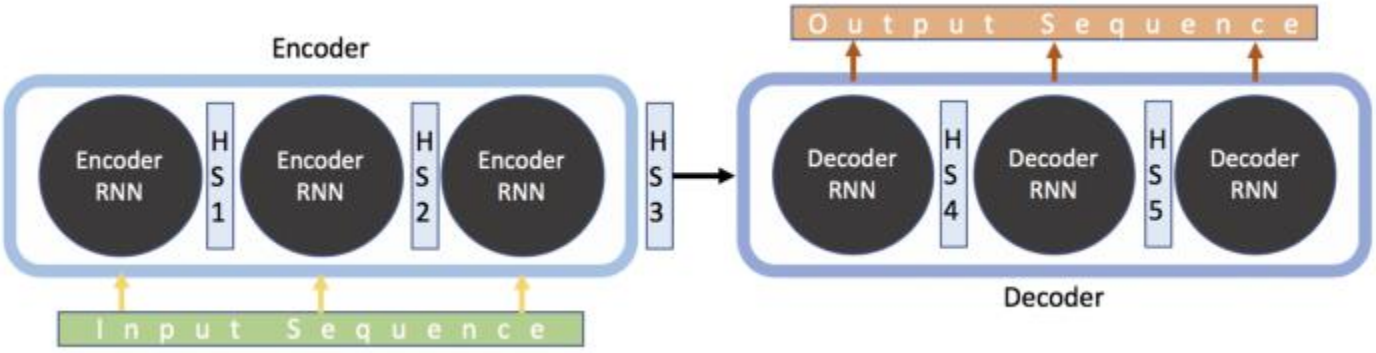


Figura 12 Modelo Seq2Seq enfocado en los contextos

En la mayoría de problemas, cada celda suele ser una red neuronal recurrente de tipo LSTM (*Long Short-Term Memory*), un tipo especial de red que trata el problema del desvanecimiento de gradiente(*vanishing gradient*).

Como se muestra en la siguiente figura, donde se muestra el interior de una red LSTM, estás analizan tres componentes: un componente para eliminar partes del contexto, otro para añadir y otro para actualizar. De esta forma son mejores cuando gestionan el contexto, y proporcion una salida mejor a la siguiente celda.

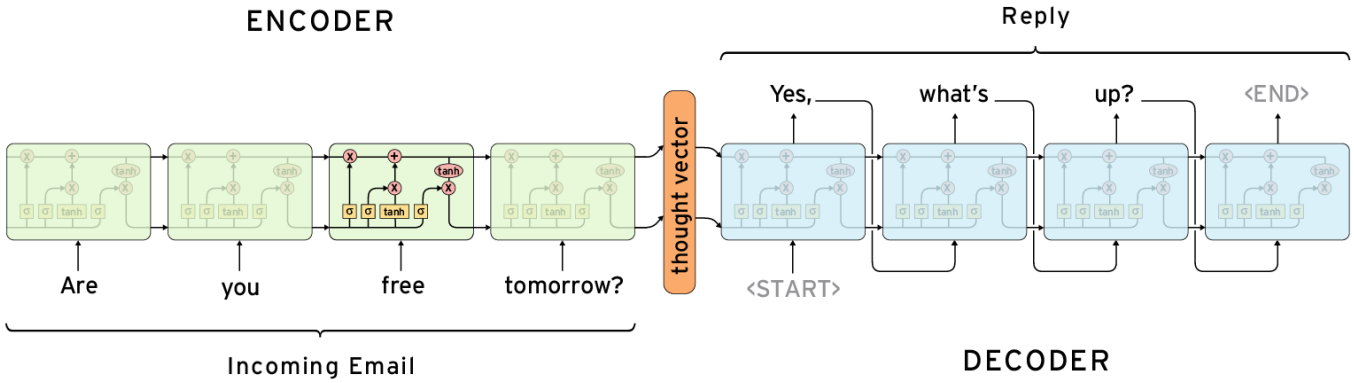


Figura 13 Modelo Seq2Seq utilizando LSTM

El principal inconveniente de este modelo se debe a que la secuencia de salida se basa, en gran medida, en el contexto definido por el estado oculto de la salida final del codificador, lo que dificulta que el modelo maneje oraciones largas. En el caso de secuencias largas, existe una alta probabilidad de que el contexto inicial se haya perdido al final de la secuencia.

3.2.1 Mecanismo de atención en Seq2Seq

Con el objetivo de resolver el problema explicado anteriormente, se añadió al modelo *Seq2Seq,* un mecanismo de atención que implica un cambio en el comportamiento del modelo básico. El objetivo es que cada parte de la salida del modelo dependa de todas las entradas del modelo.

Como se ha explicado, el codificador pasa todos los estados ocultos al decodificador. En este caso el decodificador añadirá un paso adicional antes de producir una salida con el fin de introducir la dependencia.

Así pues, el decodificador hará lo siguiente para cada celda:

1. Observará el conjunto de estados ocultos que el codificador le envió (cada estado oculto del codificador está asociado a una determinada palabra de la oración de entrada).

2. Puntuará cada estado oculto en función de la celda del decodificador, para mejorar la salida de la celda actual.

3. Multiplicará cada estado oculto por su puntuación, amplificando así los estados ocultos con puntuaciones altas y mitigando los estados ocultos con puntuaciones bajos.

El objetivo de este mecanismo es centrarse en las partes de la entrada que son relevantes para cada celda de decodificación.

A pesar de utilizar el mecanismo de atención, la principal razón por la que el modelo Seq2Seq no es tan eficaz frente a la arquitectura *Transformer* se debe a que:

1. Las redes RNN tienen una naturaleza secuencial, es decir, cada estado oculto depende de la salida del estado oculto anterior. Esto se convierte en un gran problema para las GPU (*Graphics Processing Unit*), pues a pesar de tener un enorme poder computacional, se debe esperar hasta que los datos de la red estén disponibles. Esto hace que las RNN no sean adecuada para implementarse con GPUs (que son adecuadas para procesamiento masivo en paralelo).

2. El modelo Seq2Seq es secuencial por lo que, a pesar de incluir mecanismos de atención, no tienen capacidad para importar Sistemas de Predicción Bidireccionales.

3.3 Transformer

El modelo *Seq2Seq* no funcionó tan bien debido a que, a pesar de implementar el mecanismo de atención, las redes recurrentes aún no poseían la capacidad de almacenar la suficiente información requerida en el tiempo.

El modelo basado en *Transformer* elimina las redes recurrentes y crea una nueva arquitectura, cuya estructura se muestra en la siguiente figura:

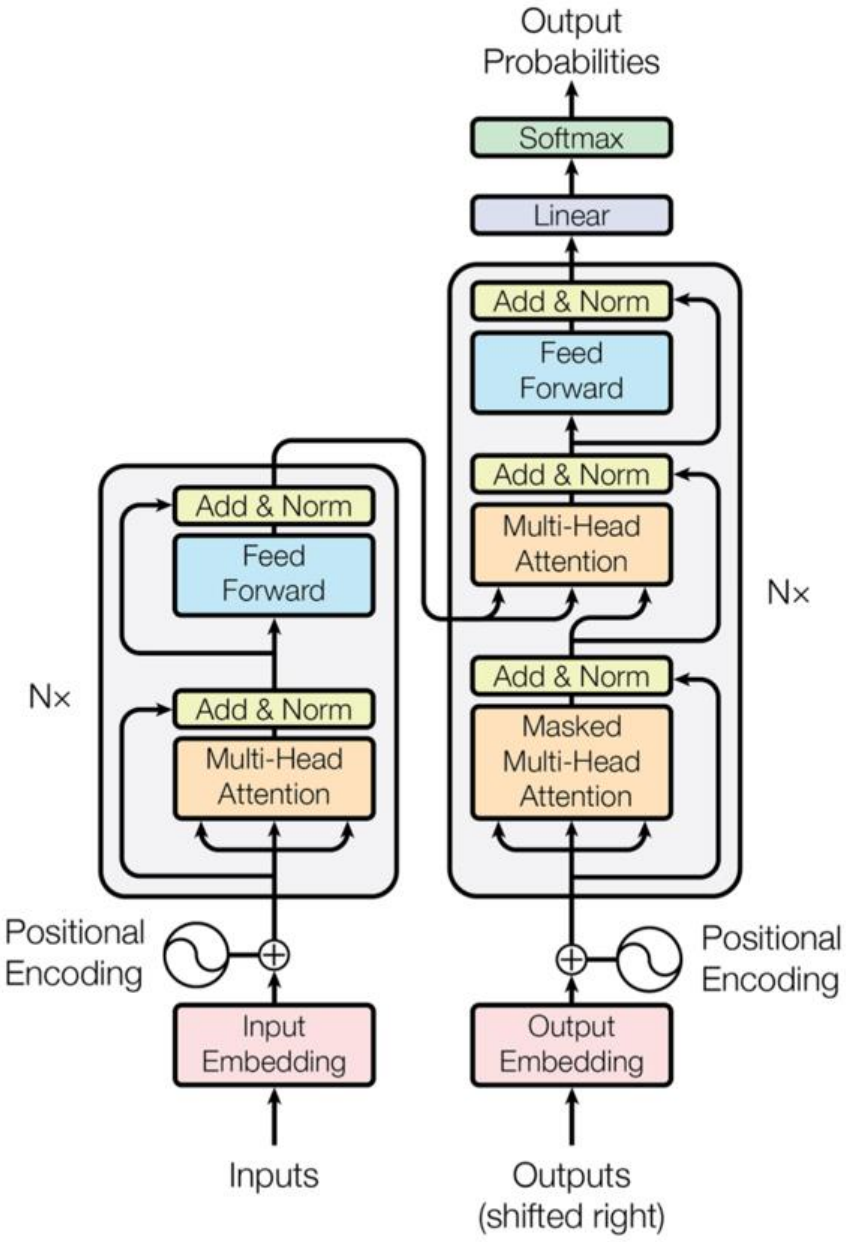


Figura 14 Arquitectura del modelo Transformer

Como el modelo *Seq2Seq*, el modelo *Transformer* también está basado en dos componentes: un decodificador y un codificador. En la figura de arriba se muestra el codificador a la izquierda y el decodificador a la derecha.

Los codificadores son todos idénticos en estructura entre sí, aunque posiblemente, cada uno de ellos tendrá un peso diferente del resto. A su vez, cada uno se divide en dos subcapas, mostradas en la siguiente figura:

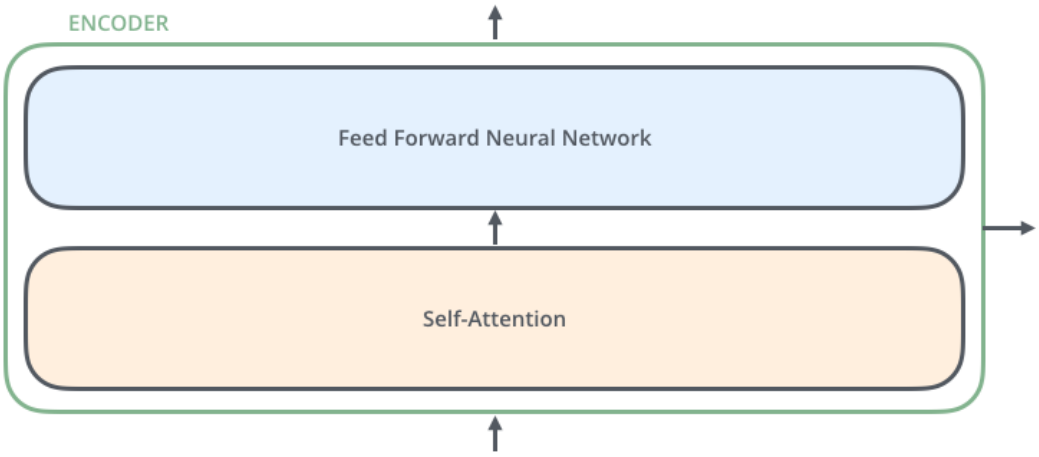


Figura 15 Codificador del modelo Transformer

La figura de arriba muestra las partes fundamentales de un codificador.

En la práctica se alimentará el codificador con toda la secuencia a la vez, y no palabra a palabra por unidad de tiempo. Esta es la razón por la que estos modelos son más potentes, porque permiten mantener la información global en la misma unidad de tiempo.

La salida del decodificador se reutiliza para su entrada, y se utiliza toda la secuencia por unidad de tiempo, del mismo como que como se hizo para el codificador. Cuando empecemos a entrenar el modelo, y no haya entradas para el decodificador, se utilizará un *token* de inicio.

La siguiente figura muestra el funcionamiento general del modelo para *Transformer* dos entradas dadas:

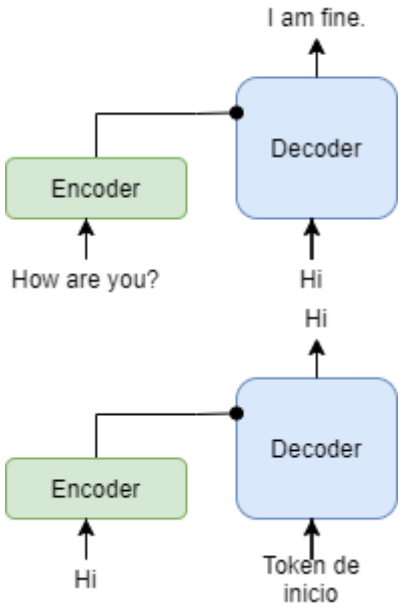


Figura 16 Funcionamiento de las entradas del modelo de Transformer

Con el objetivo de comprender el papel de cada celda de la arquitectura *Transformer* introducimos aquí las siguientes definiciones:

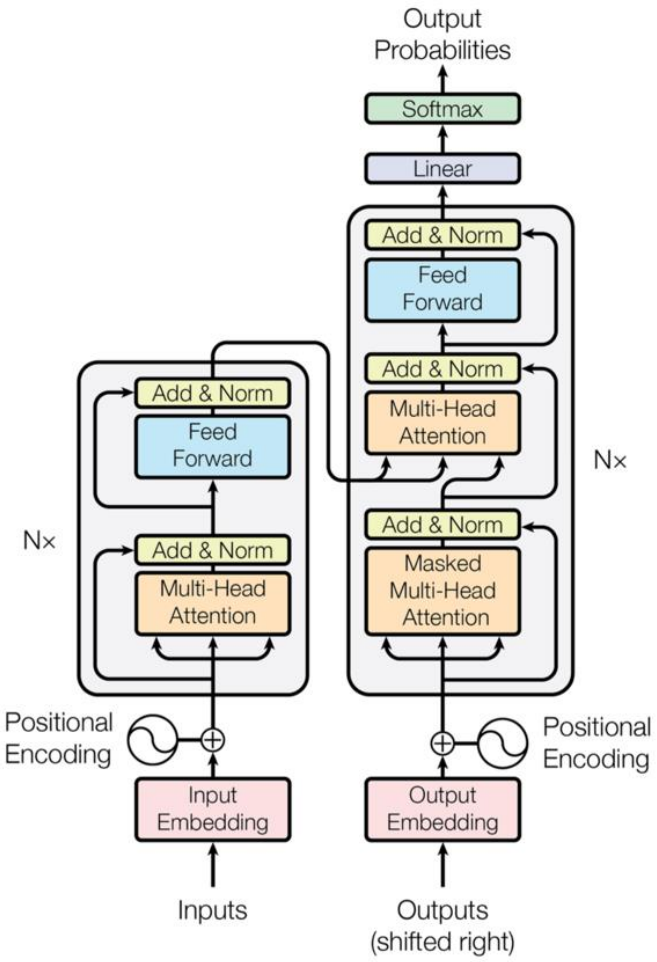
- ***Positional encoding***: Como cada palabra en una oración fluye simultáneamente a través de la pila de codificadores / decodificadores del *Transformer*, el modelo en sí, no tiene ningún sentido de posición / orden para cada palabra. En consecuencia, este proceso se introdujo para determinar qué posición tiene cada *token*.

- ***Add and Norm***: las entradas y salidas, tanto de la capa de atención de múltiples cabezas (*Multi-Head Attention*), como de la red de alimentación directa por posición, son procesadas por dos capas del tipo "agrega y normaliza"(*Add&Norm*), compuestas por una capa para juntar los resultados de la anterior capa y, después de esta, una capa de normalización. Estas capas permiten que la arquitectura *Transformer* utilice toda la secuencia en la misma unidad de tiempo, resolviendo el problema del modelo Seq2Seq, y evitando así ralentizar el procesamiento llevado a cabo por el modelo.

- ***Linear***: red neuronal simple (sin capa oculta) y completamente interconectada cuya misión es proyectar el vector producido por la pila de decodificadores en un vector mucho más grande, denominados vector *logits*.

- ***Feed Forward (FF):*** es la red neuronal clásica compuesta por distintas capas (entrada, salida y al menos una capa oculta), en la que se propaga la señal de entrada hacia delante y puede entrenarse.

- ***Softmax***: regla de cálculo que determina cuánto se expresará cada palabra la posición asociada.



El codificador utilizará la entrada de la capa *embedding*, mientras el mecanismo de atención (*Multi-Head Attention*) captará las diferentes relaciones que tiene cada palabra con el resto de palabras de esa entrada. Luego las capas *Add and Norm* aglutinarán cada *token* respecto a su estado original, y normalizando sus valores, para pasarse a una capa de FF, que tras volver a combinarse su resultado en la última capa *Add and Norm* será entrenado con la diferenciación de la composición de funciones de la celda siguiente. En el caso del decodificador, en general está última fase pretende realizar la predicción final.

Considerando las partes internas del codificador o del decodificador como si solo usaremos dos cajas negras la de atención y la de FF, el subsistema de atención trata de capturar el significado, mientras que el componente FF trata la solución que el modelo específico proporcionará.

Teniendo en cuenta lo anterior, el decodificador tiene dos capas de atención, y su objetivo es tratar dos entradas de forma simultánea en el componente FF. La intención del decodificador es utilizar salidas anteriores para su entrenamiento.

