industriales
etsii

Escuela Técnica
Superior
de Ingeniería
Industrial

UNIVERSIDAD POLITÉCNICA DE CARTAGENA

Escuela Técnica Superior de Ingeniería Industrial

Nuevas estrategias de navegación e interacción humano-robot para la introducción de robots móviles de bajo coste en entornos industriales

TRABAJO FIN DE GRADO

GRADO EN INGENIERÍA ELECTRÓNICA INDUSTRIAL Y AUTOMÁTICA

Autor: Adrián Lisón Martínez

Director: Dra. Nieves Pavón Pulido

Cartagena, 21/12/2023

Universidad
Politécnica
de Cartagena

AGRADECIMIENTOS

A mis padres por todo el apoyo que me han dado, tanto en este tramo final del grado, como durante los cuatro años que he estado cursándolo, sin su sacrificio no podría haber llegado a la situación en la que me encuentro ahora.

RESUMEN

Este trabajo fin de grado permitirá analizar las estrategias de navegación existentes para la construcción de planes globales y locales, de forma que un robot móvil de bajo coste pueda realizar tareas de ayuda y acompañamiento a operarios de entornos industriales. Para ello, se implementarán técnicas de mapeo que permitan definir el entorno donde se va a trabajar. La construcción del mapa métrico se realizará utilizando el robot real y escaneando cada rincón del área de trabajo a partir de algoritmos de creación de mapas. También, se implementará una prueba en un escenario real donde el robot tendrá que ayudar a los operarios con actividades de transporte de mercancías, usando técnicas para la interacción humano-robot, como la clasificación de rostros, el reconocimiento de gestos y el reconocimiento del lenguaje natural (PLN), el cual profundizaremos más debido a que abriremos una línea de investigación en la novedosa arquitectura Transformer.

Índice

Capítulo 1. Introducción. 10

1.1 Estado del arte. 10

1.1.1 Robótica móvil en entornos industriales 10

1.1.2 Navegación 12

1.1.3 Inteligencia Artificial (IA) 13

1.1.4 Reconocimiento y procesamiento del lenguaje natural 14

1.1.4.1 Reconocimiento y síntesis de voz 16

1.1.4.2 Técnicas clásicas de PLN 17

1.1.4.3 Redes neuronales artificiales 18

1.1.4.4 Arquitectura Transformer 19

1.1.5 Reconocimiento facial 22

1.1.6 Reconocimiento de gestos 25

1.2 Motivación y objetivos. 26

1.3 Estructura del documento 27

Capítulo 2. Diseño del sistema. 29

2.1 Diseño global del sistema. 29

2.2 Arquitectura hardware. 31

2.2.1 Robot Pioneer 3-DX. 31

2.2.2 Láser de escaneo Hokuyo UTM-30LX. 32

2.2.3 Cámara Orbecc Astra Pro. 34

2.2.4 Micrófono Hama Mic-P35 Allround. 37

2.3 Arquitectura software. 39

2.3.1 Descripción de los elementos principales de ROS. 39

2.3.2 Navegación del robot. 41

2.3.2.1 Stack Navigation de ROS. 42

2.3.2.2 Uso del simulador GAZEBO. 46

2.3.2.3 Navegación en el robot real. 47
2.3.3 Interacción natural con el robot.......... 49
2.3.3.1 Redes tipo Transformer para procesamiento de lenguaje natural. 49
2.3.3.2 Redes tipo Transformer para la clasificación de rostros. 61
2.3.3.3 Reconocimiento de gestos mediante MediaPipe.......... 65
2.3.3.4 Integración de componentes con la interfaz natural.......... 67
Capítulo 3. Análisis de resultados. 68
3.1 Pruebas realizadas.......... 68
3.1.1 Pruebas de la interfaz natural 68
3.1.1.1 Pruebas del procesamiento del lenguaje natural 68
3.1.1.2 Pruebas de reconocimiento visual 71
3.1.1.3 Pruebas de reconocimiento de gestos 73
3.1.2 Pruebas de navegación. 76
3.1.2.1 Pruebas de navegación en GAZEBO. 76
3.1.2.2 Pruebas de navegación en el robot real.......... 81
3.1.3 Integración de la interfaz natural con el sistema de navegación. 83
3.2 Resultados. 87
3.2.1 Resultados de la interacción humano-robot. 87
3.2.2 Resultados de las pruebas de navegación.......... 88
3.2.3 Resultados del proceso de integración. 88
3.3 Discusión. 88
Capítulo 4. Conclusiones y trabajo futuro. 90
4.1 Conclusiones.......... 90
4.2 Trabajo futuro. 91
Referencias. 94
Anexos. 97

Índice de figuras

Figura 1. Aspecto de robot Proteus. Robot utilizado por Amazon. 11

Figura 2. Estructura de control de navegación básica para un robot móvil. 12

Figura 3. Navegador implantado en el robot móvil Blanche de AT&T. 13

Figura 4. *Pipeline* genérico para tareas de PLN. 16

Figura 5. El modelo Transformer del paper original de *Attention is all you need*. 20

Figura 6. Progreso de los modelos Transformer hasta 2019. 21

Figura 7. Proceso simplificado de reconocimiento facial. 22

Figura 8. Alineación de caras usando 5 puntos clave de MTCNN. 23

Figura 9. Landmarks de la mano. 26

Figura 10. Diseño del robot. Parte delantera. 30

Figura 11. Diseño del robot. Parte trasera. 30

Figura 12. Pioneer 3-DX 32

Figura 13. Láser Hokuyo UTM-30LX. 34

Figura 14. Cámara Orbecc Astra Pro. 36

Figura 15. Micrófono Hama Mic-P35 Allround. 38

Figura 16. El interfaz de ROS para tele operar al robot. 43

Figura 17. Esquema de los *topics* y *nodos* del sistema. 45

Figura 18. Mi entorno simulado en Gazebo 47

Figura 19. Muestra del dataset de frases para el procesamiento del lenguaje natural. 50

Figura 20. Captura del resultado de la prueba de procesamiento del lenguaje natural (1). 70

Figura 21. Captura del resultado de la prueba de procesamiento del lenguaje natural (2). 70

Figura 22. Captura del resultado de la prueba de procesamiento del lenguaje natural (3). 70

Figura 23. Captura del resultado de la prueba de procesamiento del lenguaje natural (4). 70

Figura 24. Foto1.jpg para el reconocimiento facial. 71

Figura 25. Captura del resultado de la prueba de reconocimiento facial con la foto1.jpg. 71

Figura 26. Foto3.jpg para el reconocimiento facial. 72

Figura 27. Captura del resultado de la prueba de reconocimiento facial con la foto3.jpg. 72

Figura 28. Foto2.jpg para el reconocimiento facial. 72

Figura 29. Captura del resultado de la prueba de reconocimiento facial con la foto2.jpg. 73

Figura 30. Landmarks en una imagen de video real. 73

Figura 31. Mano en posición para el gesto parar. 74

Figura 32. Captura de la salida del código de reconocimiento de gestos. 74

Figura 33. Mano en posición para el gesto afirmativo. 75

Figura 34. Captura de la salida del código de reconocimiento de gestos (2). 75

Figura 35. Configuración *rviz* para crear el mapa métrico. 76

Figura 36. Mapa totalmente escaneado. 77

Figura 37. Archivos *launch* necesarios para el paquete *amcl*. 77

Figura 38. Configuración *rviz* para el paquete *amcl*. 78

Figura 39. Localización del robot al iniciar el paquete *amcl*. 79

Figura 40. Localización del robot al analizar el entorno. 79

Figura 41. Archivos de configuración necesarios para el paquete *move_base*. 80

Figura 42. Archivos *launch* necesarios para el paquete *move_base*. 80

Figura 43. Configuración de *rviz* para el paquete *move_base*. 80

Figura 44. Mapeado del entorno real. 81

Figura 45. Nube de partículas al iniciar el paquete de navegación en el robot real. 82

Figura 46. Localización del robot real una vez estudiado el entorno. 82

Figura 47. Prueba de la integración de los comandos de voz en el PLN 85

Figura 48. Herramienta RViz una vez publicado el topic */goal*. 86

Capítulo 1. Introducción.

En este primer capítulo, se presenta una descripción general del trabajo y establece el escenario para el resto del documento. Se tienen en cuenta varios aspectos clave, incluido el estado del arte, la motivación que llevó a la investigación, el objetivo a alcanzar y la estructura del documento.

1.1 Estado del arte.

En el ámbito de la robótica y la inteligencia artificial, la navegación autónoma y la colaboración entre humanos y robots han emergido como áreas de investigación de vital importancia. A continuación, se examinan críticamente las estrategias de navegación existentes y se analizan los avances recientes en la interacción entre humanos y robots en entornos industriales, más específicamente las técnicas de reconocimiento del lenguaje natural, facial y de gestos.

1.1.1 Robótica móvil en entornos industriales

La robótica móvil ha experimentado rápidos avances en el entorno industrial, cambiando la forma en que las empresas gestionan la logística y la producción. Los robots móviles autónomos se han convertido en aliados clave para optimizar procesos y mejorar la eficiencia en fábricas y almacenes. Estos robots pueden realizar tareas de envío, recepción y entrega automáticamente, liberando a los trabajadores de tareas repetitivas y permitiéndoles concentrarse en actividades creativas y estratégicas.

Un ejemplo destacado es el uso de robots móviles en almacenes de comercio electrónico. Empresas como Amazon han implementado robots autónomos que navegan por los pasillos de los almacenes para recoger y transportar productos (Clark, 2022).

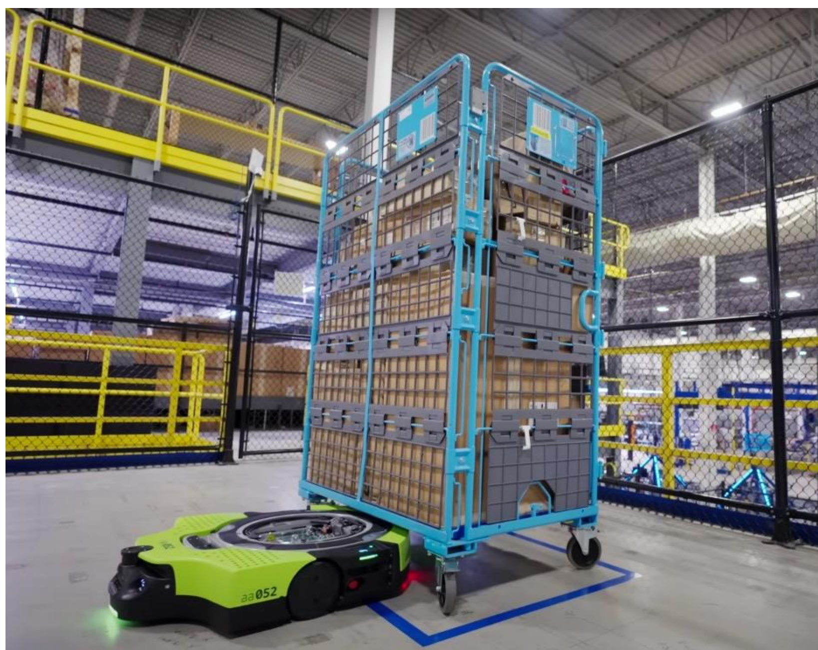

Figura 1. Aspecto de robot Proteus. Robot utilizado por Amazon.

Amazon ha presentado un robot autónomo para carga y descarga de carros de paquetería. El robot es capaz de navegar de forma segura e independiente por los almacenes de la compañía, incluso esquivando a los empleados humanos. La tecnología de seguridad, percepción y navegación del robot se basa en un haz de luz frontal que detecta elementos como trabajadores. Si un humano entra en el haz, el robot detiene su navegación y no reanuda su movimiento hasta que el operario se aleja.

Estos robots optimizan las rutas, reduciendo el tiempo necesario para completar los pedidos y mejorando la precisión en la gestión de inventario. Además, en la industria automotriz, los robots móviles pueden transportar piezas y componentes a lo largo de la cadena de producción, mejorando la eficiencia y disminuyendo el tiempo de fabricación.

En cuanto a la colaboración entre robots y operarios, los avances en reconocimiento del lenguaje natural, reconocimiento facial y de gestos están permitiendo una interacción más intuitiva y fluida. Imagine a un operario que necesita ayuda para mover una carga pesada. Mediante el reconocimiento del lenguaje natural, el operario puede dar instrucciones al robot de manera verbal, indicándole qué hacer y dónde llevar la carga. El robot, equipado con capacidades de reconocimiento facial y de gestos, puede interpretar las expresiones faciales y los gestos del

operario para comprender mejor sus necesidades y adaptar su comportamiento en consecuencia.

1.1.2 Navegación

Si nos adentramos más en el tema de la navegación de los robots, la estructura de control de navegación básica para un robot móvil es la que se muestra en la Figura 2 (Villarreal Onofre, 2021):

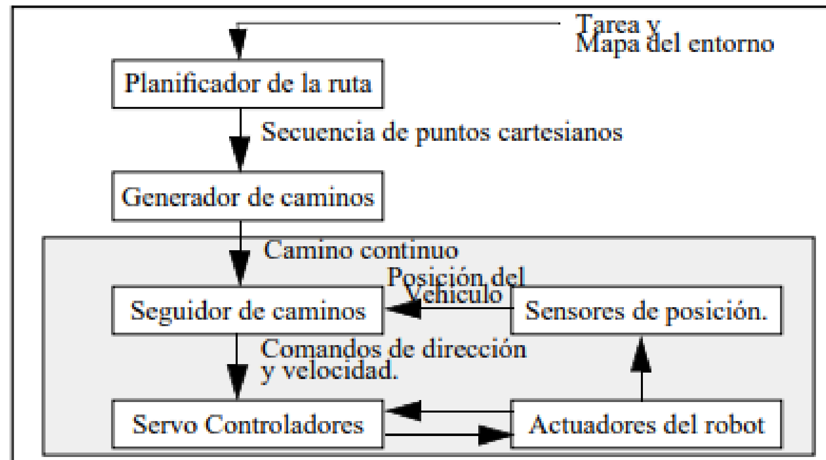

Figura 2. Estructura de control de navegación básica para un robot móvil.

La complejidad del sistema requerida para completar esta tarea depende principalmente del conocimiento del entorno de trabajo. En la Figura 2, se cree que existe un mapa del entorno que se corresponde exactamente con la realidad. Con un uso adecuado, se puede construir una carretera adecuada que cumpla los requerimientos de la misión de navegación, sin que el vehículo colisione con ningún factor ambiental. Sin embargo, los modelos de entorno que están disponibles para los robots suelen tener algunos defectos, omitiendo algunos de sus detalles. Por lo que el esquema de la Figura 2 no es del todo efectivo al no tener garantizada la construcción de carreteras.

En aplicaciones donde la información sobre el entorno de trabajo puede variar desde un conocimiento completo hasta un cierto grado de incertidumbre, se utiliza un esquema de navegación como el mostrado en la Figura 3. Este esquema fue implantado en el robot móvil Blanche de los laboratorios AT&T (Cox I. J., 1988).

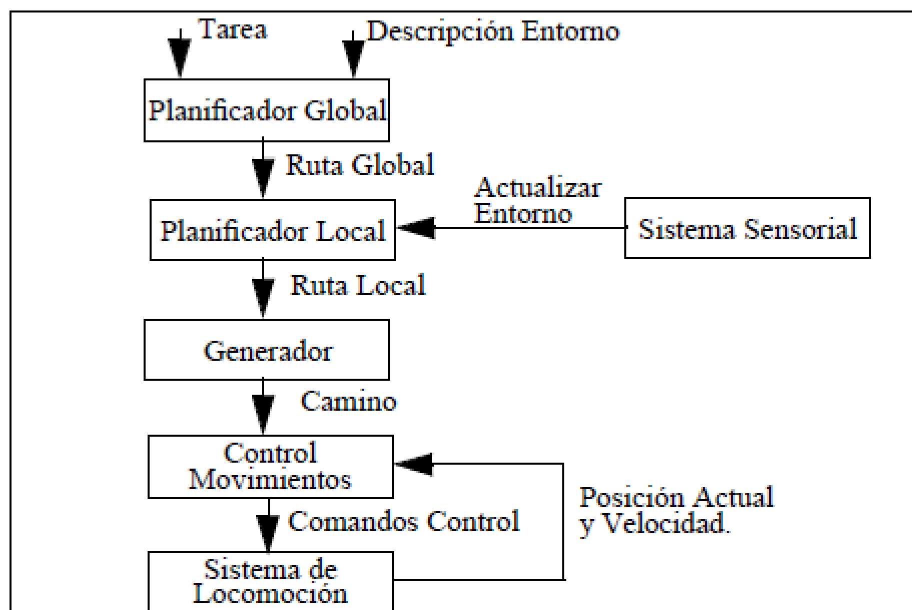

Figura 3. Navegador implantado en el robot móvil Blanche de AT&T.

La clave de este esquema para adaptarse a diversos entornos, incluso con un conocimiento incompleto del mismo, es la distinción entre planificación global y local.

1.1.3 Inteligencia Artificial (IA)

La inteligencia artificial (IA) en robots móviles se refiere a la integración de sistemas de IA en robots diseñados para moverse y operar en entornos diversos. Estos robots utilizan algoritmos y métodos de IA para planificar rutas, evitar obstáculos, interactuar con su entorno y realizar tareas específicas. La robótica de servicios, la exploración espacial, la fabricación y la logística son algunas áreas de aplicación importantes.

La IA en robots móviles se basa en diversas tecnologías:

Aprendizaje automático (machine learning): Los robots pueden aprender de la información que recopilan mientras operan, mejorando sus habilidades y permitiéndoles adaptarse a nuevas circunstancias. Esto incluye el aprendizaje por refuerzo, así como el aprendizaje supervisado y no supervisado.

Visión artificial: Los robots móviles equipados con cámaras y sensores pueden analizar y procesar datos visuales utilizando visión artificial. Son capaces de identificar cosas en su entorno, incluidas personas, obstáculos y patrones.

Sensores y percepción: Con la ayuda de la inteligencia artificial (IA), los robots pueden interpretar datos de sensores como el sensor láser LIDAR, ultrasonidos y codificadores para comprender mejor su posición, movimiento y entorno.

Planificación de movimiento: Los robots pueden elegir el mejor camino hacia un destino evitando obstáculos y teniendo en cuenta las limitaciones físicas con la ayuda de algoritmos de planificación.

Navegación autónoma: La IA permite que los robots móviles se muevan de forma autónoma en entornos desconocidos, utilizando datos de sensores para evitar colisiones y tomar decisiones de navegación en tiempo real.

Interacción humano-robot: La IA permite que los robots móviles comprendan y reaccionen a comandos verbales, gestos y otras comunicaciones humanas, lo cual es esencial en aplicaciones de servicio al cliente, asistencia personal y colaboración en entornos de trabajo.

Robótica colaborativa: La IA permite que los robots móviles trabajen de forma segura junto a las personas, colaborando en tareas desafiantes y compartiendo conocimientos.

Optimización y planificación de tareas: Los robots móviles pueden usar algoritmos de optimización para asignar tareas, planificar rutas eficientes y aumentar la eficiencia de las tareas.

1.1.4 Reconocimiento y procesamiento del lenguaje natural

El procesamiento de lenguaje natural (NLP) es una rama de la inteligencia artificial que se encarga de entender y generar lenguaje humano, que los ordenadores puedan procesar (Instituto de ingeniería del conocimiento, s.f.). Normalmente, parte de dividirlo en elementos (frases, palabras, etc.) e intentar entender las relaciones entre ellos. Es una disciplina compleja que abarca una amplia gama de tareas, desde la traducción automática hasta la comprensión del lenguaje natural.

Las funciones del NLP se pueden dividir en tres categorías principales:

- <u>Preprocesamiento</u>: Estas tareas se centran en preparar el texto para su análisis. Incluyen la segmentación de palabras, la segmentación de oraciones, la lematización o stemming, y la segmentación morfológica.

- Análisis: Estas tareas se centran en comprender la estructura y el significado del lenguaje. Incluyen el análisis sintáctico, el etiquetado gramatical (POS) y la extracción de información.
- Generación: Estas tareas se centran en generar texto nuevo. Incluyen la traducción automática, la respuesta a preguntas y la generación de formatos de texto creativos.

Segmentación de palabras

La segmentación de palabras es la tarea de dividir el texto en palabras individuales. Es una tarea necesaria para muchas otras tareas de NLP, como el análisis sintáctico y el etiquetado gramatical.

Segmentación de oraciones

La segmentación de oraciones es la tarea de dividir el texto en oraciones individuales. Es una tarea necesaria para muchas otras tareas de NLP, como la comprensión del lenguaje natural y la traducción automática.

Lematización o stemming

La lematización o stemming es la tarea de reducir las palabras a su forma base. Por ejemplo, el lema de la palabra "caminando" es "caminar", y el lema de la palabra "gatos" es "gato".

Segmentación morfológica

La segmentación morfológica es la tarea de dividir las palabras en sus partes constituyentes, como los lexemas, los morfemas derivativos y los morfemas flexivos. Por ejemplo, la palabra "caminando" se puede dividir en el lexema "caminar" y los morfemas derivativos "-ando".

Análisis sintáctico

El análisis sintáctico es la tarea de determinar la estructura sintáctica de una oración. Es una tarea necesaria para muchas otras tareas de NLP, como la comprensión del lenguaje natural y la traducción automática.

Etiquetado gramatical (POS)

El etiquetado gramatical (POS) es la tarea de asignar una categoría gramatical a cada palabra de una oración. Las categorías gramaticales más comunes son los sustantivos, los verbos, los adjetivos, los adverbios, las preposiciones, las conjunciones y los artículos.

El esquema de la Figura 4 resume de manera gráfica lo que podría ser un *pipeline* genérico para el desarrollo de sistemas de PLN.

