Implementación y evaluación de rendimiento de una aplicación para procesamiento de lenguaje natural

Por
Jorge Millán García

UNIVERSIDAD
COMPLUTENSE
MADRID

Grado en Ingeniería del Software
Facultad de Informática

Dirigido por
Francisco Igual Peña y Luis Piñuel Moreno

Implementation and performance evaluation of an application for Natural Languaje Proceessing

Madrid, 2020–2021

Resumen

El Procesamiento del Lenguaje Natural (NLP) es el campo de conocimiento de la Inteligencia Artificial encargado de investigar la manera de comunicar las máquinas con las personas mediante el uso de lenguas naturales, como el español, el inglés o el chino.

Una de las utilidades más conocidas del NLP pueden ser los asistentes virtuales o los chatbots, pero no son las únicas. Además, es importante entender que el NLP no dota de inteligencia a un chatbot, solo le da la capacidad de procesar y generar lenguaje humano. Sin embargo, existen otras muchas áreas de interés en las que se aplica de forma intensiva NLP, entre las que destacan, la comprensión de lenguaje natural, generación de lenguaje natural, recuperación de información, reconocimiento y síntesis del habla, traducción automática, resumen automático de textos o detección de sentimientos y emociones.

Muchas de estas aplicaciones, además, requieren un tiempo de respuesta reducido que, habitualmente, no permite el procesamiento remoto de los datos en grandes centros de procesamiento de datos (en la nube), sino en un dispositivo cercano a la fuente de la información (por ejemplo, un dispositivo móvil o un dispositivo en el borde de la red –Edge Computing–).

El trabajo propone la implementación de algoritmos para procesamiento de lenguaje natural tomando como base implementaciones ya existentes (por ejemplo, BERT), y sus versiones reducidas para dispositivos móviles (mobileBERT), así como la evaluación de su rendimiento en términos de tiempo de respuesta en diferentes entornos. El objetivo final es realizar un estudio sobre el rendimiento de diferentes modelos con diferentes tamaños de entrada en diferentes entornos de trabajo, para determinar la eficiencia de cada uno y compararla con el resto para obtener la mayor eficiencia posible.

Palabras clave: NLP (Procesamiento del Lenguaje), BERT, mobileBERT, AA (Aprendizaje Automatico), TesnorFlow Lite

Abstract

Natural Languaje Proccesing (NLP) is the field of knowledge of Artificial Intelligence that deals with investigating the way to communicate with people through the use of natural languajes, such as Spanish, English or Chinese.

Virtual assisteants or chatbots are one of the best known NLP utilities, but they are not the only one. In addition, it is important to understand that the NLP does not endow a chatbot with intelligence, it only gives it the ability to process and generate human languaje. However, there are many other areas of interest in where NLP is applied intensively, including natural languaje comprehension, natural languaje generation, information retrieval, speech recognition and synthesis, automatic translation, automaticsummarization of texts or detection of feelings and emotions.

Many of these applications also require a reduced response time that usually does not allow remote data processing in large data processing centers (in the cloud), but in a device close to the source of the information, (for example, a mobile device or a device on the edge of the network –Edge Computing–).

The project proposes the implementation of algorithms for natural languaje processing based on existing implementations (for example BERT) and their reduced versions for mobile devices (mobileBERT), as well as the evaluation of their perfomance in terms of response time over a set of mobile devices (for example Coral's Edge TPU). The final objective is to ocarry out a study on the perfomance of different models with different input sizes in different working enviroments, to determine the efficiency of each one and compare it with the other ones to obtain the highest possible efficiency

Keywords: NLP (Natural Languaje Processing), BERT, mobileBERT, ML (Machine Learning), TesnorFlow Lite

Agradecimientos

Me gustaría agradecer a toda mi familia, que ha estado apoyándome en todo momento y me han dado siempre ánimos para continuar, por muchas dificultades que encontrase en el camino. A todos mis amigos y compañeros de la universidad, que me han aconsejado y ayudado con ciertas partes de programación.
Por último, me gustaría hacer una mención especial a mi tutor Francisco Igual Peña, que me ha ayudado absolutamente con todo y siempre ha estado disponible para resolver todas mis dudas y los problemas que íbamos encontrando a lo largo del desarrollo de este TFG. Sin duda un tutor excelente y de los mejores profesores que he tenido.

Sobre TEFLONX

Teflon X(cc0 1.0(documentación) MIT(código))es una plantilla de LaTeX creada por David Pacios Izquierdo con fecha de Enero de 2018. Con atribuciones de uso CC0.

Esta plantilla fue desarrollada para facilitar la creación de documentación profesional para Trabajos de Fin de Grado, Trabajos de Fin de Máster o Doctorados. La versión usada es la X

V:X Overleaf V2 with XeLaTeX, margin 1in, bib

Índice general

Índice de figuras

Capítulo 1

Introducción

En esta memoria se van a reflejar los resultados del estudio realizado con modelos de MobileBERT sobre diferentes tamaños de texto de entrada.

Nos basaremos en el algoritmo de Google BERT, concretamente en la versión de móvil MobileBERT para el estudio de los modelos. Así mismo, se usará la biblioteca TensorFlow Lite, de Python para la creación, entrenamiento y evaluación de los modelos.

1.1. Objetivos

Los objetivos principales del trabajo son los siguientes:

- Generar distintos modelos MobileBERT (cuantizados y no cuantizados), entrenarlos y exportarlos a formato `tflite` para que puedan ser usados en nuestra aplicación. El objetivo de estos pasos es evaluar el rendimiento de dichos modelos para diferentes tamaños de entrada y analizar los resultados.

- Desarrollar una aplicación basada en una demo de MobileBERT, que nos permita introducir como datos de entrada: textos de diferentes tamaños, preguntas y el modelo que va a ser usado. Se espera que genere una salida con una respuesta lógica y concreta a la pregunta realizada, y el tiempo que ha tardado en generar la respuesta.

- Dotar a a aplicación de las capacidades que permitan la entrada de datos de dos maneras: Escribiéndolos a mano, tanto el texto como las preguntas, o por fichero. Así mismo, permitirá escoger si se quiere mostrar la salida por pantalla o si se quiere guardar en un fichero.

  El cometido principal de la aplicación será la automatización de la generación de respuestas y tiempos de ejecución de un mismo modelo en distintos tamaños de entrada, para poder evaluar su rendimiento.

1.2. Plan de trabajo

El trabajo realizado se divide en dos caminos paralelos: el desarrollo y mejora de la aplicación y la creación de distintos modelos que son probados constantemente en la aplicación para comprobar su funcionamiento.

Una vez se tiene la aplicación totalmente desarrollada y los modelos específicos preparados, se pasa a evaluar y analizar los resultados obtenidos en las distintas pruebas realizadas para posteriormente, obtener la conclusión final sobre el estudio de estos modelos en diferentes máquinas.

1.3. Estructura del documento

El documento se estructura en varios capítulos de la forma que sigue:

- **Capítulo 1: Introducción.** Breve resumen de los objetivos y motivación del proyecto así como la estructura del mismo
- **Capítulo 2: Deep Learning y NLP.** Explicación de diferentes conceptos y tecnologías de relevancia en nuestro proyecto.
- **Capítulo 3: Modelos en detalle.** Resumen del proceso seguido en la creación, entrenamiento y transformación de los modelos, con una breve descripción de su estructura y funcionamiento.
- **Capítulo 4: Diseño de la aplicación.** Explicación en profundidad del desarrollo de la aplicación y su funcionalidad.
- **Capítulo 5: Resultados** En este capítulo se exponen los resultados de las pruebas realizadas en la aplicación.
- **Capítulo 6: Conclusiones** Por último, se explican las conclusiones a las que se han llegado a partir de los resultados del capítulo anterior.

1.4. Entornos y lenguajes utilizados

Tanto para el desarrollo de la aplicación como para la creación de los modelos se ha usado Python como lenguaje de programación.

Python ofrece muchas posibilidades gracias a ser un lenguaje multiparadigma y multiplataforma, cuenta con multitud de bibliotecas muy interesantes y útiles.
Las mas importantes usadas en este proyecto son algunas como:

- `flask`, `mobilebert`, `pyquery` y `time` para el desarrollo de la aplicación.
- `numpy` y `Tensorflow` para la creación de los modelos.

Capítulo 2

Introduction

This document will reflect the results of the study carried out with MobileBERT models on different sizes of input text.

We will be based on the Google BERT algorithm, specifically on the mobile version (MobileBERT) for the study of the models. Likewise, the TensorFlow Lite library from Python will be used for the creation, training and evaluation of the models.

2.1. Objectives

- Generate different MobileBERT models (quantized and unquantized), train and export them to the `tflite` format so that they can be used in our application. The goal of these steps is to evaluate the performance of these models for different input sizes and analyze the results.

- Develop an application based on a MobileBERT demo, which allows us to enter as input data: texts of different sizes, questions and the model to be used. It is expected to generate an output with a logical and concrete answer to the question asked, and the time it took to generate the answer.
- The application will allow the entry of data in 2 ways: Writing them by hand, both the text and the questions, or by file. Likewise, it will allow you to choose if you want to show the output on the screen or if you want to save it in a file.

  The main objective of the application is to automate the generation of responses and execution times of the same model in different input sizes in order to evaluate its performance.

2.2. Workplan

The work carried out is divided into 2 parallel paths: The development and improvement of the application, and the creation of different models that are constantly tested in the application to verify their perfomance.

Once the application is fully developed and the specific models prepared, the results obtained in the different tests carried out are evaluated and analyzed to obtain the final conclusion on the study of these models in different environments.

2.3. Document structure

The document is structured in several chapters as follows:

- **Chapter 1: Introduction.** Summary of the objectives and motivation of the project as well as its structure.
- **Chapter 2: Deep Learning and NLP.** Explanation of different concepts and relevant technologies in our project.
- **Chapter 3: Models in details.** Summary of the process followed in the creation, training and transformation of the models, with a brief description of their structure and operation.
- **Chapter 4: App design.** In-depth explanation of the development of the application and its functionality.
- **Chapter 5: Results** This chapter presents the results of the tests carried out in the application.
- **Chapter 6: Conclusions** Finally, the conclusions reached from the results of the previous chapter are explained.

2.4. Environments and languages used

Both the development of the application and the creation of the models, Python has been used as a programming language.

Python offers many possibilities thanks to being a multi-paradigm and multi-platform language, it has a multitude of very interesting and useful libraries.

The most important ones used in this project are some like:

- `flask`, `mobilebert`, `pyquery` and `time` for the app development.
- `Numpy` and `Tensorflow` to create the models.

Capítulo 3

Deep Learning

Antes de ahondar en el cómo funciona el Deep Learning, es conveniente definir qué es y en qué se diferencia del Machine Learning. Nos hemos ayudado de los siguientes blogs: [3] y [4] y de los siguientes libros: [1] y [2].

El Deep Learning surgió como una rama del Machine Learning en el año 2011, gracias a que cada vez los ordenadores contaban con mayor capacidad de computación. Surgió como una manera de intentar mejorar el aprendizaje automático, que usaba algoritmos de regresión y árboles de decisión.

El Deep Learning introdujo un nuevo concepto llamado redes neuronales, intentando simular las conexiones neuronales que hay en nuestro cerebro para mejorar de una manera exponencial el aprendizaje de las máquinas. En la siguiente imagen 3.1 vemos una comparativa a lo largo del tiempo.

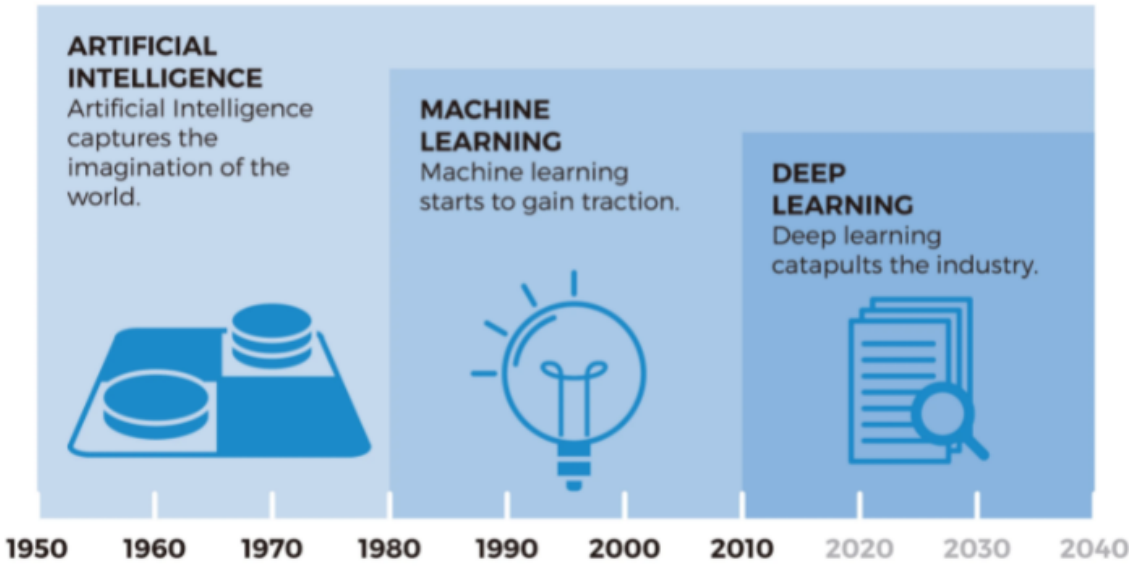

Figura 3.1: Comparativa entre AI, ML y DL a lo largo del tiempo [4]

Ahora bien, ¿cuáles son las principales diferencias entre Machine Learning y Deep Learning? Tanto el Machine Larning como el Deep Learning imitan la forma de aprender del cerebro humano, pero se diferencian en el método ya que, como hemos mencionado

antes, mientras que el Machine Learning utiliza **algoritmos de regresión y árboles de decisión**, el Deep Learning utiliza arquitectura de **redes neuronales** o *neural networks*.