3.3.1 Capa de auto-atención

El objetivo de esta capa es encontrar las relaciones existentes entre palabras de una misma secuencia, y recomponer esas sentencias en función de las relaciones descubiertas. De esta manera, cada palabra representaría no solo esa palabra sino a el resto con el que esté relacionado.

Para entender la importancia del mecanismo de auto-atención, ponemos el ejemplo siguiente:

"I couldn't write with the pencil on the paper because it was torn."

Es difícil saber a qué se refiere en esta frase, si al lápiz o al papel. En este caso, por definición, al papel, pero también podría terminar la frase por "*broken*", y entonces al algoritmo le costaría más encontrar el significado. Situaciones de este tipo son las que han motivado que el mecanismo de atención se haya ido complicándose con el desarrollo de este tipo de sistemas.

Como se explicó en el apartado anterior, las entradas del codificador fluyen primero a través de una capa de auto-atención o *self attention*, una capa que ayuda al codificador a ver otras palabras en la oración de entrada, mientras codifica una palabra específica.

Las salidas de la capa de auto-atención se alimentan a una red neuronal de FF. La misma red de FF se aplica de forma independiente a cada posición.

El decodificador tiene ambas capas, pero entre ellas hay una capa de atención que ayuda al decodificador a enfocarse en las partes relevantes de la oración de entrada (similar a lo que hace la atención en los modelos Seq2Seq).

En la siguiente imagen se ve el modelo al completo.

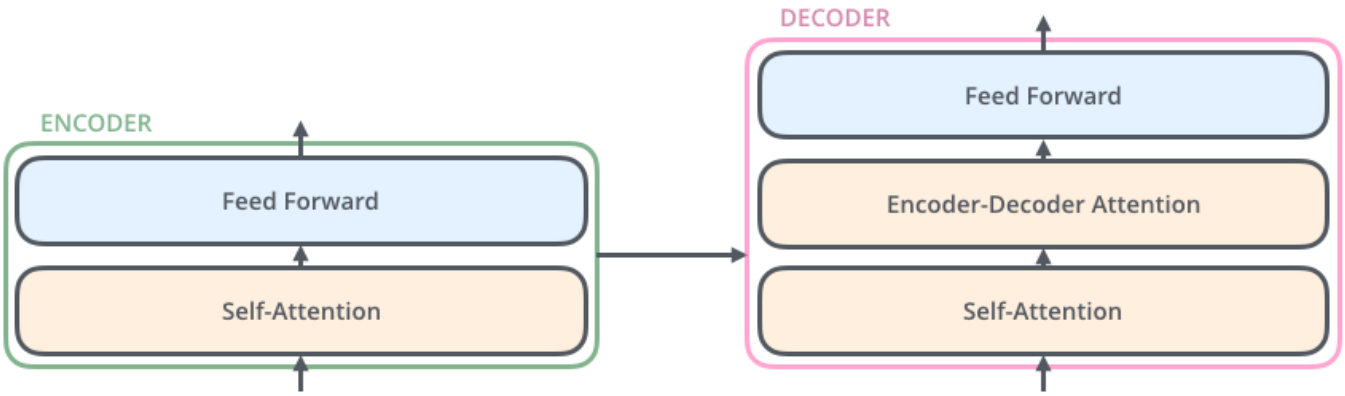


Figura 17 Modelo simplificado de Transformer

El modelo comienza pasando las palabras a vectores (capa *embedding*), esto solo ocurre en la primera capa.

Después, como ya se ha mencionado, un codificador recibe una lista de vectores como entrada. Procesa esta lista pasando estos vectores a una capa de auto-atención, que pasa a una red neuronal de FF. Luego la salida generada se envía hacia arriba al siguiente codificador en la pila, o se pasa a la pila de decodificación.

Paralelamente, como vemos en la siguiente figura, la secuencia se trata en la misma unidad de tiempo, por lo que se incrementará la potencia de la máquina, y se evitarán retardos.

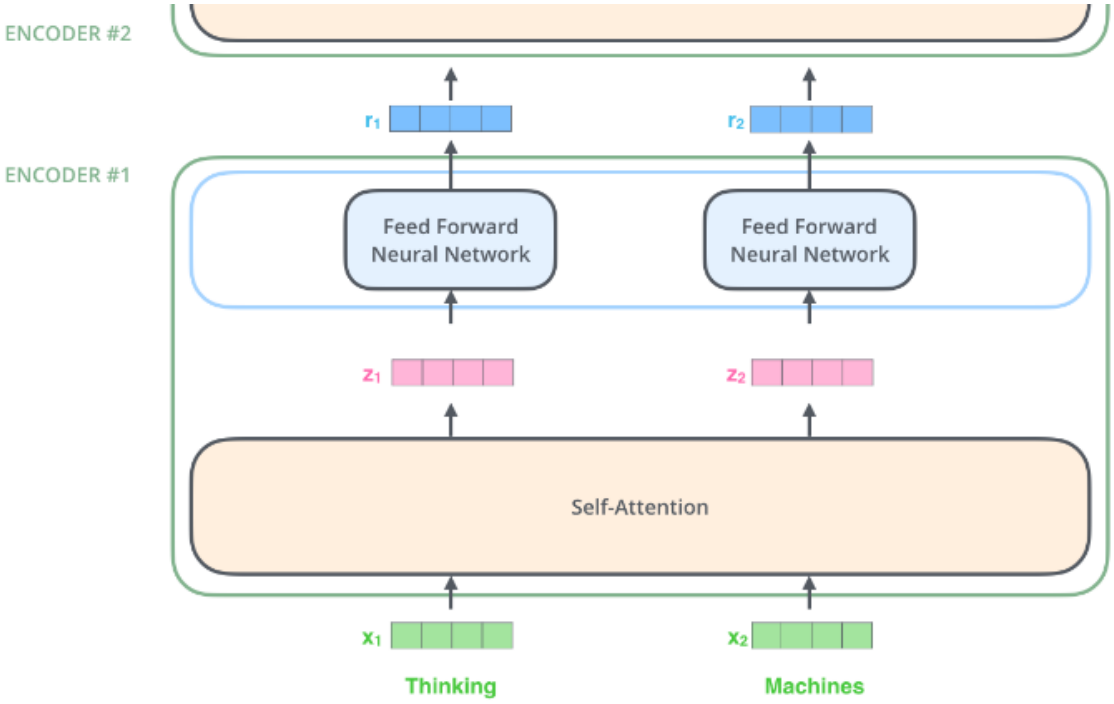


Figura 18 Codificadores apilados de la arquitectura Transformer

A medida que el modelo procesa cada palabra en cada posición de la secuencia de entrada, el mecanismo de auto-atención posibilita buscar relaciones en otras posiciones de la secuencia de entrada que puedan ayudar a aprender una mejor codificación de esta palabra.

En la figura siguiente se muestra una secuencia a la derecha y otra a la izquierda. A la derecha tenemos la palabra "*it*" resaltada y relacionada con cada *token* de la secuencia de la izquierda. Cada *token* de la izquierda representa un valor relativo, con respecto a la palabra "*it*". En este caso se trata de un ejemplo unidimensional, pero como se verá a continuación, se pueden añadir *n* dimensiones al problema.

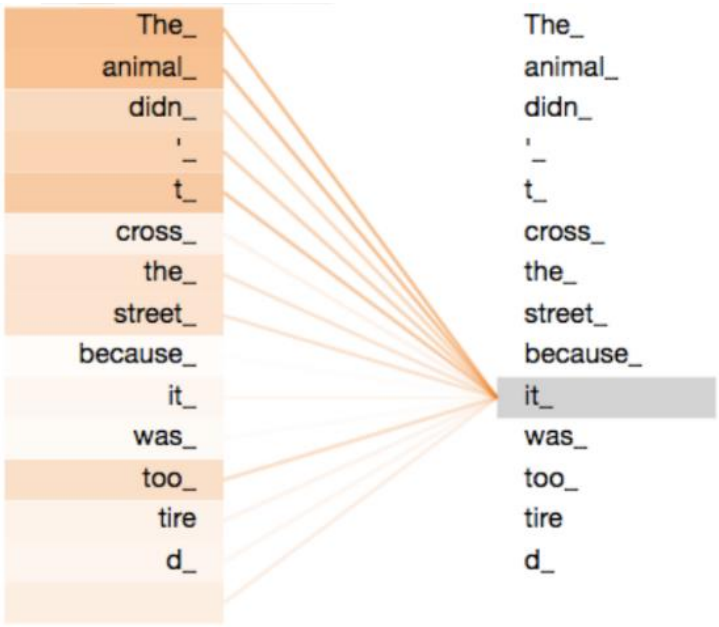


Figura 19 Ejemplo del mecanismo de auto-atención con una variable

En general el mecanismo de auto-atención sigue los siguientes pasos:

- **Primer paso**: Inicialización. La auto-atención funciona mediante tres vectores para cada uno de los vectores de entrada del codificador, es decir, en este paso se crean estos tres vectores por cada *token*. Entonces, para cada palabra, se crea un vector de **consulta** *(queries)*, un vector **clave** *(keys)* y un vector de **valor** *(values)*.

  Estos vectores se crean multiplicando la entrada por tres matrices cuyos valores se establecen durante el proceso de entrenamiento.

  La siguiente figura muestra, a modo de ejemplo, un esquema general de los vectores que se tienen para los *tokens* "*Thinking*" y "*Machines*".

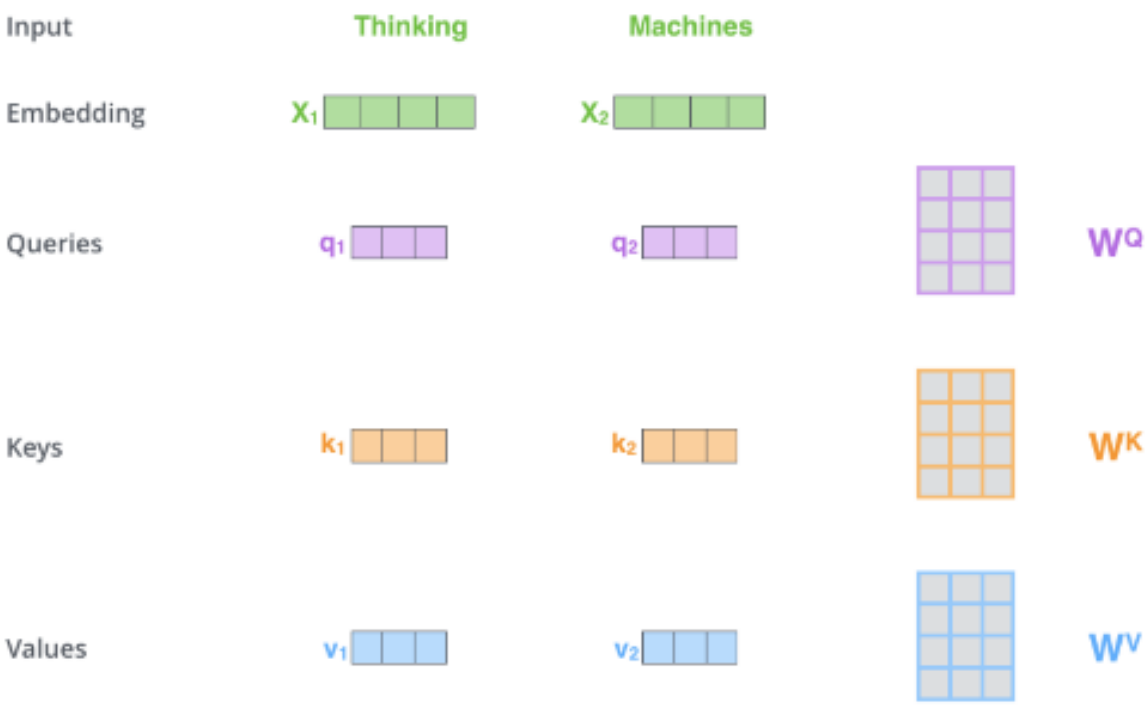


Figura 20 Ejemplo de variables de entrada para el mecanismo de auto-atención

- **Segundo paso**: Puntuación. Para ilustrar la regla de puntuación, se utilizará la siguiente figura, que describe los diferentes pasos para puntuar cada palabra de la oración de entrada respecto al término "*Thinking*". Utilizando la siguiente figura, para la palabra "*Thinking*" se necesita calificar cada palabra de la oración de entrada con esta palabra. La puntuación determinará la prioridad de una palabra en específico respecto a otras partes de la oración.

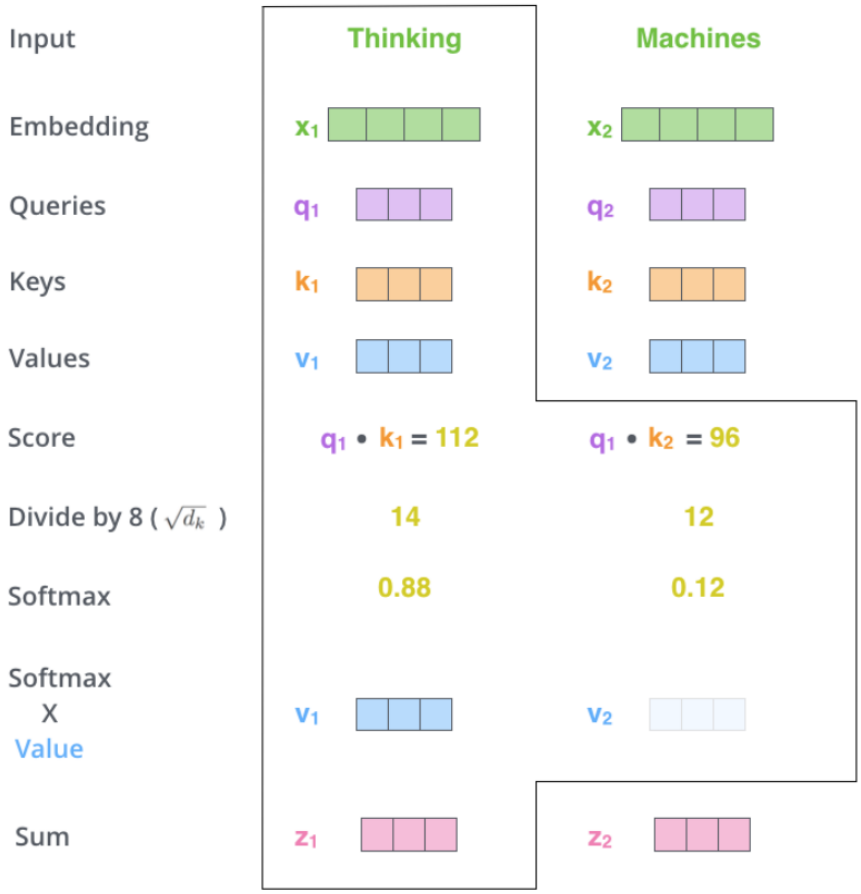


Figura 21 Ejemplo para las operaciones del mecanismo de auto-atención

La puntuación se calcula mediante el producto escalar del vector de **consulta** con el respectivo vector **clave** de la palabra que se está puntuando. Para la primera palabra, la primera puntuación sería el producto escalar de q_1 y k_1. La segunda puntuación sería el producto escalar de q_1 y k_2, y así, sucesivamente.

- **Tercer paso**: Escalado. en este caso particular, la dimensión del vector de entrada es 64, pero depende del vector utilizado como entrada. El escalado consiste en dividir las puntuaciones por 8 para este ejemplo (en general el factor de escalado es la raíz cuadrada de la dimensión del vector de entrada, en este caos, $8=\sqrt{64}$). Esto conduce a tener gradientes más estables, y que no se vean muy afectados por la multiplicación del paso anterior. En otros modelos, la regla del escalado puedo cambiar, pero ésta es la más común.

- **Cuarto paso**: Normalización. Los resultados obtenidos tras escalar los resultados por una capa de *softmax,* que normaliza las puntuaciones para que todas sean positivas y sumen 1. Esta puntuación *softmax* determina cuánto se expresará cada palabra en esta posición. Claramente, la palabra en esta posición tendrá la puntuación *softmax* más alta, pero a veces, es útil prestar atención a otra palabra que sea relevante para la palabra actual.

- Q**uinto paso**: ponderación. Consiste en multiplicar cada vector de **valor** por la puntuación *softmax*. Se pretende mantener intactos los valores de las palabras en las que queremos centrarnos y eliminar las palabras irrelevantes (multiplicándolas por números muy pequeños como 0,001, por ejemplo).

- **Sexto paso**: agregación. Se suman los vectores de **valor**, una vez ponderados, produciendo la salida de la capa de auto-atención para esta posición.

Finalmente, dado que se está tratando con matrices, podemos condensar los pasos del dos al seis en una fórmula para calcular los resultados de la capa de auto-atención.

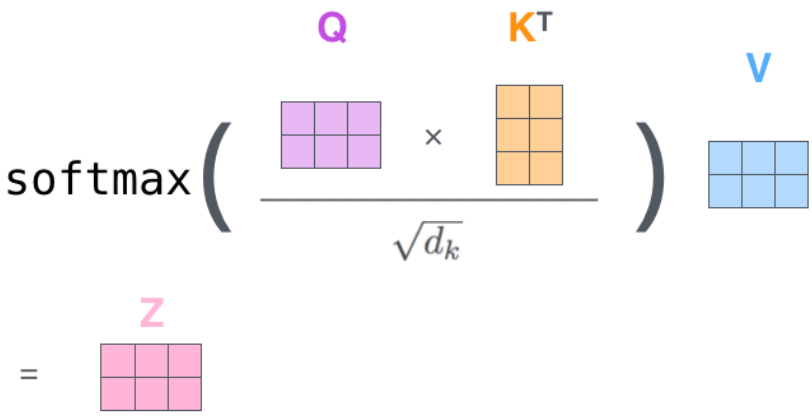


Figura 22 Ecuación simplificada del mecanismo de auto-atención

La arquitectura inicial se perfeccionó aún más, al agregar un mecanismo llamado atención de "múltiples cabezas" (*Multi-head attention)*. Mejoró el rendimiento de la capa de atención, mediante las siguientes dos formas:

En primer lugar, amplió la capacidad del modelo para enfocarse en diferentes posiciones. En el ejemplo anterior, z_1 (el resultado final de la capa) contiene un poco de información de cualquier otro *token*, pero podría darse el caso ed estar dominado por la propia palabra. Por lo que, sería más útil si se considera una frase como "*The animal don´t cross the street because it was tired.*", y se quisiera saber a qué palabra se refiere "*it*", se tendría que dar más relevancia a "*animal*" que a "*it*". En el caso de "*animal*" su palabra más relevante es ella misma.