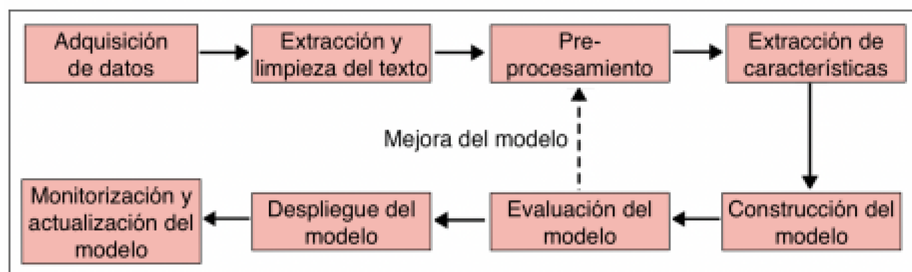

Figura 4. *Pipeline* genérico para tareas de PLN.

1.1.4.1 Reconocimiento y síntesis de voz

El reconocimiento y síntesis de voz son dos áreas de investigación en el campo del procesamiento del lenguaje natural (NLP) que se centran en la interacción entre los ordenadores y los humanos a través del lenguaje hablado.

- **Reconocimiento de voz:** Es la tarea de convertir el habla humana en texto. Se utiliza en una amplia gama de aplicaciones, como la transcripción automática de audio, la interacción con asistentes virtuales y el control de dispositivos electrónicos.
  Los sistemas de reconocimiento de voz suelen utilizar un modelo estadístico para identificar las palabras habladas. Este modelo se entrena con un conjunto de datos de audio etiquetado con las palabras correspondientes.
  Los principales desafíos del reconocimiento de voz son la variabilidad del habla humana, el ruido ambiental y la pronunciación incorrecta.
- **Síntesis de voz:** Es la tarea de generar habla humana a partir de texto. Se utiliza en una amplia gama de aplicaciones, como la creación de audiolibros, la generación de voz para personajes de videojuegos y la comunicación con personas con discapacidades auditivas. Los sistemas de síntesis de voz suelen utilizar un modelo de voz para generar

los sonidos del habla. Este modelo se crea a partir de un conjunto de datos de audio de voz humana. Los principales desafíos de la síntesis de voz son la calidad del sonido, la naturalidad de la pronunciación y la eficiencia del proceso de síntesis.

- **Aplicaciones del reconocimiento y síntesis de voz:** El reconocimiento y síntesis de voz tienen una amplia gama de aplicaciones potenciales, entre las que se incluyen:

  - Transcripción automática de audio: El reconocimiento de voz se puede utilizar para transcribir automáticamente audio a texto. Esta aplicación es útil para una amplia gama de tareas, como la creación de transcripciones de reuniones, la creación de subtítulos para vídeos y la traducción de audio.
  - Interacción con asistentes virtuales: El reconocimiento de voz se utiliza para permitir a los usuarios interactuar con asistentes virtuales a través del habla. Esta aplicación es útil para tareas como la realización de llamadas, la configuración de alarmas y la búsqueda de información.
  - Control de dispositivos electrónicos: El reconocimiento de voz se puede utilizar para controlar dispositivos electrónicos a través del habla. Esta aplicación es útil para tareas como el control de la reproducción de música, el control de la temperatura y el encendido y apagado de dispositivos.

1.1.4.2 Técnicas clásicas de PLN

Algunas técnicas clásicas de PLN utilizan modelos de machine learning (ML) (Navarro Castillo, 2022), el aprendizaje automático (del inglés, *Machine learning*) es una rama de la inteligencia artificial que permite a los sistemas aprender sin ser programados explícitamente. Los sistemas de aprendizaje automático utilizan modelos matemáticos para aprender de datos existentes. Estos modelos pueden utilizarse para realizar tareas como la predicción, la clasificación y la toma de decisiones. La fase de aprendizaje del modelo, también conocida como fase de entrenamiento, es el proceso mediante el cual el modelo analiza un gran número de datos para identificar patrones.

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

- Aprendizaje supervisado
- Aprendizaje no supervisado
- Aprendizaje por refuerzo

Aunque los tres tipos de algoritmos de *machine learning* (ML) se han utilizado en el procesamiento del lenguaje natural (PLN), los más populares son los basados en el aprendizaje supervisado. En este artículo, hablaremos del algoritmo de Naive-Bayes y de las máquinas de vectores de soporte (SVM).

- **El algoritmo de Naive-Bayes**: Basado en el teorema de Bayes, expresa la probabilidad de que ocurra un evento A, dado que ya ocurrió un evento B. El algoritmo se utiliza para clasificar datos, calculando la probabilidad de que cada dato pertenezca a una clase. Para ello, se entrena con un conjunto de datos etiquetados, de modo que el modelo aprende a clasificar nuevos datos. Matemáticamente, el teorema de Bayes se expresa mediante la fórmula:

$$P(A/B) = \frac{P(B/A) * P(A)}{P(B)}$$

- **Support Vector Machine (SVM):** Son un conjunto de algoritmos de aprendizaje supervisado que se utilizan para resolver problemas de clasificación y regresión. El objetivo de un SVM es encontrar un hiperplano que separe los datos de dos clases de manera óptima. Para ello, el algoritmo identifica los puntos de datos que están más cerca de la frontera entre las dos clases, llamados vectores de soporte. El hiperplano se encuentra lo más lejos posible de los vectores de soporte de cada clase.

1.1.4.3 Redes neuronales artificiales

Las redes neuronales artificiales son una técnica de procesamiento del lenguaje natural que imita el funcionamiento del cerebro humano. Se trata de sistemas adaptables que aprenden y mejoran continuamente a partir de datos, lo que les permite modelar relaciones y patrones entre datos, ofrecer soluciones a problemas complejos y simular el proceso del lenguaje natural.

Una red neuronal consta de neuronas o nodos interconectados y organizados por capas. La capa de entrada recibe los datos, una o más capas ocultas procesan los datos y la capa de salida genera el resultado.

Para emular el cerebro humano, una red neuronal funciona mediante el examen de los datos que recibe en su capa de entrada. La capa de entrada envía los datos a la siguiente capa, que los procesa y los envía a la siguiente, y así sucesivamente. En cada capa, las neuronas o nodos detectan y filtran patrones de gran relevancia y combinan los datos.

Cada valor de entrada se le asigna un peso que modifica el peso de entrada. Estos valores resultantes se suman y se definen mediante una función logística o sigmoide.

El sistema se basa en la premisa de que en cada conjunto de parámetros existe una manera de combinarlos para predecir un determinado resultado. La red neuronal se encarga de encontrar y aplicar la mejor combinación posible de parámetros para cierto problema.

En los modelos de lenguaje natural PLN, las redes neuronales actúan en sus primeras fases transformando las palabras del vocabulario en vectores. Para esto, toman como principio básico que, en un texto, el significado de una determinada palabra se encuentra asociado con las palabras que se encuentran a su alrededor. Estos vectores creados son empleados en operaciones sencillas para ofrecer resultados razonables a nivel semántico.

1.1.4.4 Arquitectura Transformer

Hasta el momento, hemos repasado algunas técnicas clásicas de procesamiento del lenguaje natural (PLN), así como las redes neuronales. En este capítulo, profundizaremos en la arquitectura *Transformer*, uno de los modelos más avanzados de *machine learning*, en concreto de *deep learning*.

En 2017, Google presentó el Transformer en el paper - "Attention is All You Need"-, una arquitectura de redes neuronales que revolucionó el procesamiento del lenguaje natural (PLN). El Transformer sustituyó las capas recurrentes, como las LSTM, por capas de atención (Josende, 2022).

Las capas de atención codifican cada palabra de una frase en función del resto de la secuencia, lo que permite capturar el contexto del texto. Por este motivo, los modelos basados en *Transformer* también se denominan *Embeddings Contextuales*.

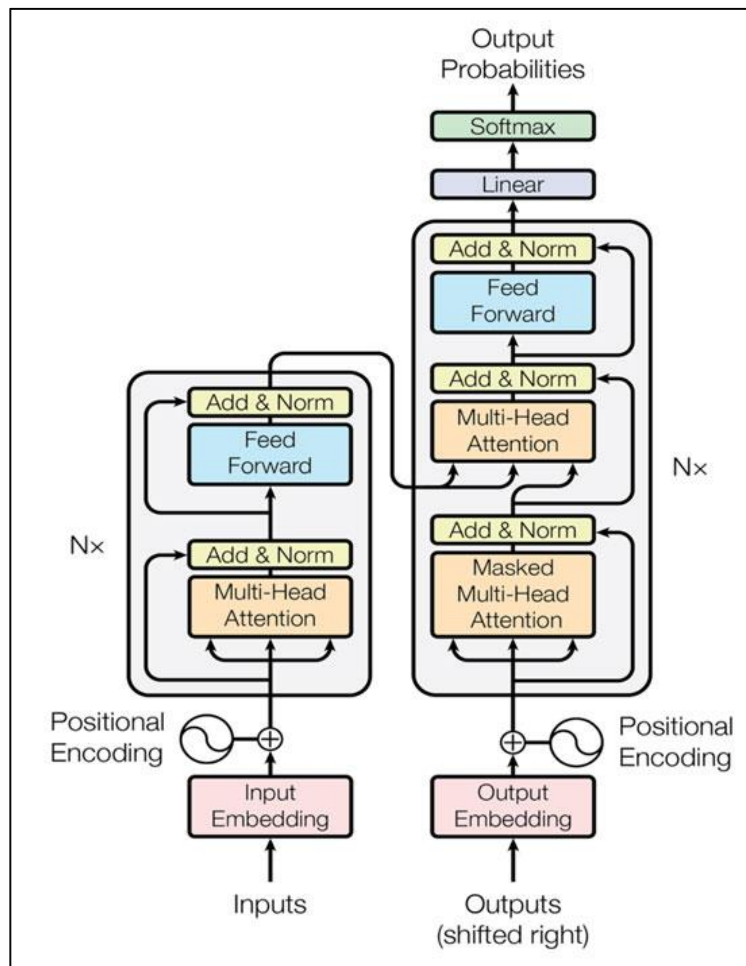

Figura 5. El modelo Transformer del paper original de *Attention is all you need.*

La arquitectura Transformer también incluye otras innovaciones, como los embeddings posicionales. Estos embeddings permiten al algoritmo conocer la posición relativa de cada palabra del texto. Esto es importante para tareas como la traducción, donde la posición de las palabras puede tener un significado gramatical o semántico. En el paper original del Transformer, se examinó la aplicación de esta arquitectura a tareas de traducción. Los resultados mostraron que el Transformer era mucho más efectivo que los métodos anteriores en esta tarea. El gráfico inferior muestra la mejora del rendimiento del Transformer en comparación con los métodos anteriores.

Y ¿cuál es el funcionamiento de los Transformer? Con Transformer, se trabaja en dos fases:

- **Pre-training**. En esta fase, el modelo aprende a comprender el lenguaje en general, tanto su estructura como el significado de las palabras. Esto se logra de

forma similar a los exámenes de idiomas, en los que el modelo debe completar frases con las palabras que faltan.

- **Fine-tuning**. Una vez pre-entrenado, el modelo se puede adaptar a tareas específicas añadiendo nuevas capas a su arquitectura y re-entrenando el modelo en esas tareas.

Y, ¿se tiene expectativas por estos modelos Transformers? La verdad es que sí, el futuro de los Transformers es muy prometedor (ver Figura 6). Los modelos de lenguaje se están volviendo cada vez más grandes y complejos, ya que son entrenados con conjuntos de datos cada vez más grandes. Esta carrera por obtener el mejor modelo de lenguaje ha llevado a una colaboración entre los grandes jugadores del mercado, que liberan sus modelos de forma gratuita. Esto ha permitido a la comunidad científica seguir desarrollando estos modelos y mejorar su rendimiento.

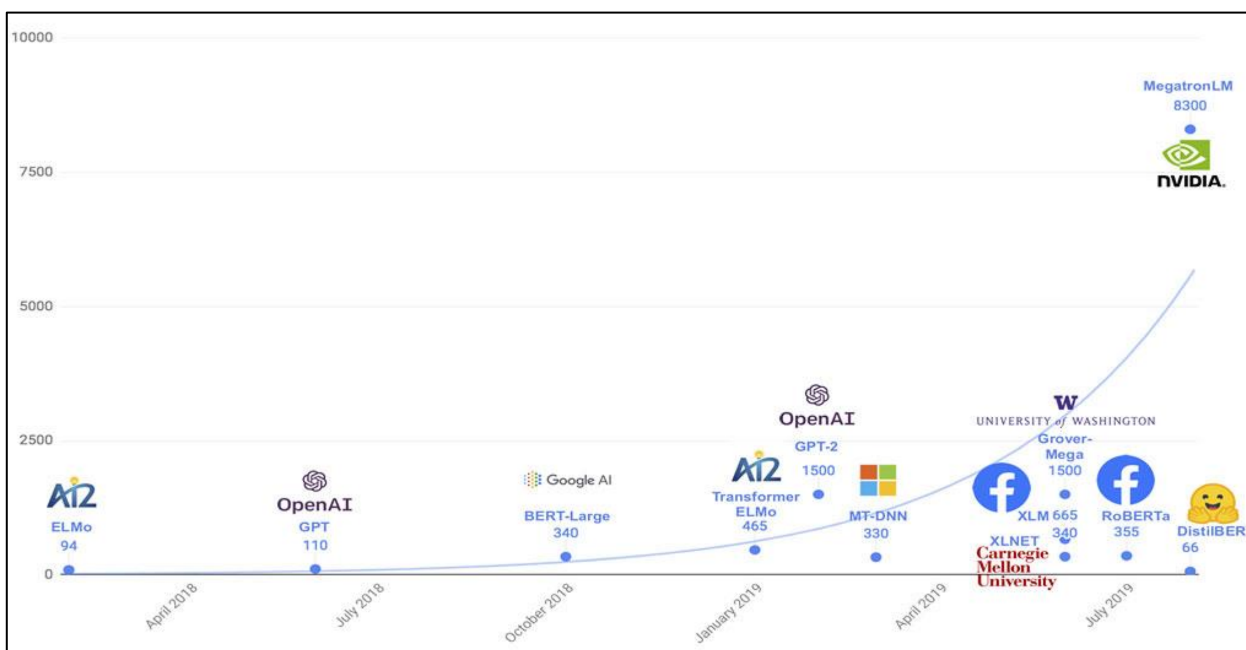

Figura 6. Progreso de los modelos Transformer hasta 2019.

1.1.5 Reconocimiento facial

El reconocimiento facial ha sido uno de los temas de investigación más destacados en el campo de la de la visión por ordenador desde los años noventa. Desde entonces, el campo ha evolucionado drásticamente, al aumentar la potencia de cálculo disponible y desarrollarse nuevos métodos desarrollados. En especial, la llegada del Deep Learning y las Convolutional Neural Networks convolucionales (CNN) han puesto los métodos a la altura del rendimiento humano en condiciones favorables.

Las tareas de reconocimiento facial pueden clasificarse en verificación e identificación facial. Un sistema de verificación facial tiene por misión confirmar si el sujeto en cuestión es la persona que proclama ser, es decir, una comparación uno a uno, mientras que un sistema de identificación facial debe determinar la identidad de la persona a partir de un conjunto de identidades conocidas, es decir, una comparación uno a muchos. No obstante, el enfoque de ambas categorías comparte un de procesamiento similar, aunque se fomentan diferentes compensaciones dependiendo la situación.

La detección de caras es el primer paso en el proceso de reconocimiento facial, que recibe como entrada una imagen sin procesar directamente de la cámara. El objetivo de este componente es localizar todas las caras dentro de una imagen, devolviendo la región que las contiene en forma de un cuadro delimitador (ver Figura 7).

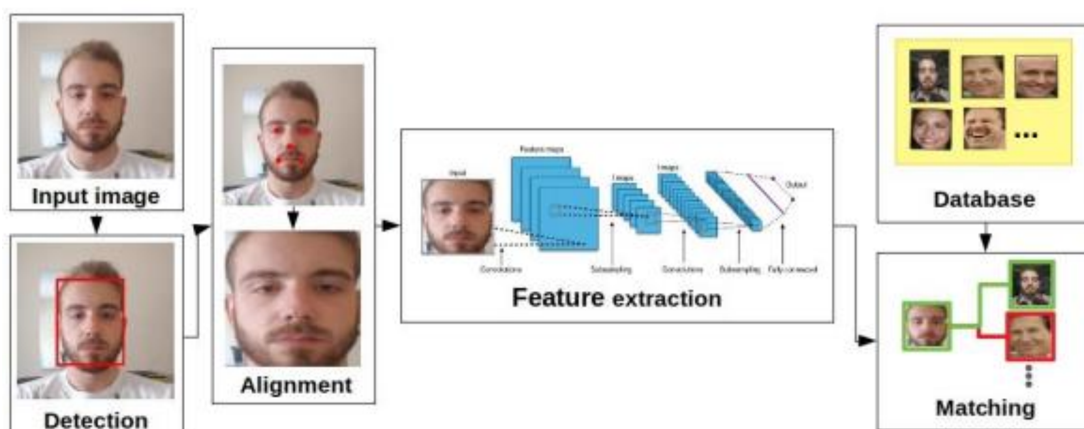

Figura 7. Proceso simplificado de reconocimiento facial.

Al igual que otras tareas relacionadas con imágenes y rostros, la detección de rostros se enfrenta a múltiples dificultades que desaconsejan el uso de enfoques tradicionales en aplicaciones del mundo real, como las distintas resoluciones de imagen, condiciones de iluminación, orientaciones de las caras, expresiones faciales, oclusiones, etc. Por eso, un método que se usa mucho para mejorar la robustez contra las variaciones de pose es la alineación facial, que se considera el mayor reto al que se enfrentan estos sistemas. Los métodos habituales de alineación facial se basan en la detección de puntos clave faciales: conjuntos de puntos de referencia situados en partes de la cara (por ejemplo, los ojos, la nariz la boca, etc.). La ubicación de estos puntos clave proporciona información sobre la información sobre la pose de la cara, que se utiliza para reducir la variabilidad de las poses que la red de la red de extracción de características.

En el caso de la alineación 2D, el proceso es bastante sencillo una vez que se han calculado los puntos clave faciales. En concreto, se utilizan transformaciones afines para normalizar la posición de estos puntos clave concretos. El modelo de transformación afín puede describir rotaciones, traslaciones, escalados y cizallamientos para convertir los puntos de una imagen desde su posición original a la deseada. Un ejemplo de estas transformaciones se muestra en la Figura 8.

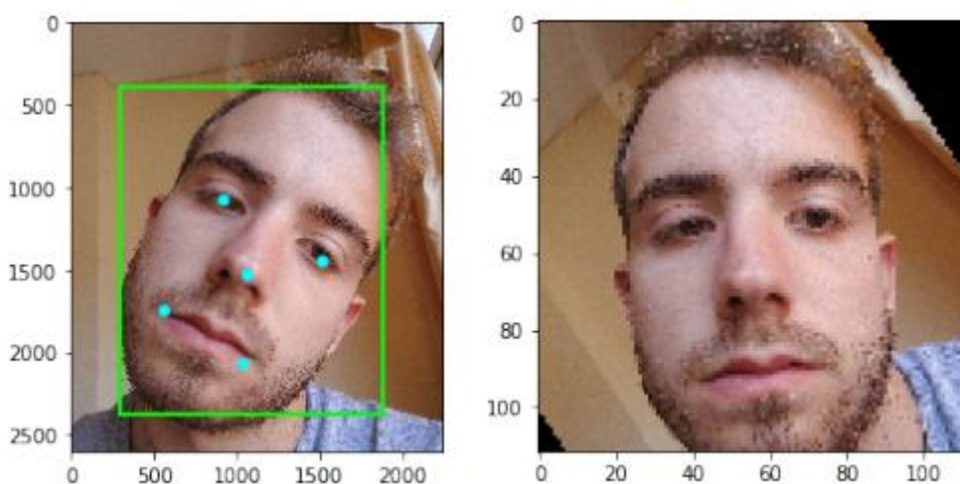

Figura 8. Alineación de caras usando 5 puntos clave de MTCNN.

MTCNN (de las siglas en inglés, *Multi-task Cascaded Convolutional Neural Networks*) (Eirea & Peña, 2022) es un marco multitarea profundo en cascada capaz de lograr la detección y alineación de caras en tiempo real explotando las similitudes entre las dos tareas. Utiliza una canalización CNN de 3 etapas: la primera etapa (P-Net) genera candidatos a cuadro delimitador,

la segunda etapa (R-Net) refina esos candidatos y la última etapa (O-Net) produce los cuadros delimitadores finales y los puntos de referencia faciales.

La red devuelve un recuadro delimitador rectangular y una lista de 5 puntos de referencia faciales por rostro. Estos puntos de referencia se corresponden con el ojo izquierdo, el ojo derecho, la nariz, la comisura izquierda de la boca y la comisura derecha de la boca (Figura 8, la primera imagen). Este conjunto de puntos es suficiente para alineación y mitigar los efectos perjudiciales de la variación de la pose.

En cambio, en este proyecto utilizaré otro tipo de estructura para clasificar rostros. Como ya he comentado, las arquitecturas de red neuronales convolucionales (CNN) han sido el estándar de facto para el reconocimiento de rostros durante muchos años. Sin embargo, las arquitecturas Transformer están ganando popularidad debido a su capacidad para aprender relaciones a largo alcance entre los píxeles de una imagen.

Este tipo de estructura, ya la he nombrado en el capítulo anterior, pero solamente enfocada al procesamiento del lenguaje natural. Es un tipo de red neuronal que se basa en la atención. La atención es una técnica que permite a la red neuronal centrarse en las partes más relevantes de una entrada. En el caso del reconocimiento de rostros, la atención permite a la red neuronal aprender las relaciones entre los diferentes rasgos faciales.

Las arquitecturas Transformer se han demostrado que son efectivas para el reconocimiento de rostros en una variedad de conjuntos de datos. En particular, las arquitecturas Transformer han demostrado ser superiores a las CNN en tareas de reconocimiento de rostros en condiciones desafiantes, como imágenes de baja resolución o caras parcialmente ocultas (Guanipa, 2022).

Una arquitectura Transformer para el reconocimiento de rostros consta de dos componentes principales: una capa de atención y una capa de transformación.