Dichas redes neuronales son mucho mas complejas que los árboles de decisión y si bien ambos métodos permiten un aprendizaje de forma supervisada o no supervisada, los modelos computacionales de Deep Learning ofrecen unos resultados muchísimo mas precisos. En la Figura 3.2 se muestra una comparativa entre ML y DL

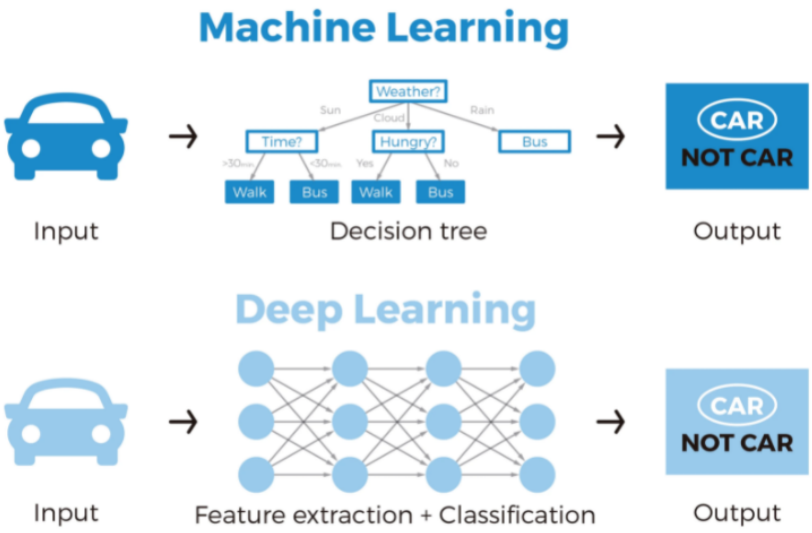

Figura 3.2: Machine Learning vs Deep Learning [4]

3.1. NLP

En esta época de avances constantes y gran revolución tecnológica, el procesamiento del lenguaje natural (NLP, del inglés *natural languaje processing*), tiene una presencia cada vez mayor en nuestra vida. Lo vemos frecuentemente en los chatbots, en las herramientas de traduccion de idiomas, en la clasificación de texto, etc...

No obstante, el NLP es considerado uno de los grandes retos de la inteligencia artificial por ser una de las tareas más complicadas y desafiantes a las que nos podemos enfrentar: ¿cómo comprender realmente el significado de un texto?¿cómo intuir sentimientos en un conjunto de palabras escritas?

Nuestro lenguaje está lleno de ambigüedades, de palabras con distintas acepciones y diversos significados según el contexto. Esto es lo que hace al NLP una de las tareas mas difíciles de dominar.

Se espera de un sistema de procesamiento del lenguaje natural 2 cosas principales:

- *Eficacia*: Que sea capaz de general una salida correcta, entendible y precisa.
- *Eficiencia*: Que el tiempo de ejecución sea el menor posible.

Si queremos que un computador entienda lo que decimos o escribimos tendrá que ser capaz de ejecutar varias tareas propias de NLP como:

- **Inferencia:** Dadas dos frases, tiene que ser capaz de predecir si la segunda frase guarda algún tipo de relacción con la primera.
- **Equivalencia semántica:** Determinar si dos preguntas semánticas son equivalentes
- **Pregunta-respuesta:** Dado un pasaje y una pregunta, tiene que ser capaz de encontrar en el texto la respuesta correspondiente a la pregunta dada.
- **Análisis de sentimientos:** Dado un texto, identificar si el sentimiento expresado es positivo, negativo o neutral.
- **Aceptabilidad linguistica:** Dada una frase, identificar si esta es gramaticalmente correcta.
- **Respuesta correcta:** Dada una frase, determinar cómo continúa en base a un número en concreto de opciones.

El texto es una de las formas más generalizada de secuencias de datos. Se puede ver como una secuencia de caracteres, o una secuencia de palabras. Los modelos de deep learning no toman texto puro como entrada, sino que **vectorizan el texto para transformarlo en tensores numéricos**. Esto se puede hacer de diferentes maneras:

- Segmentando el texto en palabras y transformar cada palabra en un vector
- Segmentando el texto en caracteres y transformar cada palabra en un vector

Los procesos de vectorización consisten en aplicar algún tipo de esquema de tokenización al texto de entrada y asociar vectores numéricos con los tokens generados. Existen varias maneras de asociar un vector a un token, dos de las más importantes son: **one-hot encoding** y **token embeddings**, también llamado word embedding(o incrustación de palabras en español), ya que este último se suele usar exclusivamente para palabras. En la figura 3.3 se muestra un ejemplo de tokenización.

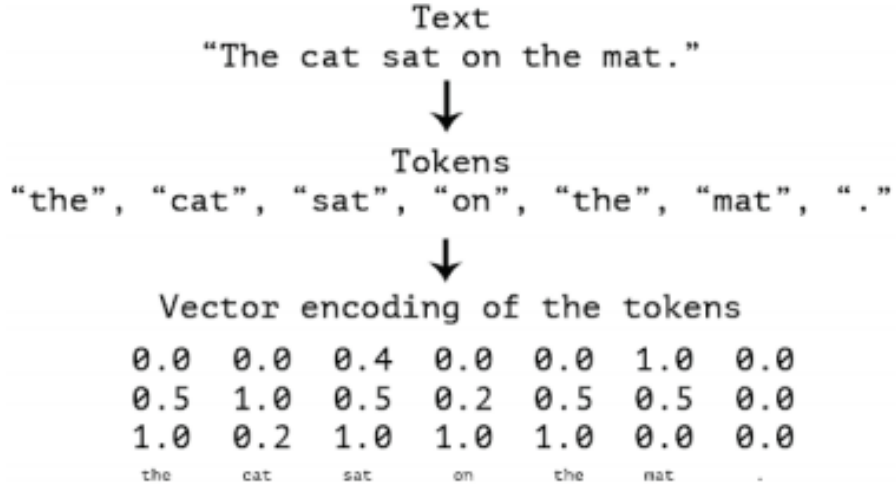

Figura 3.3: De texto a tokens y de tokens a vectores [1]

Nos centraremos en el método **token/word embedding** ya que, mientras que los vectores obtenidos con el método one-hot encoding son binarios y de dimensión muy grande,

los obtenidos por word embedding son vectores de punto flotante y de menor dimensión.

Hay dos formas principales de vectorizar tokens aplicando word embedding:

- Cargando en el modelo *words embbedings* que han sido precalculadas usando tareas de machine learning diferentes a la que queremos resolver. También llamadas words embeddings preentrenadas.
- Usando al principio, vectores de palabras al azar y calcular los vectores de palabras al mismo tiempo que se calculan los pesos de la red neuronal.

3.2. TensorFlow

Tensor Flow es es una plataforma de código abierto de machine learning. Diseñada y mantenida por Google, es una de las mayores librerías open source de machine learning del mundo e implementa muchos de los algoritmos que se necesitan en machine learning.

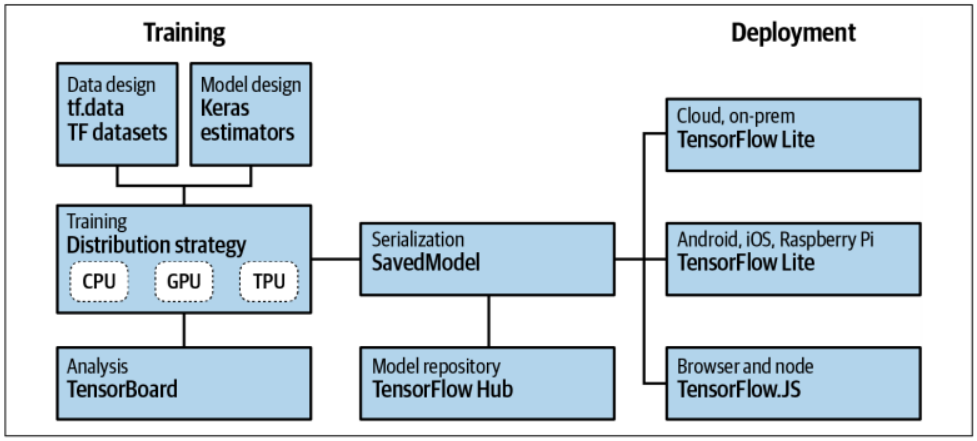

Figura 3.4: Arquitectura de alto nivel de TensorFlow [2]

Está enfocado para personas amateur hasta desarrolladores profesionales incluso investigadores que buscan explotar al máximo el potencial de la inteligencia artificial. En la Figura 3.4 se muestra la arquitectura de alto nivel que forma TensorFlow, desde el entrenamiento hasta su despliegue.

El proceso de creación de modelos de machine learning se conoce como *entrenamiento*. Durante el entrenamiento el ordenador utiliza un conjunto del algoritmos para entender y distinguir los datos y diferenciarlos entre sí. Por ejemplo, si queremos que nuestro programa reconozca imágenes de perros y gatos, se pueden usar un conjunto muy grande de imágenes de perros y gatos para crear el modelo. El programa usará el modelo para intentar diferenciar qué hace que un gato sea un gato y un perro sea un perro. Por último, una vez que el modelo ha sido entrenado, el proceso de reconocer futuras imágenes se llama **inferencia**.

Hay varias maneras de entrenar un modelo, usando una CPU(central processing unit), una GPU(graphics processing unit) o incluso una TPU(tensor processing unit), normalmente se usa un solo chip, sin embargo se puede entrenar paralelamente con múltiples chips ya que TensorFlow posibilita esta opción.

El núcleo de cualquier modelo son sus datos. Si volvemos al ejemplo anterior de perros y gatos, necesitamos muchos ejemplos de imágenes para entrenar el modelo, y manejar tal cantidad de datos puede requerir mucho más código incluso que la creación de los propios modelos. Sin embargo, TensorFlow nos ofrece distintas APIs para intentar facilitar el proceso llamadas TensorFlow Data Services las cuales incluyen muchos datasets ya preprocesados y heramientas que nos ayudarán en el manejo de los datos.

3.2.1. TensorFlow Lite

TensorFlow Lite es una versión de TensorFlow preparada para ser usada en dispositivos móviles. Cuando se crean y ejecutan modelos en la nube o en los ordenadores no tenemos que preocuparnos del consumo de batería o del tamaño de la pantalla, sin embargo, en los móviles son problemas que no podemos pasar por alto.

Por lo tanto se esperan ciertas características de un framework adaptado a dispositivos móviles:

- **Que sea ligero.** Ya que los dispositivos móviles, tienen recursos mucho más limitados que un ordenador. Por ello, los desarrolladores tienen que ser cuidadosos con los recursos que usa, no solo la aplicación, sino el framework en general. Si el framework que ejecuta el modelo es muy grande y el modelo en si mismo también lo es, puede llegar a dar problemas de rendimiento.
- **Que tenga baja latencia.** Se espera, por parte de un usuario de móvil, que las aplicaciones que use sean veloces y funcionen correctamente, sin tiempos de carga innecesarios. Por lo tanto el framework para aplicaciones basadas en Machine Learning tiene que ser rápido en cargar y en realizar la inferencia de datos.
- **Que tenga un formato de modelo eficiente.** Cuando se entrena en un supercomputador o en un servidor, normalmente no hay que preocuparse del formato del modelo, sin embargo, al ser ejecutado en un dispositivo móvil el cuál se espera que tenga una potencia de cómputo mucho menor, el formato del modelo cobra mucha más importancia. Un framework diseñado para ser usado en dispositivos móviles debería tener las herramientas necesarias para poder realizar la conversión de modelos si así se precisara.

TensorFlow Lite, por lo tanto, no es un framework para entrenar modelos, sino un conjunto de herramientas diseñadas para optimizar y facilitar el uso de aplicaciones basadas en machine learning en dispositivos móviles o sistemas empotrados. Debe ser visto por ello, como un conversor de modelos de TensorFlow a formato **.tflite** y un conjunto de intérpretes de distintos dispositivos.

En la imagen a continuación 3.5 se puede ver un esquema de conversión de modelos a tflite y los dispositivos en los que pueden ser usados dichos modelos, que como se muestra, van desde intérpretes de móvil (Andrioid o iOS) hasta intérpretes de Linux.