En segundo lugar, se le da a la capa de atención múltiples "subespacios de representación". La atención de múltiples cabezas nos permite, no solo tener un conjunto de matrices (del tipo clave, consulta y valor), sino múltiples conjuntos de matrices de ponderación del tipo consulta, clave y valor. Cada uno de estos conjuntos se inicializa aleatoriamente cuando se comienza a realizar el entrenamiento

En la siguiente figura se muestran dos espacios de representación para el ejemplo anterior.

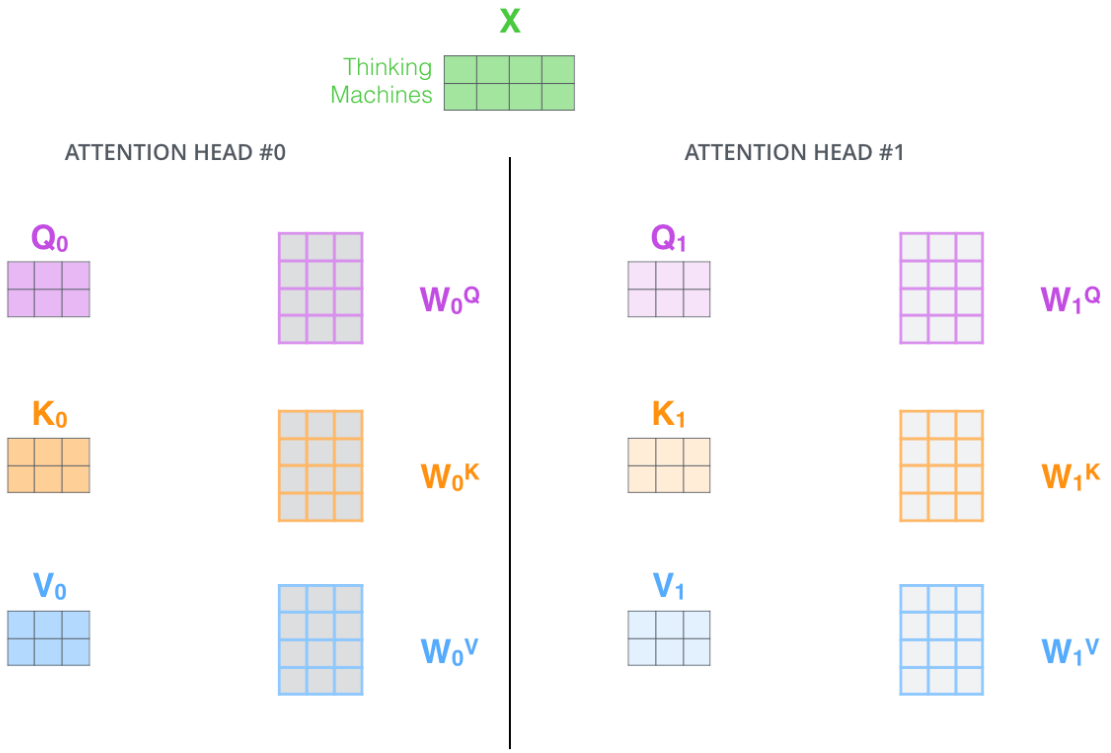


Figura 23 Ejemplo de subconjuntos del mecanismo auto-atención

Si se realiza el mismo cálculo de auto-atención que se describió anteriormente, hasta ocho veces más (igual a la raíz cuadrada de la dimensión) con diferentes matrices de peso, y se terminara con ocho matrices Z diferentes como resultados de la capa de *Multi-head attention.*

Cada una de estas matrices Z describirán el resultado final de la capa, el que utilizaremos para la red neuronal densa.

La figura siguiente muestra un esquema de caja negra del mecanismo de atención.

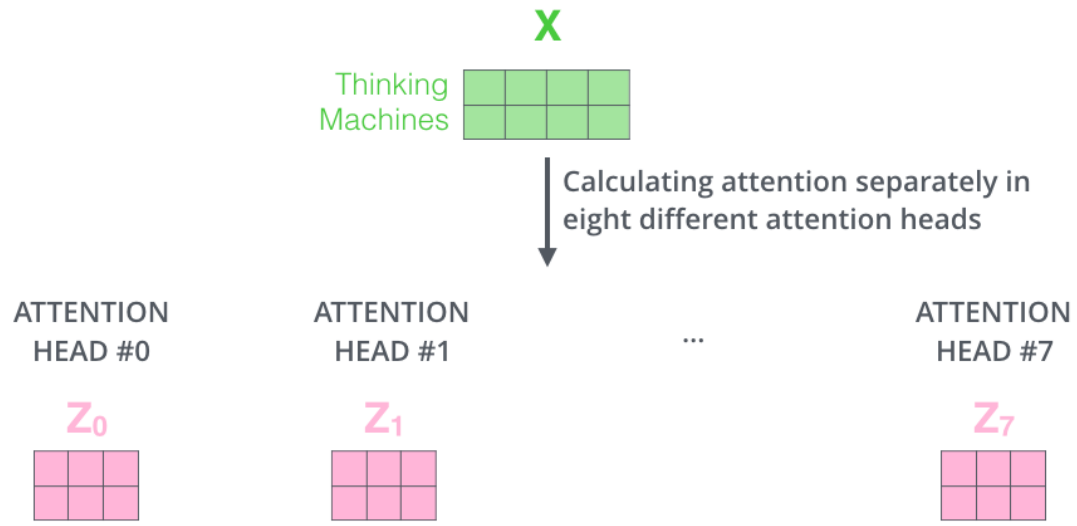


Figura 24 Ejemplo del resultado del mecanismo de multi-atención

Sin embargo, la capa siguiente no espera ocho matrices, espera una sola matriz (un vector para cada palabra). Entonces, se realizan operaciones para combinar estas ocho matrices en una única matriz.

Para ello, se concatenan las matrices y luego se multiplican por una matriz de pesos adicional W^O.

Finalmente, la siguiente figura muestra la representación para el caso del término "*it*" que apareció en el ejemplo introducido anteriormente.

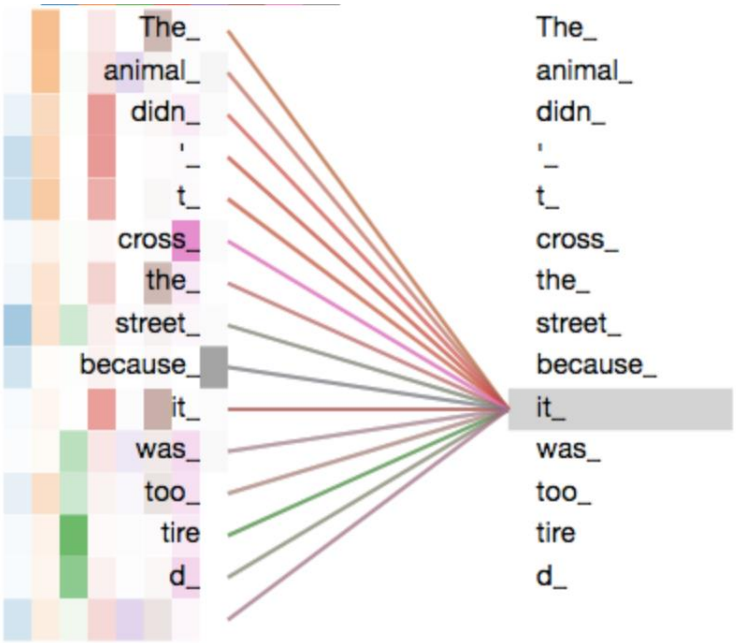
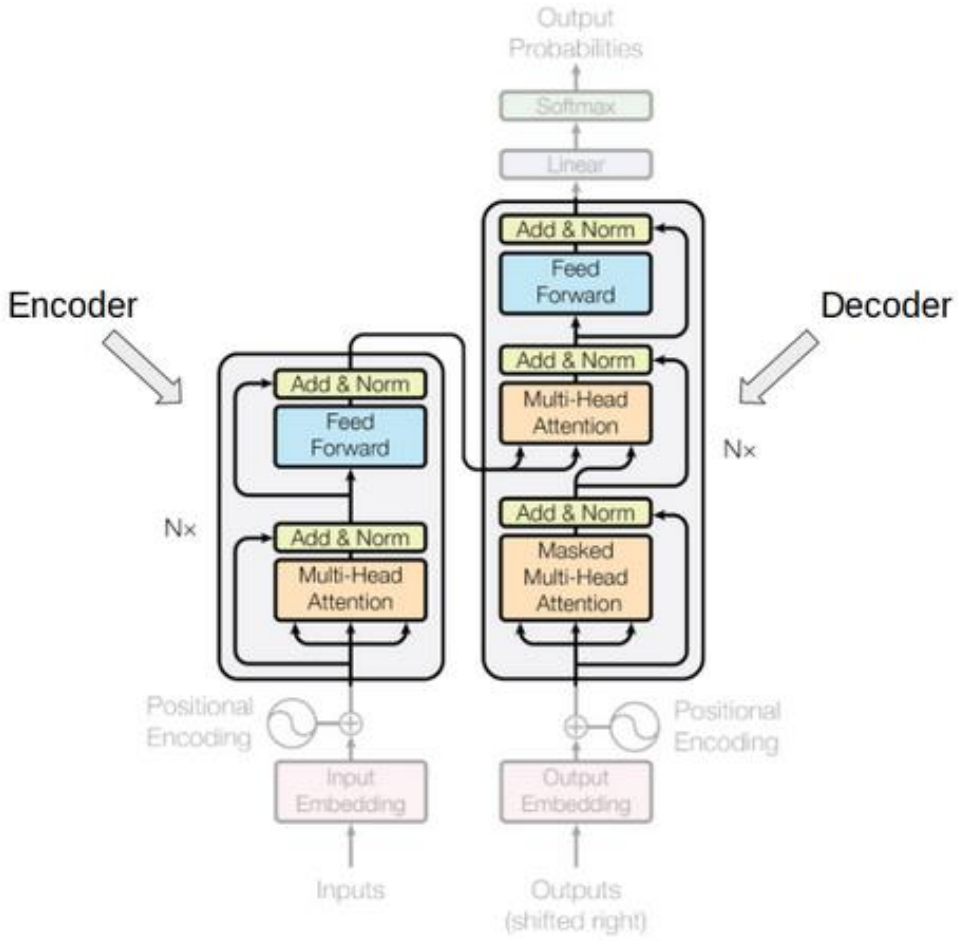


Figura 25 Atención para un token con multi-variable

En la arquitectura *Transformer*, tal y como se muestra en la siguiente figura, la capa de atención aparece tres veces; una en el codificador y dos en el decodificador.

Para las dos primeras capas de atención del codificador y del decodificador, este proceso de atención se hace varias veces con el objetivo de obtener la mayor información.



Para la segunda capa de atención del decodificador solo K y V son iguales, donde utilizará la consulta o contexto Q obtenido en el paso previo de atención del decodificador. Obteniéndose la información relevante con respecto a la salida anterior del decodificador.

Hay que tener en cuenta que la capa de atención del decodificador solo utiliza la anterior salida utilizando una operación llamada *Look ahead mask,* para solo utilizar el paso anterior. Para ello multiplica la entrada a esta capa por una matriz con el objetivo de anular las partes anteriores.

La siguiente muestra cómo se compone cada capa de atención de la arquitectura *Transformer*.

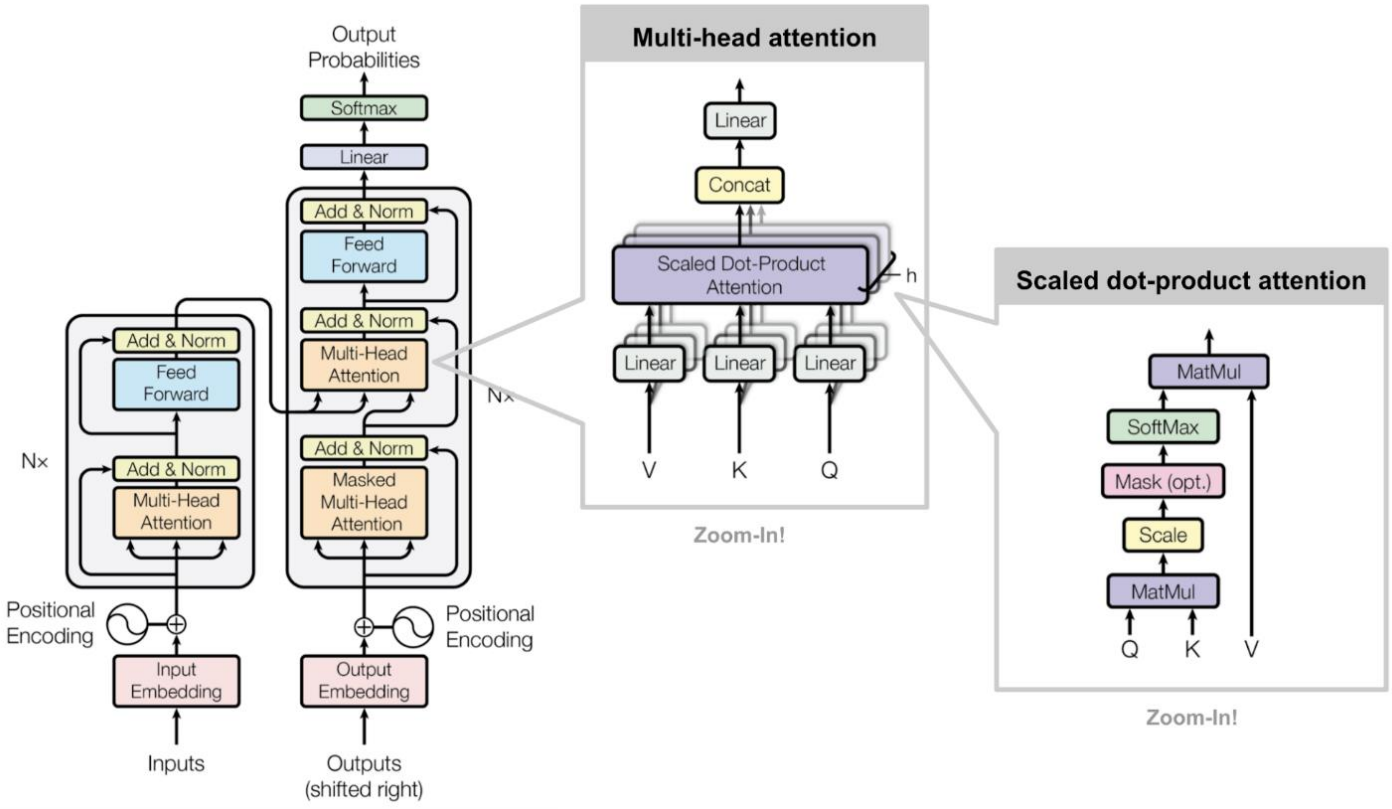


Figura 26 Detalle del mecanismo de atención en Transformer

Una vez entendido todo el proceso general, la capa de *Multi-head attention* se basa en linealizar para crear una combinación óptima de los subespacios, después se pasa por capa de atención, se concatenan y se vuelven a linealizar para obtener la mejor representación de los resultados.

Existen muchos modelos que, inspirados por la arquitectura *Transformer,* han construido sus propios modelos, de los cuales destacan dos modelos:

- **BERT** (*Bidirectional Encoder Representations from Tranformer*): El modelo está basado en organizar la parte del codificador de la arquitectura *Transformer* en varias capas. También utiliza otras técnicas para el tratamiento de los datos.

- **GPT** (Generative Pre-trained Transformer)**:** En este caso el modelo está basado en apilar en la parte del decodificador de la arquitectura *Transformer,* varias capas. Aunque, como BERT, también utiliza otras técnicas. La versión más reciente de este modelo es la 3 (GPT - 3), lanzada en mayo del 2020, la cual no es ofrecida al público, pero únicamente se permite su evaluación. Su anterior, es decir GPT-2, sí permite su uso para diferentes tareas, aunque, con respecto al número de parámetros, es de dos órdenes de magnitud inferior al modelo actual.

En la siguiente figura se muestre la evolución en parámetros de varios modelos entre ellos BERT y GPT-3.