- Capa de atención: Se encarga de aprender las relaciones entre los diferentes rasgos faciales. La capa de atención funciona de manera similar a un algoritmo de búsqueda. La red neuronal primero genera una representación de cada rasgo facial. Luego, la capa de atención utiliza estas representaciones para calcular una puntuación de atención para cada par de rasgos faciales. Las puntuaciones de atención se utilizan para determinar qué rasgos faciales son más relevantes para la tarea de reconocimiento de rostros.

- Capa de transformación: Se encarga de combinar las representaciones de los diferentes rasgos faciales para generar una representación global del rostro. La capa de transformación funciona de manera similar a una red neuronal convolucional. La red neuronal primero aplica una serie de filtros a las representaciones de los rasgos faciales. Luego, la red neuronal combina las representaciones filtradas para generar una representación global del rostro.

1.1.6 Reconocimiento de gestos

Al hablar de reconocimiento gestual o de gestos, lo normal es pensar en gestos manuales, pero hoy en día los dispositivos nos permiten la transmisión de información casi con cualquier parte del cuerpo, o de sólo una parte de este, bien pudiera ser la cabeza, los ojos, las piernas, los brazos, las manos o incluso sólo los dedos. Según estudios, existen setecientos mil mensajes no verbales que el ser humano es capaz de hacer, de los cuales quinientos mil son gestos faciales y quinientos manuales (Romera, 2015). Nosotros en esta sección profundizaremos en los gestos con las manos y más concretamente en una solución de *Mediapipe* que es la que utilizaremos (Gesture recognition, s.f.).

MediaPipe es una herramienta de Google que facilita la implementación de soluciones de machine learning en diferentes plataformas, como dispositivos móviles, web, escritorio e IoT.
MediaPipe Hands utiliza dos modelos: un modelo de detección de la palma de la mano y un modelo de referencia de la mano. El modelo de detección de la palma de la mano encuentra la posición y el tamaño de la mano en la imagen. El modelo de referencia de la mano identifica 21 puntos clave de la mano en 3D, dentro de la región de la mano detectada.

El proceso de detección de los landmarks de la mano, Figura 9,se realiza en dos pasos:

1. Detección de la palma de la mano: El modelo de detección de la palma de la mano encuentra la posición y el tamaño de la mano en la imagen.
2. Referencia de la mano: El modelo de referencia de la mano identifica 21 puntos clave de la mano en 3D, dentro de la región de la mano detectada.

Para la referencia de la mano, el modelo utiliza regresión para predecir las coordenadas de los puntos clave en 3D. El modelo también aprende una representación interna consistente de la postura de la mano, lo que le permite ser robusto incluso con las manos parcialmente visibles.

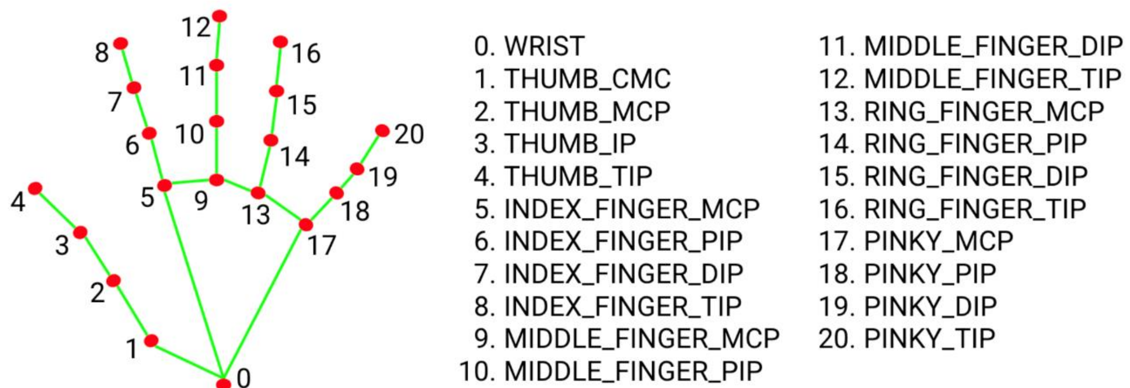

Figura 9. Landmarks de la mano.

1.2 Motivación y objetivos.

El objetivo central de este Trabajo de Fin de Grado (TFG) es desarrollar un sistema de navegación de robots que integre tecnologías híbridas de generación de mapas y reconocimiento de lenguaje natural, con el objetivo de permitir que el robot reciba comandos y orientación a través de la interacción verbal con el usuario.

Mi motivación nace porque este proyecto representa una oportunidad para avanzar en el campo de la robótica, y además tiene un impacto significativo en nuestra vida cotidiana y en la forma en que interactuamos con la tecnología. Esto abriría un mundo de posibilidades en términos de asistencia personal, automatización de tareas y colaboración hombre-máquina más fluida.

Para lograr el objetivo principal de diseñar un robot real autónomo inteligente que pueda comunicarse de forma natural con las personas, es necesario alcanzar los siguientes objetivos secundarios:

- Montaje e integración de todos los componentes (sensores y actuadores), de forma que estén sincronizados.
- Implementar y/o configurar el conjunto de módulos software necesarios para que se lleve a cabo el proceso de navegación autónoma. El robot debe de ser capaz de generar una trayectoria para alcanzar un destino y ser capaz de evitar obstáculos.

- Desarrollar un módulo que permita comunicarse de forma sencilla con el robot y que éste sepa interpretarlo.
- Implementar el modelo de detección de la palma de la mano *Mediapipe Hands.*
- Implementar el algoritmo de *TensorFlow* basado en una arquitectura Transformer para el reconocimiento facial de los usuarios.
- Abrir línea de investigación sobre la arquitectura *Transformers* cuyo objetivo sea obtener un modelo de lenguaje el cual sea más intuitivo.
- Pruebas y puesta en marcha del robot para comprobar su correcto funcionamiento.

1.3 Estructura del documento.

Este Trabajo Fin de Grado constará de los siguientes 4 capítulos:

- **Capítulo 1. Introducción.** Este capítulo actual, ofrece una panorámica introductoria del trabajo que sienta las bases para el desarrollo posterior del TFG. Se abordan diversos elementos fundamentales, englobando tanto el estado del arte, como la motivación que ha dado pie a la investigación, el objetivo que se aspira a lograr y la disposición estructural que seguirá.
- **Capítulo 2. Diseño del sistema.** En el segundo capítulo titulado se aborda de manera integral la estructura del robot en cuestión. Se exploran aspectos tanto del diseño general del sistema como de su arquitectura, que abarca tanto el hardware como el software. Se detallan los componentes clave en la arquitectura de hardware, mientras que en la arquitectura de software se describen los elementos principales de ROS, la navegación del robot y la interacción natural con el mismo. Cada sección de este capítulo contribuye al entendimiento global de cómo se ha diseñado el sistema y cómo sus componentes colaboran en su funcionamiento.
- **Capítulo 3. Análisis de resultados.** En el capítulo 3, se detallan las pruebas realizadas, incluyendo las relacionadas con la interfaz natural y la navegación. Se presentan los resultados de los métodos de Aprendizaje Automático para Procesamiento de Lenguaje Natural, las pruebas de navegación y la integración de la interfaz natural con el sistema de navegación. Finalmente, se lleva a cabo una discusión sobre estos resultados.
- **Capítulo 4. Conclusiones y trabajo futuro.** El cuarto y último capítulo, culmina el documento al presentar las conclusiones extraídas de la investigación realizada. Aquí se resumen los hallazgos clave y se evalúa en qué medida se han alcanzado los objetivos

propuestos. Además, se establece una visión hacia adelante al discutir posibles direcciones para futuras investigaciones y desarrollos en el área.

Capítulo 2. Diseño del sistema.

Como se ha comentado anteriormente, en este capítulo se examina de manera exhaustiva la estructura completa del robot en cuestión. Se analizan diversos aspectos relacionados con la planificación general del sistema, así como con su organización, que comprende tanto el hardware como el software. Se proporcionan detalles acerca de los elementos fundamentales en la estructura del hardware, al mismo tiempo que se describen los principales componentes de la arquitectura de software, incluyendo ROS, la navegación del robot y su interacción natural.

2.1 Diseño global del sistema.

El sistema propuesto para el desarrollo de un robot autónomo capaz de seguir instrucciones en lenguaje natural se compone de dos partes principales: el hardware y el software.

El hardware del sistema está formado por el robot Pioneer 3-DX (ver Figura 10 y Figura 11), al que se le han sumado los siguientes componentes:

- Un sensor láser: El sensor láser se utiliza para la navegación del robot. Proporciona una representación del entorno del robot en forma de mapa de puntos.
- Una cámara: La cámara se podría utilizar para la detección de obstáculos y la identificación de objetos, pero en este caso la vamos a emplear para el reconocimiento facial y de gestos del usuario.
- Un mini PC: El PC es la base para el desarrollo de todo el sistema software.
- Una pantalla táctil: La pantalla táctil es un complemento para poder navegar por el PC, sin tener que conectarnos de forma remota.
- Un micrófono: El micrófono se utiliza para la captura de voz del usuario.
- Un altavoz: Este componente no será utilizado en este proyecto, ya que no le hemos añadido la opción de que el robot pueda hablar.

Todos los componentes anteriores están integrados en el robot a través de una carcasa pudiendo elevar la cámara a una altura más elevada para poder visualizar mejor el espacio que rodea al robot. El láser es el único componente que se encuentra en la parte superior del robot.

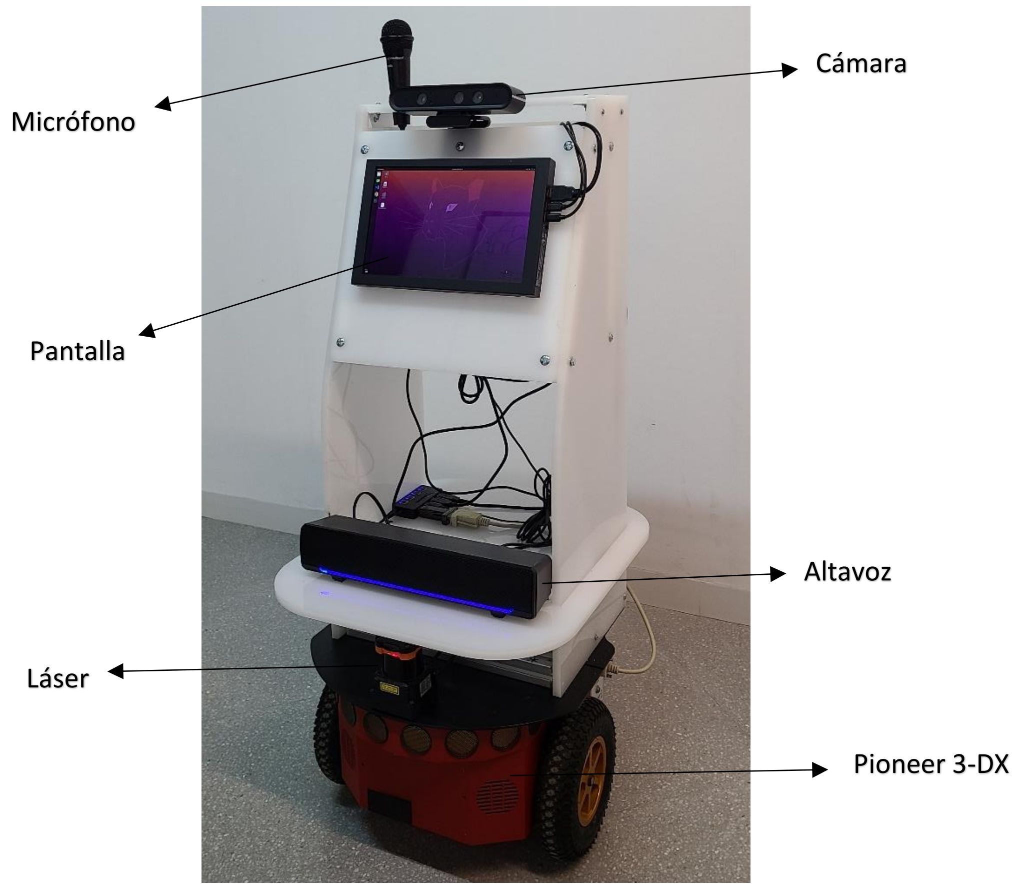

Figura 10. Diseño del robot. Parte delantera.

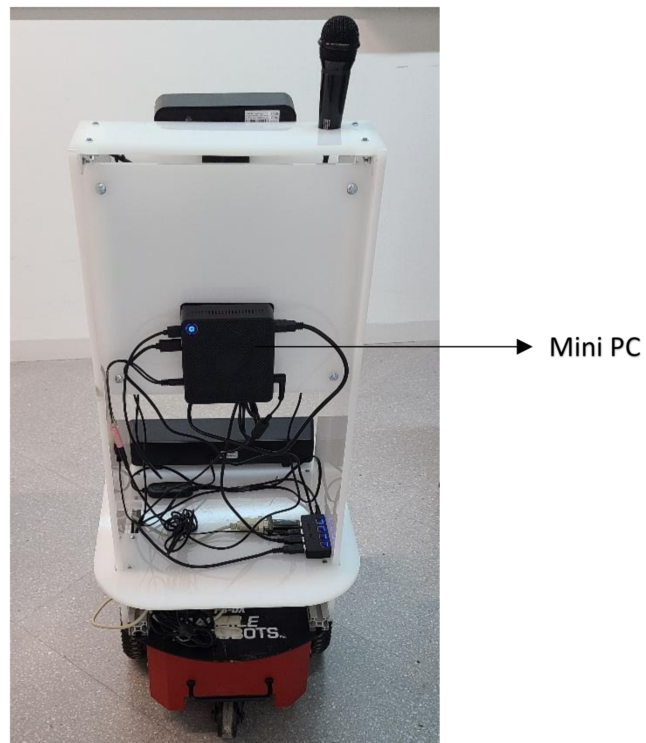

Figura 11. Diseño del robot. Parte trasera.

El software del sistema se desarrolla en ROS (*Robot Operating System*). ROS es un framework de software open source que proporciona una plataforma común para el desarrollo de robots autónomos. A partir de este entorno, junto todas las herramientas que nos proporciona (*RViz*, *Gazebo* etc.), programaremos en Python todo lo relacionado con la interfaz humano-robot, utilizando *TensorFlow*. Ya hablaré más detalladamente de todo el software más adelante.

2.2 Arquitectura hardware.

Tal y como se ha comentado brevemente en el punto anterior, la arquitectura hardware consta del robot Pioneer 3-Dx, el láser Hokuyo UTM-30LX, la cámara Orbecc Astra Pro, una pantalla táctil, el micrófono Hama Mic-P35 Allround y el mini PC. Ahora profundizaré en aquellos componentes que vamos a utilizar.

2.2.1 Robot Pioneer 3-DX.

El Pioneer 3-DX (ver Figura 12), es un robot móvil ampliamente reconocido en el campo de la robótica. Desarrollado por Adept Technology, Inc. (ahora parte de Omron Corporation), dicho se caracteriza por su configuración de dos ruedas motrices y una rueda loca de tipo castor. Esta disposición le otorga una excelente capacidad de maniobra y agilidad en diferentes entornos. Entre los componentes principales que integra, se encuentran:

1. **Array de dispositivos de ultrasonido (sonar) frontal:** Estos sensores ultrasónicos le permiten al robot detectar objetos y obstáculos en su entorno cercano, contribuyendo a su capacidad de navegación y evasión de obstáculos.
2. **Batería:** Es el componente que proporciona la energía necesaria para el funcionamiento del robot, permitiéndole realizar sus tareas y desplazamientos de manera autónoma.
3. **Encoders para las ruedas:** Estos encoders son dispositivos que ayudan a medir la velocidad y la distancia recorrida por las ruedas del robot, lo que facilita la precisión en el control del movimiento y la navegación.
4. **Microcontrolador con el firmware ARCOS:** El robot está equipado con un microcontrolador que utiliza el firmware ARCOS. Este firmware es fundamental para el control y la gestión de las funciones del robot, permitiendo su interacción con los diferentes componentes y la ejecución de tareas específicas.
5. **Paquete de desarrollo de software de robots móviles Pioneer SDK (Software Development Kit):** El SDK Pioneer proporciona un conjunto de herramientas, bibliotecas y recursos para el desarrollo de software destinado a programar y controlar el robot

Pioneer 3-DX. Esto facilita a los desarrolladores la creación de aplicaciones personalizadas y el aprovechamiento máximo de las capacidades del robot.

Figura 12. Pioneer 3-DX

En resumen, el Pioneer 3-DX es un robot móvil versátil y adaptable, con una configuración mecánica sólida y una amplia gama de sensores, complementada con un conjunto de herramientas de software que lo hacen idóneo para aplicaciones de investigación, educación y desarrollo en el campo de la robótica móvil.

2.2.2 Láser de escaneo Hokuyo UTM-30LX.

El láser de escaneo Hokuyo UTM-30LX es un dispositivo de medición láser 3D de alta velocidad y precisión. Está diseñado para aplicaciones de robótica, automatización y navegación. El UTM-30LX tiene una arquitectura hardware modular que consta de los siguientes componentes:

- Unidad de escaneo: Esta unidad contiene el láser, el espejo giratorio y los sensores de posición.
- Unidad de control: Esta unidad controla el funcionamiento del láser y el espejo giratorio.
- Unidad de interfaz: Esta unidad proporciona una interfaz para conectar el láser a un ordenador o sistema de control.

Unidad de escaneo

La unidad de escaneo del UTM-30LX contiene un láser de estado sólido de 635 nm y un espejo giratorio. El láser emite un rayo de luz que se refleja en el espejo giratorio y se dirige a un sensor de posición. El sensor de posición mide la posición del rayo de luz y la envía a la unidad de control.

Unidad de control

La unidad de control del UTM-30LX recibe los datos del sensor de posición y los utiliza para controlar el funcionamiento del láser y el espejo giratorio. La unidad de control también proporciona un puerto de comunicación para conectar el láser a un ordenador o sistema de control.

Unidad de interfaz

La unidad de interfaz del UTM-30LX proporciona una interfaz para conectar el láser a un ordenador o sistema de control. La interfaz puede ser de tipo serial, Ethernet o USB.

Las principales especificaciones técnicas del láser de escaneo Hokuyo UTM-30LX (ver Figura 13) son las siguientes:

- Distancia de medición: 0,2 a 30 m
- Anchura del haz: 0,15°
- Frecuencia de escaneo: 100 Hz
- Precisión: ±0,05°
- Resistencia al polvo y al agua: IP67

El láser de escaneo Hokuyo UTM-30LX se puede utilizar en una amplia gama de aplicaciones, entre las que se incluyen:

- Robótica: navegación de robots, manipulación de objetos, detección de obstáculos
- Automatización: control de procesos, inspección de productos, seguridad
- Navegación: sistemas de navegación autónomos, mapeo 3D

En conclusión, el láser de escaneo Hokuyo UTM-30LX es un dispositivo de medición láser 3D de alta velocidad y precisión. Su arquitectura hardware modular lo hace adecuado para una amplia gama de aplicaciones.

Figura 13. Láser Hokuyo UTM-30LX.

2.2.3 Cámara Orbecc Astra Pro.

La cámara Orbbec Astra Pro es una cámara 3D de profundidad basada en tecnología de luz estructurada. Está diseñada para aplicaciones de robótica, realidad aumentada y realidad virtual. La Astra Pro tiene una arquitectura hardware modular que consta de los siguientes componentes:

- Unidad de cámara: Esta unidad contiene la cámara RGB, la cámara de profundidad y los sensores de iluminación.
- Unidad de procesamiento: Esta unidad procesa las imágenes de profundidad y RGB para generar datos de nube de puntos.
- Unidad de interfaz: Esta unidad proporciona una interfaz para conectar la cámara a un ordenador o sistema de control.

Unidad de cámara

La unidad de cámara de la Astra Pro contiene una cámara RGB de 1920 x 1080 píxeles y una cámara de profundidad de 640 x 480 píxeles. La cámara RGB se utiliza para capturar imágenes en color, mientras que la cámara de profundidad se utiliza para capturar imágenes de profundidad. La unidad de cámara también contiene los sensores de iluminación necesarios para generar las imágenes de profundidad. Estos sensores emiten un patrón de luz estructurada que se refleja en los objetos de la escena. El patrón de luz reflejado se captura por la cámara de profundidad, que utiliza algoritmos de procesamiento de imágenes para calcular la profundidad de los objetos.

Unidad de procesamiento

La unidad de procesamiento de la Astra Pro es un circuito integrado (ASIC) personalizado diseñado por Orbbec. Este ASIC se encarga de procesar las imágenes de profundidad y RGB para generar datos de nube de puntos. Los datos de nube de puntos se almacenan en la memoria interna de la Astra Pro. Estos datos se pueden transferir a un ordenador o sistema de control a través de la unidad de interfaz.

Unidad de interfaz

La unidad de interfaz de la Astra Pro proporciona una interfaz para conectar la cámara a un ordenador o sistema de control. La interfaz puede ser de tipo USB 2.0 o Ethernet.

Las principales especificaciones técnicas de la cámara Orbbec Astra Pro son las siguientes:

- Rango de medición: 0,4 a 8 m
- Resolución de profundidad: 640 x 480 píxeles
- Frecuencia de actualización: 30 fps
- Precisión de profundidad: ±1 mm

La cámara Orbbec Astra Pro se puede utilizar en una amplia gama de aplicaciones, entre las que se incluyen:

- Robótica: navegación de robots, manipulación de objetos, detección de obstáculos
- Realidad aumentada y realidad virtual: captura de movimiento, visualización 3D
- Digitalización 3D: escaneo de objetos, mapeo 3D

En conclusión, la cámara Orbbec Astra Pro (ver Figura 14) es una cámara 3D de profundidad de alta calidad y rendimiento. Su arquitectura hardware modular la hace adecuada para una amplia gama de aplicaciones.

Figura 14. Cámara Orbecc Astra Pro.