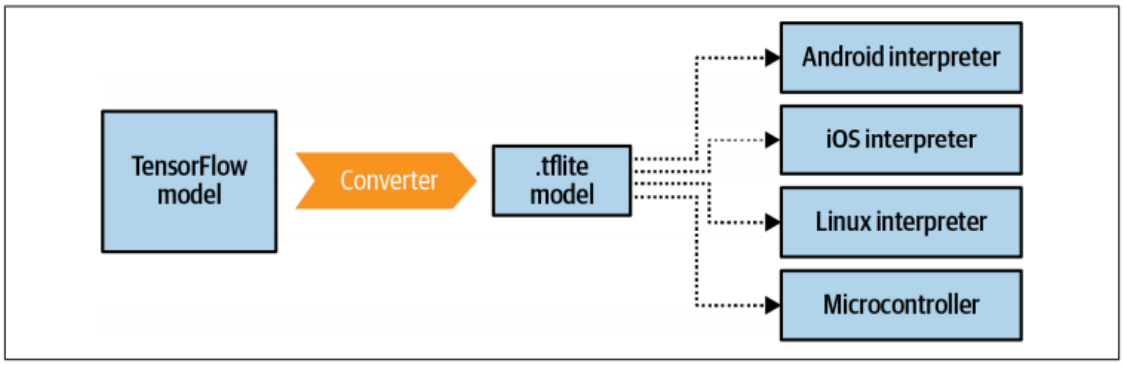

Figura 3.5: Esquema de TensorFlow Lite [2]

3.3. BERT

BERT(Bidirectional Encoder Representations for Transformers). Se trata de un sistema basado en IA para ayudar a los algoritmos de Google a entender mejor el lenguaje que utilizamos en el buscador de Google. Para lograrlo, BERT posee una característica llamada **bidireccionalidad** que consiste en analizar una oración en dos direcciones. Analiza las palabras que se encuentran tanto a la izquierda como a la derecha de una palabra clave, lo que permite entender en profundidad el contexto y la temática de toda la frase. Nos hemos ayudado del blog: [5] para complementar la explicación.

Además, BERT utiliza una red neuronal de código abierto para procesar el lenguaje natural de las búsquedas introducidas, ya que propone entrenar un modelo base que aprenda a interpretar el lenguaje en general, en vez de crear un modelo que resuelva cada tarea individualmente.

El modelo base se puede entrenar con datasets inmensos y luego re-entrenarlo con datasets más pequeños para que realize tareas más específicas.

3.3.1. Arquitectura en detalle

BERT surge de las Redes Transformers, un tipo de arquitectura creado en 2017. Las Redes Transformers procesan el texto en dos fases diferentes: una para la codificación, la cual se encarga de procesar el texto de entrada y codificarlo numéricamente para extraer su información mas relevante y otra frase para la decodificación, encargada de generar una nueva secuencia de texto.

Cada bloque está conformado por múltiples codificadores y decodificadores como se puede observar en la imágen 3.6

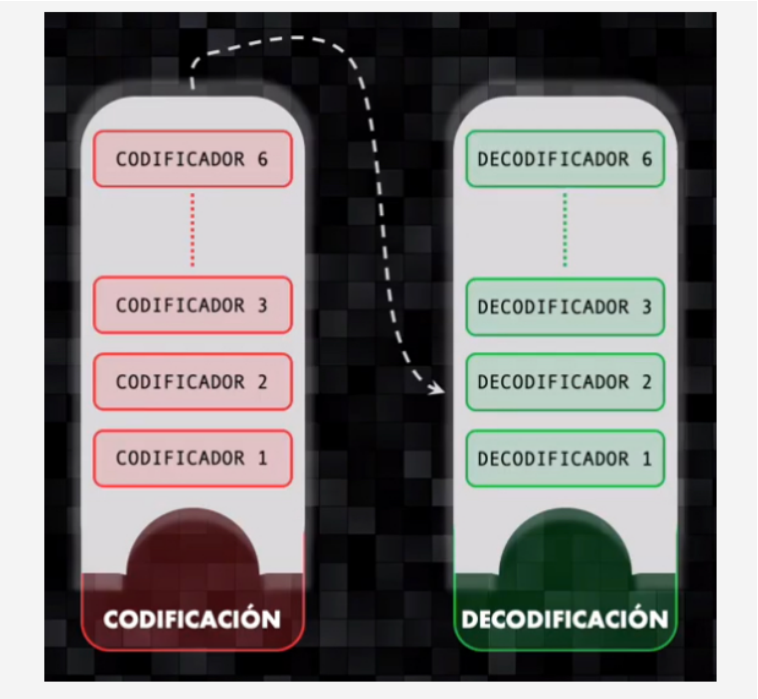

Figura 3.6: Estructura de una Red Transformer [5]

Con BERT, nos interesa un modelo capaz de **codificar** texto, por lo que omitiremos la parte de decodificación de la Red Transformer.

En las tareas vistas anteriormente, en el apartado de NLP los datos de entrada suelen estar conformados por dos elementos, ya sea dos frases, una frase y una pregunta o dos preguntas. Por lo tanto, el texto de entrada se codifica como dos frases:

a). Al principio, se incluye un token de clasificación (CLS) y para separar una frase de otra, un token de separación (SEP).

b). Después se obtienen tres representaciones por cada token:

- Un **embedding**, que representa el token como un vector.
- Una **codificación posicional**, la cual sirve para indicarle al bloque de codificación la posición relativa de cada palabra dentro de la frase, ya que todas las palabras se procesan de forma simultánea por la red.
- Adicionalmente se añade un embedding más que indica a qué **segmento** pertenece la frase. La primera frase antes del separador, se codifica con un embedding diferente a la de la segunda.

Finalmente se suman las tres representaciones y los vectores resultantes de la suma son procesados por BERT.
En la figura 3.7 se puede observar un esquema donde tenemos las tres representaciones y el vector resultante de la suma.

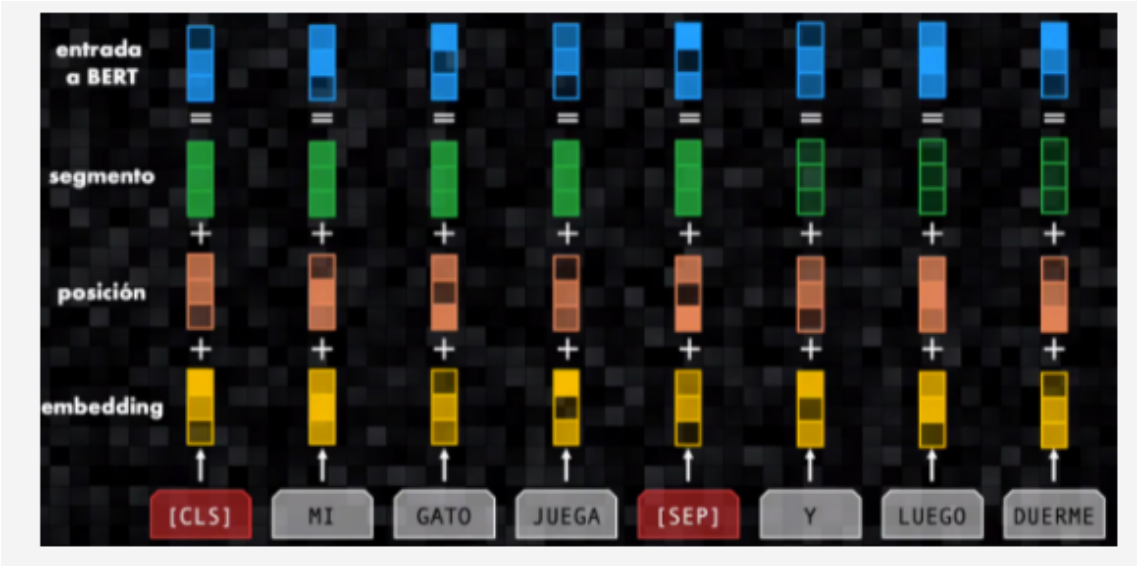

Figura 3.7: Esquema de procesamiento de texto en BERT [5]

El entrenamiento de los modelos de BERT se realiza en dos partes:

a). **Pre-entrenamiento**.

BERT es pre-entrenado usando dos datasets muy grandes, Wikipedia y Google Books, los cuales tienen en conjunto más de 3.000 millones de palabras.

Con el entrenamiento, BERT aprende a analizar el texto de manera bidireccional, lo cual implica que tiene en cuenta todo el contexto, tanto lo que está a su izquierda, como lo que está a su derecha.

Primero, se enseña al modelo a completar una palabra faltante en una frase, y como esa palabra puede estar en cualquier ubicación, se fuerza al modelo a que aprenda a analizar el texto de manera bidireccional. Aprendiendo así a comprender el lenguaje tal y como lo hacemos los humanos.

Tras esto, se enseña al modelo a predecir la continuación una frase. Dadas dos frases (A y B) el modelo tiene que averiguar si B es realmente la frase que precede a A o es una frase aleatoria, para ello durante el pre-entrenamiento, el 50 % de las veces la frase B preceda a la A y el otro 50 % no.

b). **Afinación**.

Al modelo pre-entrenado se le agregan dos capas: una red neuronal y una capa softmax. Después se presentan las frases de la tarea específica y se re-entrena el modelo de extremo a extremo, con la única diferencia de que esta nueva fase requiere menos tiempo pues el modelo ya ha sido pre-entrenado.

Para las tareas de tipo pregunta-respuesta, la red neuronal y la capa softmax, realizarán la predicción de los tokens de inicio y finalización de la respuesta encontrada dentro del texto.

3.3.2. MobileBERT

Como ya mencionamos en el apartado de TensorFlow Lite, los dispositivos móviles ofrecen una serie de desafíos a la hora de la programación de aplicaciones y más aún cuando hablamos de Deep Learning.

Básicamente, MobileBERT es una versión ligera de BERT, concretamente, 4.3 veces mas pequeña y 5.5 veces más rápida. logrando mantener resultados igual de precisos que su padre BERT.

3.4. Cuantización de modelos

La cuantización posterior al entrenamiento en el modelo es una técnica de conversión que puede reducir el tamaño del modelo y la latencia de inferencia, mejorando al mismo tiempo la velocidad de inferencia del procesador con una pequeña degradación del modelo.

Más adelante, se verán los tipos de cuantización más en detalle, a la hora de evaluar los modelos y explicar su estructura interna.

Capítulo 4

Modelos en detalle

Uno de los principales objetivos de este TFG es el desarrollo de modelos y su estudio en diferentes máquinas con diferentes entradas de datos. Los modelos en cuestión se han desarrollado en python, usando librerias de TensorFlow y exportados a formatos de tipo TensorFlow Lite(.tflite).

A continuación se ofrece una descripción del código usado en la generación, entrenamiento y evaluación de los modelos. Toda la información, pasos a seguir y código la podemos encontrar en el siguiente blog: [6].

4.1. Generación de los modelos

En esta sección se va a explicar cómo se generan los modelos paso a paso, desde importar las librerias necesarias hasta la evaluación final del modelo generado. Para poder ejecutar el script correctamente, es necesario instalar el paquete Model Maker, del cuál hablarmeos mas adelante, con el siguiente comando:

$ pip install -q tflite-model-maker

4.1.1. Model Maker

Model Maker es una biblioteca de TensorFlow Lite que nos permite simplificar el proceso de adaptación y conversión de un modelo de TensorFlow a datos de entrada particulares al implementar este modelo para aplicaciones de AA en el dispositivo.

La biblioteca se puede usar para las siguientes tareas de AA:

- **Clasificación de imágenes:** Clasifica las imágenes predefinidas.
- **Detección de objetos:** Detecta objetos en tiempo real.
- **Clasificación de texto:** Clasifica el texto en categorias predefinidas.
- **Respuesta a la pregunta BERT:** Encuentra la respuesta en un contexto determinado para una pregunta determinada con BERT.
- **Clasificación de audio:** Clasifica el audio en categorías predefinidas.

- **Recomendación:** Recomienda elementos basados en la información de contexto para el escenario en el dispositivo.

A nosotros nos interesa la parte de *Respuesta a la pregunta BERT* ya que nos ofrece funciones muy útiles como **DataLoader**, **ExportFormat** o **model_spec**.

```
import tensorflow as tf
assert tf.__version__.startswith('2')

from tflite_model_maker import model_spec
from tflite_model_maker import question_answer
from tflite_model_maker.config import ExportFormat
from tflite_model_maker.question_answer import DataLoader
```

Figura 4.1: Librerias y paquetes necesarios [6]

Tras importar las librerias y módulos como se ve en la imágen 4.1, hay que elegir una especificación de modelo que represente un modelo para la respuesta a la pregunta. Tenemos tres para elegir:

a). **mobilebert_qa**. 4,3 veces mas pequeño y 5,5 más rápido que bert_qa al tiempo que logra resultados competitivos, adecuado para escenarios en dispositivos móviles.

b). **mobilebert_qa_squad**. Igual que mobilebert_qa pero el modelo inicial está reentrenado en SQuAD1.1.

c). **bert_qa**. Modelo BERT estándar que se usa ampliamente en tareas de PNL

Vamos a usar mobilebert_qa_squad ya que al estar entrenado en SQuAD1.1, es capaz de cubrir más rápido la tarea de respuesta a preguntas(Figura 4.2)

```
spec = model_spec.get('mobilebert_qa_squad')
```

Figura 4.2: Especificación de modelo mobilebert_qa_squad [6]

SQuAD(*Standford Question Answering Dataset*) es una colección de pares de preguntas y respuestas derivadas de artículos de Wikipedia donde cada respuesta de cada pregunta es un segmento de texto. SQuAD1.1 posee mas de 100.000 preguntas y respuestas de más de 500 artículos diferentes.