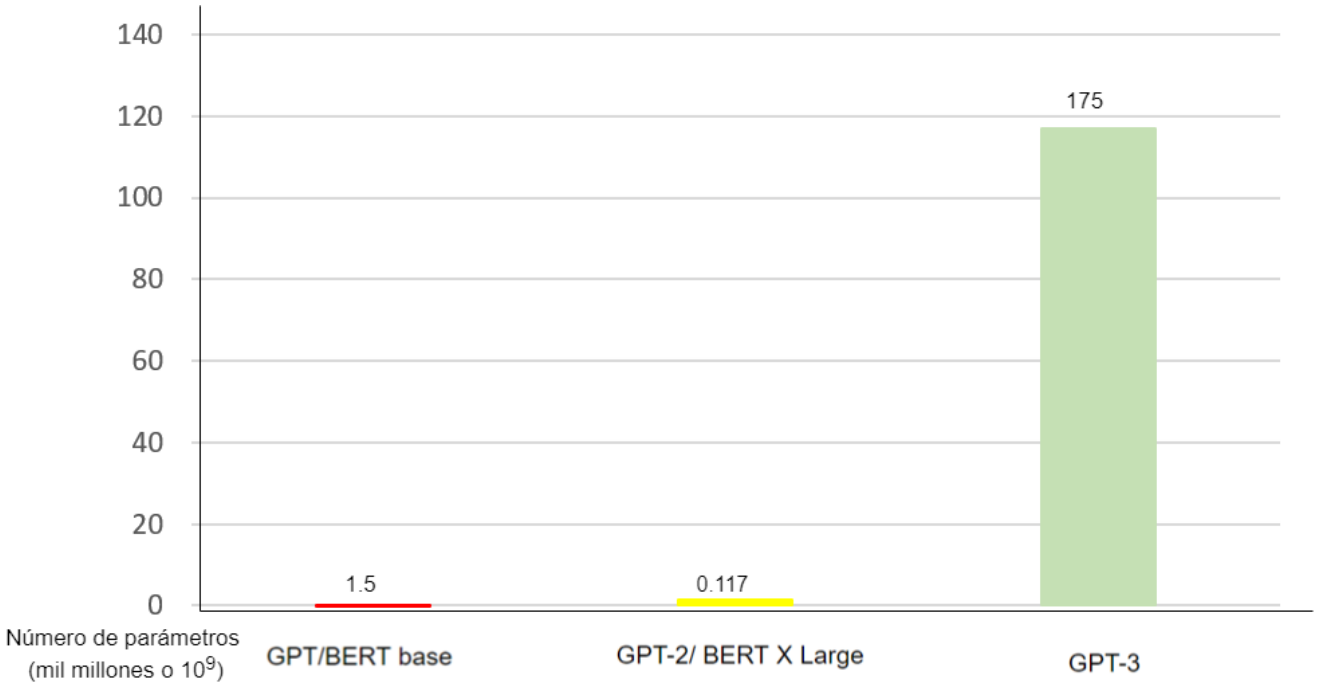


Figura 27 Comparación del número de parámetros de los principales modelos basados en Transformer

3.4 Fine tunning

En este aparto describiremos la idea general del aprendizaje por transferencia (*Transfer learning*). La idea es tratar de crear un modelo para resolver un problema general y luego lo se usa para abordar otra tarea que está relacionada.

En nuestro caso, el modelo del lenguaje se entrenará empleando un gran *corpus* compuesto por información del Wikipedia, entre otros sitios web. El modelo se entrenará de forma semi supervisada, es decir, no hay una tarea específica con el objetivo de ser capaces de producir un modelo de representación de palabras a partir de los textos.

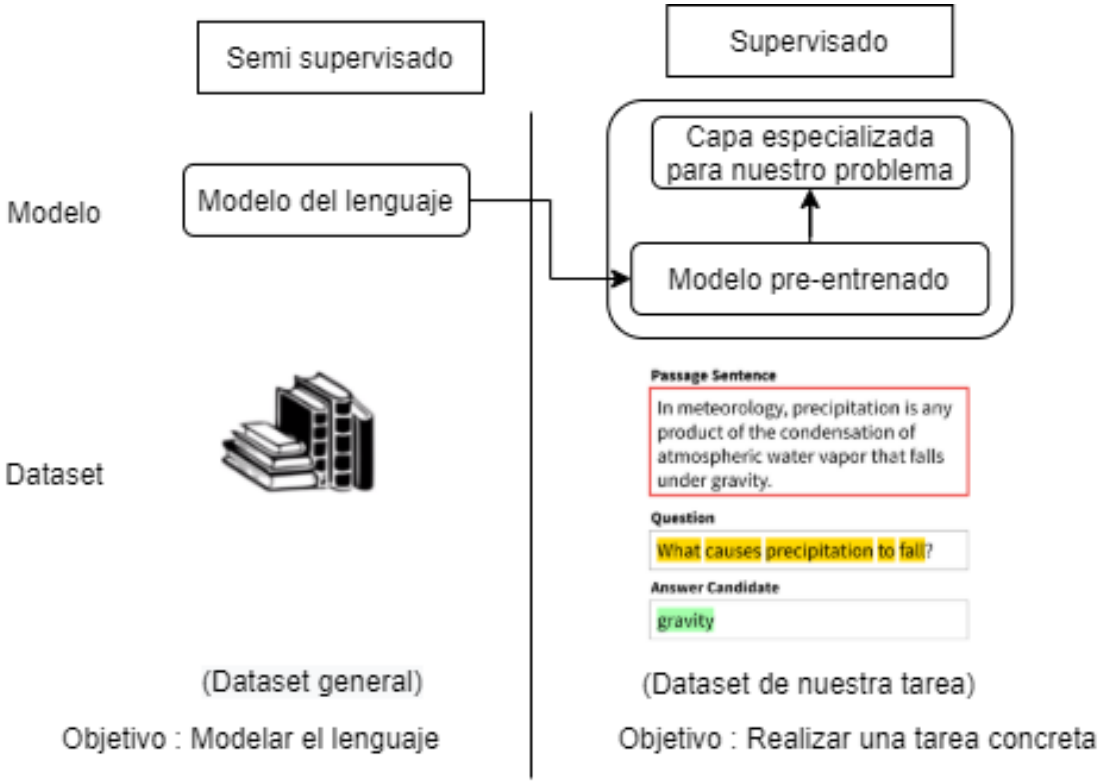


Figura 28 Aprendizaje por transferencia

Como ejemplo, se supone que se oculta la última parte de la siguiente frase.

How are you today? I am fine, thank you.

Y pedimos a nuestro modelo que sea capaz de predecir las siguientes palabras. Es decir, como entradas y salidas de nuestro modelo, tendremos:

Entrada: How are you today?

Salida: I am fine, thank you.

Esto significa que el modelo entiende cómo funciona el contexto, es capaz de relacionar las palabras entre sí, y por último dar un significado final a la frase.

Así pues, una vez que tenemos este modelo general del lenguaje, se puede introducir nuestro *dataset* como entrada y obtendremos una representación asociada a este *dataset* mediante vectores. Acerca de estos vectores, palabras y contextos, se suponen que transmiten la información de la forma más eficiente y más versátil posible.

Como nuestro objetivo es aplicarlo a una tarea específica, tenemos que poder transferir lo que aprendimos a nuestro modelo para una tarea específica, para la cual basta con añadir una pequeña capa de salida al final.

Esta capa de salida será entrenada en esta fase supervisada y nos permitirá resolver nuestra tarea específica. En nuestra aplicación práctica usaremos una red neuronal densamente conectada.

La fase semi supervisada se llama el pre-entrenamiento y a la parte supervisada es la denomina ajuste fino o *fine tunning*.

El *fine tunning* es mucho menos pesado que el pre-entrenamiento porque necesitaremos muchos menos datos. Así mismo, también requiere mucho menos tiempo porque hay una gran parte del trabajo que ya se ha hecho, en concreto, todo aquello referido a la comprensión del lenguaje.

3.5 Modelo BERT

BERT (*Bidirectional Encoder Representations from Transformer*) es un sistema que modela el lenguaje, con el fin de dar una representación de una secuencia, en forma numérica.

Fue revolucionario porque pudo combinar la arquitectura *Transformer* con el modelo bidireccional.

Los conceptos "Bidirectional", "Encoder", y de "Transformer" son conceptos que se explican aquí brevemente, para información más detallada, consúltense sus respectivos apartados o el Anexo. Brevementes:

- Se dice que el modelo es "*Bidirectional*", porque utiliza, para cada *token,* el resto de *tokens* para predecirlo.
- El que se diga que el modelo es "*Encoder Representation from Transformer*" se refiere a que solo utiliza una representación de la parte de codificación de la arquitectura *Transformer.*

La combinación de estos conceptos agrupados implica que BERT esté compuesto por un conjunto de modelos agrupados que pretenden comportarse como un modelo no supervisado, persiguiendo la mejor representación posible de las palabras y secuencias, en términos de eficiencia y flexibilidad.

El sistema que se va a utilizar fue construido por Google, y entrenado con un *corpus* grande, durante mucho tiempo y requiriendo mucha potencia de procesamiento para el establecimiento de los pesos del algoritmo.

Obtendremos el modelo BERT pre-entrenado a través de la página de Tensorflow, donde está subido un modelo ya listo para utilizarse dentro de este entorno. Es decir, han construido un modelo, lo han entrenado y lo han subido a esta página, pero el código del modelo también está a libre disposición del público y se puede ver cómo está construido.

Nota: En la bibliografía están los enlaces para descargar el modelo desde la página de Tensorflow.

Por tanto, utilizaremos BERT para realizar un *fine tuning*, un entrenamiento que requerirá poca potencia de procesamiento en comparación al modelo original del lenguaje, que es la parte del proceso más intensiva, computacionalmente, y que ya está realizada, en este caso por Google.

Por tanto, BERT se caracteriza porque:

- No es un modelo específico, es un modelo general que se utiliza como herramienta para obtener un modelo de lenguaje.
- Produjo un mayor entendimiento del algoritmo en las palabras y sentencias del contexto.
- Algoritmo desarrollado por Google publicado a finales del 2018.
- Fue uno de los mayores hitos en 6 años, afirmó Google, clasificándose como el mejor en muchas tareas.
- Está siendo usado en el motor de búsqueda de Google.

Existen muchos modelos que han surgido a empleándose en BERT. La siguiente figura muestra estos modelos, y a partir de dónde se han obtenido.

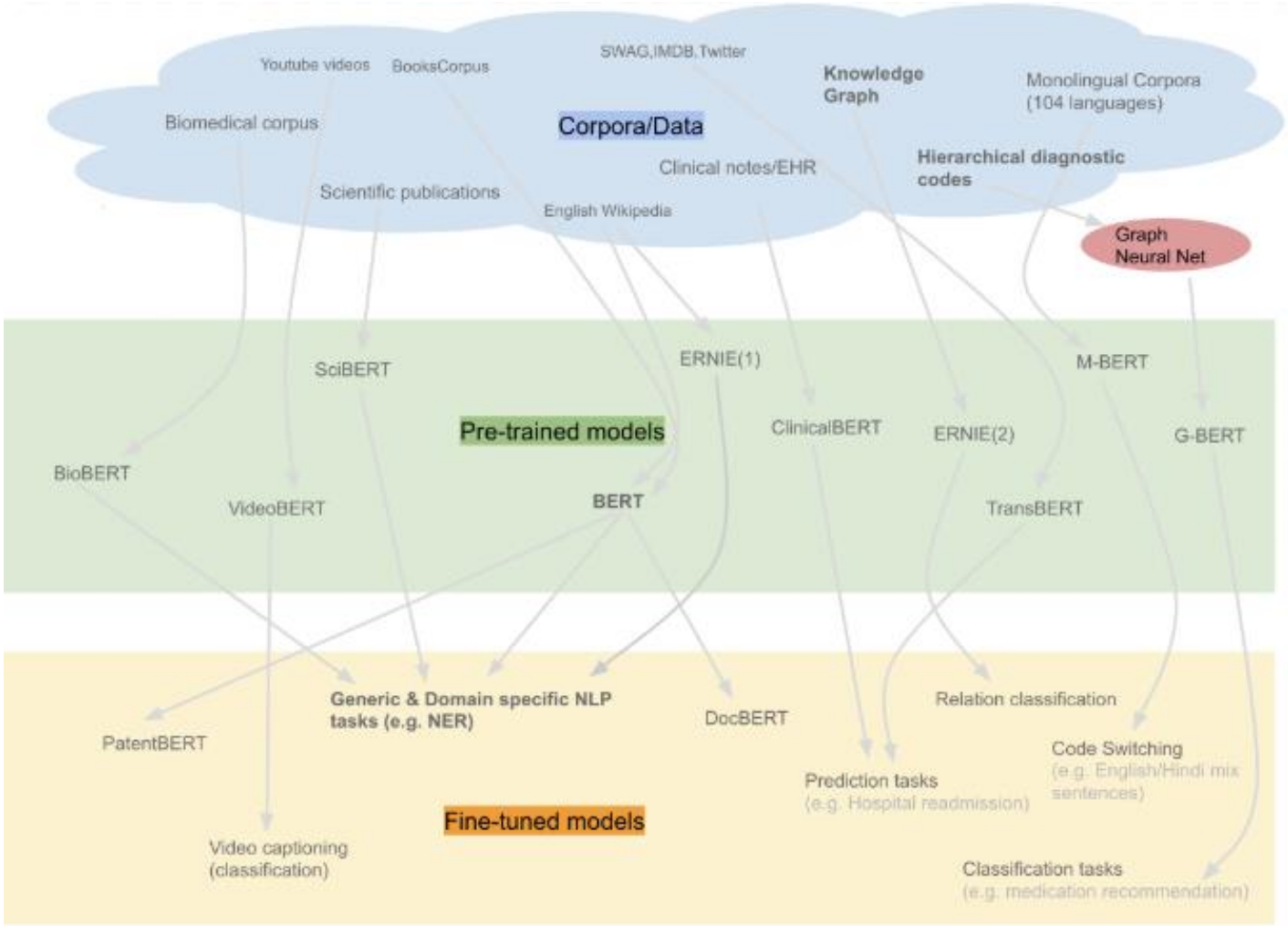


Figura 29 Modelos surgidos a raíz del modelo BERT

3.5.1 Estructura de BERT

En principio, BERT es efectivo para tareas de clasificación, aunque nosotros lo utilizaremos, mediante el mecanismo *fine tunning*, para nuestra tarea de QA (Query Answering).

Una vez entendida la arquitectura *Transformer*, BERT se basa en una pila de codificadores, como se muestra a continuación:

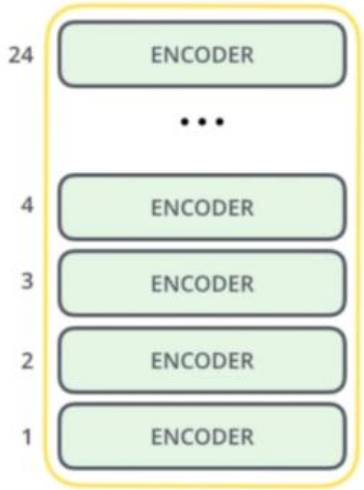


Figura 30 Codificadores en pila

La estructura interna de cada codificador empleado por BERT es la siguiente:

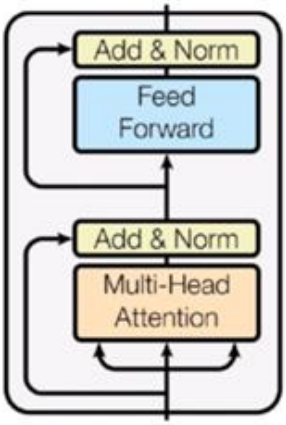


Figura 31 Codificador del modelo Transformer

3.5.2 Tokenización

Los modelos que se utiliza comienzan convirtiendo la secuencia (*string*) de entrada en vectores, tal y como se muestra a continuación:

Figura 32 Esquema de la conversión de secuencias a vectores

Para ello se puede utilizar *WordPiece tokenizer*, un proceso que es caracterizado por:

- Utilizar 30522 palabras. No es un número grande, en comparación con todas las existentes en el idioma inglés, pero es un número que normalmente se encuentra en los textos.
- Manejar el hecho que una palabra esté compuesta por otras.
- Capaz de manejar nuevas palabras que están compuestas por otras ya conocidas. Por tanto, si encuentra una palabra desconocida, la dividirá en otras más simples para obtener un significado completo.

3.5.3 Entradas y salidas

Las entradas y salidas del modelo se disponen tal y como se muestra a continuación:

[CLS] + Sentencia de A + [SEP] + Sentencia de B

Donde [CLS] representa el comienzo de la sentencia a clasificar y [SEP] es un separador para diferenciar cada sentencia (opcional). Mediante la separación de sentencias, el modelo podrá dar diferentes significados a cada sentencia. En nuestro caso, una sentencia significará la pregunta, que queremos formular, y la siguiente secuencia el contexto al que se refiere la pregunta.

El codificador es la parte fundamental de BERT y necesitará:

- Valores vectorizados de las palabras (*token embeddings*)
- Indicación de cuál es la primera y segunda secuencia a utilizar (*segment embedding*)
- La posición de cada palabra (*position embedding*)

La representación final de las entradas de BERT sería la siguiente:

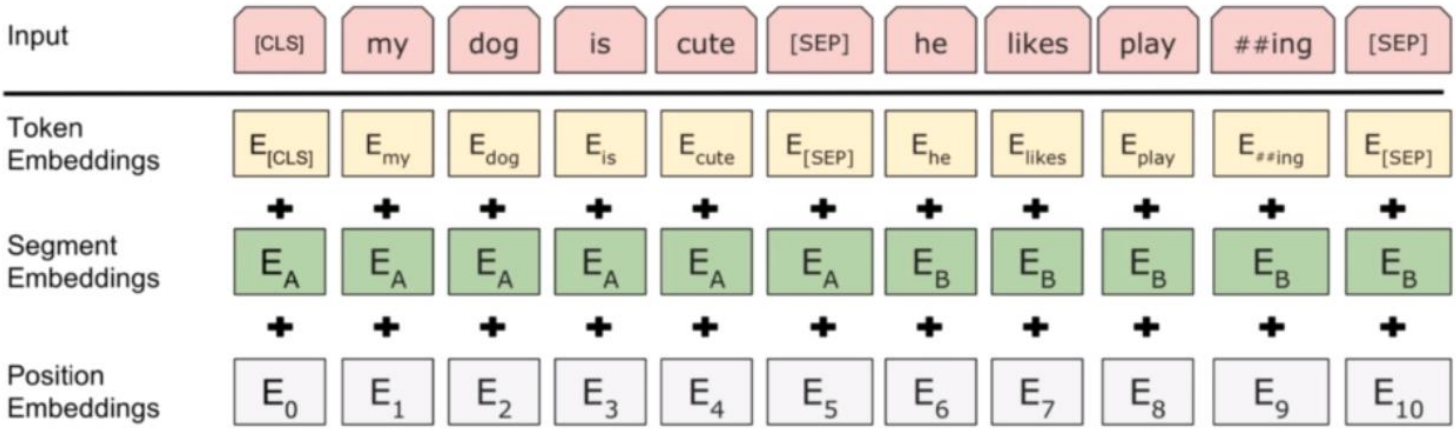


Figura 33 Ejemplo de entradas del modelo BERT

Las salidas son de dos tipos:

-Salidas para los *token*s [CLS], dadas por el vector NSP (*next sentence prediction*), un vector que clasificará el tipo de salida.