2.2.4 Micrófono Hama Mic-P35 Allround.

El micrófono Hama Mic-P35 Allround es un micrófono de condensador unidireccional con una respuesta de frecuencia de 50 a 16000 Hz. Está diseñado para aplicaciones de voz y audio general. El Mic-P35 Allround tiene una arquitectura hardware sencilla que consta de los siguientes componentes:

- Cápsula de micrófono: Esta cápsula es el elemento sensible al sonido del micrófono. Está fabricada con un diafragma de condensador que se mueve en respuesta a las ondas sonoras.
- Circuito de preamplificador: Este circuito amplifica la señal del micrófono para que sea compatible con la mayoría de los dispositivos de grabación.
- Conexión: El micrófono tiene una conexión de 3,5 mm que se conecta a la entrada de micrófono de un ordenador, una grabadora u otro dispositivo de grabación.

Cápsula de micrófono

La cápsula de micrófono del Mic-P35 Allround es una cápsula electret de condensador. Esta cápsula es pequeña y ligera, lo que la hace ideal para su uso en aplicaciones móviles. El diafragma de la cápsula está fabricado con un material piezoeléctrico que genera una carga eléctrica en respuesta a las ondas sonoras. Esta carga eléctrica es amplificada por el circuito de preamplificador.

Circuito de preamplificador

El circuito de preamplificador del Mic-P35 Allround es un circuito integrado que amplifica la señal del micrófono. El circuito de preamplificador tiene un nivel de ganancia de 36 dB, que proporciona una señal suficientemente fuerte para la mayoría de los dispositivos de grabación.

Conexión

El micrófono tiene una conexión de 3,5 mm que se conecta a la entrada de micrófono de un ordenador, una grabadora u otro dispositivo de grabación. La conexión de 3,5 mm es un estándar común que se utiliza en la mayoría de los dispositivos de audio.

Las principales especificaciones técnicas del micrófono Hama Mic-P35 Allround son las siguientes:

- Tipo: Micrófono de condensador unidireccional
- Respuesta de frecuencia: 50 a 16000 Hz
- Impedancia de salida: 2200 Ω
- **Nivel de presión sonora (SPL): -58 dB (1 kHz, 0,1 % THD)
- Sensibilidad: -40 dB (1 kHz, 0 dB SPL)
- Conexión: 3,5 mm

El micrófono Hama Mic-P35 Allround se puede utilizar en una amplia gama de aplicaciones, entre las que se incluyen:

- Grabación de voz: Para grabar voz, podcasts, etc.
- Grabación de audio general: Para grabar música, entrevistas, etc.
- Streaming de audio: Para transmitir audio en vivo a través de Internet.

En conclusión, el micrófono Hama Mic-P35 Allround (ver Figura 15) es un micrófono de condensador unidireccional de calidad y rendimiento. Su arquitectura hardware sencilla lo hace fácil de usar y configurar.

Figura 15. Micrófono Hama Mic-P35 Allround.

2.3 Arquitectura software.

El objetivo de este TFG es diseñar un robot con navegación autónoma que pueda interaccionar con las personas de forma sencilla. Para ello, se ha utilizado un modelo de aprendizaje profundo gratuito que no requiere conexión a Internet y se ha adaptado al proyecto, dicha librería se llama *TensorFlow.*

La integración de los componentes del sistema se ha llevado a cabo mediante ROS, una plataforma de software de código abierto que proporciona herramientas para la programación, la simulación y la monitorización de robots. Además, se ha utilizado el lenguaje *Python*, que es el lenguaje de programación más popular para el aprendizaje automático, y *Gazebo*, un simulador de robots que permite realizar simulaciones en un entorno 3D.

2.3.1 Descripción de los elementos principales de ROS.

El Robot Operating System (ROS) es un framework de software para el desarrollo de robots y sistemas robóticos. Fue creado por el instituto de investigación Willow Garage en 2007 bajo licencia BSD y todos sus programas son de código abierto (OSS), lo que permite su uso gratuito tanto para la investigación como para fines comerciales.

ROS ofrece varias características clave (Repositorio UPCT, s.f.) que lo hacen ideal para el desarrollo de robots, como:

- Control de dispositivos de bajo nivel.
- Abstracción de hardware.
- Sistema de comunicación de procesos basado en el paso de mensajes.
- Sistema de administración de paquetes.
- Diseño de software distribuido basado en el concepto de paquete software.
- Implementación de utilidades que se utilizan comúnmente en la comunidad para robots.

Dentro del ecosistema ROS, existen varios elementos que son fundamentales para su funcionamiento y flexibilidad (Wiki ros, s.f.):

- **Nodos:** Son unidades de ejecución individuales en ROS. Cada nodo realiza tareas específicas y puede comunicarse con otros nodos mediante mensajes, servicios y parámetros.

- **Mensajes:** Los nodos se comunican intercambiando mensajes. Los mensajes definen la estructura de los datos compartidos entre los nodos y permiten la transferencia de información relevante, como coordenadas, imágenes, velocidades, etc.
- **Topics (Tópicos):** Los topics son canales de comunicación unidireccional a través de los cuales los nodos envían y reciben mensajes. Los nodos pueden publicar mensajes en un topic o suscribirse a otro para recibir mensajes relevantes.
- **Servicios:** Los servicios permiten la comunicación de solicitud-respuesta entre nodos. Un nodo puede ofrecer un servicio y otros nodos pueden enviar solicitudes para obtener una respuesta del nodo de servicio.
- **Parámetros:** Los parámetros son valores que pueden ser ajustados en tiempo de ejecución para modificar el comportamiento de los nodos. Son utilizados para configurar aspectos específicos del sistema robótico.
- **Bags (Bolsas):** Las bolsas son archivos de registro que almacenan datos publicados en los topics durante la ejecución de nodos. Esto permite reproducir y analizar experimentos en un momento posterior.
- **Arquitectura Cliente-Servidor:** ROS permite una arquitectura flexible cliente-servidor, lo que significa que los nodos pueden estar distribuidos en diferentes computadoras y comunicarse a través de la red.
- **Herramientas de Visualización:** ROS proporciona herramientas de visualización para ayudar a comprender y depurar el comportamiento del sistema. Ejemplos incluyen Rviz para visualización 3D y rqt para interfaces gráficas.
- **Paquetes:** Los paquetes son la unidad básica de organización en ROS. Contienen nodos, mensajes, topics, servicios, bibliotecas y archivos de configuración relacionados con una funcionalidad específica.
- **Catkin:** Es el sistema de construcción utilizado en ROS para compilar y administrar paquetes.
- **Simuladores:** ROS se integra con varios simuladores (como Gazebo) que permiten probar y desarrollar código robótico en un entorno virtual antes de implementarlo en un robot físico.
- **Comunidad y Documentación:** ROS cuenta con una gran comunidad de desarrolladores y usuarios que contribuyen con paquetes, tutoriales y soporte. La documentación oficial y los foros son recursos valiosos para aprender y resolver problemas.

2.3.2 Navegación del robot.

La navegación en el contexto de ROS se refiere al conjunto de técnicas y herramientas utilizadas para permitir que un robot se mueva de manera autónoma en su entorno, evitando obstáculos y alcanzando objetivos específicos. La navegación en ROS (Villarreal Onofre, 2021) involucra varios componentes y procesos que se combinan para lograr un movimiento seguro y eficiente. A continuación, describo los elementos clave relacionados con la navegación en ROS:

- **Costmap y Obstacle Layer:** Uno de los pilares fundamentales de la navegación en ROS es la utilización de costmaps y el Obstacle Layer. Estos elementos permiten al robot desarrollar una comprensión detallada del entorno que le rodea. Los costmaps son representaciones en dos dimensiones del terreno, mostrando áreas transitables y obstáculos detectados. El Obstacle Layer se encarga de actualizar esta información en tiempo real a partir de los datos de los sensores del robot. Esta combinación permite al robot planificar rutas que evitan colisiones y optimizan su movimiento en entornos complejos y cambiantes.
- **Global Planner:** Desempeña un papel crucial al trazar la ruta general que el robot seguirá para alcanzar su destino. Basado en algoritmos como A* o Dijkstra, este componente toma en cuenta el costmap y el objetivo deseado para generar una ruta general que guía al robot a través de áreas transitables y evita zonas de obstáculos.
- **Local Planner:** Mientras que el Global Planner define la ruta general, el Local Planner se concentra en los detalles del movimiento autónomo. Utilizando información en tiempo real de los costmaps y los sensores del robot, el Local Planner genera comandos de control precisos para sortear obstáculos y mantener el curso planificado. Este proceso asegura que el robot navegue de manera suave y segura incluso en entornos desafiantes.
- **Move_base:** Es un paquete de ROS que actúa como el núcleo de control para la navegación autónoma en ROS. Reúne todos los componentes previamente mencionados, como el Global y Local Planner, para crear un sistema integrado que permite al robot navegar con autonomía. Este paquete simplifica la configuración y ejecución de la navegación.
- **TF (Transform Library):** En el ámbito de la navegación, la Transform Library es esencial para garantizar la consistencia espacial. Facilita la transformación de datos entre distintos sistemas de coordenadas, como el marco de referencia del robot y el marco

del mapa. Esta coherencia es esencial para asegurar que los datos de sensores y planificación estén en sintonía, lo que es crucial para un movimiento preciso.

- **Gazebo:** Como he mencionado anteriormente, Gazebo es simulador 3D integrado con ROS, desempeña un papel vital en el proceso de navegación. Permite probar algoritmos de navegación en entornos virtuales antes de implementarlos en un robot físico. Esta capacidad de experimentación previa permite optimizar los algoritmos y garantizar un movimiento autónomo exitoso.
- **SLAM (Simultaneous Localization and Mapping):** En situaciones en las que el entorno es desconocido, la técnica de Simultaneous Localization and Mapping (SLAM) es crucial. ROS ofrece el uso de gmapping, un algoritmo SLAM que permite al robot construir un mapa mientras navega y, al mismo tiempo, estimar su posición en el mapa en tiempo real. Esto resulta invaluable para crear mapas precisos y completos de entornos previamente no explorados.
- **RViz:** Es una herramienta de visualización en tiempo real en ROS. Permite observar el progreso del robot en tiempo real, visualizar costmaps, inspeccionar rutas planificadas y ejecutadas, y ajustar parámetros sobre la marcha.

2.3.2.1 Stack Navigation de ROS.

Para la navegación del robot, ROS dispone de un paquete llamado *Navigation* (Repositorio UPCT, s.f.). Este paquete cubre:

- Publicación de datos de sensores.
- Publicación de información de odometría.
- Configuración de las transformadas.
- Construcción de un mapa.
- Localización.
- Planificación global y local de caminos.

La pila de Navegación en 2D toma datos de la odometría, sensores y objetivos y genera comandos de velocidad que se envían a la base móvil. Este paquete está centrado en la generación de trayectorias de forma autónoma para un robot, dado un punto de destino. Para ello, emplea los datos recibidos de odometría y sensores exteroceptivos que le permitan recopilar información métrica de su entorno.

Para la construcción del mapa se emplea ***slam_gmapping***, un nodo de ROS del paquete *Gmapping*. Este nodo toma los datos publicados en el *topic scan* por el láser, que junto con la información aportada por los *frames baselink* y *odom* (un *frame* es el sistema de coordenadas de un objeto o del conjunto global del sistema), permite la construcción de un mapa en 2D de la estancia en la que se encuentra el robot. Para que el mapa sea lo más completo posible, el robot debe recorrer la entorno para que el láser recoja el mayor número de datos de la geometría de la habitación. Para ello he usado el paquete de ROS ***rqt.*** Este paquete permite tele operar el robot mediante una interfaz que aparece en pantalla (ver Figura 16).

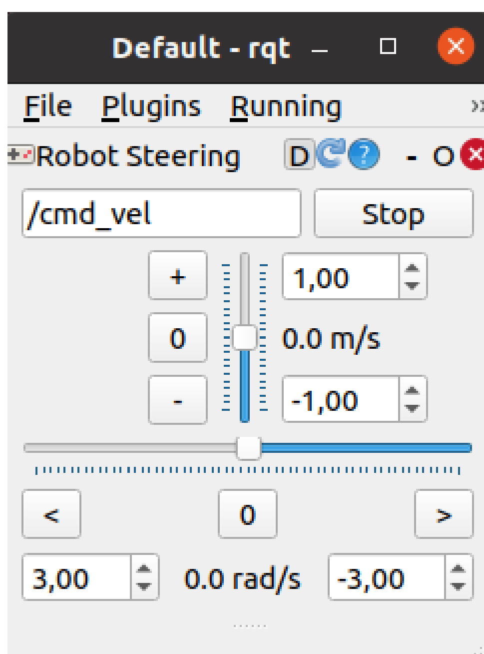

Figura 16. El interfaz de ROS para tele operar al robot.

Una vez hecho el mapa, se guarda usando el paquete ***map_server***. Para guardar el mapa, se escribe lo siguiente en una terminal:

```
~$ rosrun map_server map_saver -f {nombre_del_mapa}
```

Este comando genera y guarda los archivos *map.pgm* y *map.yaml* en la carpeta personal del usuario. El archivo de tipo *pgm* contiene la imagen del mapa, mientras que el archivo tipo *yaml* contiene la información necesaria para la navegación por ese mapa.

El paquete AMCL de ROS es un sistema probabilístico de localización para la movilidad de un robot en 2D. Este paquete usa un filtro de partículas para rastrear la pose del robot por un mapa

conocido. Para incluir estas funciones en el robot se deben crear dos archivos tipo *launch*, que se incluirán en el *launch* final del robot junto con el nodo *move_base*. El archivo ***amcl.launch,*** este archivo contiene todos los parámetros que AMCL necesita, y el archivo ***mapa.launch***, para cargar el mapa creado con *gmapping*. Una vez creados estos 2 archivos, podemos combinarlos en 1 para proporcionar una navegación autónoma basada en mapas y amcl, yo lo he guardado como ***amcl_todo.launch***.

Para la configuración del nodo *move_base* se tiene que crear una carpeta llamada *config*, donde se crean cuatro archivos que contienen la configuración de las diferentes partes que componen la navegación autónoma del robot. En el archivo ***costmap_common_params.yaml*** se establecen los parámetros comunes del mapa global y local. Los parámetros para la localización global se introducen en el archivo ***global_costmap_params.yaml*** y los parámetros para la localización local se introducen en el archivo ***local_costmap_params.yaml.*** Por último, en el archivo ***trayectory_planner.yaml***, se establecen los parámetros necesarios para el cálculo de trayectorias para la navegación autónoma, considerando que el robot utilizado es no holónomo.

Ahora se deben crear 2 archivos del tipo *launch*, en su capeta correspondiente. El primero de ello agrupará los 4 archivos nombrados anteriormente, lo llamaremos ***planificacion_movebase.launch*** y el segundo será el encargado de combinar el paquete de planificación de *move_base*, archivo *launch* anterior, junto con los archivos *launch* de *amcl*, mapas y el robot, lo guardaremos como ***todo_planificacion.launch***.
La Figura 17 muestra el esquema de los topics y nodos del sistema.

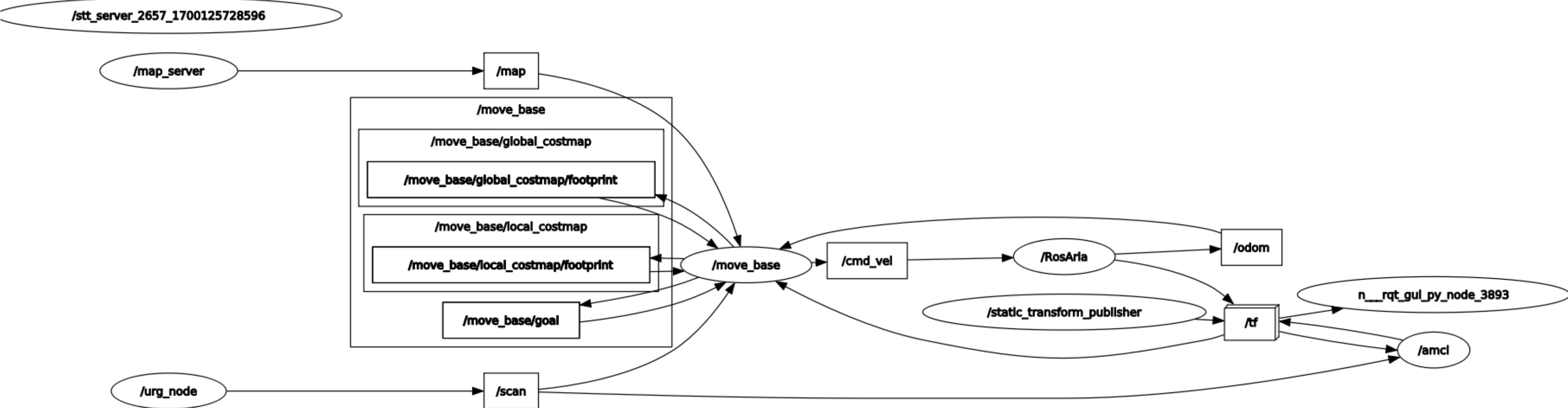

Figura 17. Esquema de los *topics* y *nodos* del sistema.

2.3.2.2 Uso del simulador GAZEBO.

La simulación robótica es una herramienta esencial en el tool-box robótico. Un simulador bien diseñado permite probar rápidamente algoritmos, robots diseñados y realizar pruebas de regresión utilizando escenarios realistas. Gazebo ofrece la posibilidad de simular con precisión y eficiencia poblaciones de robots en entornos interiores y exteriores complejos. Genera tanto la realimentación de los sensores como las interacciones físicas entre objetos tratándolos como cuerpos rígidos, a través de gráficos de alta calidad e interfaces programables. Además de tener una comunidad activa.

La descripción visual del robot, así como las propiedades relacionadas con la dinámica y la interacción con el entorno simulado se definen en archivos que utilizan el formato URDF con extensiones *xacro*, que permiten definir propiedades, e incluso incluir operaciones para facilitar la parametrización de los archivos que describen los modelos de robots.

Los elementos visuales están asociados a *plugins* que simulan el comportamiento de los diferentes elementos de un robot, por ejemplo, simulan la cinemática, permitiendo calcular la odometría a partir de las consignas de velocidad publicadas en un topic (normalmente llamado */cmd_vel*), y publican dicha odometría en otro topic (generalmente llamado */odom*). La descripción de estos elementos se realiza en un archivo con extensión .gazebo.

Por defecto, Gazebo genera un mundo vacío cuando se inicia. Dentro de la aplicación se pueden insertar figuras geométricas e iluminación, modificar su tamaño, color y posición. Esto es útil para hacer cambios en tiempo de ejecución. Sin embargo, es más eficiente tener un archivo predefinido con todos los modelos que se desean incluir en el simulador. De esta manera no se tienen que meter manualmente cada vez que se arranque la aplicación. En mi caso, he tenido que crear mi propio mundo, que se asemeje a la nave industrial donde va a operar el robot. Una vez creado dicho mundo lo he tenido que guardar para seguir trabajando sobre él. La Figura 18 muestra una captura del mundo simulado en Gazebo.

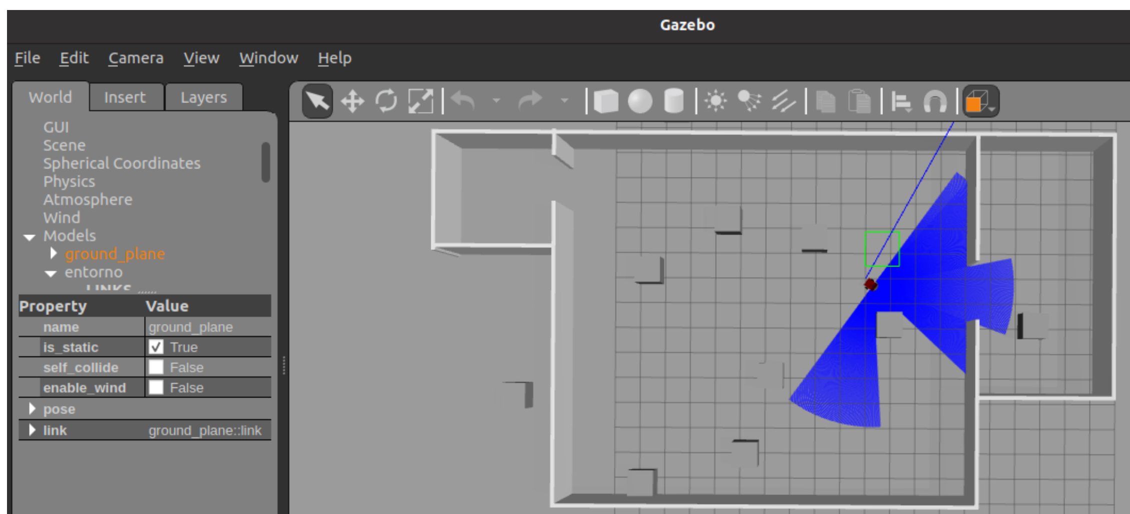

Figura 18. Entorno simulado en Gazebo

Podemos observar que dicho entorno se divide en 3 partes, una sala pequeña, otra de tamaño medio y otra sala principal más grande. También vemos la presencia de cajas repartidas por el mapa, que nos han ayudado a un escaneo de mayor calidad con el *gmapping*, así como nos servirán de obstáculos para los tests simulados.

2.3.2.3 Navegación en el robot real.

Antes de nada, hay que explicar la manera en la que nos conectamos al pc del robot, ya que en mi caso lo he hecho de forma remota.

Por ello debemos de utilizar el comando:

```
~$ ssh {Direccion_IP_Robot} -ljuno
```

Con ese comando nos conectamos de forma remota al robot, ahora tenemos que ejecutar el *roscore* del robot, y a continuación dar los permisos necesarios para habilitar el uso del sistema de rotación y el láser. Esto se habilita con los comandos:

```
~$ sudo chmod 777 /dev/ttyUSB0
```

```
~$ sudo chmod 777 /dev/ttyACM0
```

Una vez hayamos hecho esto, estamos preparados para lanzar, en el pc del robot, el paquete “pioneerjuno.launch” para poder manejarlo de forma remota.