El siguiente paso es obtener un dataset, de triples pregunta-respuesta-contexto para entrenar nuestro modelo, para ello usaremos TriviaQA el cual nos ofrece hasta 650.000 triples de pregunta-respuesta-contexto, al ser tan grande usaremos un subconjunto de unos 8.000 elementos(Figura 4.3). Podemos obtenerlo del siguiente enlace:

`https://storage.googleapis.com/download.tensorflow.org/models/tflite/dataset/triviaqa-web-tr`
`json`

Usando el método de Model Maker `DataLoader.from_squad` podemos cargar y preprocesar los datos en formato SQuAD tanto los datos de entrenamiento como los de validación del modelo (Figura 4.4).

```
{"question": "What areas did Beyonce compete in when she was growing up?",
 "id": "56be85543aeaaa14008c9065",
 "answers": [{"text": "singing and dancing", "answer_start": 207}],"is_impossible": false}


{"question": "When did Beyonce start becoming popular?",
 "id": "56be85543aeaaa14008c9063",
"answers": [{"text": "in the late 1990s", "answer_start": 269}], "is_impossible": false}

{"question": "How many records has Beyonce sold in her 19 year career?",
 "id": "56bf725c3aeaaa14008c9647",
"answers": [{"text": "118 million", "answer_start": 393}], "is_impossible": false}
```


Figura 4.3: Ejemplos de pares pregunta-respuesta en SQuAD1.1 [6]

```
train_data = DataLoader.from_squad(train_data_path, spec, is_training=True)
validation_data = DataLoader.from_squad(validation_data_path, spec, is_training=False)
```

Figura 4.4: Carga y preprocesamiento de los datos [6]

4.2. Entrenamiento de los modelos

Antes de profundizar en el entrenamiento de los modelos vamos a ver ciertas variables dignas de mención relaccionadas con la especificación del modelo(spec), las cuales nos van a ayudar a entender las características del modelo.

Se muestran entre paréntesis los valores por defecto.

- **seq_len (384):** Representa la longitud del texto de entrada.
- **query_len (64):** Representa la longitud de la pregunta de entrada.
- **predict_batch_size (8):** Tamaño del lote de entrada para la predicción.
- **name (MobileBert):** Devuelve el nombre del modelo soportado.
- **batch_size (32):** Número de muestras que se van a utilizar en cada paso de la formación del modelo.
- **epochs (2):** Número de iteraciones que va a realizar el modelo sobre los datos de inferencia.

El número de epochs y de batch_size se puede modificar en base a nuestros objetivos.

Aumentar el tamaño de epochs supondrá una **mejora de la precisión** pero es posible que se produzca un **sobreajuste** ya que se está entrenando con los mismos datos muchas veces, por lo que, al usar el modelo para responder a preguntas muy diferentes sobre textos también muy diferentes a los que usamos en el entrenamiento, la respuesta será muy ambigua incluso sin sentido.

Modificar el tamaño de batch_size puede suponer un consumo de recursos de memoria menor y realmente no tiene un impacto negativo significativo.

En una de las máquinas donde se probaron los modelos, se tuvo que reducir el batch_size en el script de craci´on del modelo para evitar problemas de memoria.

Una vez explicado esto precedemos a la creación del modelo con el batch_size y epochs que consideremos oportunas, recordemos que estas variables se pueden omitir en el script para usar sus valores por defecto (Figura 4.5).

```
model = question_answer.create(train_data, model_spec=spec, epochs=5, batch_size=64)
```

Figura 4.5: Creación y entrenamiento del modelo [6]

Una vez creado el modelo, podemos ver un resumen detallado de la estructura del modelo (Figura **??**)

```
model.summary()
```

Figura 4.6: Resumen del modelo [6]

Esta función nos ofrece información del modelo que ha sido creado. Concretamente nos ofrece una visión de los diferentes tensores y su tipo, la altura y anchura de sus matrices y si están concetados entre sí (Figura 4.7, mucha de la información de cada variable se ha obtenido de la siguiente página de documentación de MobileBERT: [7]).

- **input_word_ids:** Los ids de las palabras son índices de tokens, es decir, representaciones numéricas de las palabras del texto de entrada transformadas a tokens para poder ser tratadas en un modelo de Deep Learning.
- **input_mask:** El valor mask en modelos de AA, suele referirse a un valor que nos ayuda a localizar cuál es el valor más importante a tratar de los datos de entrada. En MobileBERT se refiere concretamente a un valor que nos ayuda a determinar que tokens deberían tratarse primero y cuáles no, esto depende de varios factores como la longitud de la secuencia.
- **input_type_ids:** En los modelos de pregunta-respuesta, se necesita definir al menos dos tipos de ids para ayudar a determinar y tokenizar las frases, como el tipo classifier(CLS) y el tipo separator(SEP).
- **hub_keras_layer_v1v2:** Esta capa envuelve un objeto que actúa como una capa de tipo Keras, y está conectada con las tres capas anteriores.
- **start_positions:** Esta capa determina las etiquetas de las posiciones(índices) del inicio del intervalo de etiquetado usado para calcular la pérdida(loss) de clasificación de los tokens.
  Esta capa está concectada con la capa hub_keras_layer_v1v2
- **start_positions:** Esta capa determina las etiquetas de las posiciones(indices) del fin del intervalo de etiquetado usado para calcular la pérdida(loss) de clasificación de los tokens.
  Esta capa está concectada con la capa hub_keras_layer_v1v2.

También podemos realizar una evaluación del modelo con los datos de validación (validation_data), esto lo veremos en el apartado de evaluación del modelo.

```
Model: "model"
__________________________________________________________________________________________________
Layer (type)                    Output Shape         Param #     Connected to
==================================================================================================
input_word_ids (InputLayer)     [(None, 384)]        0
__________________________________________________________________________________________________
input_mask (InputLayer)         [(None, 384)]        0
__________________________________________________________________________________________________
input_type_ids (InputLayer)     [(None, 384)]        0
__________________________________________________________________________________________________
hub_keras_layer_v1v2 (HubKerasL {'end_logits': (None 24582914    input_word_ids[0][0]
                                                                 input_mask[0][0]
                                                                 input_type_ids[0][0]
__________________________________________________________________________________________________
start_positions (Lambda)        (None, None)         0           hub_keras_layer_v1v2[0][1]
__________________________________________________________________________________________________
end_positions (Lambda)          (None, None)         0           hub_keras_layer_v1v2[0][0]
==================================================================================================
Total params: 24,582,914
Trainable params: 24,582,914
Non-trainable params: 0
```

Figura 4.7: Resumen del modelo

Por último, debemos convertir y exportar el modelo al formato de TensorFlow Lite para que pueda ser usado en aplicaciones de AA. Al ser una aplicación de móvil se recomienda cuantizar el modelo para reducir su tamaño y aumentar la rapidez de la ejecución, pero esto lo veremos en el apartado siguiente.

Los formatos de exportación permitidos para el modelo pueden ser unos de los siguientes:

- **ExportFormat.TFLITE**
- **ExportFormat.VOCAB**
- **ExportFormat.SAVED_MODEL**

De forma predeterminada se exporta solo el modelo de TensorFlow Lite con metadatos (Figura 4.8).

```
model.export(export_dir='.')
```

Figura 4.8: Exportación del modelo a formato tflite [6]

4.3. Modelos en detalle

Para la realización de este trabajo se han usado diferentes tipos de modelos con diferentes tipos de cuantizacion o diferentes tamaños de longitud de secuencia, es decir, diferentes tamaños de texto de entrada que acepta el modelo. A continuación se explicarán en detalle la estructura de los diferentes modelos en detalle y más adelante, en el apartado de **Resultados**, se valorarán los datos obtenidos.

Se han usado tres modelos de cada tipo de cuantización los cuales difieren entre sí en la longitud del texto de entrada que aceptan los modelos(**seq_len**). Se ha usado

seq_len de tamaño: **198, 384 y 512**. Para ello, solo hay que modificar el valor de seq_len de la especificación del modelo antes de crearlo, con la siguiente línea de código: **spec.seq_len = 512**

Los modelos de 192 y 384 han sido creados bajo un entorno de Google Colab, mientras que el modelo de 512 ha sido creado en una de las máquinas donde también se probó su eficiencia, por problemas con la memoria. Además, este modelo ha tenido que usar un **batch_size** menor para ser entrenado, lo que implica que son tomados menos datos en cada iteración mientras el modelo es entrenado, por lo que ralentiza el proceso de entreno pero no cambia significativamente la precisión en relación a los otros dos modelos.

4.3.1. Tipos de modelo

En total, tenemos seis modelos que han sido probados en nuestras dos máquinas. En la siguiente tabla (4.1) se muestra el tamaño de cada modelo:

	192	**384**	**512**
No cuantizado	24,8 MB	25,1 MB	25,3 MB
float16()	47,5 MB	47,7 MB	47,6 MB

Cuadro 4.1: Tamaño de los modelos usados

Observamos que el tamaño de cada modelo aumenta en base al tamaño de seq_len con los que son creados y entrenados. En este caso ha aumenado 300Kb al pasar de 192 a 384 y ha aumentado otros 200Kb al pasar de 384 a 512.

Para entender mejor la estructura interna de los modelos usaremos una herramienta llamada Netron.

Netron es un visualizador de redes neuronales que nos permite ver el árbol de decisión y las capas que lo forman y todas sus propiedades, entre las que se encuentran:

- El tipo de la capa
- El nombre de la capa
- Si la capa es entrenble
- El tipo de los datos de entrada
- El tipo de los datos de salida

En primera instancia, la única diferencia entre los modelos del mismo tipo de cuantización pero diferente seq_len, es precisamente esto último, lo cual podemos comprobarlo en las imagenes de netron, las cuales muestran la estructura de un modelo 384 (Figura 4.9) y otro 512 (Figura 4.10) con el mismo tipo de cuantización.

Dada la magnitud de estos modelos sería un arduo trabajo explicar su estructura paso por paso, por lo tanto escogemos una capa común, la que está rodeada de color rojo, y la explicaremos brevemente.

Observamos dos tipos de inputs: **data** y **shape**, en el primero tenemos una matriz de cuatro dimensiones, de tipo float32, en la que se encuentran los datos de entrada. En el segundo, observamos que tenemos un array de tipo int32 con tres elementos, los cuales

equivalen a los tres valores(**input_words_ids, input_mask, input_type_ids**) que comentamos previamente en la parte del resumen del modelo *model.summary*.

Por último, observamos que como salida (output) tenemos una matriz de tres dimensiones de tipo float32 cuyos valores, que forman la estructura de la matriz, coinciden con los valores del input shape. Esto es así porque esta capa en concreto se encarga de remodelar los datos de entrada en base a los parámetros del input shape, para devolver como salida los datos ya remodelados a las siguientes capas.

En resumen, observamos que no hay diferencia entre la estructura del modelo de 384 y de 512, ya que son el mismo tipo de modelo con tamaños de entrada de datos distinto.