-Salidas para los otros *tokens,* dados por el vector Mask LM (*Mask Language Model*)

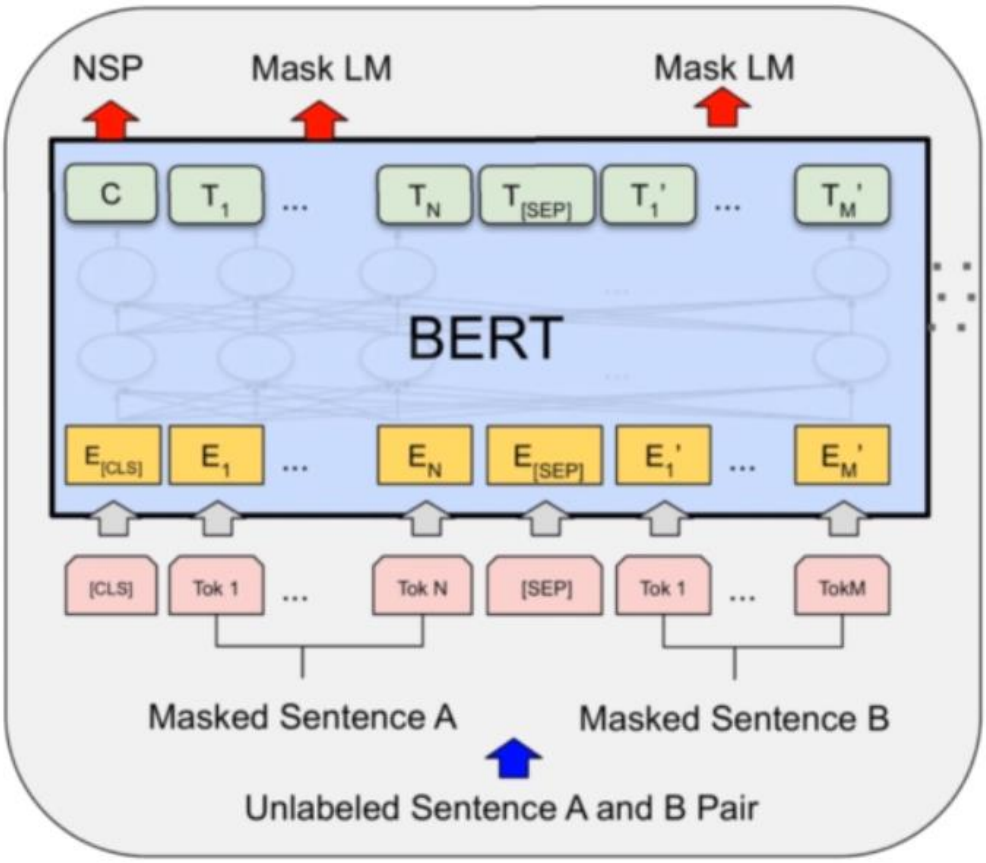


Figura 34 Esquema para las entradas y salidas del modelo BERT

En la figura de arriba se muestra como fluye la información dentro del modelo BERT. Se pueden observar sus entradas separadas con [SEP] y el *token* [CLS]. Luego se ve como pasa a través del modelo y termina dando el resultado.

El modelo concreto de BERT que vamos a utilizar tendrá dos salidas diferenciadas y que se denominan *pooled_output* y *sequence_output.* La primera salida hace referencia al *token* [CLS], y la segunda salida hace referencia al resto de *tokens.*

El siguiente ejemplo muestra cómo es la entrada de BERT y cómo es la salida.

Teniendo la siguiente frase que queremos introducir a BERT.

"Hello, this is an example"

Se empieza introduciendo los *tokens* de clasificación y de separación-

[CLS]Hello, this is an example. [SEP]

Y como sabemos, todos son *tokens* asique en realidad quedaría algo así:

[CLS][Hello][,][this][is][an][example][.][SEP]

Así pues, hay 9 tokens en total. BERT devolverá una salida de dimensión [1,9,768] para la salida *sequence_output* (se supone que el *batch* es de uno y que no utilizaremos la salida *pooled_output*).

Para nuestro modelo, nuestra salida siempre dará 768 dimensiones de embedding por cada *token* esta vendrá representada por la variable sequence_output, y la salida para la clasificación vendrá dada por pooled_output.

3.5.4 Bidireccionalidad

Como se comentó, BERT es un modelo basado en apilar codificadores. La siguiente imagen muestra cómo están apilados estas celdas tipo *Transformer*.

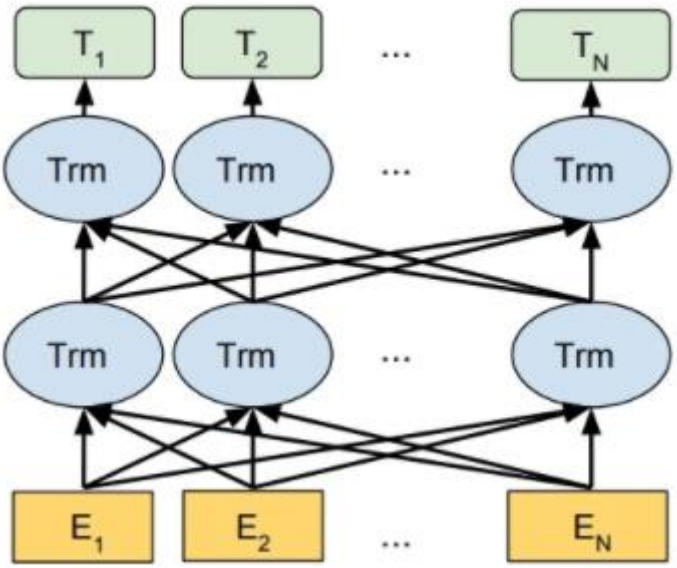


Figura 35 Apilamiento de codificadores en BERT

En la figura anterior vemos cómo BERT está compuesto por celdas, llamadas "Trm" por su arquitectura *Transformer*. Cada celda está conectada con cada una de las celdas de la siguiente capa. En la figura de arriba solo aparecen dos capas, pero el modelo puede tener más. Esta arquitectura particular es la que origina que BERT sea bidireccional.

En contraste, otros modelos no bidireccionales, pero que tiene una gran importancia en el campo del PLN son GPT y ELMo. Sus respectivas arquitecturas se muestran en la siguiente figura.

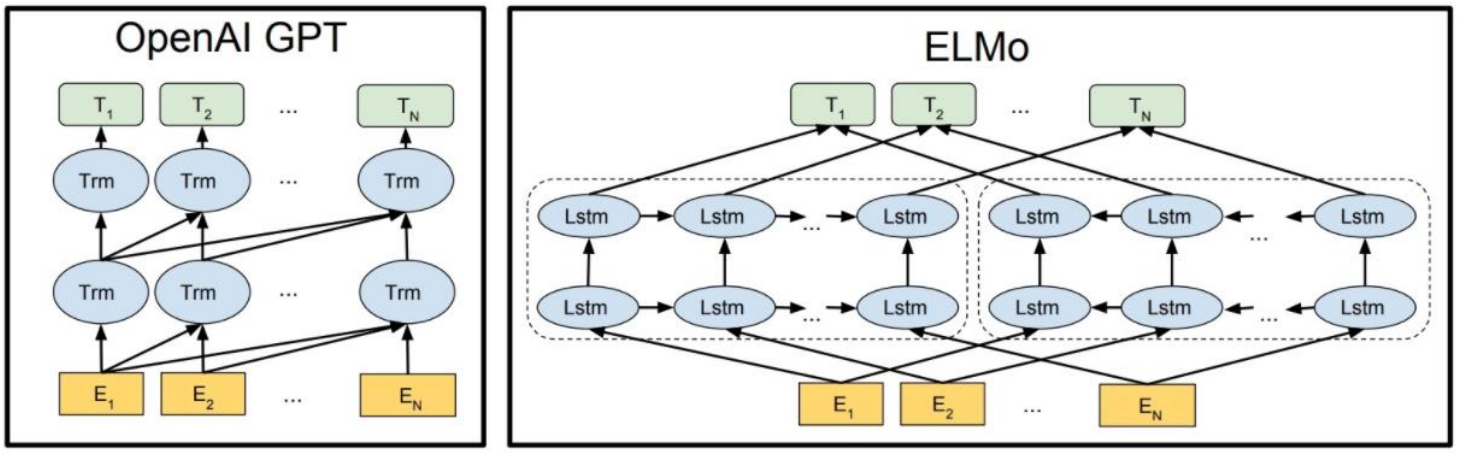


Figura 36 Arquitecturas de GPT y ELMo

Como se aprecia en la figura de arriba, en contraste con el modelo BERT, los modelos GPT Y ELMo están conectados de forma diferente.

El modelo GPT, también compuesto de celdas provenientes del modelo *Transformer,* (en este caso, por decodificadores)*,* están conectados cada celda con las celdas de la capa siguiente, pero solo si éstas son la frontal o las de la derecha. Por tanto, podemos decir que GPT es un modelo de predicción unidireccional

El modelo ELMo, esta vez compuesto por redes neuronales de tipo LSTM, tiene dos conjuntos de celdas relacionadas con diferentes direcciones. Es decir, en el conjunto de la izquierda se relacionan de derecha a izquierda y en el de la derecha de izquierda a derecha. El último paso que realiza (cuando obtiene las soluciones T_i) realiza una operación para juntar ambos resultados, como puede ser una media. En este caso, el modelo se dice que es semi – bidireccional.

3.5.5 Modelo de Lenguaje Enmascarado

Una de las razones por las que el modelo BERT obtiene buenos resultados, en el actual estado del arte para el Procesamiento del Lenguaje Natural, es debido a que es capaz de utilizar la bidireccionalidad correctamente mediante técnicas como el Modelo de Lenguaje Enmascarado

La razón de uso es debido a que los modelos bidireccionales que se utilizan suelen tener varias capas que provocan que a veces la predicción del *token* de salida sea el mismo *token* de entrada.

El Modelo de Lenguaje Enmascarado o *Masked Language Model* es un tipo de modelo donde se enmascara una o más palabras en una secuencia y hacemos que el modelo prediga esas palabras enmascaradas dadas las otras palabras en la secuencia. El objetivo de entrenar el modelo de esta forma esencialmente provoca aprender ciertas (pero no todas) las propiedades estadísticas de las palabras de la secuencia.

El modelo se ejemplifica en la siguiente figura:

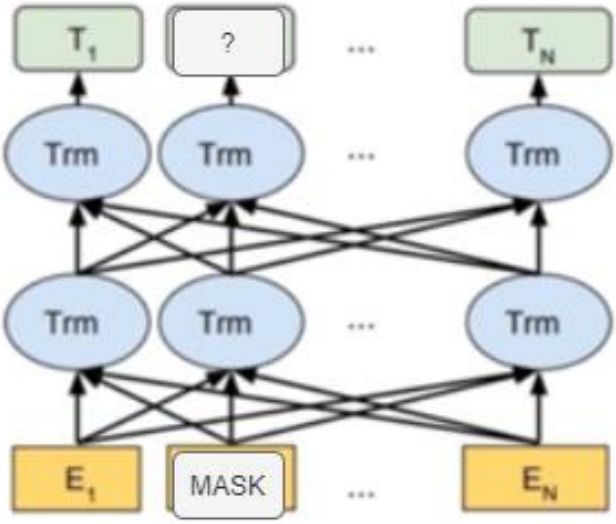


Figura 37 Modelo BERT enfocado en el Modelo de Lenguaje Enmascarado

En la figura de arriba, se muestra el modelo BERT enmascarando el *token* que se quiera predecir, para lo cual se utilizará toda la secuencia.

Este proceso de enmascaramiento no se realiza para todas las palabras de la secuencia, sino que se enmascara, aleatoriamente, solo un 15% de los *tokens*. La razón por la que se produce solo para el 15%, se debe a que se pretende introducir un ligero aprendizaje de forma aleatoria, sin afectar drásticamente al resultado.

A parte de esta técnica, también se utilizan otras técnicas para mejorar el modelo.

3.5.6 Predicción de la siguiente secuencia

Para nuestro problema concreto de "Responder a Preguntas" se va a utilizar una técnica llamada *Next sequence prediction,* que mejora la bidireccionalidad del modelo BERT.

El modelo BERT tiene un mecanismo para clasificar secuencias (mediante el *token* [CLS]), de modo que cuando maneja dos sentencias, encuentra si están relacionadas o cual es la continuación natural de la anterior. A esta técnica se la denomina *Next sequence prediction*.

El ejemplo siguiente ejemplifica del uso de esta técnica:

Figura 38 Ejemplo Next sequence prediction

Como se puede observar, el modelo BERT clasifica pares de sentencias como *IsNextSentence o NotNextSentence*. Para ello, el vector [CLS] tendrá dos valores porcentuales indicando cuanto de relacionadas están cada secuencia entre sí.

Por tanto, lo que se hará será dar a BERT la entrada con la pregunta y el contexto separados por [SEP]. Al final, se aplicará una red neuronal densa que recibirá la salida de BERT, y que consta con dos neuronas en la última capa, la primera nos dará una puntuación sobre si es el comienzo de la pregunta y la segunda si es el final de la pregunta.

3.5.7 Funcionamiento de BERT

Una vez pre-entrenado BERT mediante un *corpus* grande, se van a utilizar la técnica de *Fine tunning* para entrenarlo en una tarea específica.

Como se ha explicado en los apartados anteriores, la salida de BERT es un *token* clasificador que indica que las secuencias de entrada son equivalentes, y junto con una predicción de las secuencias de entrada.

Este resultado se entregará a la siguiente capa de nuestro modelo específico, que estará compuesto por una red densa con dos neuronas finales que nos dirán dos números. Estos dos números son las posiciones que serán utilizadas para buscar un String en el contexto de partida, con la intención de que contenga nuestra respuesta a la pregunta planteada.

Por tanto, el entrenamiento consistirá en obtener ese String del contexto, compararlo con la respuesta objetivo e iterar hasta que tenga la mayor aproximación a la respuesta objetivo.

3.6 Comparación de modelos

Existen varios modelos relevantes en el campo de Procesamiento del Lenguaje Natural. Los modelos se suelen comparar mediante sus hiperparámetros del tipo L (número de capas de codificadores), H (dimensión del *embedding* o tamaño oculto), y A (número de cabezas de atención).

La siguiente tabla muestra la comparación de los diferentes modelos en base a los valores de estos hiperparámetros.

Modelo	L	H	A
BERT Base	12	108M	768
BERT Large	24	334M	1024
BERT X Large	24	1270M	2048
GPT-2	12-48	117M-1542M	768-1600
GPT-3	12-96	125M-175B	768-12288
ALBERT	12-24	12M-235M	768-4096
ELMo	5	94M	1024

Figura 39 Tabla de comparación de modelos de aprendizaje

En el caso de BERT se indican diferentes alternativas, mientras que para el resto se indica el rango de valores, entre los que suele estar cada modelo.

Para el caso de BERT solo se han utilizado los modelos de tipo *uncased o* modelos que no utilizan mayúsculas.

Existen más variables que deberían estudiarse para una comparación efectiva entre los modelos. Ejemplos de este tipo de parámetros son el *learning rate, batch size,* etc. Incluso el resultado de cada modelo varía dependiendo de cómo se ha entrenado y cuál es el problema propuesto.

La siguiente figura muestra una comparación de varios modelos y cómo ha aumentado el número de hiperparámetros en el tiempo.

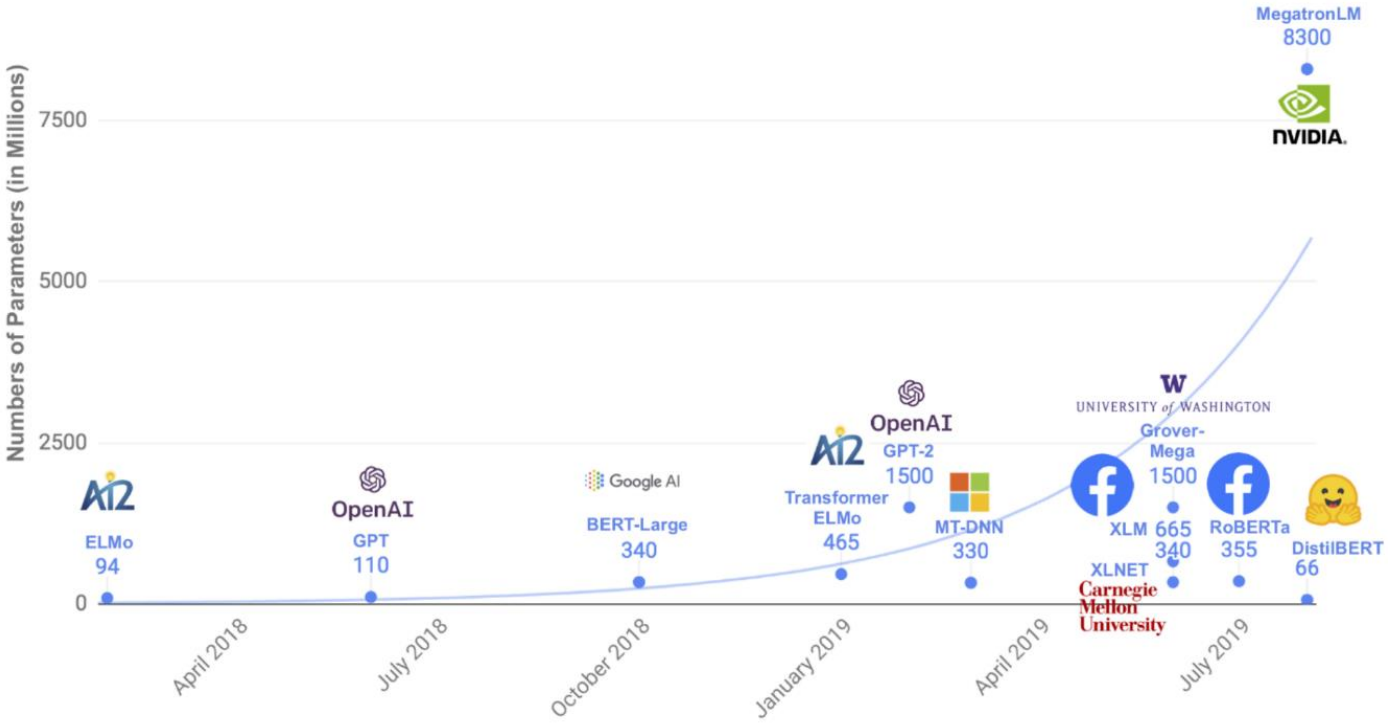


Figura 40 Comparación de modelos

Capítulo 4

Caso de estudio

En este capítulo se va a tratar el desarrollo de nuestro caso de estudio. Éste se ha implementado mediante el lenguaje de programación Python sobre un intérprete especializado en tareas analíticas denominado Jupyter Notebook. Para etapas como el entrenamiento, recomiendo utilizar Google Colab o un entorno de desarrollo que favorezca la computación, como pueden ser mediante TPU (*Tensor Processing Unit*), en lugar de Jupyter Notebook.