Una vez conectados al robot, para la implementación de la navegación, se siguieron los mismos pasos descritos anteriormente en la simulación, con algunas consideraciones adicionales:

- El mapeo del entorno real se llevó a cabo teleoperando manualmente el robot por la zona de pruebas mientras se ejecutaba el nodo *slam_gmapping*. Esto permitió generar un mapa 2D detallado del entorno real, incluyendo la geometría exacta y la ubicación de los obstáculos.
- Se tuvo especial cuidado en obtener lecturas precisas de los sensores durante el mapeo, evitando deslizamientos y errores en la odometría que pudieran distorsionar el mapa. Para ello fue crítico mantener una velocidad moderada y una trayectoria suave durante el teleoperado.
- El mapa generado del entorno real se procesó para optimizarlo y limpiar ruido antes de utilizarlo para la navegación autónoma. Esto ayuda a mejorar el rendimiento de la localización y la planificación de rutas.
- Los parámetros de configuración en los archivos de *launch* y *YAML* se ajustaron a las características específicas del robot y el entorno real, principalmente en términos de tamaño, velocidades y resolución del sensor láser.
- Se comprobó cuidadosamente la transformación de coordenadas entre el láser, la base del robot y el mapa para asegurar consistencia espacial y lecturas precisas. Pequeños errores pueden degradar el rendimiento.
- Tras lanzar la navegación con *AMCL* y *move_base*, se realizaron múltiples pruebas designando objetivos de navegación en el mapa y verificando que el robot era capaz de alcanzarlos libremente sin colisiones.
- Cuando surgían problemas de navegación, se ajustaron de forma iterativa los parámetros de *costmaps*, tolerancia de obstáculos y planificación de trayectorias.

Con estos pasos, se logró implementar la navegación autónoma del robot real en el entorno mapeado, permitiéndole navegar hacia objetivos arbitrarios indicados en el mapa.

2.3.3 Interacción natural con el robot.

En esta sección se describen los componentes adicionales, a nivel de software, que se han integrado para realizar las tareas de reconocimiento de voz y procesamiento del lenguaje natural, reconocimiento de personas y reconocimiento de gestos.

2.3.3.1 Redes tipo Transformer para procesamiento de lenguaje natural.

Antes de explicar el proceso que he llevado a cabo con las redes tipo Transformer para el procesamiento del lenguaje natural, hay que tener un algoritmo que nos permita reconocer y sintetizar la voz de la mejor manera, para tener entradas a nuestro Transformer lo más limpias y fiables posibles. En mi caso, he utilizado varias librerías como son:

Vosk: Librería de Python para reconocimiento de voz offline. Usa modelos entrenados de Kaldi para transcribir audio a texto. Permite flujos de audio en tiempo real para reconocimiento continuo.

Sounddevice: Librería para reproducir y grabar audio con Python. Permite streams de audio en tiempo real.

Pyttsx3: Librería de síntesis de voz para Python. Permite convertir texto a speech usando distintos motores de voz.

También he utilizado otras librerías como *rospy* o *std_msgs* que nos permiten trabajar, publicando y suscribiéndonos, a los topics de ROS para la comunicación entre nodos, ya que la entrada de voz será a través del micrófono que está en el robot y necesito saber en que topic está publicando el micrófono.

Una vez transcrita la voz, se puede pasar al reconocimiento del lenguaje natural. Este proceso permite que la máquina entienda el texto.

Los Transformers son una novedosa solución para el reconocimiento del lenguaje natural. Se dividen en dos fases: pre-entrenamiento y fine-tuning.

Para contextualizar, los códigos utilizados para ambas fases se basan en los tutoriales de TensorFlow (Modelo transformador para la comprensión del lenguaje., s.f.). He realizado algunos cambios, como modificar las líneas que cargan el conjunto de datos, las que recorren el dataset y las de los hiperparámetros.

A medida que explico las partes del código, adjunto dicho código, excepto en la fase de fine-tuning, que solo incluyo algunas partes, ya que esta fase es mucho más extensa, por lo que la incluyo como anexo al final del documento.

Conjunto de datos

Es lógico que para entrenar un modelo primero debemos tener el conjunto de datos, en mi caso lo he hecho de forma manual. Este dataset (ver Figura 19) estará dividido por conjuntos de palabras separadas por el tabulador y estos datos corresponderán a posibles órdenes que le puedo dar al robot (columna izquierda) y lo que quiero que entienda el robot (columna derecha).

El dataset cuenta con más de 200 maneras diferentes de decirle al robot que tiene que ir a la zona que le digamos, en el mapa real le he asignado 7 zonas objetivo, que debe saber llegar una vez que le digamos cualquier orden.

```
Por favor, ve rápidamente a la zona uno.	Ir zona uno.
Apresúrate a la zona dos ahora mismo por favor.	Ir zona dos.
No te detengas hasta llegar a la zona tres por favor.	Ir zona tres.
Por favor, dirígete a toda prisa hacia la zona cuatro.	Ir zona cuatro.
Date prisa por favor en llegar a la zona cinco.	Ir zona cinco.
Camina veloz hacia la zona seis por favor.	Ir zona seis.
Ve urgente por favor a la zona siete.	Ir zona siete.
Por favor mueve los pies rápido hacia la zona uno.	Ir zona uno.
No te retrases, ve ligero a la zona dos por favor.	Ir zona dos.
Por favor apresúrate yendo a la zona tres.	Ir zona tres.
Muévete más rápido hacia la zona cuatro por favor.	Ir zona cuatro.
Por favor acelera el paso hacia la zona cinco.	Ir zona cinco.
Camina con presteza a la zona seis por favor.	Ir zona seis.
Ve raudo y veloz a la zona siete por favor.	Ir zona siete.
Ve a toda marcha a la zona uno por favor.	Ir zona uno.
```

Figura 19. Muestra del dataset de frases para el procesamiento del lenguaje natural.

Fase de pre-entrenamiento

Para tener un visión general y clara de cómo funciona este código, aclararé de forma clara y concisa los puntos relevantes de esta arquitectura:

- Se carga el dataset de ejemplos de texto (órdenes y respuestas del robot).
- Se crea un vocabulario [1]BERT a partir de este dataset, tanto para las órdenes como para las respuestas, utilizando la función *bert_vocab_from_dataset*. Esto genera un vocabulario de los tokens (palabras y subpalabras) más comunes en el dataset.

```
bert_tokenizer_params=dict(lower_case=True)
reserved_tokens=["[PAD]", "[UNK]", "[START]", "[END]"]

bert_vocab_args = dict(
    # The target vocabulary size
    vocab_size = 8000,
    # Reserved tokens that must be included in the vocabulary
    reserved_tokens=reserved_tokens,
    # Arguments for `text.BertTokenizer`
    bert_tokenizer_params=bert_tokenizer_params,
    # Arguments for
`wordpiece_vocab.wordpiece_tokenizer_learner_lib.learn`
    learn_params={},
)
```

- Se guardan estos vocabularios en archivos de texto.

```
def write_vocab_file(filepath, vocab):
  with open(filepath, 'w') as f:
    for token in vocab:
      print(token, file=f)

write_vocab_file('orden_vocab.txt', orden_vocab)
write_vocab_file('robot_vocab.txt', robot_vocab)
```

[1] BERT (Bidirectional Encoder Representations from Transformers) es un modelo de lenguaje preentrenado de Google AI, entrenado en un conjunto de datos masivo de texto y código. BERT es un modelo de transformador de doble dirección, lo que significa que puede aprender relaciones entre palabras en una secuencia tanto en la dirección anterior como posterior.

- Se crean tokenizadores de TensorFlow Text usando estos vocabularios BERT. Estos tokenizadores segmentarán el texto en tokens basándose en el vocabulario aprendido.

```
orden_tokenizer = text.BertTokenizer('orden_vocab.txt',
**bert_tokenizer_params)
robot_tokenizer = text.BertTokenizer('robot_vocab.txt',
**bert_tokenizer_params)

# Tokenize the examples -> (batch, word, word-piece)
token_batch = en_tokenizer.tokenize(robot_examples)
# Merge the word and word-piece axes -> (batch, tokens)
token_batch = token_batch.merge_dims(-2,-1)
# Lookup each token id in the vocabulary.
txt_tokens = tf.gather(robot_vocab, token_batch)
# Join with spaces.
tf.strings.reduce_join(txt_tokens, separator=' ', axis=-1)
words = en_tokenizer.detokenize(token_batch)
tf.strings.reduce_join(words, separator=' ', axis=-1)
```

- Se añaden tokens especiales como [START] y [END] al principio y final de las secuencias.

```
START = tf.argmax(tf.constant(reserved_tokens) == "[START]")
END = tf.argmax(tf.constant(reserved_tokens) == "[END]")

def add_start_end(ragged):
  count = ragged.bounding_shape()[0]
  starts = tf.fill([count,1], START)
  ends = tf.fill([count,1], END)
  return tf.concat([starts, ragged, ends], axis=1)

words = en_tokenizer.detokenize(add_start_end(token_batch))
tf.strings.reduce_join(words, separator=' ', axis=-1)
```

- Se crea una clase *CustomTokenizer* que encapsula el tokenizer de TensorFlow Text y expone métodos para tokenizar, detokenizar (reconstruir el texto), hacer lookups de IDs a tokens, etc.

```
def cleanup_text(reserved_tokens, token_txt):
  # Drop the reserved tokens, except for "[UNK]".
  bad_tokens = [re.escape(tok) for tok in reserved_tokens if tok !=
"[UNK]"]
  bad_token_re = "|".join(bad_tokens)

  bad_cells = tf.strings.regex_full_match(token_txt, bad_token_re)
  result = tf.ragged.boolean_mask(token_txt, ~bad_cells)
```

```
  # Join them into strings.
  result = tf.strings.reduce_join(result, separator=' ', axis=-1)

  return result


token_batch = en_tokenizer.tokenize(en_examples).merge_dims(-2,-1)
words = en_tokenizer.detokenize(token_batch)
cleanup_text(reserved_tokens, words).numpy()


class CustomTokenizer(tf.Module):
  def __init__(self, reserved_tokens, vocab_path):
    self.tokenizer = text.BertTokenizer(vocab_path, lower_case=True)
    self._reserved_tokens = reserved_tokens
    self._vocab_path = tf.saved_model.Asset(vocab_path)

    vocab = pathlib.Path(vocab_path).read_text().splitlines()
    self.vocab = tf.Variable(vocab)

    ## Create the signatures for export:

    # Include a tokenize signature for a batch of strings.
    self.tokenize.get_concrete_function(
        tf.TensorSpec(shape=[None], dtype=tf.string))

    # Include `detokenize` and `lookup` signatures for:
    #   * `Tensors` with shapes [tokens] and [batch, tokens]
    #   * `RaggedTensors` with shape [batch, tokens]
    self.detokenize.get_concrete_function(
        tf.TensorSpec(shape=[None, None], dtype=tf.int64))
    self.detokenize.get_concrete_function(
          tf.RaggedTensorSpec(shape=[None, None], dtype=tf.int64))

    self.lookup.get_concrete_function(
        tf.TensorSpec(shape=[None, None], dtype=tf.int64))
    self.lookup.get_concrete_function(
          tf.RaggedTensorSpec(shape=[None, None], dtype=tf.int64))

    # These `get_*` methods take no arguments
    self.get_vocab_size.get_concrete_function()
    self.get_vocab_path.get_concrete_function()
    self.get_reserved_tokens.get_concrete_function()
```

```
  @tf.function
  def tokenize(self, strings):
    enc = self.tokenizer.tokenize(strings)
    # Merge the `word` and `word-piece` axes.
    enc = enc.merge_dims(-2,-1)
    enc = add_start_end(enc)
    return enc

  @tf.function
  def detokenize(self, tokenized):
    words = self.tokenizer.detokenize(tokenized)
    return cleanup_text(self._reserved_tokens, words)

  @tf.function
  def lookup(self, token_ids):
    return tf.gather(self.vocab, token_ids)

  @tf.function
  def get_vocab_size(self):
    return tf.shape(self.vocab)[0]

  @tf.function
  def get_vocab_path(self):
    return self._vocab_path

  @tf.function
  def get_reserved_tokens(self):
    return tf.constant(self._reserved_tokens)

tokenizers = tf.Module()
tokenizers.pt = CustomTokenizer(reserved_tokens, 'orden_vocab.txt')
tokenizers.en = CustomTokenizer(reserved_tokens, 'robot_vocab.txt')
```

- Se guarda esta clase *CustomTokenizer* como un modelo de TensorFlow *SavedModel* para poder cargarla después e inferir sobre nuevos ejemplos de texto.

```
model_path = '/content/gdrive/MyDrive/lista/SavedModel'
tf.saved_model.save(tokenizers, model_path)

reloaded_tokenizers = tf.saved_model.load(model_path)
reloaded_tokenizers.en.get_vocab_size().numpy()
```

- Al hacer inferencia, el texto de entrada se tokeniza en IDs usando el vocabulario BERT, se pueden reconstruir los tokens con el *lookup* y finalmente reconstruir el texto original con *detokenize*.

```
tokens = reloaded_tokenizers.en.tokenize(['Hello TensorFlow!'])
tokens.numpy()
text_tokens = reloaded_tokenizers.en.lookup(tokens)
round_trip = reloaded_tokenizers.en.detokenize(tokens)
```

Fase de fine-tuning

Debido a que en esta fase el código es mucho más extenso, comentaré las partes más relevantes, junto con la estructura que tiene que seguir. El código completo está en Anexos.

- Tokenización: Utiliza tokenizadores BERT pre-entrenados para convertir las frases a secuencias numéricas de tokens. Permite representar las palabras como vectores densos.

```
tokenizers = tf.saved_model.load(model_path)

encoded = tokenizers.en.tokenize(en_examples)
round_trip = tokenizers.en.detokenize(encoded)
```

- Positional Encoding: Incorpora información sobre la posición de cada token en la frase mediante funciones senoidales. Esto permite modelar el orden secuencial.

```
def positional_encoding(length, depth):
  depth = depth/2

  positions = np.arange(length)[:, np.newaxis]     # (seq, 1)
  depths = np.arange(depth)[np.newaxis, :]/depth   # (1, depth)

  angle_rates = 1 / (10000**depths)         # (1, depth)
  angle_rads = positions * angle_rates      # (pos, depth)

  pos_encoding = np.concatenate(
      [np.sin(angle_rads), np.cos(angle_rads)],
      axis=-1)

  return tf.cast(pos_encoding, dtype=tf.float32)
```

- Encoder: Consta de múltiples capas idénticas de atención autoregresiva (*self-attention*). Modela interacciones globales entre todos los tokens de entrada para obtener una representación vectorial rica y contextualizada.

```
class Encoder(tf.keras.layers.Layer):
  def __init__(self, *, num_layers, d_model, num_heads,
               dff, vocab_size, dropout_rate=0.1):
    super().__init__()

    self.d_model = d_model
    self.num_layers = num_layers

    self.pos_embedding = PositionalEmbedding(
        vocab_size=vocab_size, d_model=d_model)

    self.enc_layers = [
        EncoderLayer(d_model=d_model,
                     num_heads=num_heads,
                     dff=dff,
                     dropout_rate=dropout_rate)
        for _ in range(num_layers)]
    self.dropout = tf.keras.layers.Dropout(dropout_rate)

  def call(self, x):
    # `x` is token-IDs shape: (batch, seq_len)
    x = self.pos_embedding(x)  # Shape `(batch_size, seq_len, d_model)`.

    # Add dropout.
    x = self.dropout(x)

    for i in range(self.num_layers):
      x = self.enc_layers[i](x)

    return x  # Shape `(batch_size, seq_len, d_model)`.
```

- Decoder: Genera la traducción de forma autorregresiva, token por token. Utiliza atención causal sobre la salida generada y atención cruzada sobre la salida del *encoder* para enfocarse en partes relevantes del contexto.

```
class Decoder(tf.keras.layers.Layer):
  def __init__(self, *, num_layers, d_model, num_heads, dff, vocab_size,
               dropout_rate=0.1):
    super(Decoder, self).__init__()

    self.d_model = d_model
    self.num_layers = num_layers

    self.pos_embedding = PositionalEmbedding(vocab_size=vocab_size,
                                             d_model=d_model)
    self.dropout = tf.keras.layers.Dropout(dropout_rate)
    self.dec_layers = [
        DecoderLayer(d_model=d_model, num_heads=num_heads,
                     dff=dff, dropout_rate=dropout_rate)
        for _ in range(num_layers)]

    self.last_attn_scores = None

  def call(self, x, context):
    # `x` is token-IDs shape (batch, target_seq_len)
    x = self.pos_embedding(x)  # (batch_size, target_seq_len, d_model)

    x = self.dropout(x)

    for i in range(self.num_layers):
      x  = self.dec_layers[i](x, context)

    self.last_attn_scores = self.dec_layers[-1].last_attn_scores

    # The shape of x is (batch_size, target_seq_len, d_model).
    return x
```

- Attention: Mecanismo clave que permite relacionar tokens distantes en las frases mediante el cálculo de *queries*, *keys* y *values*. Existen varias variantes como *self-attention*, cruzada, causal, etc.

```
class BaseAttention(tf.keras.layers.Layer):
  def __init__(self, **kwargs):
    super().__init__()
    self.mha = tf.keras.layers.MultiHeadAttention(**kwargs)
    self.layernorm = tf.keras.layers.LayerNormalization()
    self.add = tf.keras.layers.Add()


class CrossAttention(BaseAttention):
  def call(self, x, context):
    attn_output, attn_scores = self.mha(
        query=x,
        key=context,
        value=context,
        return_attention_scores=True)

    # Cache the attention scores for plotting later.
    self.last_attn_scores = attn_scores

    x = self.add([x, attn_output])
    x = self.layernorm(x)

    return x


class GlobalSelfAttention(BaseAttention):
  def call(self, x):
    attn_output = self.mha(
        query=x,
        value=x,
        key=x)
    x = self.add([x, attn_output])
    x = self.layernorm(x)
    return x
```

- <u>Optimización</u>: Entrena el modelo para predecir la traducción correcta de cada frase de entrada mediante *Adam optimizer* y *learning rate schedule*. Utiliza máscara en la *loss* para ignorar tokens de *padding*.

```
class CustomSchedule(tf.keras.optimizers.schedules.LearningRateSchedule):
  def __init__(self, d_model, warmup_steps=4000):
    super().__init__()

    self.d_model = d_model
    self.d_model = tf.cast(self.d_model, tf.float32)

    self.warmup_steps = warmup_steps

  def __call__(self, step):
    step = tf.cast(step, dtype=tf.float32)
    arg1 = tf.math.rsqrt(step)
    arg2 = step * (self.warmup_steps ** -1.5)

    return tf.math.rsqrt(self.d_model) * tf.math.minimum(arg1, arg2)

learning_rate = CustomSchedule(d_model)

optimizer = tf.keras.optimizers.Adam(learning_rate, beta_1=0.9,
beta_2=0.98, epsilon=1e-9)
```

- <u>Inferencia</u>: Para traducir una nueva frase, se pasa por el *encoder* y luego el *decoder* predice la traducción paso a paso hasta generar el token [END].

```
class Translator(tf.Module):
  def __init__(self, tokenizers, transformer):
    self.tokenizers = tokenizers
    self.transformer = transformer

  def __call__(self, sentence, max_length=MAX_TOKENS):
    assert isinstance(sentence, tf.Tensor)
    if len(sentence.shape) == 0:
      sentence = sentence[tf.newaxis]

    sentence = self.tokenizers.pt.tokenize(sentence).to_tensor()

    encoder_input = sentence

    start_end = self.tokenizers.en.tokenize([''])[0]
    start = start_end[0][tf.newaxis]
    end = start_end[1][tf.newaxis]
```

```
    # `tf.TensorArray` is required here (instead of a Python list), so
that the
    # dynamic-loop can be traced by `tf.function`.
    output_array = tf.TensorArray(dtype=tf.int64, size=0,
dynamic_size=True)
    output_array = output_array.write(0, start)

    for i in tf.range(max_length):
      output = tf.transpose(output_array.stack())
      predictions = self.transformer([encoder_input, output],
training=False)

      # Select the last token from the `seq_len` dimension.
      predictions = predictions[:, -1:, :]  # Shape `(batch_size, 1,
vocab_size)`.

      predicted_id = tf.argmax(predictions, axis=-1)


      # decoder as its input.
      output_array = output_array.write(i+1, predicted_id[0])

      if predicted_id == end:
        break

    output = tf.transpose(output_array.stack())
    # The output shape is `(1, tokens)`.
    text = tokenizers.en.detokenize(output)[0]  # Shape: `()`.

    tokens = tokenizers.en.lookup(output)[0]

    # `tf.function` prevents us from using the attention_weights that
were
    # calculated on the last iteration of the loop.
    # So, recalculate them outside the loop.
    self.transformer([encoder_input, output[:,:-1]], training=False)
    attention_weights = self.transformer.decoder.last_attn_scores

    return text, tokens, attention_weights
```

En esta fase también habrá que ir ajustando los diferentes parámetros del modelo para obtener una mejor salida y unas predicciones acertadas. Los hiperparámetros que hay que revisar son los siguientes:

Buffer size → Tamaño del buffer para shuffling durante el entrenamiento.

Batch size → Número de muestras por batch.

Num layers → Número de capas.

D model → Dimensión de los vectores internos.

Num heads → Número de cabezas (atención paralel

Dff → Dimensión del feed forward layer.

Vocab size → Tamaño del vocabulario.

Dropout rate → Tasa de dropout para regularización.

Key dim → Dimensión del vector key utilizado para calcular las puntaciones de atención.

Learning rate → Tasa de aprendizaje.

Beta 1 → Momentum para estimar el primer momento.

Beta 2 → Momentum para estimar el segundo momento.

Epsilon → Para evitar división por cero.

Decay → Tasa de decay del learning rate.

Epochs → Número de pases completos del algoritmo de entrenamiento sobre el conjunto de datos.

2.3.3.2 Redes para la clasificación de rostros.

Para crear un modelo de red neuronal para el reconocimiento facial, se utilizará un modelo de ejemplo de *TensorFlow* de clasificación de imágenes (Clasificacion de imágenes, s.f.). Este modelo se ha diseñado para cargar eficientemente los datos de más de 30000 imágenes de flores desde el disco y para identificar y mitigar el sobreajuste. En nuestro caso, tendremos que cargar y crear un conjunto de datos de los rostros a clasificar.