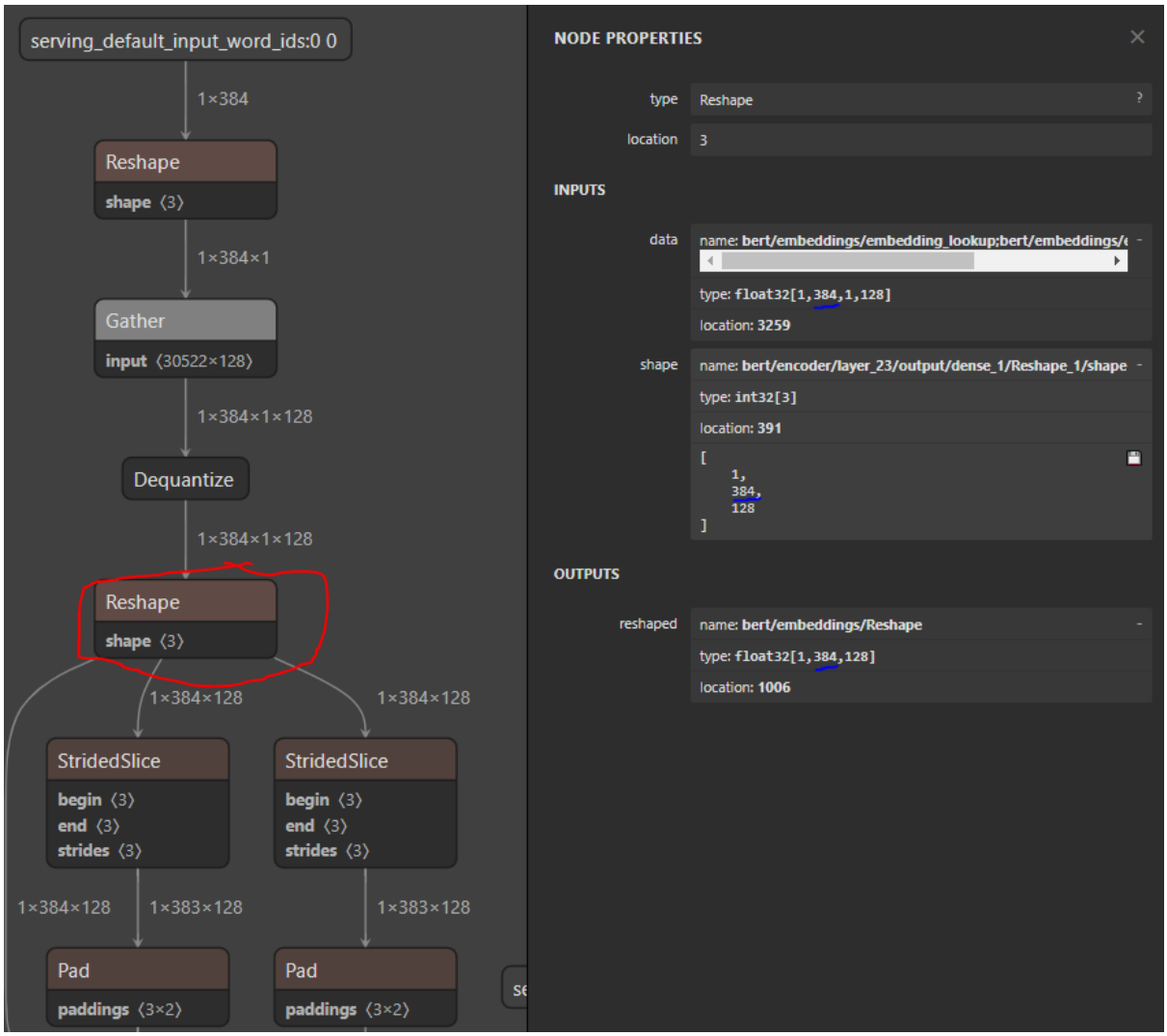

Figura 4.9: Modelo de 384

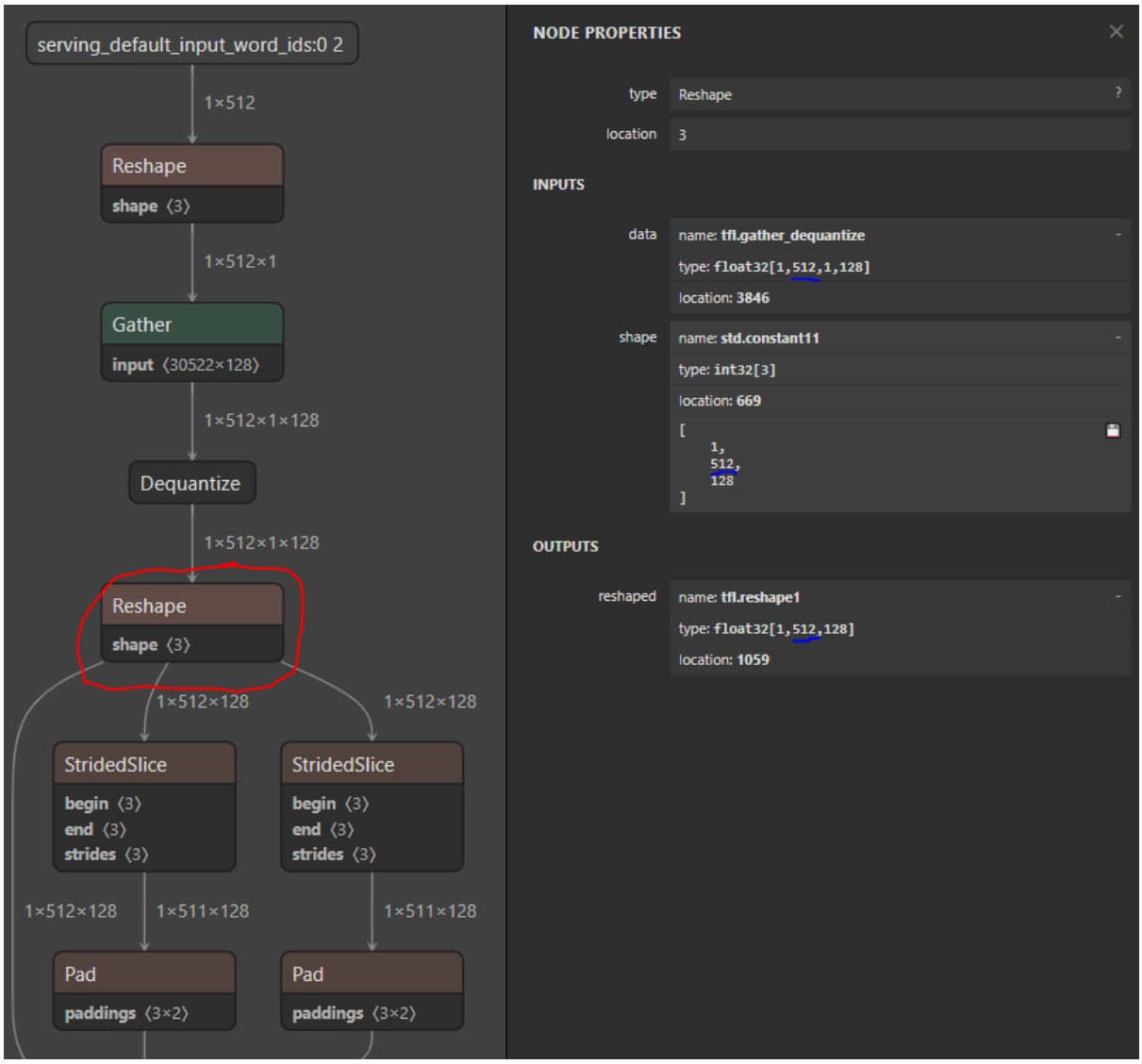

Figura 4.10: Modelo de 512

Si bien parece que la comparación entre los dos modelos anteriores era algo trivial, es necesaria para poder recalcar la importancia de la comparación que se va a realizar a continuación. En este caso vamos a comparar dos modelos, pero esta vez tendrán el mismo seq_len: **384** pero distinto tipo de cuantización, uno será un modelo **No Cuantizado** (Figura 4.11) y el otro estará cuantizado en **float16()** (Figura 4.12).

En las dos imagenes siguientes se puede ver la estructura inicial del árbol de un modelo No Cuantizado de 384 y otro cuantizado a float16 de 384. Observamos que ya al comienzo la estructura es diferente:

- El modelo No Cuantizado, remodela los datos de entrada(capa Reshape), despues los recolecta(Capa Gather) y por último los descuantiza.
- El modelo cuantizado a float16, fusiona el proceso de descuantización y el de remodelado de datos(Capa Reshape) y devuelve el resultado a la capa Gather como inputs.

Esto se debe a que, en el modelo No cuantizado, la capa de Descuantización trata datos enteros de 8 bits y los transforma en datos de tipo float 32 para ser usados por las capas siguientes, sin embargo la descuantización del modelo float16 trata datos de tipo float16 y los transforma en datos de tipo float32.

La primera imagen nos ofrece una breve descripción de la capa de descuantización del modelo No Cuantizado, en la que podemos ver que cuenta con un input: una matriz de cuatro dimensiones de tipo int8 y un output: una matriz de cuatro dimesniones de float32, la estructura del input y del output es la misma, por lo que deducimos que esta capa se encarga de cambiar el tipo de datos, en base a un valor **q** y transformarlos a datos de tipo float16.

En la segunda imagen, vemos también una breve descripción de la capa de descuantización del modelo cuantizado a float16, sin embargo, en este caso el input es una matriz de dos dimensiones de tipo float16 y el output, aunque no se muestre en la imagen, es una matriz de dos dimesiones de tipo float32. Por lo tanto, la función de esta capa es la misma que el modelo No Cuantizado, solo cambia el tipo de datos a ser descuantizados.

Como hemos visto la estructura y la forma de tratar los datos es diferente, esto afectará al rendimiento de cada modelo, pero esto lo veremos más adelante, en el apartado de resultados