En el último apartado, se mostrará los resultados obtenidos a lo largo de este proyecto.

4.1 Datos utilizados

El modelo necesita un *dataset* o datos de entrada y salida para entrenarse. Se va a utilizar una *dataset* llamado SQuAD.

SQuAD (*Stanford Question Answering Dataset*) es un conjunto de datos estructurados creados por *crowdsourcing.*

Los datos están listos para utilizarse para la compresión de texto mediante máquinas. Básicamente, los datos consisten en preguntas planteadas por *crowdworkers* a partir de un conjunto de artículos de Wikipedia, entre otras fuentes.

Las preguntas se hacen para cuando la respuesta es un segmento del texto. Mediante el contexto se empieza a construir el *dataset*.

Los datos que se van a utilizar provienen de la versión de SQuAD 1.1 y se componen de:

- **`train-v1.1`:** es el *dataset* utilizado para hacer el entrenamiento y está representado en formato `.json`, en este archivo es donde están almacenadas las secuencias que se utilizan para entrenar el modelo.

- **`dev-v1.1.json`:** es el *dataset* utilizado para hacer la evaluación, también está en formato `.json`.

- **`vocab.txt`:** describe los *tokens* utilizados para identificar a cada palabra. Su estructura está compuesta por una lista vertical de 30522 palabras.

En el siguiente texto se describe la estructura del archivo. Respecto a la notación del contenido, las partes que están en cursiva y en color azul indican el contenido que debería colocarse en esa parte del *dataset*.

La estructura de los *dataset* de entrenamiento y evaluación es la siguiente;

```
{ "data":[{
            "title": "título"
             "paragraphs":[ {
                   "context":  "Secuencias del contexto",
                    "qas": [
                  {
               "answers": [{
                    "answer_start": "Tipo entero",
                    "text": "Secuencia de la respuesta"
               }
               {…}
// Puede contener varias respuestas a una pregunta
               ]
               "question": "Secuencia de la pregunta"
               "id" : "Identificador de la pregunta"
               }
               {…}
// Puede contener más respuestas-preguntas
              ]
         }
         {…}
// Puede contener más contextos
         ]}
         {…}
// Puede contener más párrafos
 ]}
```

4.2 Modelo utilizado

4.2.1 Resumen de la aplicación práctica

Con el objetivo de entender la aplicabilidad de los modelos, se presenta aquí un resumen esquemático de la aplicación práctica que se desarrollarán en los siguientes apartados.

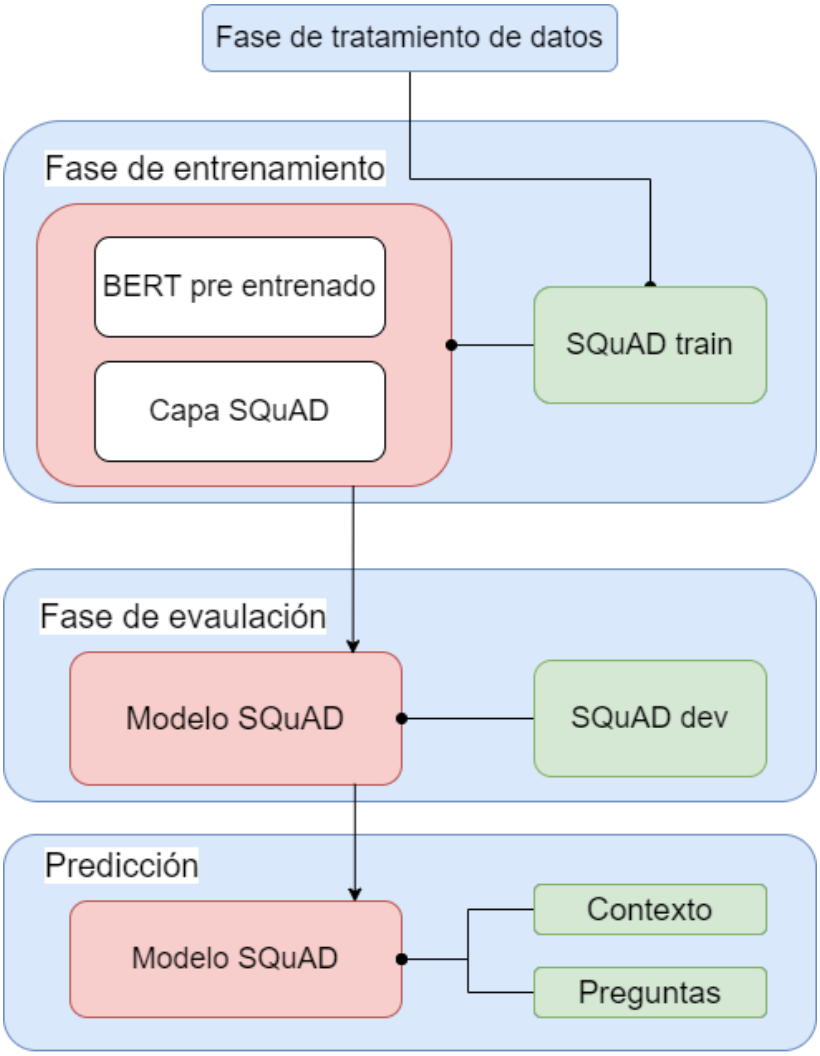


Figura 41 Flujo de trabajo seguido en el desarrollo del caso de estudio

4.2.2 Librerías utilizadas

En este apartado se describen las librerías utilizadas para describir el entorno de desarrollo del caso de estudio planteado.

- **`Tensorflow`**: es una biblioteca de código abierto para aprendizaje automático e implementar un amplio abanico de tareas de aprendizaje. Fué desarrollada por Google para satisfacer sus necesidades.

Esta librería va a ser la principal herramienta que utilizada para entrenar y usar tanto el modelo como los datos. Se va a utilizar la versión dos, en concreto, la 2.4.0.

Las sublibrerías y funciones utilizadas son:

- **`tf.keras.layers.Layer`**: clase que normalmente contiene el resto de capas de la librería `keras`.

  Una capa es un objeto invocable que toma como entrada uno o más tensores y proporcionan salidas de uno o más tensores. Está formada por los siguientes componentes:

  - **Estados** o variables de peso (variables de ponderación): definidas en el constructor `__init__()` o en el `build()`. Normalmente, cuando se crea una capa personalizada, primero se instancia mediante la super clase y luego se invoca al método `call()`.
  - **`__init__():`** define los atributos en una capa personalizada, y crea las variables de estado de la capa que no dependen del tipo de entrada, usando `add_weight()`.
  - **`call():`** método utilizado para realizar el computo.
    Puede incluir varios argumentos y, después de haber llamado a `build()`, realizará la lógica de la aplicación de la capa para los tensores de entrada (que deben ser pasados como argumento). Dos de los argumentos que están reservados y son de uso opcional en `call()` son:

    -`training` (booleano que determina si la llamada está en modo de inferencia o el modo de entrenamiento)

    -`mask` (booleano para utilizar o no un vector enmascarado)

- **`tf.keras.layers.Dense:`** implementación de una red neuronal densamente conectada.

- **`tf.unstack`**: descomprime la dimensión dada de un rango R de un tensor en a un rango R-1, devolviendo un tensor.

- **`tf.keras.initializers.TruncatedNormal:`** inicializador que genera una distribución normal truncada.

- **`tf.transpose:`** transpone A , donde A es un tensor.

- **`tf.keras.Model:`** se utilizará para crear y almacenar nuestro modelo, el cual contendrá conjuntos de capas. El objeto contiene funciones de entrenamiento y de inferencia.

- **`tf.keras.backend.sparse_categorical_crossentropy:`** método utilizado para añadir una capa de *softmax* con *cross-entropy* .

- **`tf.reduce_mean:`** método para calcular la media de los elementos para un tensor dado.

- **`tf.keras.metrics.Mean:`** calcula la media ponderada de los valores dados.

- **`tf.train.Checkpoint:`** permite almacenar objetos que se puedan rastrear, pudiendo los guardar y restaurar. Cada objeto guardado será llamado punto de control.

- **`tf.train.CheckpointManager:`** gestiona múltiples puntos de control para mantener cierta integridad y eliminar los que no sean necesarios.

- **`tf.GradientTape:`** biblioteca para calcular la derivada (direccional) de la función perdida (función error entre la salida de un modelo y la salida esperada). Una vez calculado el gradiente, podremos utilizar una función de optimización para el modelo.

- **`tf.io.gfile.GFile:`** librería utilizada para escribir datos de `Tensorflow` a archivos.

  - **`Tensorflow_hub:`** es una biblioteca que utilizaremos para guardar nuestros modelos, y de esta forma reutilizar las para diferentes tareas.
    - **`hub.KerasLayer:`** nos permitirá añadir un modelo guardado al modelo que se está utilizando, como si fuese una capa más dentro del modelo.

- **`Oficial.nlp`**: biblioteca que nos proporciona un conjunto de herramientas de PLN, usando *Tensorflow* 2. En concreto, se van a utilizar las siguiente sub bibliotecas de esta biblioteca:

  - **`Optimization:`** para utilizar distintos tipos de optimizadores, en concreto se utilizará uno llamado `Adam Weight Decay`.
  - **`Official.nlp.bert.tokenization:`** utilizado para la tokenización de elementos en las sentencias mediante el método `FullTokenizer.`

- **`Official.nlp.bert.input_pipeline:`** utilizado para manejar los datos para del entrenamiento. En concreto, al utilizar SQuAD, se utilizará `create_squad_dataset().`

- **`official.nlp.data.squad_lib:`** utilizado para manejar los datos para el entrenamiento del modelo. Está especializado para utilizar los datos de SQuad. Las funciones que se han importado son:

  -**`Read_squad_examples()`**: utilizado para tratar las entradas iniciales del modelo, las entradas que deben tener la estructura del *dataset* explicado anteriormente. Recibe un primer archivo mediante un directorio para ser tokenizado a través del segundo archivo que permite esta tokenización. Después devuelve otro archivo nuevo al directorio indicado.

  Otras de sus funciones son `FeatureWriter(), Write_predictions(), Convert_examples_to_features()` o `Generate_tf_record_from_json_file()`.

Otras librerías conocidas, que se han utilizado son:

- **`Numpy`:** extensión de Python, que le agrega mayor soporte para vectores y matrices.
- **`Random`**: extensión de Python para crear números aleatorios.
- **`Math`:** extensión de Python para utilizar funciones matemáticas.
- **`Time`:** extensión de Python para utilizar funciones relacionadas con el tiempo.
- **`Json`:** extensión de Python con funciones para administrar archivos tipo `.json`.
- **`Collections`**: implementa módulos de tipos de datos especializada de contenedores como `containers`, `dict`, `list`, `set`, y `tuple`.
- **Os:** proporciona una forma portátil de utilizar la funcionalidad dependiente del sistema operativo.

4.2.3 Construcción del modelo

En este apartado se va a explicar los pasos que se han realizado para obtener el modelo final.

4.2.3.1 Preprocesamiento de datos

Tras importar las librerías explicadas anteriormente, se requiere transformar los datos de entrenamiento de SQuAD a datos utilizables por las librerías de `Tensorflow`. Para ello se utilizará la función `generate_tf_record_from_json_file()`, dejando escrita, su salida, en un archivo llamado `train_meta_data`, divido en cuatro partes. Este archivo ya estará listo para ser utilizado, también se inicializa el tamaño del *batch* para el entrenamiento, en este caso,que será de tamaño 4.

Una vez dado este paso ya se tendrán los datos "tokenizados". Se llama a la función `create_squad_dataset()` que, a partir del archivo anteriormente guardado, creará el *dataset* de entrenamiento. Tras este paso ya se tienen listos los datos para entrenar el modelo.

4.2.3.2 Capa de SQuAD

En este apartado se van a definir las funciones que describen el modelo a entrenar. Esta etapa estará compuesta por una capa llamada "Capa SQuAD" y el modelo pre-entrenado BERT, tal y como se muestra a continuación:

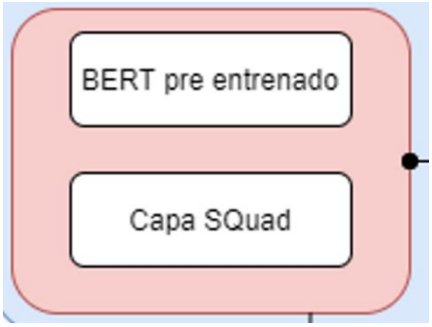


Figura 42 Modelo en la fase de entrenamiento

En primer lugar, se creará una capa compuesta por redes neuronales densamente conectadas, la cual será entrenada para devolver dos valores numéricos.

Estos valores representarán dos posiciones en el contexto, que determinarán de dónde se obtendrán el resultado, una vez obtenido el resultado. El resultado calculado se podrá comparar con el resultado esperado y, de esta forma, en base a la evaluación del error cometido, empezar el aprendizaje de las neuronas de la red densamente conectada.

Se empieza definiendo la clase **`BertSquadLayer`**, clase que hereda `tf.keras.layers.Layer`. Dentro de esta capa, tendremos los siguientes métodos:

- **Método para inicializar** la capa donde se inicializará la red neuronal densa (llamada `final_dense`). Indicándose que la salida son dos números (`units = 2`), el comienzo y el final que para el `kernel initializer`, se selecciona una distribución normal truncada con desviación de 0.02.

- **Método para utilizar la capa** (método `call()`), donde utilizaremos la red neuronal densa, introduciéndole las entradas. Como ya se ha explicado, la salida de la red neuronal son dos valores posición. Antes de retornar estos valores se ha tenido que tratar los vectores de `Tensorflow`, transponiendo la matriz (`tf.transpose`) y separando los valores (`tf.unstack`)para que el método solo devuelva dos variables.

4.2.3.3 Modelo completo

Después de definir la capa neuronal personalizada y, una vez entrenada, se crea un modelo nuevo heredado de la clase `tf.keras.Model`, llamado **`BERTSquad`**. Tendrá los siguientes dos métodos importantes:

- **Método para inicializar** el modelo contiene la inicialización del modelo a través de `hub.KerasLayer()`, con el atributo de entrenamiento activado, y la inicialización de la capa `BertSquadLayer`, anteriormente explicada.

- **Método para utilizar el modelo** (`call()`), el cual permitirá introducir las entradas que pasarán por el modelo pre-entrenado BERT y después su salida entrará en la capa `BertSquadLayer`, encargada de devolver las dos posiciones buscadas del contexto.

4.2.3.4 Función de perdida

El modelo anterior ya estaría completo, y ya se podría empezar a crear salidas, dadas unas entradas. Después, estas salidas deberán utilizarse para, dados unos objetivos, poder empezar a entrenar el modelo.

La función de perdida ayudará a realizar esta tarea, evaluando la diferencia entre la salida esperada(objetivo) y la salida en ese momento de nuestro modelo.

Como ya se ha explicado, la salida y el objetivo son dos posiciones (téngase en cuenta que se está simplificando, ya que en realidad son dos vectores posicionales). Con estos valores se calculará, para cada uno de ellos (en este caso uno para el comienzo y otro para el final), la función de coste a través del método `sparse_categorical_crossentropy()` de `tf.keras.backend`.

Se activará, el atributo `from_logits` para usar solo valores reales, sin aplicar ninguna regla `softmax` a los valores originales. Se hará lo mismo para la perdida de las posiciones finales.

Finalmente, se calcula la pérdida total haciendo la media entre estos dos valores. Es decir:

$$P_{total} = \frac{P_{incio} + P_{final}}{2}$$

Figura 43 Ecuación de la pérdida total

Más adelante, se utilizarán los valores devueltos por esta función de perdida para tareas como calcular los gradientes o ir mostrando, por cada época de entrenamiento, cuanto se acerca al resultado final.

Como se puede intuir, el modelo solo va a calcular cuánto de cerca está de encontrar la solución dada para el *dataset* predefinido.

4.2.3.5 Entrenamiento

Una vez definidos, el modelo y la función de perdida, se puede comenzar el entrenamiento. Una vez instanciado los hiperparámetros y el optimizador utilizado sobre el modelo definido, comenzará a iterar sobre el modelo para acercar al resultado objetivo, y hasta que se terminen las épocas de entrenamiento.

Definimos las siguientes hiperparámetros:

- El tamaño del entrenamiento, en este caso de 88641
- El tamaño máximo del *batch* que se propagarán por la red. En este caso 2000
- Número de `batches`. En este caso 4
- Número de épocas 3
- Tasa de aprendizaje (*learning rate*), en este caso de $5 \cdot 10^{-5}$
- Número de pasos para cambiar la tasa de aprendizaje, en este caso 200.

Describiendo el proceso más en detalle, se comienza añadiendo estas variables al modelo.

Dado el número máximo de un lote de entrenamiento (*batch*) se generarán los correspondientes conjuntos de datos a partir de `train_dataset`.

Se inicializa un optimizador dado por la librería `official.nlp` que requerirá una tasa de aprendizaje, numero de pasos de entrenamiento, y el número de pasos necesarios para cambiar esta tasa.