El modelo se construirá utilizando el siguiente flujo de trabajo básico de aprendizaje automático:

1. **Examinar y comprender el conjunto de datos**: Se evaluará el conjunto de datos para determinar su tamaño, distribución y calidad.
2. **Construir una canalización de entrada**: Se creará una canalización que permita cargar los datos del conjunto de datos de forma eficiente.
3. **Construir el modelo**: Se diseñará un modelo de red neuronal que sea adecuado para la tarea de reconocimiento facial.
4. **Entrenar el modelo**: Se entrenará el modelo utilizando el conjunto de datos de entrenamiento.

5. **Probar el modelo**: Se evaluará el rendimiento del modelo utilizando el conjunto de datos de prueba.

6. **Mejorar el modelo**: Si el rendimiento del modelo no es satisfactorio, se pueden realizar mejoras al modelo o al conjunto de datos.

Todo el código utilizado para esta parte se encuentra en los Anexos.

El nuevo conjunto de datos lo hemos creado en una carpeta llamada "faces", la cual podemos encontrarnos imágenes de dos categorías: *adrian* y *otras*. En *adrian* hay imágenes mías, que son con las que el sistema será capaz de identificar al usuario, mientras que en la carpeta *otras* se encuentran imágenes de otras personas, para que el sistema sea capaz de identificar que, efectivamente, esas personas no son el usuario. A diferencia del gran número de imágenes que se cargan con el dataset de TensorFlow, en este caso, la carpeta *adrian* solo hay 13 imágenes de mi cara y en la carpeta *otras* hay 150 imágenes de otros rostros randoms, generados por inteligencia artificial [Referencia] (esas caras no existen).

Una vez ya tenemos una buena recopilación de rostros, podemos empezar con esta red tipo Transformer.

- Importamos y cargamos los datos

```
data_dir = pathlib.Path('/Users/adria/Documents/Faces')
```

- Definimos varios parámetros para el procesamiento del conjunto de datos, en este caso las imágenes serán de una dimensión 500 x 500 para que el modelo las emplee en el entrenamiento y validación.

```
batch_size = 150
img_height = 500
img_width = 500
```

- El 80% de las imágenes va destinado a entrenamiento, mientras que el 20% restante se empleará para validar el modelo.

```
train_ds = tf.keras.utils.image_dataset_from_directory(
  data_dir,
  validation_split=0.2,
  subset="training",
  seed=123,
  image_size=(img_height, img_width),
  batch_size=batch_size)
```

```
val_ds = tf.keras.utils.image_dataset_from_directory(
  data_dir,
  validation_split=0.2,
  subset="validation",
  seed=123,
  image_size=(img_height, img_width),
  batch_size=batch_size)
```

- Se deben estandarizar los datos de entrada, ya que los valores del canal RGB de las imágenes están en el rango [0, 255] y una red neuronal debe de tener unos valores de entrada pequeños para un mayor rendimiento. Por ello, se estandarizan los valores para el rango [0, 1].

```
normalization_layer = layers.Rescaling(1./255)
normalized_ds = train_ds.map(lambda x, y: (normalization_layer(x), y))
image_batch, labels_batch = next(iter(normalized_ds))
first_image = image_batch[0]
```

- Creamos y compilamos el modelo. El modelo secuencial consta de tres bloques de convolución con una capa de agrupación máxima en cada uno de ellos. Hay una capa totalmente conectada con 128 unidades encima que se activa mediante una función de activación de ReLU. Para este modelo, se ha elegido optimizador Adams.

```
num_classes = len(class_names)

model = Sequential([
  layers.Rescaling(1./255, input_shape=(img_height, img_width, 3)),
  layers.Conv2D(16, 3, padding='same', activation='relu'),
  layers.MaxPooling2D(),
  layers.Conv2D(32, 3, padding='same', activation='relu'),
  layers.MaxPooling2D(),
  layers.Conv2D(64, 3, padding='same', activation='relu'),
  layers.MaxPooling2D(),
  layers.Flatten(),
  layers.Dense(128, activation='relu'),
  layers.Dense(num_classes)
])

model.compile(optimizer='adam',
              loss=tf.keras.losses.SparseCategoricalCrossentropy(from_log
its=True),
              metrics=['accuracy'])
```

- Entrenamos el modelo.

```
epochs=10
history = model.fit(
  train_ds,
  validation_data=val_ds,
  epochs=epochs
)

acc = history.history['accuracy']
val_acc = history.history['val_accuracy']

loss = history.history['loss']
val_loss = history.history['val_loss']

epochs_range = range(epochs)
```

- Para comprobar el funcionamiento del modelo, ahora cargamos una imagen que no esté dentro del conjunto de datos con el que hemos entrenado y validado el modelo.

```
from PIL import Image

# Ruta de la imagen
image_path = '/Users/adria/Descargas/foto1.jpg'

# Cargar la imagen usando Pillow
img = Image.open(image_path)
img = img.resize((img_height, img_width))  # Ajustar al tamaño objetivo

img_array = tf.keras.utils.img_to_array(img)
img_array = tf.expand_dims(img_array, 0) # Crear un lote (batch)

# Resto del código (suponiendo que `model` y `class_names` ya están
definidos)
predictions = model.predict(img_array)
score = tf.nn.softmax(predictions[0])

print(
    "Esta imagen probablemente pertenece a {} con una confianza del
{:.2f} por ciento."
    .format(class_names[np.argmax(score)], 100 * np.max(score))
)
```

- Podremos comprobar si el modelo es bueno o mala fijándonos que imprime por pantalla, ya que nos dirá si la imagen está en *adrian* o en *otras*, junto con un porcentaje de confianza

2.3.3.3 Reconocimiento de gestos mediante MediaPipe

Para la detección de gestos, he utilizado una herramienta de *MediaPipe* (*MediaPipe Hands*) comentada en el punto 1.1.6. Basándonos en los landmarks de la mano (Figura 9), he hecho un código en Python, el cual detecte cuando la palma de la mano está abierta (apuntando a la cámara del robot) y he asignado ese gesto a que el robot tiene que parar.

Para realizar lo comentado:

- Lo primero que hay que hacer es importar MediaPipe en el código, para poder utilizar todas sus herramientas.
- Definimos la función gesto parar.

```
def gesto_parar():
    print("Gesto parar detectado")
    # Gesto con la mano abierta
```

- Abrimos la cámara del dispositivo para poder capturar el video.

```
cap = cv2.VideoCapture(0)
```

- Inicializamos todos los parámetros de la herramienta Hands, configuramos la imagen de video que nos saldrá, obtenemos las coordenadas de los landmarks necesarios para ese gesto, guardamos las coordenadas en una variable y asociamos el gesto cuando dichos landmarks se encuentren en cierta posición.

```
with mp_hands.Hands(
    model_complexity=0,
    min_detection_confidence=0.5,
    min_tracking_confidence=0.5) as hands:
  while cap.isOpened():
    success, image = cap.read()
    if not success:
      print("Ignoring empty camera frame.")
      # If loading a video, use 'break' instead of 'continue'.
      continue
```

```
    # To improve performance, optionally mark the image as not writable
to
    # pass by reference.
    image.flags.writeable = False
    image = cv2.cvtColor(image, cv2.COLOR_BGR2RGB)
    results = hands.process(image)

    # Draw the hand annotations on the image.
    image.flags.writeable = True
    image = cv2.cvtColor(image, cv2.COLOR_RGB2BGR)
    if results.multi_hand_landmarks:
      for hand_landmarks in results.multi_hand_landmarks:
        # Obtener las coordenadas de los puntos clave
        pinky_tip_x =
hand_landmarks.landmark[mp_hands.HandLandmark.PINKY_TIP].x
        pinky_tip_y =
hand_landmarks.landmark[mp_hands.HandLandmark.PINKY_TIP].y
        pinky_dip_y =
hand_landmarks.landmark[mp_hands.HandLandmark.PINKY_DIP].y
        thumb_tip_x =
hand_landmarks.landmark[mp_hands.HandLandmark.THUMB_TIP].x
        index_finger_dip_y =
hand_landmarks.landmark[mp_hands.HandLandmark.INDEX_FINGER_DIP].y
        index_finger_tip_y =
hand_landmarks.landmark[mp_hands.HandLandmark.INDEX_FINGER_TIP].y
        middle_finger_dip_y =
hand_landmarks.landmark[mp_hands.HandLandmark.MIDDLE_FINGER_DIP].y
        middle_finger_tip_y =
hand_landmarks.landmark[mp_hands.HandLandmark.MIDDLE_FINGER_TIP].y
        ring_finger_dip_y =
hand_landmarks.landmark[mp_hands.HandLandmark.RING_FINGER_DIP].y
        ring_finger_tip_y =
hand_landmarks.landmark[mp_hands.HandLandmark.RING_FINGER_TIP].y


        # Asociar gestos con funciones
        if (middle_finger_tip_y < pinky_tip_y) & (middle_finger_tip_y <
wrist_y) & (thumb_tip_x > pinky_tip_x) & (middle_finger_tip_y <
middle_finger_dip_y) & (index_finger_tip_y < index_finger_dip_y) &
(ring_finger_tip_y < ring_finger_dip_y) & (pinky_tip_y < pinky_dip_y):
            gesto_parar()


        # Dibujar los puntos clave y las conexiones en la imagen
        mp_drawing.draw_landmarks(
            image,
            hand_landmarks,
            mp_hands.HAND_CONNECTIONS,
```

```
            mp_drawing_styles.get_default_hand_landmarks_style(),
            mp_drawing_styles.get_default_hand_connections_style())

    # Flip the image horizontally for a selfie-view display.
    cv2.imshow('MediaPipe Hands', cv2.flip(image, 1))
    if cv2.waitKey(5) & 0xFF == 27:
      break
```

- Al igual que se puede crear el gesto parar, podemos asociar cualquier posición de la mano (configurando la condición del if) a otra acción que podría hacer el robot

2.3.3.4 Integración de componentes con la interfaz natural

Una vez explicados los pasos y códigos utilizados para la interacción humano-robot, así como para la navegación, es el momento de explicar cómo se podrían combinar.

Yo he utilizado los mismos códigos expuestos anteriormente, pero con pequeñas modificaciones para que el robot reaccione una vez que nosotros le enviemos una consigna en forma de gesto, imagen o habla.

En el caso del procesamiento del lenguaje natural, hay que modificar la entrada (input) del modelo, para que, en vez de ser de forma escrita, sea reconocida y sintetizada mediante la voz, con las librerías **Vosk**, **Sounddevice** y **Pyttsx3.** Una vez hecho esto, el modelo nos dará una predicción y esa predicción estará vinculada con un diccionario, donde para cada predicción están relacionadas unas coordenadas del mapa. Esto se ha implementado dentro del mismo código del Transformer del PLN para que dependiendo de la predicción que nos del modelo, envíe todos los *topics* relacionados con la planificación de trayectorias (*move_base*), y más concretamente el topic */goal* para saber el punto donde tiene que ir.

La integración del reconocimiento de gestos también es muy sencilla, ya que lo que hay que hacer es implementar la herramienta en la cámara del robot y cuando reconozca algún gesto, este vaya ligado a algún topic del robot para que este pueda realizar la función asignada. Por ejemplo, yo he implementado la función de *gesto_parar()* y una vez que, gracias a los landmarks de la mano, la cámara detecte que estamos haciendo ese gesto, le enviamos al topic */cmd_vel* del robot el valor 0 para que se pare.

Todo lo anterior es compatible con la navegación, ya que el archivo *todo_planificacion.launch* se lanzará en el pc y tras él, se lanzarán los archivos del procesamiento del lenguaje natural, así como el de gestos, para que cuando le hablemos al robot sepa dónde ir y cuando le hagamos algún gesto reaccione consecuentemente.

Capítulo 3. Análisis de resultados.

En este capítulo se presenta el análisis de los resultados obtenidos en las pruebas de navegación realizadas con el robot real y de forma simulada. En ambos casos, las pruebas se han realizado en un entorno cerrado y con obstáculos.

3.1 Pruebas realizadas.

Para un completo análisis de todos los factores a tener en cuenta para tener una interacción humano-robot optima, se han hecho una serie de pruebas que se dividen en:

- Pruebas para la interfaz natural, que abarcan la manera en la que he hecho todo lo relacionado con el procesamiento de lenguaje natural y el reconocimiento de rostros y de gestos.
- Pruebas para la navegación del robot, que evalúan la capacidad del robot para moverse de forma autónoma por su entorno.
- Pruebas de interconexión, que analizan la capacidad en la que se han interconectado ambas.

3.1.1 Pruebas de la interfaz natural

A continuación, expondré de forma dinámica las pruebas que he realizado, para que veáis que todos los algoritmos, estructuras y códigos explicados anteriormente funcionan perfectamente y pueden ser aplicados, no solo en este proyecto, sino en cualquier otro en el que sean necesarios.

3.1.1.1 Pruebas del procesamiento del lenguaje natural

Una vez explicado el código de la estructura Transformer, para el procesamiento del lenguaje natural, y las librerías necesarias para la síntesis y reconocimiento de la voz. Tenemos que ir ejecutando el modelo para ver si sus predicciones son buenas.

Al lanzar el modelo, vemos que nos ofrece predicciones muy malas en las primeras pruebas, y esto es debido a que el dataset que he creado es muy pequeño comparado con el modelo de prueba que nos ofrece *TensorFlow*, y, por tanto, los parámetros del código están puestos para ese tipo de dataset. Por lo que, para datasets pequeños nos saldrán pésimos resultados.

A continuación, mostraré las reglas que he seguido yo, en los hiperparámetros de cada parte del código:

- Para el dataset:
  - buffer_size: Igual o parecido al tamaño del dataset, ≈ 210
  - batch_size: Igual o parecido al tamaño del dataset, ≈ 210
  - num_layers: 2 o 3 capas
  - d_model: 64
  - num_heads: 4
  - dff: 128
  - vocab_size: Según el vocabulario, pero valores pequeños como 2000-5000
  - dropout_rate: 0.3 o 0.4

- Para el codificador (Encoder):
  - num_layers: 2
  - d_model: 64
  - num_heads: 2
  - dff: 128
  - vocab_size: input vocab size, ej. 4000

- Para el decodificador (Decoder):
  - num_layers: 2
  - d_model: 64
  - num_heads: 2
  - dff: 128
  - vocab_size: target vocab size, ej. 5000

- Para el optimizador Adams:
  - learning_rate: Un valor pequeño como 1e-3 o 1e-4. Con datasets pequeños un LR alto puede inestabilizar el entrenamiento.
  - beta_1: El valor default de 0.9 está bien.

- beta_2: El valor default de 0.999 también es bueno. Se puede probar con 0.98 como está en el código también.
- epsilon: 1e-7 u 1e-9 está bien. Este valor previene división por cero.
- decay: No es necesario decay con datasets pequeños. Se puede omitir.

Y el parámetro que más me ha influenciado en los resultados del modelo es *epochs* (épocas en español), indica el número de pases completos del algoritmo de entrenamiento sobre el conjunto de datos y hace que para datasets pequeños se necesiten más épocas para converger. A este parámetro le he dado un valor de 300.

Una vez modificados estos parámetros vemos que la predicción se ajusta a lo que queremos que haga, como podemos observar en las capturas siguientes.

```
Input:          : Robot, lleva esta pieza a la zona cuatro
Prediction      : ir zona cuatro
Ground truth    : Ir zona cuatro.
```

Figura 20. Captura del resultado de la prueba de procesamiento del lenguaje natural (1).

```
Input:          : No vayas a la zona seis, mejor acercate a la zona uno
Prediction      : ir zona uno
Ground truth    : Ir zona uno.
```

Figura 21. Captura del resultado de la prueba de procesamiento del lenguaje natural (2).

```
Input:          : Ve a la zona tres en vez de ir a la zona cuatro
Prediction      : ir zona tres
Ground truth    : Ir zona tres.
```

Figura 22. Captura del resultado de la prueba de procesamiento del lenguaje natural (3).

```
Input:          : En la zona seis está el documento, ve a por el
Prediction      : ir zona seis
Ground truth    : Ir zona seis.
```

Figura 23. Captura del resultado de la prueba de procesamiento del lenguaje natural (4).

3.1.1.2 Pruebas de reconocimiento visual

Para hacer pruebas al Transformer, se emplearán imágenes que el modelo no conoce, ya que estas no están incluidas en el conjunto de datos con el que se entrena y valida el modelo.

Debido a que no tenemos un conjunto de imágenes muy amplio, tras varios intentos fallidos, modificando parámetros del modelo, el modelo está listo y funciona correctamente.

Primero se le aporta una imagen mostrada en la Figura 24. Este rostro es de una persona que no soy yo, por lo que le debería corresponder a la clasificación *otras.*

Figura 24. Foto1.jpg para el reconocimiento facial.

```
1/1 [==============================] - 0s 43ms/step
Esta imagen probablemente pertenece a otras con una confianza del 55.08 por ciento.
```

Figura 25. Captura del resultado de la prueba de reconocimiento facial con la foto1.jpg.

El modelo clasifica la imagen foto1.jpg como *otras* con un 55.08% de confianza (ver Figura 25), lo que es correcto.

Si probamos con otro rostro (ver Figura 26) obtenemos que dicho rostro pertenece en un 69.46 % de confianza a *otras*.

Figura 26. Foto3.jpg para el reconocimiento facial.

```
1/1 [==============================] - 0s 70ms/step
Esta imagen probablemente pertenece a otras con una confianza del 69.46 por ciento.
```

Figura 27. Captura del resultado de la prueba de reconocimiento facial con la foto3.jpg.

Ahora probaré con una imagen mía (ver Figura 28), a ver que salida nos ofrece.

Figura 28. Foto2.jpg para el reconocimiento facial.

Y obtenemos una salida bastante buena como podemos ver en la siguiente captura.

```
1/1 [==============================] - 0s 148ms/step
Esta imagen probablemente pertenece a adrian con una confianza del 96.07 por ciento.
```

Figura 29. Captura del resultado de la prueba de reconocimiento facial con la foto2.jpg.

He probado con diferentes imágenes a parte de las expuestas anteriormente y he observado que con imágenes en las que salga mi rostro (*adrian*) las reconoce con un intervalo de confianza muy alto, más de un 90% en todos los casos. Pero, en el caso de imágenes con rostros diferentes al mío (*otras*), el modelo no está tan seguro y nos ofrece porcentajes más variados, desde un 55% hasta un 90% de confianza.

3.1.1.3 Pruebas de reconocimiento de gestos

Una vez lanzado el código para el reconocimiento de gestos, podremos observar que cada vez que detecte una mano la cámara nos superpondrá los landmarks, tal y como se ve en la Figura 30.

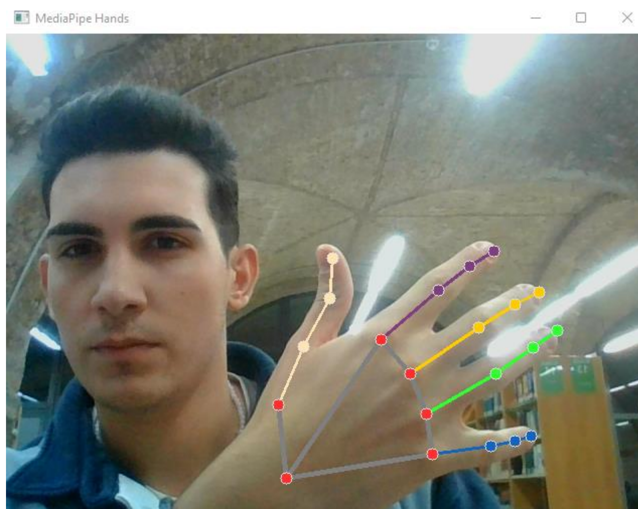

Figura 30. Landmarks en una imagen de video real.

Ahora probaré si la función *gesto_parar()* funciona correctamente cuando detecte la mano en esa posición (ver Figura 31).

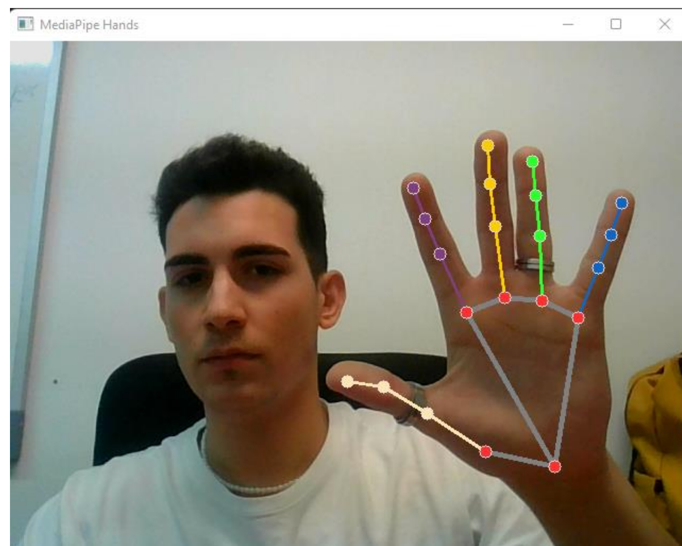

Figura 31. Mano en posición para el gesto parar.

Para ver si de verdad la detecta debería de imprimir “Gesto parar detectado” tal y como se puede observar en la siguiente captura.

```
Gesto parar detectado
Gesto parar detectado
Gesto parar detectado
Gesto parar detectado
Gesto parar detectado
Gesto parar detectado
Gesto parar detectado
Gesto parar detectado
Gesto parar detectado
Gesto parar detectado
```

Figura 32. Captura de la salida del código de reconocimiento de gestos.