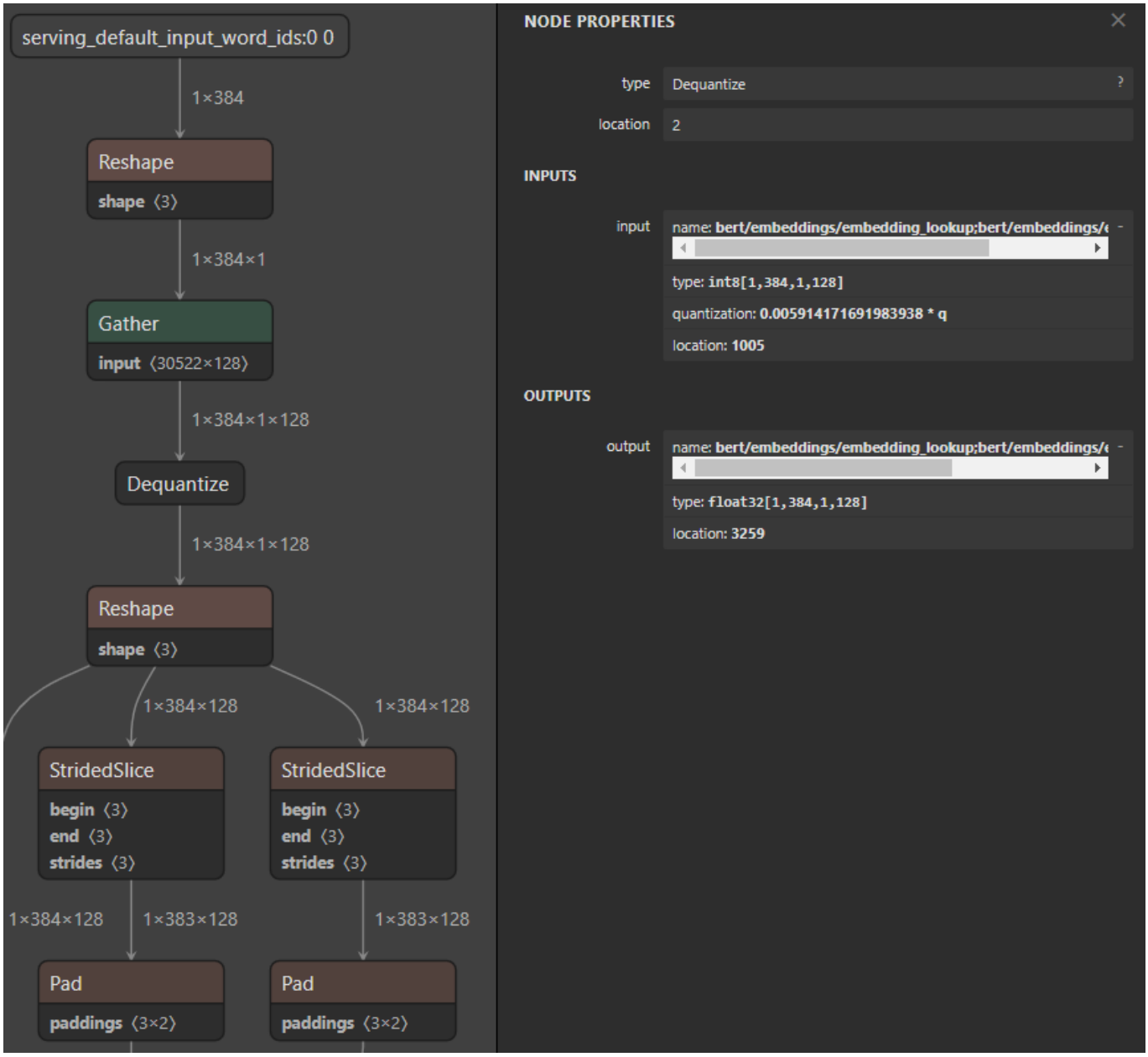

Figura 4.11: Modelo No cuantizado

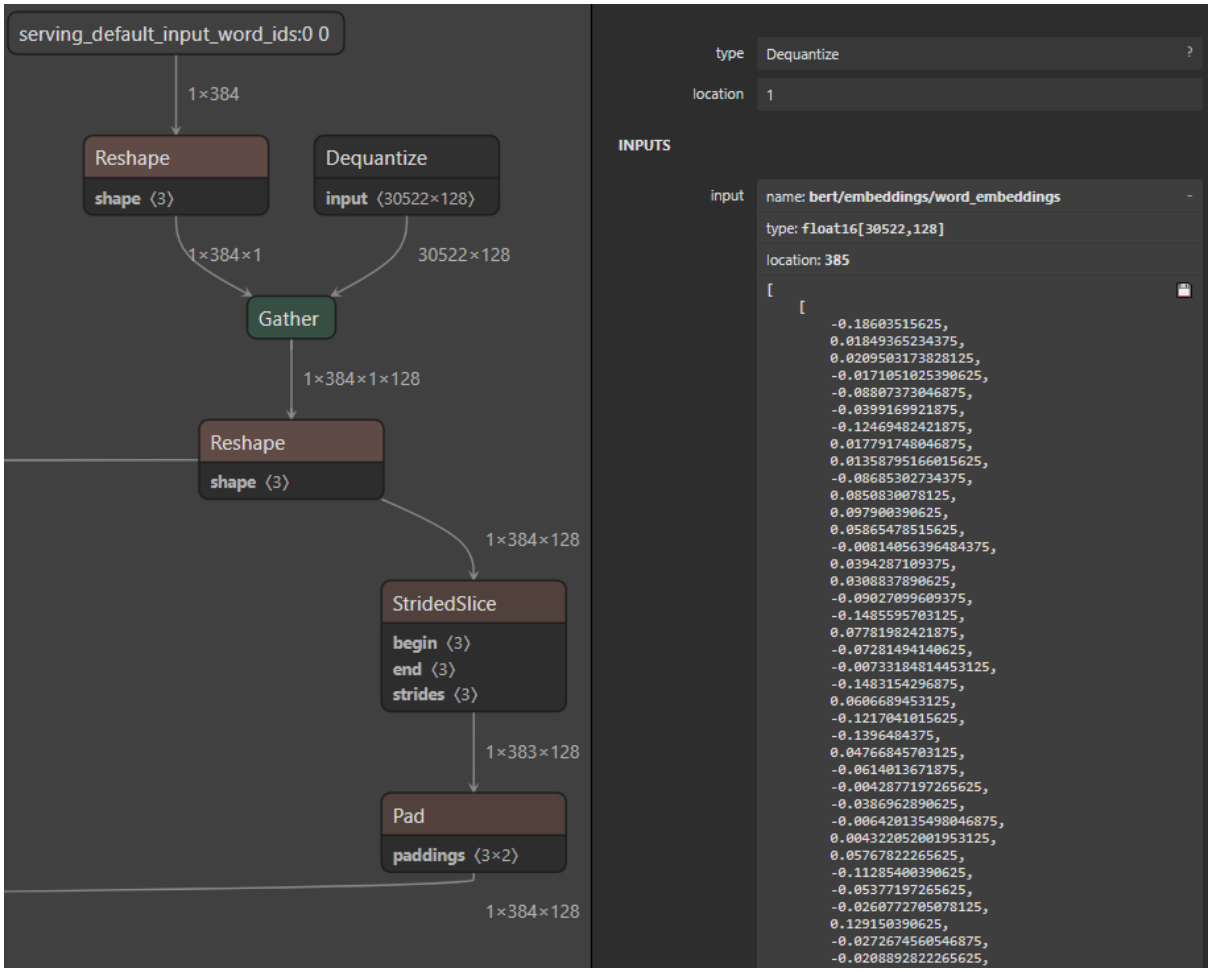

Figura 4.12: Modelo cuantizado a float16

4.3.2. Evaluación de los modelos

Evaluar los modelos con los datos de validación nos proporciona un diccionario de métricas(**f1** y **exact_match**). Cada modelo ofrece unas métricas diferentes, en nuestro caso, como hemos usado datos de validación y de entrenamiento de tipo SQuAD, obtenemos estas métricas al llamar a la función **model.evaluate(validation_data)** (Figura 4.13), de hecho, las métricas cambian dependiendo la versión de SQuAD usada (SQuAD1.1 ofrece métricas distintas a SQuAD2.0).

```
INFO:tensorflow:Made predictions for 200 records.
INFO:tensorflow:Made predictions for 200 records.
INFO:tensorflow:Made predictions for 400 records.
INFO:tensorflow:Made predictions for 400 records.
INFO:tensorflow:Made predictions for 600 records.
INFO:tensorflow:Made predictions for 600 records.
INFO:tensorflow:Made predictions for 800 records.
INFO:tensorflow:Made predictions for 800 records.
INFO:tensorflow:Made predictions for 1000 records.
INFO:tensorflow:Made predictions for 1000 records.
INFO:tensorflow:Made predictions for 1200 records.
INFO:tensorflow:Made predictions for 1200 records.
{'exact_match': 0.5884353741496599, 'final_f1': 0.6621698029861295}
```

Figura 4.13: Evaluación del modelo [6]

- **exact_match:** Esta métrica nos proporciona información de la precisión que tiene el modelo ante casos de acierto, es como la métrica accuracy en otros modelos.
- **final_f1:** Muestra la métrica f1, que se refiere a la combinación entre las medidas de precisión y exhaustividad que tiene el modelo. Se calcula con la fórmula: 2*((precision*recall)/(precision*recall)).

Se ha realizado una evaluación para cada modelo existente, en formato estandar(mobileBERT), en formato tflite sin cuantizar y en formato tflite cuantizado a float 16(). Esto se ha realizado para modelos de 192, 384 y 512. Los resultados son los siguientes:

- **Valores model(192):** 'exact_match': 0.568027, 'final_f1': 0.636059
- **Valores model.tflite(192):** 'exact_match': 0.568027, 'final_f1': 0.642020
- **Valores model_float16.tflite(192):** 'exact_match': 0.578231, 'final_f1': 0.641615
- **Valores model(384):** 'exact_match': 0.595238, 'final_f1': 0.669312
- **Valores model.tflite(384):** 'exact_match': 0.588435, 'final_f1': 0.662347
- **Valores model_float16.tflite(384):** 'exact_match': 0.595238, 'final_f1': 0.669312

Observamos que no hay mucha diferencia en las evaluaciones de los distintos modelos, ya que su eficacia radica en las épocas con las que ha sido entrenado, la cuantización o el cambio de formato a tflite no cambia mucho, si bien parece que los modelos con mayor seq_len tienen una eficacia relativamente mayor a los de menor seq_len no consideraremos esto como algo relevante ya que la diferencia es mínima.

Capítulo 5

Diseño de la aplicación

La aplicación consiste en una página web, donde podemos comprobar la eficacia de un modelo concreto para textos de tamaño variable. Ofrece la posibilidad de cargar el texto de entrada a mano o en un txt así como cargar una o varias preguntas a mano o en un txt.

Se puede acceder a la aplicación a través del siguiente enlace:

https://drive.google.com/drive/folders/1fgMaRBAAC4mIp12vsopys2JLIRCiEj7l?usp=sharing

La aplicación ha sido desarrollada, así como el script, a partir de otra aplicación mucho más simple que se puede encontrar en el siguiente github: [8]

El principal objetivo es permitir la posibilidad de cargar textos de entrada muy grandes y poder modificarlos de manera sencilla, así como poder cargar una batería de preguntas muy grande y poder modificarlas de manera sencilla añadiendo o eliminando preguntas antes de ejecutar la aplicación.

También nos permite seleccionar el modelo que queremos usar. La única condición es que el modelo se encuentre en el path: **Aplicación/demo/mobilebert**.

5.1. Entrada de datos

Existen dos maneras de introducir datos a la aplicación, a mano o por fichero. La idea de esto es, que si se quiere hacer copia-pega de un texto, no sea necesario pasarlo a un fichero y abrirlo desde la aplicación, sino solo pegarlo en una sección para ser tratado y lo mismo con las preguntas. Pero si por otra parte tenemos ya preparado nuestro texto en un fichero y una gran batería de preguntas puedas pasarlas directamente por fichero.

En la siguiente imagen(5.1) podemos ver la parte de carga de datos de entrada, cada input corresponde a un fichero distinto. Por orden descendente: fichero de texto de entrada, fichero de preguntas y escoger el modelo de los posibles en una lista desplegable.
Queda a elección del usuario si quiere introducir 2 ficheros,(de texto y preguntas) el texto por fichero y las preguntas a mano o cualquier otra combinación. La aplicación ofrece la libertad de poder escoger entre múltiples opciones.

Una vez se tienen los ficheros seleccionados, ya se pueden cargar los datos pulsando el botón verde *Cargar* y aparecerá abajo el contenido de todos los ficheros que hemos sele-

Figura 5.1: Vista de la carga por fichero de datos de la aplicación

cionado previamente, así como el modelo.

En la imagen a continuación(5.2), observamos el contendido de los dos ficheros cargados, y el nombre del modelo que hemos cargado y el seq_len del mismo(será necesario para el correcto funcionamiento de la aplicación). Si, por ejemplo, no hubiesemos seleccionado fichero para las preguntas, aparecería ese campo sin contenido.

El formato al introducir los datos por fichero ya está preparado para ser procesado, por lo que, si queremos modificar o añadir algo a mano debemos seguir ciertas reglas de formato.

- **Texto:** El texto debe tener sentido semántico, si queremos introducir más texto o modificarlo hay que tener especial atención con las faltas de ortografía y los espacios entre palabras ya que afectará al resultado obtenido.
- **Preguntas:** Cada pregunta debe tener el formato de frase acabada con signo de interrogación y cada pregunta debe ir en una sola línea en el caso de que haya más de una pregunta.
- **Modelos:** El nombre del modelo que vamos a usar tiene que coincidir con el nombre del fichero .tflite que hay en el path correspondiente, y no podemos seleccionar más de un modelo por ejecución, el seq_len debe coincidir con el del modelo, esto se hace automático al escoger un modelo de los disponibles, pero si queremos introducir un modelo propio hay que tenerlo en cuenta.

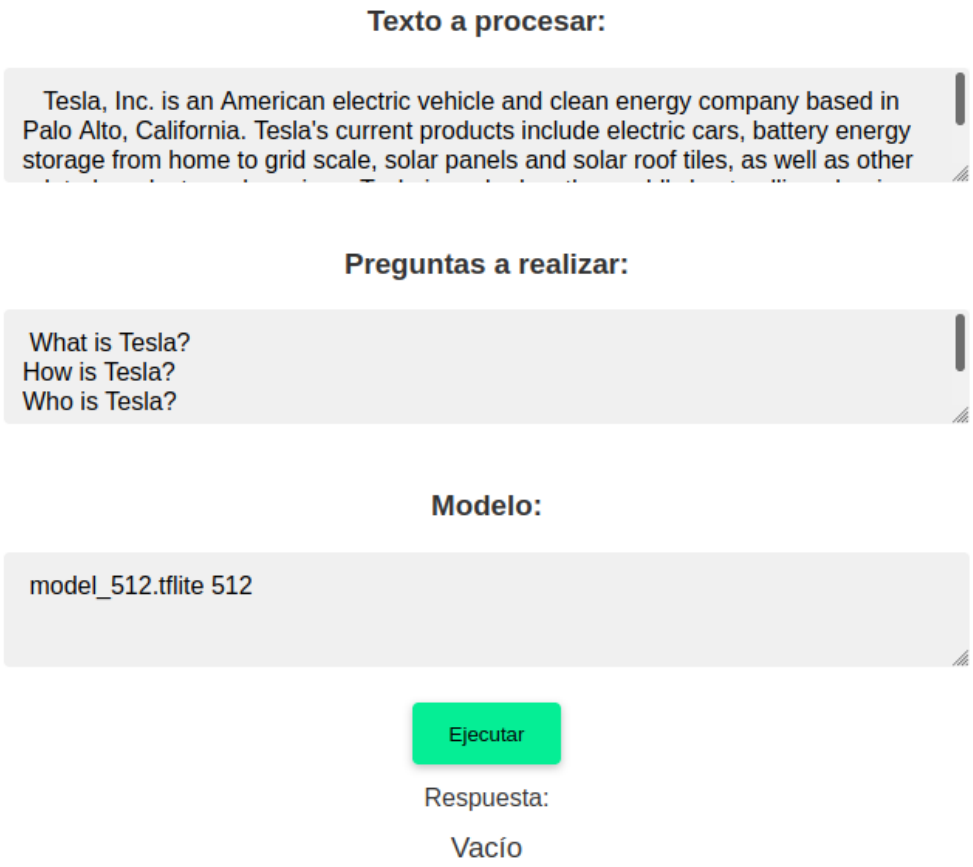

Figura 5.2: Vista de los datos a procesar de la aplicación

5.2. Funcionamiento interno

5.2.1. Frontend

En esta sección vamos a ver internamente cómo funciona el código de carga, preprocesamiento y tratado de datos de la aplicación. Lo primero que veremos es el documento html.
El documento se encuentra en el path: **Aplicación/templates/index.html**.
En la imagen a continuación (5.3), observamos el formulario para cargar los datos y las tres áreas de texto donde aparecerán visualmente los datos cargados, así como el botón de ejecutar el cuál ejecuta el script que está en el documento html que veremos más adelante.
Por último, vemos la sección donde aparecerá la respuesta a la pregunta tras haber terminado la ejecución. Por defecto aparece la palabra Vacío.