Una vez que tenemos el optimizador inicializado y la función de perdida definida (`squad_loss_fn`) se puede añadir al modelo para que los utilice cuando necesite entrenarse.

Así mismo, se inicializa una variable llamada `train_loss` mediante `tf.keras.metrics.Mean`, y que servirá para mostrar la media de la perdida.

Antes de empezar a entrenar, se carga el último punto de guardado de las variables de entrenamiento mediante el método `CheckpointManager()`.

El algoritmo de entrenamiento consta de dos bucles, uno dentro del otro, para iterar sobre las épocas, y otro para cada época. Cada época itera sobre los ejemplos del lote de el *batch* de entrenamiento, la cual estará compuesta por la entrada . Para la iteración de cada época se seguirán los siguientes pasos:

1. Se inicializará la variable `tape` mediante tf.GradientTape().

2. Se utilizará el modelo `bert_squad()` con las entradas de la iteración actual, devolviendo una salida para ese *batch*. Con esta salida se calcula la perdida mediante `squad_loss_fn` y el objetivo actual.

3. Se calcularán los gradientes para cada variable entrenable del modelo mediante `trainable_variables` (variable interna del modelo de keras).

4. Se indicará que realice la optimización de forma lineal, como se ha explicado, y se hará una optimización preliminar para mejorar la optimización posterior, a modo de calentamiento (`warmups`).

Además, se va calculando, para cada *batch,* la media de la perdida mediante `train_loss()` y se muestra por pantalla, para ver cómo evoluciona. También se va guardando el estado, por cada iteración completada, sobre el *batch*.

Por cada iteración de una época, solo se muestra el tiempo que ha consumido el entrenamiento.

4.2.3.6 Evaluación

Una vez que ya se tiene nuestro modelo entrenado, se pasa la fase de evaluación. En esta fase se obtendrá el modelo con los datos de evaluación ya metidos. Para ello, el modelo pre-entrenado BERT deberá tener los mismos *tokens* que el *dataset* que le se introduce.

Se empieza tratando los datos tal y como se hizo para el entrenamiento, y guardándose estos datos. Esta vez utilizaremos la variable `eval_writer` para actualizar este archivo y de esta forma las modificaciones, que se haga a esta variable, se actualizarán en el archivo, para poder utilizarse después.

También se va a utilizar un "tokenizador" o *tokenizer* que convierte secuencias a *tokens*. Este "tokenizador" se va a extraer a partir del modelo pre-entrenado de BERT. Para ello, se volverá a instanciar su capa con una variable, y a partir de esta, se obtendrán los *tokens* de su vocabulario, y el valor booleano para los casos de mayúsculas o minúsculas.

Se creará una función `_append_feature` que nos permitirá controlar como saldrá la salida de la "tokenización" que posteriormente se va a realizar. Se basará en una función que almacena las salidas en el archivo tipo `.tf_record`, y si la salida ha sido comprimida, es decir, tiene el valor `is_padding` como *true,* entonces almacenará este dato en un array independiente de su código para poder utilizarlo después en la predicción.

Después de este paso, se utilizará un método específico para el *dataset* de SQuAD que recibirá el tokenizador anterior, el *dataset* de evaluación, y variables sobre sus dimensiones, como son:

- Máxima longitud de secuencia, en este caso, de 384
- Número de divisiones del contexto o `doc_stride`, en este caso, 128
- Longitud máxima de la consulta, en este caso, 64
- Tamaño del *batch,* en este caso, 4
- Se establece que no es entrenable y añadimos la función `_append_feature`

Una vez actualizado el archivo `.tf_record`, se crean nuestros *dataset* finales a través de este archivo mediante `create_squad_dataset()`, ya mostrado.

Una vez establecido el *dataset*, se puede comenzar con las operaciones de predicción.

Para almacenar las predicciones, se crea un tipo `collection` llamado `RawResult` con los valores para el identificador de la tupla de los valores de posición de inicio y fin.

Para empezar a realizar las predicciones, se va a necesitar en un momento transformar las salidas finales, las cuales utiliza tipos provenientes de `Tensorflow`. Para ello, se convertirá a un tipo de dato lista mediante la librería `numpy` para poder utilizarlo sin `Tensorflow`, esta función se llamará `get_raw_results`.

Se itera sobre el *dataset* de evaluación, y por cada iteración solo se cogen de la entrada los identificadores únicos y se introducen en el modelo, que devolverá la salida con dos posiciones, con esas dos posiciones y el identificador. Con estas salidas se crea un diccionario el cual se pasa a la función `get_raw_results()` que permitirá añadir los tres valores importantes de la salida.

Se escriben las predicciones en archivos `.json`, mediante la librería pre-implementada para SQuAD llamada `write_predictions`, donde introducimos: los resultados obtenidos anteriormente, los ejemplos junto con sus casos comprimidos y los directorios donde se quieren guardar.

4.2.3.7 Tratamiento de la predicción

Se va a utilizar el "tokenizador" implementado por `FullTokenizer`. Este "tokenizador" se va a extraer del modelo pre-entrenado de BERT. Para ello, se volverá a instanciar su capa a una variable y de esta obtenemos los *tokens* de su vocabulario, así como el valor booleano para mayúsculas o minúsculas.

Se van a crear unas funciones para tratar la salida obtenida:

- `is_whitespace()`: para devolver un booleano en función de si son espacios saltos de línea …
- `whitespace_split()`: toma un texto y lo divide por espacios creando una lista de listas.
- `tokenize_context()`: toma una lista de palabras (devuelta por `whitespace_split()`) y "tokeniza" cada palabra. También realiza un seguimiento, para cada *token* nuevo, de su palabra original, dado parámetro `text_words`.
- `get_ids()`: para obtener el identificador dado un *token* mediante el "tokenizador".
- `get_mask()`: obtiene el *token* máscara
- `get_segments()`: obtiene los identificadores de un segmento de *tokens*.
- `create_input_dict()`: toma una pregunta y un contexto de tipo `String` y devuelva un diccionario con los 3 elementos necesarios para el modelo. También devuelve los identificadores del contexto y la longitud de cada pregunta.

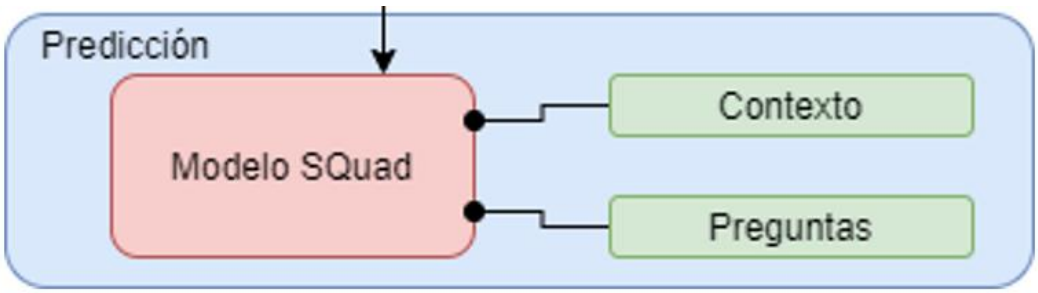


Figura 44 Fase de predicción

4.2.3.8 Predicciones

En este momento, se tiene el modelo listo para utilizarse, y se puede empezar a utilizarlo, lo cual se necesita un contexto y una pregunta a este contexto. Este procedimiento se muestra a continuación:

Figura 45 Entradas y salidas de la predicción

En primer lugar, se utiliza `create_input_dict()` que permite obtener la entrada como diccionario, los *tokens*, los identificadores del contexto, y las longitudes de las preguntas.

Se calcula la predicción utilizando el modelo y la entrada con el atributo de entrenamiento desactivado.

El resultado estará almacenado en las variables `start_logits` y `end_logits`. Esta salida se procesa para convertirla en información fuera del contexto de `Tensorflow` usando `numpy`.

Se realizan un par de iteraciones para comprobar que el comienzo de la respuesta es superior al fin de la respuesta.

Después se utilizan los valores de salida para mostrar por pantalla la respuesta en texto.

4.2.3.9 Carga del modelo

En este apartado se describirán las tareas que se deberían hacer si solo se quiere ejecutar el modelo, sin pasar por la parte de entrenamiento y evaluación.

Para implementar esta operación se necesitará ejecutar:

- La primera parte del *notebook* o la fase Importar dependencias para cargar las librerías.
- La tercera parte o fase de construcción del modelo para cargar las funciones que se utilizará sobre el modelo.
- La sexta parte o fase de predicciones propias, pero solo será necesario ejecutar la parte de utilidades, para tratar los datos de salida.

Para terminar, se ejecutará el último bloque llamado "Cargar el modelo", se carga el modelo y dentro de la última celda del *notebook*, se especificará el contexto y la pregunta, para finalmente ejecutarla y obtener la solución.

4.3 Tecnologías utilizadas

En este apartado se va a mostrar qué tecnologías se han utilizado para la realización del caso práctico y qué otras alternativas hay para su ejecución.

El lenguaje utilizado es, como ya se ha explicado, es Python y su librería más destacable es Tensorflow.

Para el desarrollo de la aplicación se ha utilizado Jupyter Notebook. Este es un entorno interactivo basado en la web para crear documentos tipo Jupyter Notebook (como `.ipynb`). La razón de su uso se debe a su amplio uso en la comunidad y a que muestra, de forma amigable, el código.

A parte de Jupyter Notebook, existen muchas otras alternativas para ejecutar el código. Se recomienda utilizar GoogleColab ya que permite utilizar sus TPU para la computación. Otra alternativa es añadir el uso de algún entorno de despliegue de aplicaciones dentro de un contenedor, como es Docker, y de esta forma, controlar mejor el proceso de creación.

4.4 Resultados

En este apartado se muestran los resultados obtenidos para una serie de contextos y preguntas, asociadas a cada contexto.

Escenario 1: Contexto relacionado con las rede neuronales:

> Briefly, a neural network is defined as a computing system that consist of a number of simple but highly interconnected elements or nodes, called 'neurons', which are organized in layers which process information using dynamic state responses to external inputs. This algorithm is extremely useful, as we will explain later, in finding patterns that are too complex for being manually extracted and taught to recognize to the machine. In the context of this structure, patterns are introduced to the neural network by the input layer that has one neuron for each component present in the input data and is communicated to one or more hidden layers present in the network; called 'hidden' only due to the fact that they do not constitute the input or output layer. It is in the hidden layers where all the processing actually happens through a system of connections characterized by weights and biases (commonly referred as W and b: the input is received, the neuron calculate a weighted sum adding also the bias and according to the result and a pre-set activation function it decides whether it should be 'fired' or activated. Afterwards, the neuron transmits the information downstream to other connected neurons in a process called 'forward pass'. At the end of this process, the last hidden layer is linked to the output layer which has one neuron for each possible desired output.

Se le preguntan las siguientes preguntas y nos responde las siguientes respuestas para cada caso.

Caso 1

Pregunta: Where does the computation happens in a neural network?

Respuesta: "a number of simple but highly interconnected elements or nodes, called 'neurons',"

Caso 2

Pregunta: "How is called the process of transmitting information inside a net?"

Respuesta: "dynamic state responses to external inputs."

Caso 3

Pregunta: "Why are some layers called hidden?"

Respuesta: "input or output"

Algunas respuestas se comprenden, excepto la última que podría ser más confusa debido a que le falta más información. Si se remarca en el texto, se ve que es lo que realmente nos quiere decir, como se muestra a continuación.

> Briefly, a neural network is defined as a computing system that consist of a number of simple but highly interconnected elements or nodes, called 'neurons', which are organized in layers which process information using dynamic state responses to external inputs. This algorithm is extremely useful, as we will explain later, in finding patterns that are too complex for being manually extracted and taught to recognize to the machine. In the context of this structure, patterns are introduced to the neural network by the input layer that has one neuron for each component present in the input data and is communicated to one or more hidden layers present in the network; called 'hidden' only due to the fact that they do not constitute the input or output layer. It is in the hidden layers where all the processing actually happens through a system of connections characterized by weights and biases (commonly referred as W and b: the input is received, the neuron calculate a weighted sum adding also the bias and according to the result and a pre-set activation function it decides whether it should be 'fired' or activated. Afterwards, the neuron transmits the information downstream to other connected neurons in a process called 'forward pass'. At the end of this process, the last hidden layer is linked to the output layer which has one neuron for each possible desired output.

Escenario 2: En el siguiente contexto trata de cómo funciona un circuito.

> An electrical circuit is composed of a source of electrical power, two wires that can carry electric current, and a light bulb. One end of both the wires is attached to the terminal of a cell while their free ends are connected to the light bulb. The electrical circuit is broken when the bulb is switched off. To light up the bulb, the electrical circuit needs to be completed that is, a connection be made between the light bulb and the wires so that the latter can transmit electric current to the bulb. For this the electrical circuit needs to be closed.
>
> The wires in an electrical circuit are made of a material called conductor that helps these transmit electricity. These wires have low resistance to electric current. Copper and aluminum are usually used as the wire material in fluorescent bulbs. However, in an incandescent bulb, the electric current passes through a thin strip of tungsten wire called a filament. The filament becomes heated and produces light.

Caso 1

Pregunta: When can we broke and electrical circuit?

Respuesta: "when the bulb is switched off."

Caso 2

Pregunta: "How is a wire made?

Respuesta: "conductor "

Caso 3

Pregunta: "What is the difference within incandescent and fluorescent bulbs?"

Respuesta: "Copper and aluminum are usually used as the wire material in fluorescent bulbs. However, in an incandescent bulb, the electric current passes through a thin strip of tungsten wire called a filament."

En el caso 2 tenemos el mismo problema que en el anterior contexto. Si se observa el texto, vemos que es correcta.

> An electrical circuit … closed.
>
> The wires in an electrical circuit are made of a material called conductor that helps these transmit electricity. These wires have low resistance to electric current. Copper and aluminum are usually used as the wire material in fluorescent bulbs. However, in an incandescent bulb, the electric current passes through a thin strip of tungsten wire called a filament. The filament becomes heated and produces light.

Estos son los resultados tras el uso de la aplicación práctica.

La siguiente figura muestra la evolución de la función de pérdida (error) durante el proceso de entrenamiento.

Figura 46 Perdida durante el entrenamiento

Como vemos, empieza aumentando ligeramente el error, luego baja hasta 0.6, y aumenta drásticamente hasta 1.7, se supone que es debido a la presencia de *outlayers*, y luego termina bajando otra vez hasta 0.6

Finalmente, la duración del entrenamiento ha sido de 15612.34 segundos o aproximadamente 4 horas y 30 minutos, que junto a la ejecución del resto del código ha tardado 5 horas de 27 minutos.

La razón de estos números tan elevados es debido a que no se han utilizado todas las características de la GPU (*Graphics Processing Unit*) del ordenador en el que se ejecutó. Sin embargo, usando Google Colab y utilizando la TPU (*Tensor Processing Unit*) que nos proporcionan, el proceso total se queda en 1 hora y 13 minutos, por lo que tenemos que es cuatro veces más rápido utilizando TPU.

Capítulo 5

Conclusiones y trabajo futuro

En este capítulo se presentan las conclusiones que se han alcanzado tras el desarrollo de este proyecto. Se plantearán posibles mejoras al modelo planteado y mejores alternativas a este modelo.

Durante el transcurso de este proyecto, he afrontado dificultades como son la compresión de redes recurrentes aplicadas al Lenguajes Natural. Más concretamente a la aplicación real de las redes LSTM, o la compresión de la arquitectura *Transformer.*

5.1 Conclusiones

Tras haber investigado sobre el campo del Procesamiento del Lenguaje Natural, la conclusión a la que puedo llegar es que el potencial y expectativas que tiene es demasiado grande como para no convertirse en un referente de progreso para este siglo.

La conclusión del proyecto, sobre la tecnología utilizada, es que actualmente ya se tiene la tecnología necesaria para crear productos útiles para la sociedad.

Actualmente, la mejor utilidad que se podría hacer con este tipo de modelos es la creación de modelos que permitan la autogeneración de código. El programa recibiría una entrada con los requisitos de un proyecto de software, este devolvería el software como solución. Aunque, el caso de estudio no produzca este resultado, ya existen modelos, como GPT-3, que permita una solución similar.

No obstante, sustentar una solución empresarial en manos de una máquina es demasiado arriesgado, por lo que siempre se requerirá un ingeniero de *software* que supervise este tipo de máquinas.

5.2 Trabajo futuro

El modelo utilizado tiene muchas carencias, aunque es rápido cuando devuelve una respuesta. Algunas de estas carencias son que no da respuestas validas y totalmente comprensibles, no responde a preguntas genéricas, y no acepta tamaños grandes de entradas.

Para solventar estos problemas se deberán hacer los siguientes ajustes:

- Cambiar de la versión base del modelo BERT a la versión X Large, para que tenga mayor número de parámetros.
- Cambiar la versión de los datos de entrenamiento a la versión 2 de SQuAD, la cual contiene un mayor rango de preguntas y respuesta.
- Utilizar otras técnicas para mejorar el comportamiento interno, o para construir secuencias bien construidas a partir de la respuesta y la pregunta.

También, como se ha expuesto, existen modelos mucho más potentes que BERT como es el caso de GPT-3 para utilizarse.

Anexo

Hay dos apartados para las redes neuronales y el *embedding*.

1 Redes Neuronales

Las redes neuronales son modelos computacionales compuestos por neuronas conectadas entre sí, para transmitir la información de entrada, que posteriormente será sometida a una serie de operaciones que modifican el valor del resultado.

1.1 Funcionamiento de una neurona

Cada conexión tiene un valor y un peso, la neurona internamente tiene una función interna, y un *bias*, como se muestra en la siguiente imagen.