Para comprobar que no siempre sale por pantalla “Gesto parar detectado”, he creado otra función llamada *gesto_afirmativo()* y he programado la condición de los landmarks para que cuando estén en esa posición llamen a la función *gesto_afirmativo()* (ver Figura 33 y Figura 34)

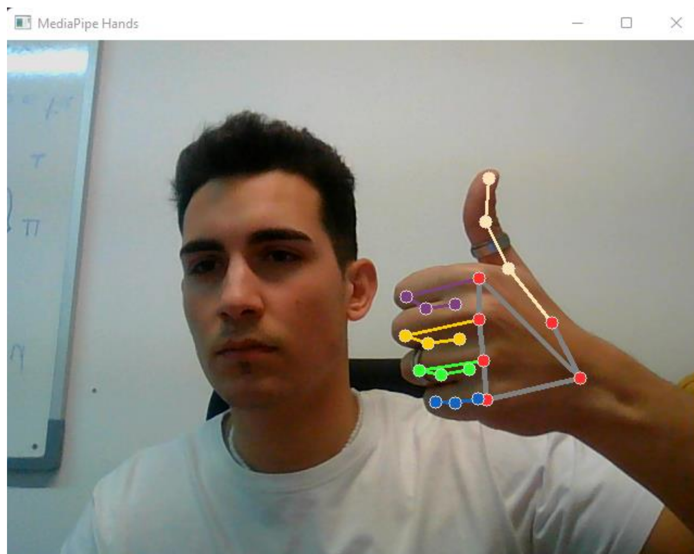

Figura 33. Mano en posición para el gesto afirmativo.

```
Gesto afirmativo detectado
Gesto afirmativo detectado
Gesto afirmativo detectado
Gesto afirmativo detectado
Gesto afirmativo detectado
Gesto afirmativo detectado
Gesto afirmativo detectado
Gesto afirmativo detectado
Gesto afirmativo detectado
Gesto afirmativo detectado
```

Figura 34. Captura de la salida del código de reconocimiento de gestos (2).

Como podemos observar cuando realizamos el gesto afirmativo, como se ve en la Figura 33, obtenemos de salida la captura anterior.

3.1.2 Pruebas de navegación.

Este punto contendrá las explicaciones de la forma en la que he realizado las pruebas de navegación, tanto de forma virtual, como en el robot real.

3.1.2.1 Pruebas de navegación en GAZEBO.

En el punto 2.3.2 Navegación del robot, he descrito todo el tema más teórico en el asunto de navegación y ahora ese contenido será apoyado por capturas para contextualizarlo y ver qué resultados obtenemos al hacer dichos pasos en Gazebo.

1. Lanzamos la simulación del entorno y la herramienta *rviz*, junto a *rqt* para tele operar al robot.

```
~$ roslaunch robot_gazebo mibot.launch
```

```
~$ rosrun rviz rviz
```

```
~$ rqt
```

2. Configuramos rviz (ver Figura 35), para ver cómo se va creando el mapa, añadiendo los topics */LaserScan*, /RobotModel y */*Map, que este nos debe aparecer una vez hayamos lanzado el nodo *slam_gmapping*.

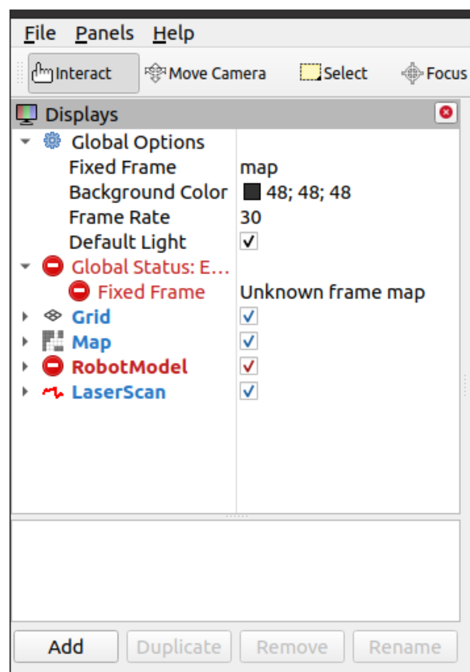

Figura 35. Configuración *rviz* para crear el mapa métrico.

3. Lanzamos el nodo *slam_gmapping* del paquete *gmapping*.

```
~$ gmapping slam_gmapping _base_frame:=baselink
```

4. Comenzamos a tele operar el robot (Vemos que se va creando el mapa). Vamos moviendo el robot por todo el mapa, para que lo escanee en su totalidad (ver Figura 36).

Figura 36. Mapa totalmente escaneado.

5. Guardamos el mapa utilizando el paquete *map_server*.

```
~$ rosrun map_server map_saver -f mapanave
```

6. Una vez que se tiene el mapa, utilizamos un paquete de Navigation, el paquete *amcl*

7. Configuramos los diferentes archivos *launch* del paquete nombrados en el punto 2.3.2.1 (ver Figura 37).

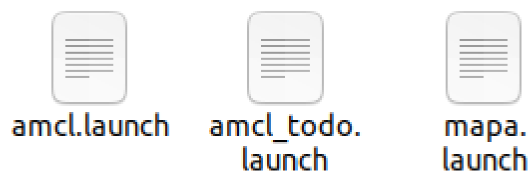

Figura 37. Archivos *launch* necesarios para el paquete *amcl*.

8. Ahora necesitamos añadir más topics en *rviz*, a parte del láser, el mapa y el modelo del robot, añadiremos los topics de la pose con covarianza y el array de partículas, */PoseWithCovariance* y */PoseArray*, respectivamente (ver Figura 38).

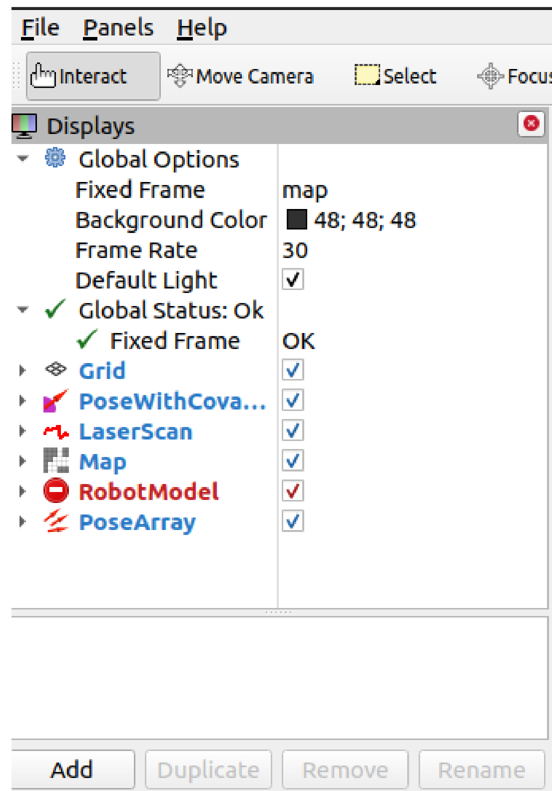

Figura 38. Configuración *rviz* para el paquete *amcl*.

9. Lanzamos el archivo *amcl_todo.launch*.

```
~$ roslaunch robot_gazebo amcl_todo.launch
```

10. Ahora teleoperamos el robot y vemos cómo se comporta la nube de partículas. Observamos que al principio el robot no sabe exactamente en qué zona del mapa se encuentra, pero al cabo de unos segundos, cuando se va moviendo y analizando el entorno, la precisión con la que él sabe en qué parte del mapa está es más exacta. Lo podemos ver en la Figura 39 y Figura 40.

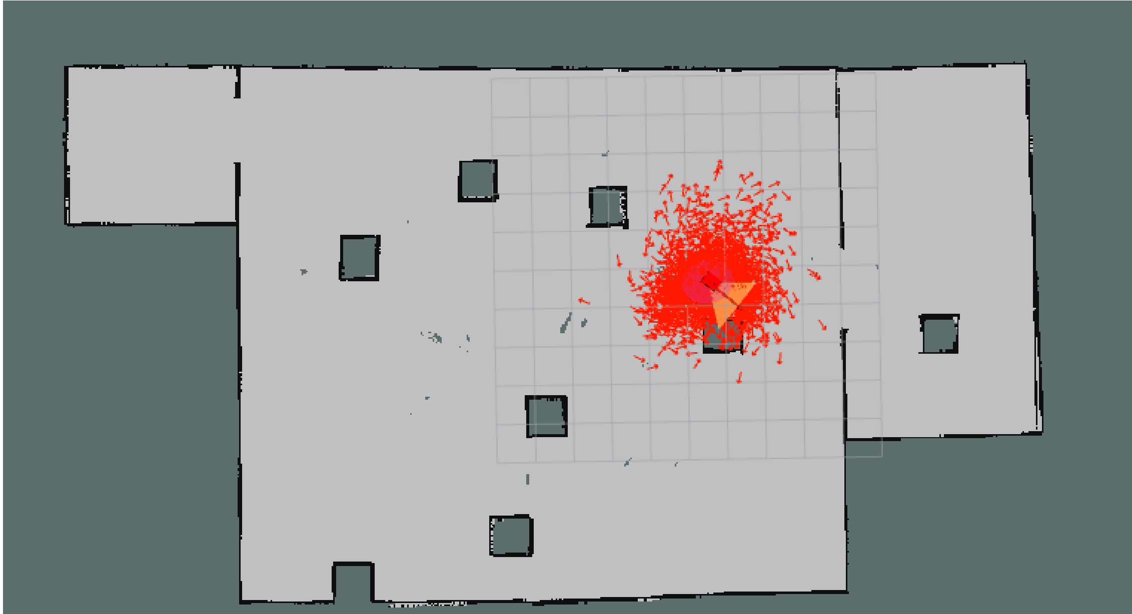

Figura 39. Localización del robot al iniciar el paquete *amcl*.

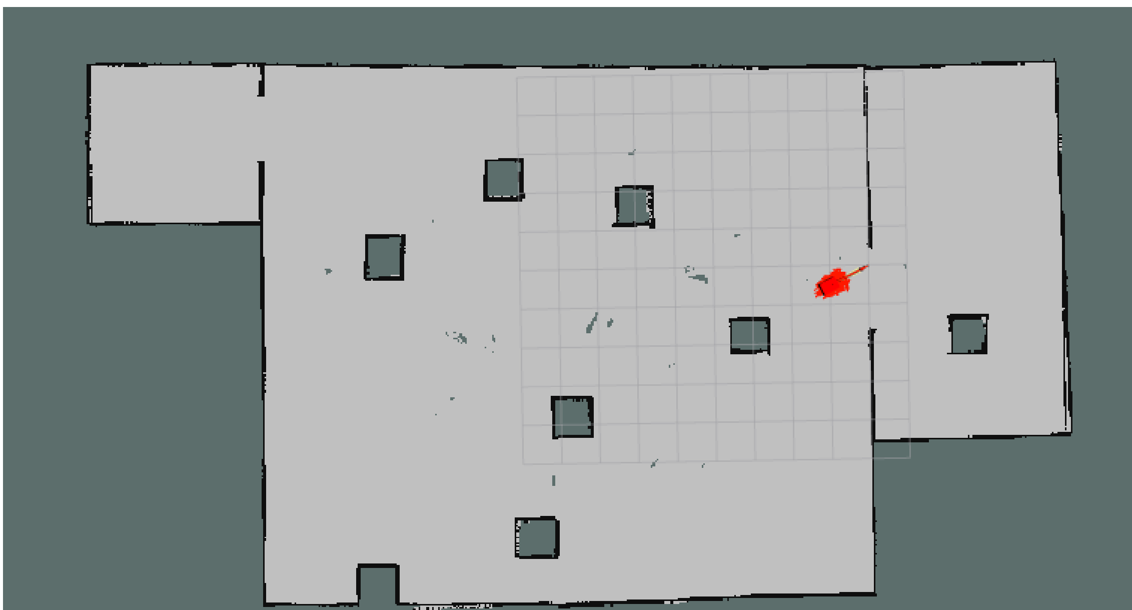

Figura 40. Localización del robot al analizar el entorno.

11. Para realizar tareas de planificación tenemos el paquete *move_base*. Se usa de forma similar a AMCL, pero en este caso hay que crear una carpeta, llamada *config*, dentro del paquete *robot_gazebo*, e incluir los 4 archivos de configuración mencionados en el punto 2.3.2.1. También crear los 2 archivos *launch* correspondientes (ver Figura 42).

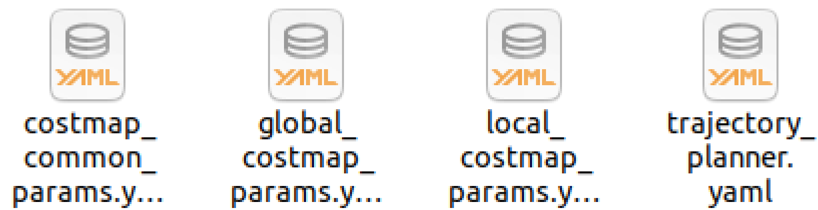

Figura 41. Archivos de configuración necesarios para el paquete *move_base*.

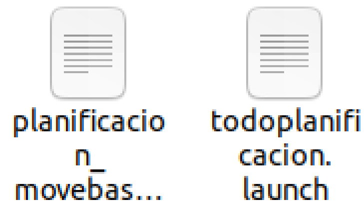

Figura 42. Archivos *launch* necesarios para el paquete *move_base*.

12. Lanzamos el archivo *todo_planificacion.launch*.

```
~$ roslaunch robot_gazebo todo_planificacion.launch
```

13. Añadimos en rviz los topics anteriores más los destinados hacia la planificación que son */Path*, tanto para el planificador global como local, */Map*, tanto para el planificador global como local y */Polygon* (ver Figura 43).

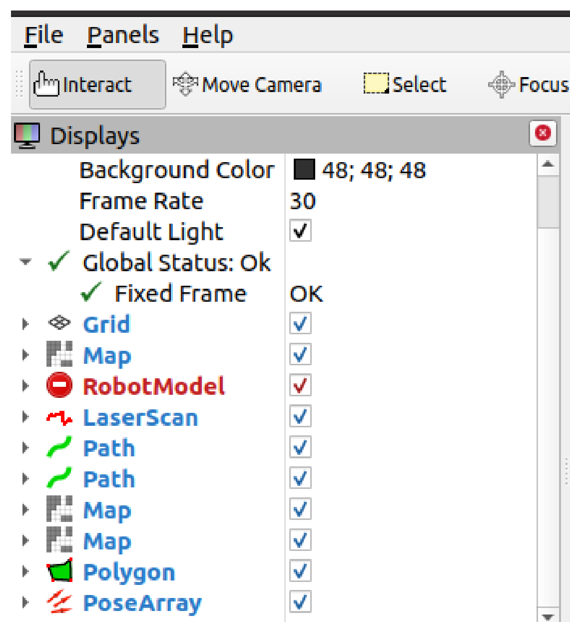

Figura 43. Configuración de *rviz* para el paquete *move_base*.

14. Para terminar, marcamos en *rviz* un punto del mapa, pinchando anteriormente en “2D Nav Goal”, y vemos si de forma autónoma el robot consigue llegar, si el proceso se realiza incorrectamente debemos de revisar los parámetros de configuración.

3.1.2.2 Pruebas de navegación en el robot real.

Para realizar ensayos en un entorno real, se han aplicado los pasos ya explicados en el capítulo anterior, pero esta vez en un marco donde se ha podido observar cómo ha sido el funcionamiento de la navegación cuando se aplica sobre un robot real. Estas pruebas se han hecho en una parte del edificio de industriales de la Universidad Politécnica de Cartagena (ver Figura 44, Figura 45 y Figura 46).

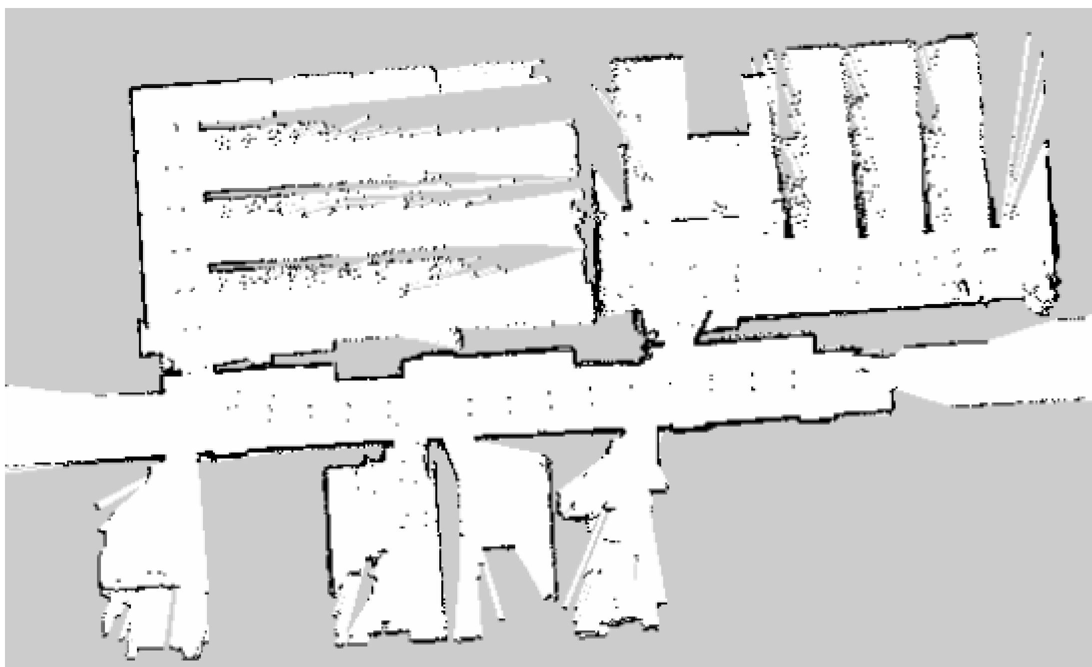

Figura 44. Mapeado del entorno real.

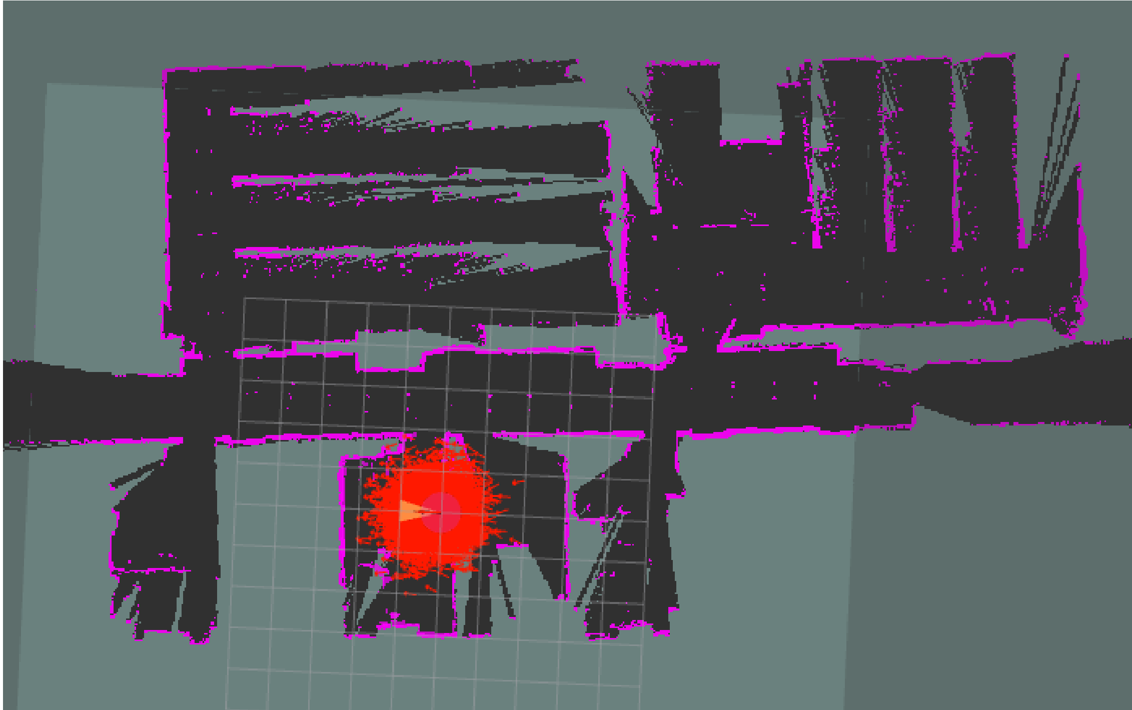

Figura 45. Nube de partículas al iniciar el paquete de navegación en el robot real.

Figura 46. Localización del robot real una vez estudiado el entorno.

Como podemos observar, el paquete *amcl* funciona perfectamente y el robot se ubica bien una vez examinado el entorno.

3.1.3 Integración de la interfaz natural con el sistema de navegación.

Como ya sabemos, gracias al punto 2.3.3.4, el proceso de integración del PLN y de los gestos con la navegación, consta en añadir a los códigos expuestos, tanto en el punto 2.3.3.1 como en el 2.3.3.3, de los *topics* de la navegación necesarios para hacer que el robot sea capaz de reaccionar dependiendo de lo que le llegue a través de esos *topics*.