Realmente, la vista de la aplicación es bastante simple, su principal función es poder ofrecer comodidad visual al usuario para cargar los ficheros o modificar manualmente los datos de forma intuitiva.

```
<body>
    <h2>MobileBERT</h2>
    <div>
        <h3>Carga el texto de entrada y las preguntas, o escribelo manualmente</h3>
       <form action="{{ url_for('gettext') }}" method="POST" enctype="multipart/form-data">
       <label for="myfile">Texto de entrada: </label>
       <input id="selector" type="file" name="myfile"><br>
        <label for="myfile2">Preguntas: </label>
        <input id="selector" type="file" name="myfile2"><br>

       <!-- <label for="myfile3">Modelo: </label>
        <input id="selector" type="file" name="myfile3"><br>
        -->
        <label for="myfile3">Modelo: </label>
<select name="modelo">
   <option selected value="0"> Elige un modelo </option>
       <optgroup label="Modelos Cuantizados">
       <option value="model_float16_192.tflite">model_192</option>
       <option value="model_float16_384.tflite"">model_384</option>
       <option value="model_float16_512.tflite"">model_512</option>
   </optgroup>
   <optgroup label="Modelos No Cuantizados">
       <option value="model_192.tflite">model_192</option>
       <option value="model_384.tflite">model_384</option>
       <option value="model_512.tflite">model_512</option>
   </optgroup>
</select>
      <input id="cargar" type="submit" value="Cargar">
      </div>
      <div>
        <h3>Texto a procesar:</h3>
        <textarea id="content"> {{textoString}} </textarea>
    </div>
    <div>
        <h3>Preguntas a realizar:</h3>
        <textarea id="question"> {{preguntaString}} </textarea>
    </div>
    <div>
        <h3>Modelo:</h3>
        <textarea id="modelo"> {{modeloString}} </textarea>
    </div>
    </form>

    <div>
        <button onclick="myFunction()">Ejecutar</button>

        Respuesta: <span id="answer">Vacío</span>
    </div>
```

Figura 5.3: Documento html

Una vez vista la parte mas relevante de html, dentro del mismo documento tenemos un script que nos sirve para comunicar con el back, para llevar los datos del front hacia el main.py y para llevar también el resultado de la ejecución(la respuesta) hacia el front para mostrarla al usuario.

En la siguiente imagen (5.4) se muestra la función que recoge la pregunta, el texto y el modelo (question, content, modelo) de las áreas de texto donde se muestran tras cargarlos por fichero o escribiéndolos manualmente. Estos datos se llevan a una query que a su vez se pasa a una petición fetch para llevar los datos al backend donde serán tratados en el fichero main.py, el resultado de la petición será la respuesta a la pregunta que se mostrará al usuario en su campo correspondiente.

```
54          function myFunction() {
55
56              const question = document.getElementById("question").value;
57              const content = document.getElementById("content").value;
58              const modelo = document.getElementById("modelo").value;
59                  console.log(question)
60                  console.log(content)
61                  console.log(modelo)
62              const params = { question, content, modelo };
63
64              var esc = encodeURIComponent;
65              var query = Object.keys(params)
66                  .map(k => esc(k) + '=' + esc(params[k]))
67                  .join('&');
68
69              fetch('http://localhost:5000/api?' + query, {
70                  method: 'GET', // or 'PUT'
71                  headers: {
72                      'Content-Type': 'application/json',
73                  },
74              })
75                  .then(response => response.json())
76                  .then(data => {
77                      console.log('Success:', data);
78                      document.getElementById("answer").innerText = data["answer"]
79                  })
80                  .catch((error) => {
81                      console.error('Error:', error);
82                  });
83
84          }
85      </script>
86
```

Figura 5.4: Documento html(script)

También tenemos una parte de estilo css que se encuentra dentro del documento html, realmente esta parte no es muy relevante ya que simplemente da estilo a la pagina web.

5.2.2. Backend

Nuestro fichero de código principal se encuentra en el path:**Aplicacion/Demo/main.py** al principio se importan los paquetes y librerías necesarios.

El método **gettext()** Figura 5.5 se encarga de cargar los datos y transformarlos a string para renderizar la plantilla del html con los datos recogidos, es llamado cuando pulsamos el botón *Cargar* tras haber seleccionado los ficheros que queremos.

En el caso de los ficheros de Texto y Preguntas, se leen y el contenido se guarda en una varibale de tipo string, sin embargo, en el modelo, se guarda el nombre del modelo que se ha seleccionado y el tamaño de seq_len que se le asigna dependiendo del nombre que tenga.

Todo esto se devuelve a la plantilla html para mostrarlo al usuario.

```
app = Flask(__name__)

@app.route('/gettext', methods=['GET','POST'])
def gettext():

    if request.method == 'POST':
                |
        file = request.files['myfile']
        filename = file.filename
        #print (os.path.abspath(filename))
        texto = file.read()
        textoString = texto.decode('UTF-8')


        file = request.files['myfile2']
        filename = file.filename
        pregunta = file.read()
        preguntaString = pregunta.decode('UTF-8')


        #file = request.value['myfile3']
        #print(file)
        #modeloString = os.path.abspath(file.filename)
        #modeloString = file.filename


        modelo = request.form["modelo"]
        if modelo == "model_192.tflite":
                modeloString = modelo + " " +  "192"
        if modelo == "model_384.tflite":
                modeloString = modelo + " " +  "384"
        if modelo == "model_512.tflite":
                modeloString = modelo + " " +  "512"
        if modelo == "model_float16_192.tflite":
                modeloString = modelo + " " +  "192"
        if modelo == "model_float16_384.tflite":
                modeloString = modelo + " " +  "384"
        if modelo == "model_float16_512.tflite":
                modeloString = modelo + " " +  "512"


        return render_template("index.html", textoString=textoString, preguntaString=preguntaString, modeloString=modeloString)


    else:
        result = request.args.get['myfile']
    return result
```

Figura 5.5: Método gettext() de main.py

El método **run()** es llamado tras pulsar el botón *Ejecutar*, recoge el texto, las preguntas y el modelo con su seq_len de la vista.
Crea un objeto de tipo BERT al que se le pasa el path del modelo que hemos seleccionado y el seq_len. Como hemos comentado, los modelos necesitan estar previamente en el path: **Aplicación/demo/mobilebert**, una vez esté allí podemos tener los que queramos y seleccionarlos.

Es importante hacer un breve inciso en este punto, ya que también se ha modificado el **__init__.py** (Figura 5.6) dentro de la carpeta mobilebert. En principio este código no es necesario modificarlo, pero al decidir poder seleccionar manualmente el modelo, es necesario hacer un par de cambios. En la imagen a continuación se observa cómo se recibe por parámetro el path del modelo y posteriormente se crea el intérpetre con una función de tflite donde se recoge el path del modelo. Así mismo, llevamos el seq_len y lo igualamos a max_lenght.

```
class MobileBERT:
        def __init__(self,modelpath, seq_len):
                self.max_length = seq_len
                self.tokenizer = bert.bert_tokenization.FullTokenizer(__file__.replace("__init__.py","vocab.txt"), True)
                with tf.device('/CPU:0'):
                        self.interpreter = tf.lite.Interpreter(model_path=__file__.replace("__init__.py",modelpath.strip()))
                        #self.interpreter = tf.lite.Interpreter(model_path=modelpath)
                        self.interpreter.allocate_tensors()
                        self.input_details = self.interpreter.get_input_details()
                        self.output_details = self.interpreter.get_output_details()
```

Figura 5.6: Parte modificada del fichero __init__.py

Continuando con la explicación de código del método run(), se guarda la batería de preguntas para ser tratadas en una lista. Tras esto, empieza la ejecución profunda de la aplicación(línea 96 - 98), primero se genera un timer para calcular el tiempo de ejecución que se tarda en generar las respuestas a las preguntas, después se guarda en un fichero llamado Answer la pregunta, la respuesta y el tiempo que ha transcurrido y por último se devuelve la respuesta desde el servidor al html para ser visualizada por el usuario.

```
71 @app.route('/api', methods=['GET','POST'])
72 def run():
73     modelpath = request.args.get('modelo', default=None, type=str)
74     question = request.args.get('question', default=None, type=str)
75     content = request.args.get('content', default=None, type=str)
76 
77     values = modelpath.split()
78     print("###############"+values[0])
79     print("###############"+values[1])
80 
81     bert = MobileBERT(values[0], int(values[1]))
82 
83     nlines = len(question.splitlines())
84 
85     questionArr = question.splitlines()
86     for x in questionArr:
87         if x == " ":
88                 questionArr.remove(x)
89                 break
90 
91     print(questionArr)
92     print(content)
93     respuesta1=''
94     cont=0
95     for q in questionArr:
96         start_time = time()
97         answer = bert.run(q,content)
98         elapsed_time = time()-start_time
99         print("Elapsed time: %f seconds." %elapsed_time)
100        x = open("Answer",'a')
101        x.write("La pregunta es:" + q + "\n" + "La respuesta es:" + answer + " y ha transcurrido: " +       str(elapsed_time) + " ms" + "\n" + "\n")
102        x.close
103 
104        print(answer)
105 
106        respuesta1 = respuesta1 + "LA RESPUESTA ES: " + answer + " y ha transcurrido: " + str(elapsed_time) + " ms" + "\n" + "\n"
107        cont+=1
108        if cont == nlines:
109                x = open("Answer",'a')
110                x.write("#################################################################################" + "\n" + "#################################
   + "\n" + "\n")
111                x.close
112                break
113 
114    return {"answer": respuesta1}
```

Figura 5.7: Método run() de main.py

5.3. Ejecución de la aplicación

La ejecución de la aplicación es ciertamente sencilla, los pasos son los siguientes:

a). Abrir una ventana de comandos en el path: **Aplicación/demo** y ejecutamos el comando: **python3 main.py** para encender el servidor local.

b). Una vez el servidor esté encendido hay que abrir un explorador, como firefox y escribir: **http://localhost:5000**. Al ser un servidor local no es necesario tener conexión a internet.

Mientras el servidor esté encendido, podremos obtener información en el terminal que nos informa del estado del servidor, así como de los datos que se están tratando, los tiempos de ejecución etc...

c). Cuando queramos detener el servidor, pulsaremos Control+C en el terminal para finalizar el proceso.

5.4. Script de automatización

Ya que la finalidad de este TFG es la evaluación de los modelos, es necesario realizar muchas ejecuciones, obtener el tiempo de cada ejecución y realizar una media total para poder obtener unos datos consistentes. Para ello sería necesario realizar muchas ejecuciones en la aplicación e ir apuntando todas lo cuál supone un gasto de tiempo y esfuerzo considerable. Por lo tanto, se ha creado un script para automatizar el proceso.

El script se basa en ejecutar la función bert.run() **100** veces para recoger información de los tiempos de ejecución. Consideramos que con 100 veces es suficiente para obtener una muestra considerable de los datos. Así mismo se calculará la **media** y **desviación típica** para ayudar a la comparación entre la eficiencia de los diferentes modelos.

A continuación se muestra una imagen (5.8) del script que se explicará brevemente:

```
1
2 from mobilebert import MobileBERT
3 from time import time
4 import statistics as stats
5 f = open("textoPrueba_384.txt",'r')
6 content = f.read()
7 f.close()
8 q = open("Question.txt",'r')
9 question = q.read()
10 q.close()
11 media = 0
12 lista_elem = []
13 y = open("Time.txt","a")
14 for x in range(0, 100):
15         #aux = x*250
16
17         bert = MobileBERT(media)
18         start_time = time()
19         answer = bert.run(question,content)
20         elapsed_time = time()-start_time
21         print("Elapsed time: %f seconds." %elapsed_time)
22         lista_elem.append(elapsed_time)
23         #media = media + elapsed_time
24         #x = open("Answer",'a')
25         #x.write("Para tamaño de entrada " + str(aux) + " la respuesta es:" + answer + "\n" + "\n")
26         #x.close
27         y.write("El tiempo transcurrido es:" + str(elapsed_time) + "\n")
28
29 else:
30         #media = media / 100
31         media= stats.mean(lista_elem)
32         desviacion = stats.stdev(lista_elem)
33         print("Tiempo medio: %f seconds." %media)
34         print("Desviación típica: %f seconds." %desviacion)
35         y.write("La media de los tiempos es:" + str(media) + "\n")
36         y.write("La desviación típica de los tiempos es:" + str(desviacion) + "\n")
37         y.close()
```

Figura 5.8: Script de automatización de ejecución

Ya que la finalidad del script es ser usado para realizar muchas ejecuciones, no se necesita una interfaz gráfica, basta con ejecutar el comando **python3 main.py** en el path **script/demo** para empezar con la ejecución.

Es necesario realizar ciertos cambios en el script cada vez que se quiere cambiar de modelo:

- Cada vez que se cambie de modelo, dependiendo del seq_len que tenga, es necesario modificar el fichero de lectura del texto(línea 6), ya que con diferentes tamaños de secuencia, son necesarios diferentes tamaños de texto.
- También es necesario modificar manualmente el fichero __init__.py para que recoja el modelo en concreto que queremos evaluar así como el seq_len del mismo.

El script nos devuelve un fichero de texto donde tenemos los tiempos de ejecución de cada iteración así como la media y desviación típica de los 100 tiempos.

Los resultados de todas estas ejecuciones con diferentes modelos en distintas máquinas lo veremos en el apartado resultados.