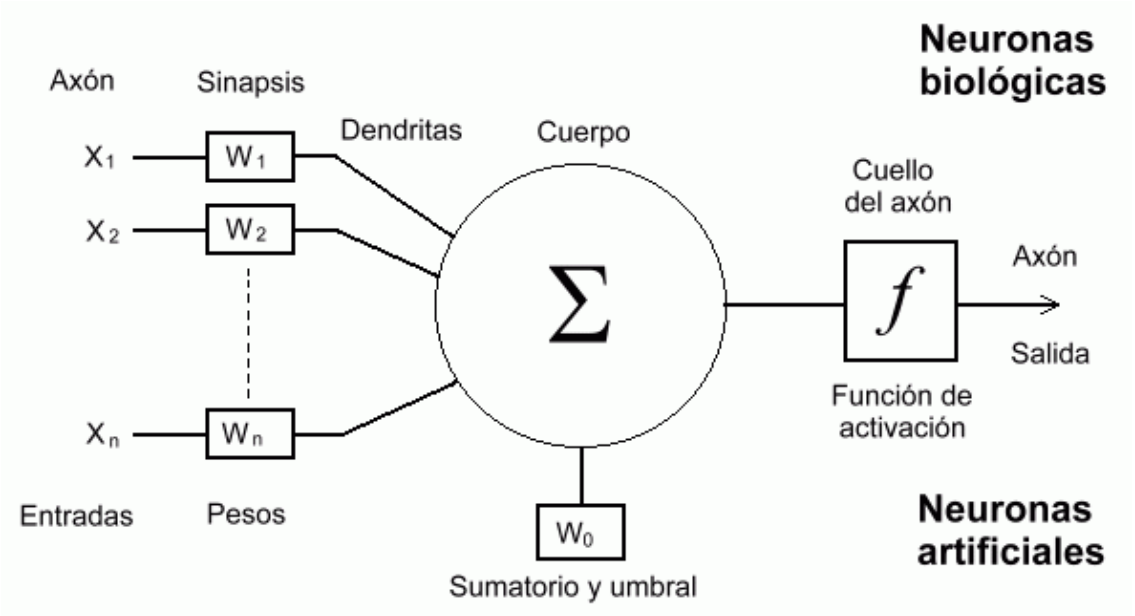


Figura 47 Esquema de una neurona

Normalmente la función interna es un sumatorio que utiliza las entradas y el *bias* para calcular su salida que es pasada por una función de activación.

La función de activación limitará la amplitud de la salida de nuestra neurona en función de su valor da salida.

1.2 Arquitectura de las redes neuronales

Las redes neuronales son agrupaciones de capas de neuronas que se envían sus resultados entre sí, habiendo capas de entrada y de salida. En la siguiente figura se muestra un ejemplo de una red neuronal densamente conectada:

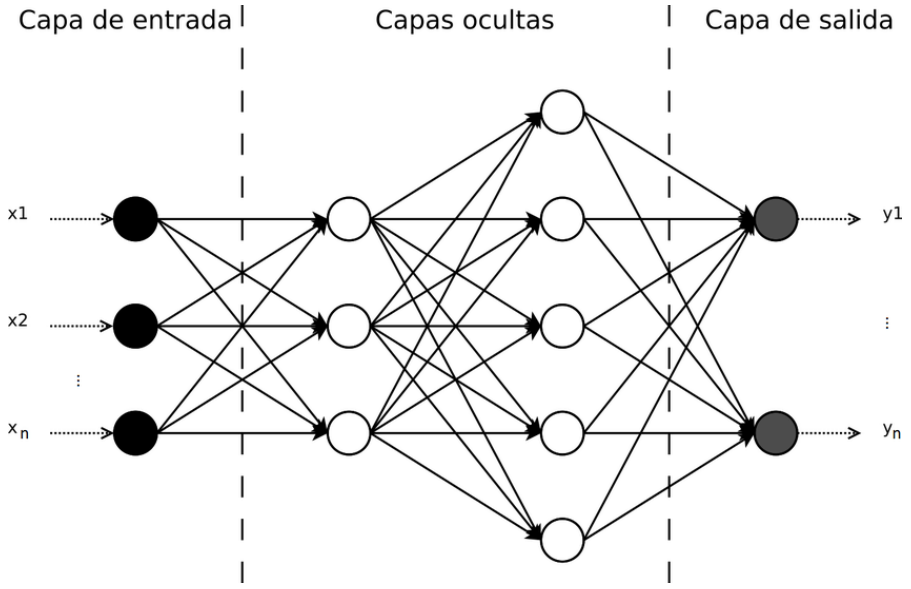


Figura 48 Esquema de una red neuronal

Red neuronal densamente conectada: Tipo de red neuronal en la que todas las neuronas de la capa anterior están conectadas con la capa actual, así como las de la capa actual con las de la siguiente.

El objetivo de una neurona es de que, a partir de un conjunto de datos, obtener una información más elaborada y compleja. De esta forma, cuando se conectan varias neuronas se pueden obtener una mayor complejidad de los datos. La tarea del ingeniero será encontrar los parámetros de cada neurona, adecuados para que la salida sea equivalente a la solución del problema.

Tras hacer la primera operación en la red obtenemos una salida, la salida no suele equivaler al resultado esperado por lo que se requiere modificar los pesos de la red para acercarse al resultado esperado.

Uno de los objetivos de una red es automatizar el cambio de la red, un ejemplo de utilidad es el método de *backpropagation*.

Backpropagation o propagación hacia atrás, cuyo objetivo es modificar las neuronas necesarias con el fin que la red pueda dar siempre los resultados esperados.

Este proceso empieza comparando su salida obtenida con la salida esperada con esto obtenemos una función de coste.

Después se usan **funciones de optimización**, tomando como ejemplo la función de descenso por gradiente (*gradient descent)* el error se convierte en un vector gradiente con el que obtendremos la dirección y la cantidad que queremos modificar del resultado.

Para ello, el vector gradiente multiplicado por una **tasa de aprendizaje** nos acercará o alejará a la solución, hasta obtener un resultado que no suponga una variación del grande del coste.

El objetivo es cambiar cada neurona para acercarnos a la solución, en general se resuelve la composición de funciones entre la función de la neurona (el sumatorio), la función de activación y la función del error optimizada.

Finalmente, se resuelve el sistema de ecuaciones para cada neurona y se pasa hacia la capa anterior hasta llegar al inicio de la red.

Cada vez que se realiza este proceso se le denomina **iteración.**

1.3 Entrenamiento de la red neuronal

Cuando se entrena una red neuronal se tiene un conjunto de datos o *dataset* que pasará por la red, normalmente este *dataset,* es demasiado grande para que la red neuronal pueda manejarlo de una vez, por lo que se divide en lostes (*batches*) para que haga una iteración para sobre un *batch* y pase al siguiente.

Uno de los objetivos del aprendizaje automático es crear modelos generales, si la red neuronal fuese tan grande como el *dataset*, se estaría creando un modelo que memorizaría ese *dataset* concreto, y se comportaría mal como modelo general (posiblemente fallaría más ante ejemplos nunca vistos).

Cuando la red ha hecho iteraciones sobre el entrenamiento, todos los *batches* del *dataset*, es cuando el modelo ha concluido una época (*epoch*). A pesar de que el modelo haya recorrido todo el *dataset*, muchos modelos requieren varias épocas hasta obtener resultados útiles.

1.4 Red neuronal recurrente

Las Redes Neuronales Recurrentes (RNN) son redes neuronales, que incluyen en su topología de red, bucles de realimentación, permitiendo, gracias a ellos, que la información persista durante varias épocas de entrenamiento.

Aunque existen diversas modificaciones de este tipo de neuronas, en general esta neurona añade a su entrada salidas anteriores que haya tenido, de esta forma utiliza las iteraciones anteriores para su entrenamiento.

Son útiles para computar las interacciones entre distintas partes de la secuencia. Aun así, la utilización de redes de tipo LSTM (RNN especializadas en entender sucesos pasados), suelen presentar el problema de no poder entender la secuencia de derecha a izquierda o viceversa, por lo que se suelen sustituir, en el campo del PLN, por la arquitectura *transformer*.

Capa de normalización: la capa de normalización pretende corregir el desajuste en los datos, mediante diferencias entre escalas de vectores. Es decir, una variable no puede tomar valores significativamente más grandes que el resto de variables, comparativamente. Está compuesta por dos capas densamente conectadas y una capa *softmax* entre ellas.

2 Embedding

Se usarán *embdeddings* porque las palabras, como secuencias de símbolos, no pueden ser procesados por un ordenador, y se requiere tener asociada una representación numérica de cada palabra.

Lo que haremos será obtener el índice de la matriz de *embeddings* (lista de vectores de cada palabra en nuestro vocabulario), correspondiente a una palabra dada. A partir del índice se obtendrá el vector del *embedding*, que se multiplicará por la matriz de contexto, obteniendo así, otro vector de salida tipo *softmax,* de dimension igual al índice de *embeddings.* Se activarán así, aquellas palabras que puedan considerarse como equivalentes en significado, dada la matriz de contexto.

Normalmente, si el significado es equivalente, significa que las matrices de contexto son parecidas (el contexto es parecido) y que los valores del *embedding* están cerca.

La figura siguiente representa lo explicado anteriormente:

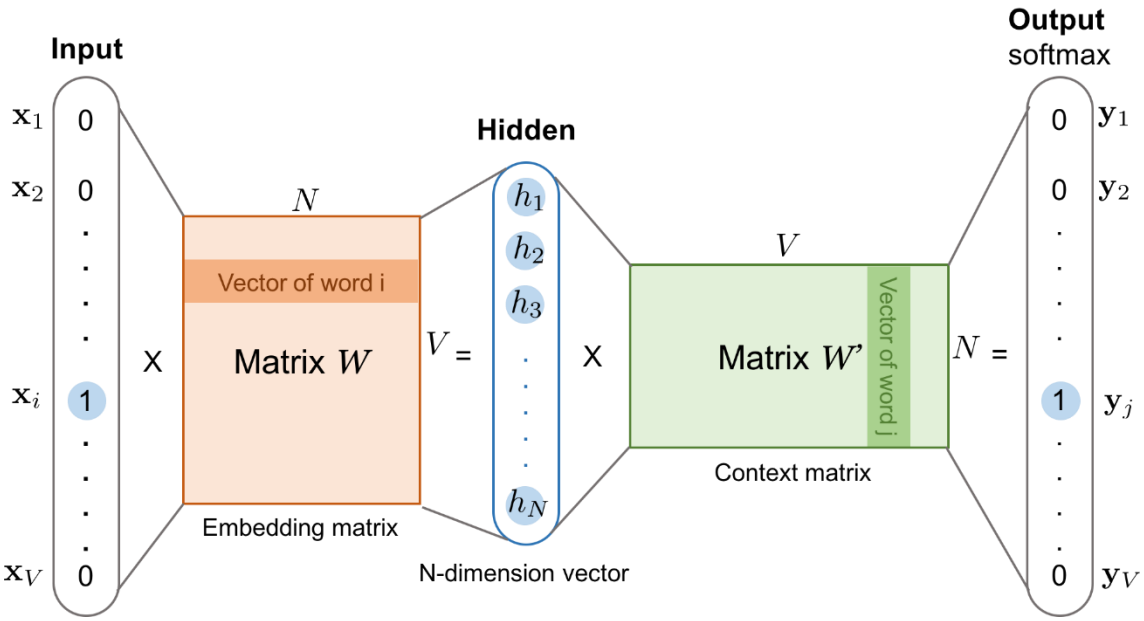


Figura 49 Matriz de embeddings

Este proceso es reversible, por lo que se puede obtener lo que significa el *embedding* después de utilizarlo.

Para tener una idea de lo que se pretende con el *embedding*, utilizaremos la siguiente figura:

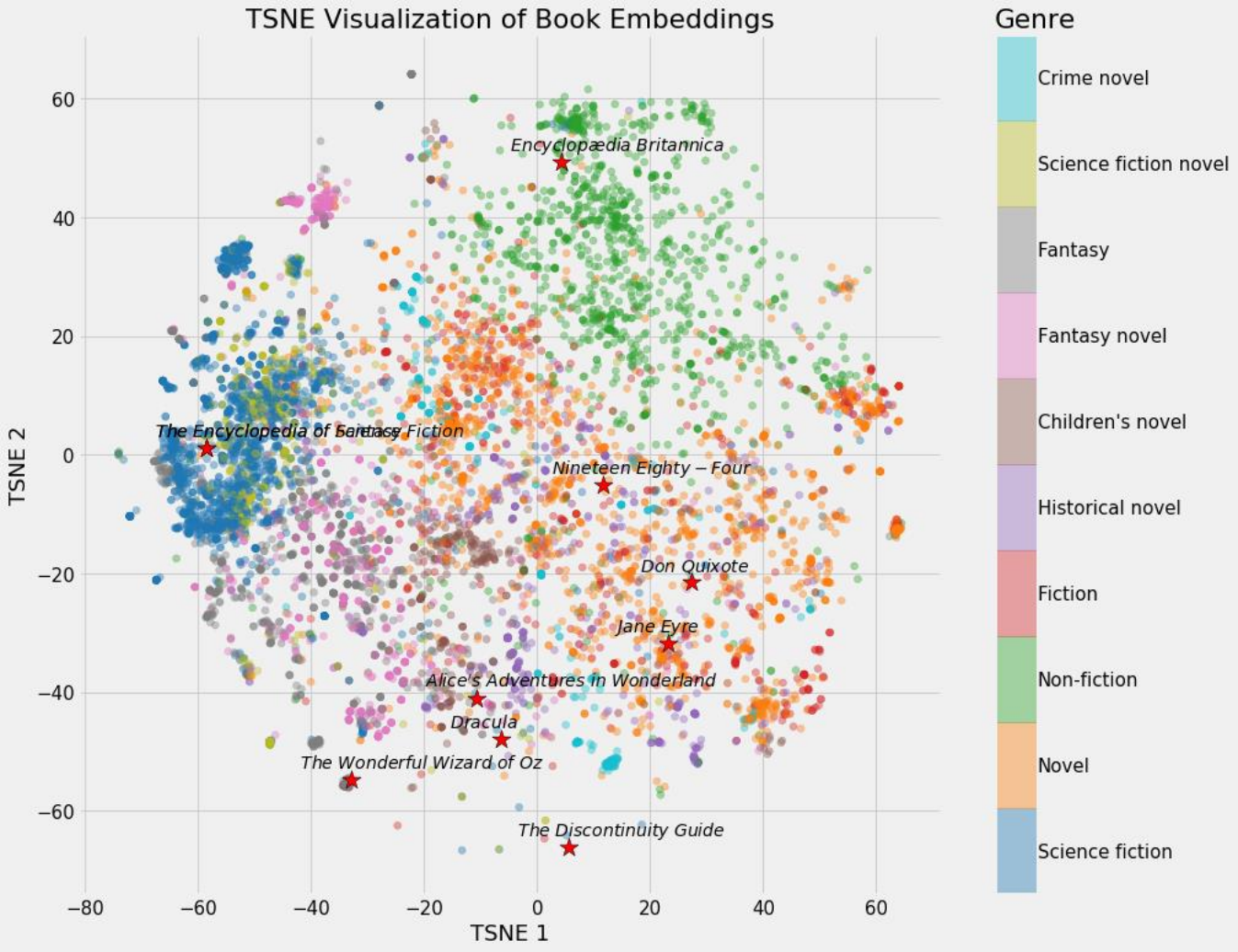


Figura 50 Mapa embedding

Como se aparece en la figura, el *embedding* pretende representar las palabras a través de diferentes dimensiones. En este caso, se representan tres, pero en la realidad se utilizan muchas más dimensiones.

Referencias

Mckinsey. *Jobs lost, jobs gained: What the future of work will mean for jobs, skills, and wages.*

Disponible en https://www.mckinsey.com/featured-insights/future-of-work/jobs-lost-jobs-gained-what-the-future-of-work-will-mean-for-jobs-skills-and-wages

Consultado el 10 de agosto de 2020

Sujay S Kumar. *Reducing Model Size.*

https://sujayskumar.com/2020/04/24/reducing-model-size/

Consultado el 10 de agosto de 2020

NVIDIA Corporation. *BERT Question Answering Inference with Mixed Precision.*

https://aihub.cloud.google.com/p/products%2Ffb38fa2f-f246-43c8-b611-c82948fc6d85

Consultado el 10 de agosto de 2020

Badreesh Shetty. *Natural Language Processing (NLP) for Machine Learning.*

https://towardsdatascience.com/natural-language-processing-nlp-for-machine-learning-d44498845d5b

Consultado el 10 de agosto de 2020

Rewon Child y Scott Gray. *Generative Modeling with Sparse Transformer.*

https://openai.com/blog/sparse-transformer/

Consultado el 10 de agosto de 2020

Dario Amodei y Danny Hernandez. *AI and Compute*

https://openai.com/blog/ai-and-compute/

Consultado el 10 de agosto de 2020

Ria Kulshresthe. *Understanding Attention In Deep Learning.*

https://towardsdatascience.com/attaining-attention-in-deep-learning-a712f93bdb1e

Consultado el 10 de agosto de 2020

Tensorflow. *bert_en_uncased_L-12_H-768_A-12.*

https://tfhub.dev/tensorflow/bert_en_uncased_L-12_H-768_A-12/1

Consultado el 10 de agosto de 2020

Kelly Bowden. *Google BERT: What Is It and How to Optimize for It.*

https://www.webfx.com/blog/internet/google-bert/

Consultado el 10 de agosto de 2020

Prateek Joshi. *Transfer Learning for NLP: Fine-Tuning BERT for Text Classification.*

https://www.analyticsvidhya.com/blog/2020/07/transfer-learning-for-nlp-fine-tuning-bert-for-text-classification/

Consultado el 10 de agosto de 2020

Lista de acrónimos

ALBERT: A Lite BERT

BERT: Bidirectional Encoder Representations from Transformer

CLS: Classification (clasificación)

ELMo: Embeddings from Language Models

GPT: Generative Pretrained Transformer

LSTM: Long Short-Term Memory

NLP: Natural Language Processing (PLN, Procesamiento del Lenguaje Natural)

NLG: Natural Language Generation

NLU: Natural Language Understanding

QA: Question Answering (Respuesta a preguntas)

RNN: Recurrent Neural Network

SEP: Separator (separador)

Seq2Seq: Secuence to Secuence (modelo secuencia a secuencia)

SQuAD: Stanford Question Answering Dataset

TPU: Tensor Processing Unit