Capítulo 6

Resultados

En estan sección se van a estudiar los datos obtenidos de todas las ejecuciones que hemos realizado de los diferentes modelos en diferentes máquinas con diferentes características. A la máquina virtual creada en mi ordenador personal con VirtualBox la llamaremos **MiMáquina**, y a la otra máquina, proporcionada por el tutor con acceso en remoto la llamaremos **Patones**. Más adelante se explicarán las diferencias entre sí.

Como ya hemos comentado antes, hemos utilizado seis modelos en total, los cuáles pueden ser cuantizados y no cuantizados y de diferente tamaños de seq_len, concretamente **192, 384 o 512**.

Como hemos mecionado en el capítulo anterior, se ha ejecutado cada modelo 100 veces gracias al script de automatización de la ejecución, de las 100 ejecuciones estudiaremos la media y la desviación típica, ya que son los valores que mayor conocimiento nos pueden aportar a la hora de medir tiempo de ejecución.

Para cada modelo de diferente tamaño de datos de entrada, es necesario así mismo un texto de diferente tamaño.

- Para modelos de seq_len = **192** tenemos un texto de seq_len (una vez tokenizado), de tamaño **189** que equivale a unas 115 palabras
- Para modelos de seq_len = **384** tenemos un texto de seq_len (una vez tokenizado), de tamaño **379** que equivale a unas 260 palabras
- Para modelos de seq_len = **512** tenemos un texto de seq_len (una vez tokenizado), de tamaño **505** que equivale a unas 356 palabras

Para la correcta medición de los tiempos las ejecuciones se han realizado sin procesos que puedan estar consumiendo memoria como algún buscador de internet o algo por el estilo para que sean lo mas limpias y precisas posible.

Antes de ver los resultados y analizarlos, vamos a entrar en detalle sobre las características de la máquina virtual donde hemos realizado las ejecuciones.
Ya que las ejecuciones utilizan la CPU, dependiendo del tipo de CPU, potencia de cómputo, chaché, etc... los resultados pueden variar como veremos más adelante.

	Patones	**MiMáquina**
Modelo CPU	Intel(R) Core(TM) i9-9900K	Intel(R) Core(TM) i5-7400
Frecuencia	3.60 GHz	3.00 GHz
Tamaño de Caché (L3)	16384 Kb	6144 KB
CPU cores	8	1

Cuadro 6.1: Comparación entre MiMáquina y Patones

Usando el comando: **cat proc/cpuinfo**, podemos ver más a fondo la información sobre la máquina. En la tabla 6.1 podemos observar con mas detenimiento las características más relevantes de las 2 máquinas:

Como podemos observar, el CPU de la máquina Patones es más nuevo y potente que el CPU de MiMáquina, por lo que se esperan resultados mejores en Patones, lo cuál confirmaremos a continuación.

6.1. MiMáquina

Los resultados de 100 ejecuciones de cada modelo en MiMáquina es el siguiente:

- **Modelo No Cuantizado de 192**
  - Media: **0.5597 segundos**
  - Desviación típica: **0.0359**
- **Modelo Cuantizado en float16 de 192**
  - Media: **0.4562 segundos**
  - Desviación típica: **0.0464**
- **Modelo No Cuantizado de 384**
  - Media: **1.2063 segundos**
  - Desviación típica: **0.1548**
- **Modelo Cuantizado en float16 de 384**
  - Media: **0.8829 segundos**
  - Desviación típica: **0.06582**
- **Modelo No Cuantizado de 512**
  - Media: **1.6322 segundos**
  - Desviación típica: **0.1111**
- **Modelo Cuantizado en float16 de 512**
  - Media: **1.1996 segundos**
  - Desviación típica: **0.1074**

Como podemos observar, existe una relación entre el tiempo de ejecución y el tamaño de datos de entrada del modelo, lo cual es lógico, ya que mientras más datos tenga que procesar mayor tiempo de ejecución requerirá para obtener la respuesta.

Vemos que entre los modelos de 192 y 384 hay un aumento de más del 100 % ya que el modelo de 192 No Cuantizado tiene una media de **0.5597 segundos** y el modelo de 384 No Cuantizado tiene una media de **1.2063 segundos**. La misma relación aproximadamente para los mismos modelos pero Cuantizados.
Son resultados esperados, ya que al duplicar la seq_len se duplica aproximadamente el tiempo de ejecución.

Siguiendo esta misma línea, si comparamos los modelos de 384 con los de 512 vemos una relacción parecida, ya que en este caso hay un aumento de un 75 % ya que el modelo de 384 No Cuantizado tiene una media de **1.2063 segundos** y el modelo de 512 No Cuantizado tiene una media de **1.6322 segundos**. La misma relacción aproximadamente para modelos Cuantizados. Igual que los modelos de 192 y 384, son resultados esperados ya que al aumentar un 75 % el seq_len, aumenta un 75 % el tiempo de ejecución.

Con lo cuál podemos decir que:
La relación entre el tamaño de seq_ len de los modelos es directamente proporcional al tiempo de ejecución de los mismos.

Veamos ahora la relación entre modelos de mismo seq_len Cuantizados y No Cuantizados.

- **Modelos de 192:** Los modelos Cuantizados de 192 tienen una mejora de tiempo en relación a los No Cuantizados de un **25 %** ya que la media del modelo No Cuantizado es **0.5597 segundos** y la del Cuantizado es **0.4562 segundos**.
- **Modelos de 384:** Los modelos Cuantizados de 384 tienen una mejora de tiempo en relación a los No Cuantizados de un **26 %** ya que la media del modelo No Cuantizado es **1.2063 segundos** y la del Cuantizado es **0.8829 segundos**.
- **Modelos de 512:** Los modelos Cuantizados de 512 tienen una mejora de tiempo en relación a los No Cuantizados de un **26 %** ya que la media del modelo No Cuantizado es **1.6322 segundos** y la del Cuantizado es **1.1996 segundos**.

Vemos que el rendimiento aumenta un **25 %** aproximadamente usando modelos Cuantizados.

Por último, es relevante comentar que la desviación tipica de cada modelo en relación a su media es relativamente baja, lo que implica cierta consistencia en cada ejecución.

6.2. Patones

Los resultados de 100 ejecuciones de cada modelo en Patones es el siguiente:

- **Modelo No Cuantizado de 192**
  - Media: **0.2628 segundos**
  - Desviación típica: **0.0024**
- **Modelo Cuantizado en float16 de 192**
  - Media: **0.1841 segundos**
  - Desviación típica: **0.0001**
- **Modelo No Cuantizado de 384**
  - Media: **0.5546 segundos**
  - Desviación típica: **0.0004**
- **Modelo Cuantizado en float16 de 384**
  - Media: **0.3435 segundos**
  - Desviación típica: **0.0007**
- **Modelo No Cuantizado de 512**
  - Media: **0.7751 segundos**
  - Desviación típica: **0.0004**

- **Modelo Cuantizado en float16 de 512**
  - Media: **0.4761 segundos**
  - Desviación típica: **0.0006**

Para no repetir lo que hemos comentado en el apartado anterior, de los resultados en Patones podemos deducir que sigue existiendo una relación directamente proporcional entre en tiempo de ejecución y el tamaño de los datos de entrada, así como entre los modelos Cuantizados y los No Cuantizados.

Lo que sí es relevante comentar es la gran disminución del tiempo de ejecución que hay entre los tiempos de MiMáquina y los de Patones. **Las ejecuciones realizadas en Patones son más del doble de rápidas que las ejecuciones en MiMáquina**.

De hecho, la desviación típica de los tiempos de Patones es tan ínfima que deducimos que cada ejecución tarda prácticamente lo mismo que el resto, es decir, son muy consistentes.

Capítulo 7

Conclusiones

Vistos los resultados del capítulo 6, podemos afirmar lo siguiente:

- Los modelos con seq_len mas grande tardan más tiempo en dar respuesta a la pregunta, siguiendo una relación directamente proporcional, en la que al doblar la seq_len se dobla el tiempo de respuesta.
- Los modelos cuantizados con float16 tienen un rapidez de respuesta mayor que los modelos no cuantizados. Aproximadamente son un 25 % más rápidos.
- Los modelos ejecutados en el entorno virtual de Patones ofrecen un tiempo de ejecución mucho menor que los modelos ejecutados en el entorno virtual de MiMáquina. Concretamente ofrecen un tiempo de respuesta menos de la mitad que el que ofrecen en MiMáquina.

Si bien los modelos Cuantizados, No cuantizados, con diferente seq_len etc... Son modelos todos diferentes entre sí, es lógico pensar que cada uno ofrece un tiempo de respuetsa distinto. Pero en base a los resultados de las ejecuciones en llas dos máuinas diferentes, mismos modelos han dado tiempos de respuesta muy diferentes.

Con esto concluimos que el tiempo que se tarda en obtener una respuesta depende de:

- **El tamaño de seq_len**
- **La cuantización (en este caso en float16)**
- **La máquina donde se ejecute, ya que depende de las características concretas de la CPU**

El objetivo principial de este trabajo era el estudio y evaluación del rendimiento de los modelos, si bien es importante remarcar que estos modelos han sido entrenados con 2 épocas, muy pocas para el gran conjunto de datos con los que han sido entrenados.
Si ejecutasemos estos modelos en la aplicación frente a textos y preguntas comunes no darían respuesta o sería una respuesta inútil, ya que los modelos no han sido creados para ser usados sino para ser evaluados.

Aun con todo, se han probado modelos de muchas épocas diferentes y **se ha llegado a la conclusión de que el modelo entrenado con 29-31 épocas es el modelo que**

nos devuelve la respuesta más racional.

Este modelo no se encuentra estudiado ni evaluado como los otros porque, como se mencionó en el apartado del entrenamiento del modelo, las épocas no interfieren en el rendimiento del modelo sino en su precisión.

No obstante como un **posible trabajo futuro**, sería muy interesante encontrar el modelo perfecto, el cuál probablemente sería un modelo de entre 30 y 40 épocas, cuantizado en float16. El problema de este camino de trabajo es que el entrenamiento de un modelo así llevaría alrededor de 1 día entero. Por lo que requiere mucho tiempo para realizar pruebas con diferentes seq_len, y otros valores.
También puede ser interesante intentar la cuantización de otro tipo, como por ejemplo **int8**, la cuál por algún motivo que no logramos deducir, no es soportada por nuestros modelos, aunque en el manual de tensorflow ofrece dicha posibilidad.

Capítulo 8

Conclusions

Considering the results of chapter 6, we can affirm the following:

- Models with a larger seq _len take longer to answer the question, following a directly proportional relationship, in which doubling the seq _len doubles the response time.
- The models quantized with float16 have a faster response speed than the non-quantized models. They are approximately 25 % faster.
- Models executed in the virtual environment of Patones offer us a much shorter execution time than the models executed in the virtual environment of VirtualBox. Specifically, they offer a response time less than half that of VirtualBox.

Although the Quantized models, Unquantized models, with different seq _len etc ... are all different models, it is logical to think that each one offers a different response time. But based on the results of the runs in the 2 different environments, the same models have given very different response times.

So we can conclude that the time it takes to obtain an answer depends on:

- **The size of seq _len**
- **Quantization (in this case on float16)**
- **The environment where it runs**

The main objective of this work was the study and evaluation of the performance of the models, although it is important to note that these models have been trained with 2 epochs, very few for the large data set with which they have been trained..
If we run these models in the application against texts and common questions, they would not give an answer or it would be a useless answer, since the models have not been created to be used but to be evaluated.

Even so, models from many different epochs have been tested and **it has been concluded that the model trained with 29-31 epochs is the model that gives us the most rational answer**.

This model has not been studied or evaluated like the others because, as mentioned in the model training section, the epochs do not interfere with the performance of the model but rather with its precision.

However as a **possible future work**, it would be very interesting to find the perfect model, which would probably be a model between 30 and 40 epochs, quantized in float16. The problem with this work path is that training such a model would take about 1 full day. So it takes a long time to test with different seq_len, and other values

It may also be interesting to try quantization of another type, such as **int8**, which for some reason that we cannot deduce, is not supported by our models, although the tensorflow manual offers this possibility.

Bibliografía

[1] François Chollet, *DEEP LEARNING with Python*, MEAP Edition, Version 6, November 2017.

[2] Laurence Moroney, *AI and Machine Learning for Coders*, *O'REILLY*, October 2020.

[3] SAS, *SAS Institute inc, 2020*, Deep Learning Qué es y porqué es importante https://www.sas.com/es_es/insights/analytics/deep-learning.html

[4] BISMART, *Bismart, 2021*, ¿Cuál es la diferencia entre el machine learning y el deep learning? https://blog.bismart.com/es/diferencia-machine-learning-deep-learning

[5] Miguel Sotaquirá, *Codificandobits, 2021*, BERT: el inicio de una nueva era en el Natural Language Processing https://www.codificandobits.com/blog/bert-en-el-natural-language-processing

[6] *Google*, Respuesta a la pregunta BERT con TensorFlow Lite Model Maker https://www.tensorflow.org/lite/tutorials/model_maker_question_answer

[7] *huggingface*, MobileBERT *Documentación de MobileBERT* https://huggingface.co/transformers/model_doc/mobilebert.html

[8] *gemnde001*, MobileBERT *Github* https://github.com/gemde001/MobileBERT