En el caso del **procesamiento del lenguaje natural**, en vez de añadir de forma manual las entradas se ha implementado el siguiente código para que procese y sintetice la voz a través del micrófono del robot y la predicción obtenida por el modelo la pase al diccionario de zonas creado para saber a qué coordenadas del mapa corresponde cada zona y enviar al topic */goal* de la navegación esa dirección.

```python
# Diccionario que asigna nombres de zonas a coordenadas (x, y)
zonas = {
    "ir zona uno": (0.0, 0.0),
    "ir zona dos": (5.94, -2.52),
    "ir zona tres": (5.84, -4.93),
    "ir zona cuatro": (-8.26, -3.44),
    "ir zona cinco": (-5.92, -5.8),
    "ir zona seis": (6.18, -8.6),
    "ir zona siete": (-12.2, -6.32)
}

def move_to_zone(texto_zon):
    client = actionlib.SimpleActionClient('move_base', MoveBaseAction)
    client.wait_for_server()

    # Obtener las coordenadas asociadas con el nombre de la zona
    x, y = zonas.get(texto_zon, (0.0, 0.0))

    goal = MoveBaseGoal()
    goal.target_pose.header.frame_id = "map"
    goal.target_pose.pose.position.x = x
    goal.target_pose.pose.position.y = y
    goal.target_pose.pose.orientation.w = 1.0
    print("Sending goal to x=%.2f, y=%.2f" % (x, y))
    print("Sending goal to %s" % goal.target_pose.pose.position)

    client.send_goal(goal)
    rospy.sleep(0.1)
    client.wait_for_result()
```

```
def print_translation(sentence, tokens, ground_truth):
  print(f'{"Input:":15s}: {sentence}')
  print(f'{"Prediction":15s}: {tokens.numpy().decode("utf-8")}')
  print(f'{"Ground truth":15s}: {ground_truth}')

import rospy
from std_msgs.msg import String

sentence=""
last_sentence=""

def comandosvoz_callback(data):
    global sentence
    sentence = data.data.lower()

terminar = False

rospy.init_node('listener', anonymous=True)
# Suscribirse al topic "comandosvoz"
rospy.Subscriber("comandosvoz", String, comandosvoz_callback)

while not terminar:
   print("Comando voz: ", sentence)
   if sentence == "":
       continue
   if sentence.lower() == 'puedes salir':
       terminar = True
       sentence = last_sentence
   else:
       last_sentence = sentence
       ground_truth = input("Introduzca la verdad: ")
       translated_text, translated_tokens, attention_weights =
translator(
       tf.constant(last_sentence))
       print_translation(sentence, translated_text, ground_truth)
       texto_zon = translated_text.numpy().decode("utf-8")
       move_to_zone(texto_zon)
print("Saliendo...")
```

En esta parte del código de PLN se observa lo comentado, primero se reconoce la voz a través del micrófono del robot, que guarda esa frase en el topic */comandosvoz*, una vez tengamos la frase de entrada la enviamos a traducirla, para que el robot la analice y sepa exactamente a que zona tiene que ir, luego se asigna, gracias al diccionario creado, la predicción a unas de las coordenadas de las zonas y por último se envía el topic */goal* al robot para que genere la planificación de trayectorias hacia ese punto.

Para que sea más visual, se han tomado unas capturas de los resultados para comprender el proceso explicado. En la Figura 47, vemos que el robot está esperando un comando de voz y una vez lo tiene, procede a realizar los pasos que he nombrado anteriormente, finalizando con asignar una coordenada al topic */goal*.

```
Comando voz:
Comando voz:
Comando voz:
Comando voz:
Comando voz:
Comando voz:
Comando voz:
Comando voz:
Comando voz:
Comando voz:
Comando voz:
Comando voz:
Comando voz:
Comando voz:
Comando voz:
Comando voz:
Comando voz:
Comando voz:
Comando voz:
Introduzca la verdad: ir zona tres
Input:          : robot llevó la herramienta la zona tres
Prediction      : ir zona tres
Ground truth    : ir zona tres
Sending goal to x=5.84, y=-4.93
Sending goal to x: 5.84
y: -4.93
z: 0.0
```

Figura 47. Prueba de la integración de los comandos de voz en el PLN

En la Figura 48, se observa la parte de la interfaz gráfica *RViz*, donde se puede ver que se ha publicado el topic */goal* en dicha interfaz (flecha verde) y el robot se está dirigiendo a él.

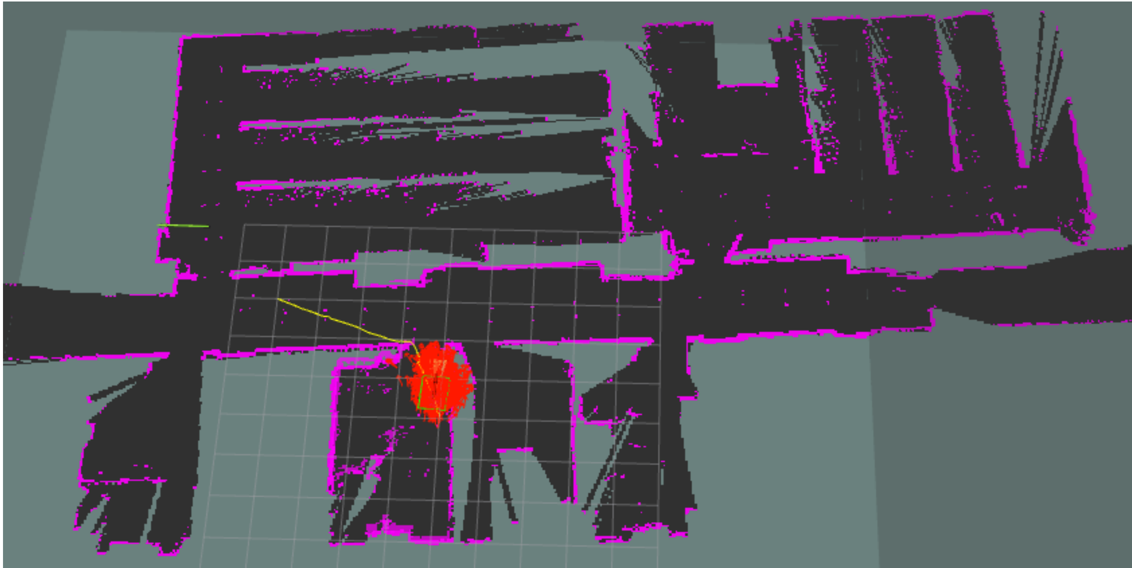

Figura 48. Herramienta RViz una vez publicado el topic */goal*.

Para la implementación del **reconocimiento de gestos** con la navegación, he llevado a cabo lo comentado anteriormente. Se ha implementado un nodo llamado "gesture_control" que se suscribe a un “topic” de imagen de una cámara y publica comandos de velocidad en otro “topic”. La comunicación entre nodos se realiza con:

1. Subscriber:

- Se suscribe al tópico *"/camera/color/image_raw*" que publica mensajes de tipo *Image* de *sensor_msgs*.
- La función "*image_callback*" se ejecuta cada vez que se recibe una nueva imagen.

2. Publisher:

- Crea un publicador "*cmd_vel_pub*" en el tópico "*cmd_vel*" de tipo *Twist* de *geometry_msgs*.
- Cuando detecta gestos, publica mensajes de velocidad 0 para detener el robot.

3. Comunicación entre nodos:

- El nodo "*gesture_control*" recibe imágenes del tópico de la cámara mediante un *Subscriber*.

- Procesa las imágenes con *OpenCV* y *MediaPipe* para detectar gestos.
- Publica comandos de velocidad en el tópico "*cmd_vel*" mediante un *Publisher*.
- Se comunica así con otro nodo que controle la velocidad del robot a través del tópico "*cmd_vel*".

3.2 Resultados.

Una vez hechas todas las pruebas, se pueden observar resultados interesantes en cada una de ellas.

3.2.1 Resultados de la interacción humano-robot.

Como ya he mostrado en el punto anterior, tras numerosos cambios y horas y horas de intentos, he obtenido resultados bastante buenos en todas las pruebas realizadas.

Las **pruebas de reconocimiento facial** han sido un éxito, aunque se podría mejorar, ya que las pruebas presentan una confianza entre el 55% y el 90%. Como se puede observar en la Figura 25 y Figura 29, atendiendo a la última época, se pueden observar unos valores de precisión de entrenamiento y de validación con una diferencia de un 45%, lo que indica la posibilidad de un sobreajuste. Este sobreajuste está más presente en la parte de los rostros *otras*, ya que cuando se trata de *adrian* el sobreajuste no supera el 10%. La presencia de sobreajuste alerta de que el conjunto de datos no es lo suficientemente amplio, esto se puede solucionar engordando dicho conjunto de datos.

Las **pruebas de reconocimiento de gestos** también han sido superadas con rigor, ya que he hecho un programa el cual es muy robusto, detectando con gran facilidad, cuando se hace el gesto de parar y envía una señal al robot con una consigna de velocidad cero para que pare.

Y, por último, con las **pruebas para el procesamiento del lenguaje natural** se obtienen predicciones exactas de lo que el robot tiene que hacer. En el PLN que le he implementado al sistema, se le puede decir al robot cualquier frase para que se dirija a la zona que queramos, que el robot lo entenderá perfectamente e irá al lugar. Un hincapié, es que el modelo de reconocimiento de voz, si no nos encontramos en un lugar con poco ruido, y no decimos la frase bien clara, tiene problemas, como se puede ver en la Figura 47, que en vez de transcribir “lleva”, ha puesto “llevó” y se ha omitido la vocal “a”, aun así, la arquitectura Transformer es tan robusta que puede predecir la salida, incluso teniendo fallos en la entrada.

3.2.2 Resultados de las pruebas de navegación.

En lo referente a las **pruebas de navegación**, la simulación en Gazebo responde adecuadamente, generando correctamente un mapa del escenario y pudiendo navegar de forma autónoma por dicho escenario mediante el uso de un mapa conocido. Por otra parte, he observado que carece de grandes resultados cuando es lanzado en un robot real, lo que dificulta una buena navegación autónoma.

Durante las pruebas reales, se han aplicado diferentes configuraciones de los parámetros *costmap_common_params*, *global_costmap_params*, *local_costmap_params* y trayectory_*planner*, para obtener mejores resultados, pero aun así no se ve una notoria mejoría.

También, resulta interesante apuntar que al robot le cuesta mucho atravesar espacios más estrechos, como puertas, ya que como pasa a unos 15 centímetros de cada lado del marco de la puerta, se piensa que va a colisionar he intenta no pasar.

3.2.3 Resultados del proceso de integración.

Los resultados obtenidos en el proceso de integración son similares a los que hemos obtenido haciendo cada prueba por separado, ya que si la implementación es buena y hemos hecho que la combinación de todos los algoritmos funcione bien, debemos obtener resultados muy parecidos. Para tener ese correcto funcionamiento y una integración optima sí que he necesitado números intentos debido a que surgen fallos en la comunicación con el robot, con la suscripción en los *topics* o en el mismo código que hay que ejecutar.

3.3 Discusión.

En este capítulo, se han presentado los resultados obtenidos en el desarrollo de un sistema de interacción humano-robot para un entorno industrial. La metodología de dividir el problema en varios códigos/módulos ha sido la adecuada porque ha permitido realizar las diferentes pruebas por separado y, por tanto, solucionar problemas de una forma más sencilla.
En general, los resultados son positivos, pero también se han identificado algunas limitaciones que deben abordarse en futuras investigaciones.

Ventajas del sistema

El sistema propuesto presenta una serie de ventajas frente a otros sistemas de interacción humano-robot existentes:

- Utiliza un enfoque multimodal, combinando el reconocimiento de imágenes, reconocimiento de gestos y procesamiento del lenguaje natural. Esto permite una interacción más natural y fluida con el usuario.
- El sistema es flexible y adaptable a diferentes entornos industriales.
- El uso de modelos de Inteligencia Artificial (concretamente Deep Learning) que no requieren de conexión a internet permite que el sistema funcione de forma autónoma en entornos industriales sin acceso a internet.
- El uso de ROS ha permitido una correcta integración de todos los componentes de una forma sencilla mediante el modo de comunicación cliente-servidor que permite sincronizar los nodos.
- Al utilizar la arquitectura Transformer para el procesamiento de lenguaje natural en vez de otro tipo de redes neuronales, se pueden procesar secuencias de cualquier longitud. Las redes neuronales recurrentes, por ejemplo, tienen problemas para procesar secuencias largas, ya que su memoria es limitada. En cambio, los Transformers utilizan un mecanismo de atención que les permite centrarse en las partes más relevantes de la secuencia, independientemente de su longitud.

Limitaciones del sistema

Las principales limitaciones del sistema propuesto son las siguientes:

- La navegación autónoma del robot en el mundo real no es tan precisa como en la simulación. Esto se debe a una serie de factores, y todos relacionados con la navegación (move_base) del robot real, ya que es incapaz de llegar al punto objetivo a pesar de que en la simulación si lo alcance.
- El robot tiene dificultades para atravesar espacios estrechos. Esto se debe a que los algoritmos de navegación consideran que los obstáculos están más cerca de lo que realmente están.
- El modelo de reconocimiento de voz tiene problemas relacionados con el ruido y en ocasiones ciertas palabras no son reconocidas en el proceso de inferencia.
- La red neuronal para el reconocimiento de personas es capaz de distinguir entre el usuario y un desconocido. Sin embargo, el hecho de diseñar el modelo de forma binaria limita al sistema a poder reconocer sólo a una persona. Para poder reconocer a más de una persona, se debería crear más carpetas de imágenes, obteniendo un modelo por persona que se desea reconocer. Esto aumentaría en gran medida el conjunto de datos y la carga de trabajo del sistema.

Capítulo 4. Conclusiones y trabajo futuro.

Para finalizar, se hará un repaso de los resultados obtenidos en el trabajo, cómo se lograron y por qué se utilizaron las metodologías descritas anteriormente. Además, se mencionarán algunos proyectos que podrían realizarse a partir de este, como añadir nuevas líneas de investigación o mejorar algunas limitaciones.

4.1 Conclusiones.

El objetivo principal del proyecto es diseñar un robot real autónomo inteligente que permita comunicarse con las personas y desplazarse por un entorno real para el transporte de mercancías o de ayuda a operarios. Este objetivo se ha cumplido, aunque con capacidades de navegación autónoma limitadas pero funcionales.

Se han dado ciertos problemas con las pruebas para el robot real, relacionados con odometría. Las pruebas en el entorno simulado nos han ayudado a ver que la planificación (move_base) funciona bien, pero cuando se aplica a un entorno real falla. Para resolver esto debemos o investigar más en move_base, añadiendo nuevos plugins, o de aplicar otros tipos de navegación como Pure Pursuit, PID Control o Visual Servoing.

Además, se ha desarrollado un módulo de red neuronal para el reconocimiento de personas. Este módulo puede distinguir entre el usuario y una persona desconocida. El uso de TensorFlow y Keras ha simplificado el desarrollo del software. Sin embargo, el modelo tiene un alto sesgo al reconocer personas que no son el usuario (*adrian*). Este sesgo podría afectar al rendimiento del modelo. Para solucionarlo, se puede aumentar el conjunto de datos de entrenamiento. Esto se puede hacer recopilando más imágenes del usuario y de otras personas que puedan encontrarse en el entorno en el que se utilizará el robot. Con esta técnica, se pueden crear modelos para diferentes usuarios. Esto permitiría al robot aprender a distinguir a varios usuarios de forma escalable.

En la parte del reconocimiento de voz han surgido algunos problemas, al no reconocer siempre todas las frases, solamente en aquellos casos que se las demos nítidas y sin ruido de fondo. Pero, esto no quita que el modelo del procesamiento del lenguaje natural, hecho con una arquitectura Transformer tenga resultados muy aceptables obteniendo un campo en el que se puede avanzar mucho.

La división del trabajo en diferentes módulos fue un punto importante en el desarrollo del robot. Esto permitió programar cada sistema de forma independiente, facilitando el trabajo de comprobación y corrección de errores.

Para integrar los diferentes componentes se utilizó ROS, que facilita la sincronización del sistema mediante una comunicación cliente-servidor. El lenguaje de programación base del proyecto era C++, pero TensorFlow utiliza Python. Por lo tanto, fue necesario aprender Python y escribir código en este lenguaje para que la implementación de TensorFlow y otros códigos, como el de MediaPipe, funcionaran correctamente.

En conclusión, se logró cumplir el objetivo principal del trabajo: construir un robot real con capacidades de navegación autónoma y de interacción con personas, aunque con algunas limitaciones.

4.2 Trabajo futuro.

Para abordar las limitaciones identificadas, se proponen las siguientes líneas de investigación:

- Mejorar la precisión de la navegación autónoma del robot en el mundo real, utilizando nuevas técnicas para adaptar los algoritmos de navegación a las condiciones reales del entorno.
- Desarrollar algoritmos de navegación que tengan en cuenta la presencia de obstáculos estrechos.
- Mejorar el rendimiento del modelo de reconocimiento de voz, utilizando técnicas de reducción de ruido o de mejora de la calidad de la señal.

Además de estas líneas de investigación específicas, también se podrían explorar otras posibilidades para mejorar el sistema propuesto, como:

- Incorporar técnicas de aprendizaje profundo para mejorar el rendimiento de los algoritmos de reconocimiento facial, reconocimiento de gestos y procesamiento del lenguaje natural.
- Desarrollar un sistema de comunicación bidireccional que permita al robot responder a las preguntas y peticiones del usuario.
- Integrar el sistema con otros sistemas industriales, como los sistemas de control de producción o los sistemas de seguridad.

En general, los resultados obtenidos en este trabajo son prometedores y demuestran el potencial de los sistemas de interacción humano-robot para mejorar la productividad y la seguridad en entornos industriales. Sin embargo, es necesario abordar las limitaciones identificadas para que el sistema pueda ser utilizado de forma segura y eficiente en entornos reales.

Referencias.

[1] Abadi M, Isard M. & Murray D. (18 de junio de 2017). *A computational model for TensorFlow: an introduction*.

[2] Alonso, J. L. (16 de junio de 2022). *Incentro*. Obtenido de https://www.incentro.com/es-ES/blog/que-es-tensorflow

[3] Cabello Pardos, E. (2004). *Técnicas de reconocimiento facial mediante redes neuronales.*

[4] Clark, M. (22 de junio de 2022). *The Verge*. Obtenido de https://www.theverge.com/2022/6/21/23177756/amazon-warehouse-robots-proteus-autonomous-cart-delivery

[5] *Clasificacion de imágenes*. (s.f.). Obtenido de TensorFlow: https://www.tensorflow.org/tutorials/images/classification?hl=es-419

[6] Cox, I. J. (25 de abril de 1988). *Blanche: Position Estimation for an Autonomous Robot Vehicle*. IEEE.

[7] Cox, W. N. (1988). Local path control for an autonomous vehicle Proceedings. En *International Conference on Robotics and Automation* (págs. 1504-1510). Philadelphia, PA, USA: IEEE.

[8] Dellaert F., Fox D., Birgard W. & Thrun S. (1999). Monte Carlo localization for mobile robots. En *IEEE International Conference on Robotics and Automation* (págs. 1322–1328).

[9] Eirea, M., & Peña, F. (2022). *Reconocimiento de rostros aplicado a robots de servicio.*

[10] Gamboa Montero, J. J. (2015). *Corrección de odometría empleando visual servoing en ROS.*

[11] *Generated Photos*. (s.f.). Obtenido de https://generated.photos/

[12] *Gesture recognition*. (s.f.). Obtenido de MediaPipe: https://developers.google.com/mediapipe/solutions/vision/gesture_recognizer#get_started

[13] Guanipa, H. D. (2022). *Comparación de modelos de Machine Learning para la clasificación de imágenes.*

[14] H. Durrant-Whyte & T. Bailey. (2006). "Simultaneous localization and mapping: part i," . En I. R. Magazine.

[15] *Instituto de ingeniería del conocimiento*. (s.f.). Obtenido de https://www.iic.uam.es/inteligencia-artificial/procesamiento-del-lenguaje-natural/

[16] Josende, A. P. (2022). *Machine learning basado en modelos transformer aplicado a la traducción automática.*

[17] Kyrarini, M. (2021). A Survey of Robots in Healthcare. *Technologies*. Obtenido de https://doi.org/10.3390/technologies9010008

[18] Mei Wang, W. D. (2018). *Deep Face Recognition: A Survey.*

[19] *Modelo transformador para la comprensión del lenguaje.* (s.f.). Obtenido de TensorFlow: https://www.tensorflow.org/text/tutorials/transformer?hl=es-419

[20] Navarro Castillo, A. (2022). *Técnicas de machine learning para procesamiento del lenguaje natural.*

[21] *Navegación*. (s.f.). Obtenido de Husarion: https://husarion.com/tutorials/ros-tutorials/9-navigation/

[22] O. Patsadu, C. Nukoolkit & B. Watanapa. (2012). Human gesture recognition using Kinect camera . En *Computer Science and Software Engineering (JCSSE).*

[23] Pérez, A. E. (6 de Julio de 2022). *OpenWebinars*. Obtenido de https://openwebinars.net/blog/robotica-movil-que-es-y-sus-aplicaciones/

[24] Pino, A. F. (16 de diciembre de 2019). *PARTICULARIDADES DE LA INTERACCIÓN HUMANO-ROBOT (HRI)*.

[25] *Repositorio UPCT*. (s.f.). Obtenido de https://repositorio.upct.es/

[26] RobotShop. (s.f.). *RobotShop*. Obtenido de https://eu.robotshop.com/es

[27] Romera, J. A. (2015). *idus.us.es.* Obtenido de https://idus.us.es/bitstream/handle/11441/36940/TFG%20Jos%E9%20Andr%E9s%20Mill%E1n%20Romera.pdf;jsessionid=D642C4A6945BF26048CB1F41296D3ECE?isAllowed=y&sequence=4

[28] Sánchez Granero, A. (2021). *Puesta a punto de un robot real autónomo inteligente usando ROS.*

[29] *Tokenizadores de subpalabras*. (s.f.). Obtenido de TensorFlow: https://www.tensorflow.org/text/guide/subwords_tokenizer?hl=es-419

[30] Tunstall L., von Werra L. & Wolf T. (2022). *Natural Language Processing with Transformers.* Sebastopol, CA: O'Reilly.

[31] Villarreal Onofre, R. L. (2021). *Un framework basado en ros para la navegación de robots móviles autónomos.*

[32] *Wiki ros*. (s.f.). Obtenido de https://wiki.ros.org/ROS/Introduction#What_is_ROS.3F

Anexos.

Anexo I. Costmap_global_params.yaml

```
global_costmap:
    update_frequency: 5
    publish_frequency: 2.5
    transform_tolerance: 0.5
    static_map: true
    resolution: 0.01
    inflation_radius: 5.0
    cost_scaling_factor: 4.0
    transform_tolerance: 1.0
    plugins:
     - {name: inflation_layer,    type:
"costmap_2d::InflationLayer"}
     - {name: static_layer,       type: "costmap_2d::StaticLayer"}
     - {name: obstacles_layer,    type: "costmap_2d::VoxelLayer"}
```

Anexo II. Costmap_common_params.yaml

```
obstacle_range: 6.0
raytrace_range: 8.5
footprint: [[0.25, 0.25], [0.25, -0.25], [-0.25, -0.25], [-0.25,
0.25]]
map_topic: /map
subscribe_to_updates: true
global_frame: map
robot_base_frame: base_link
always_send_full_costmap: true
static_layer:
    map_topic: /map
    subscribe_to_updates: true

plugins:
    - {name: inflation,   type: "costmap_2d::InflationLayer"}
    - {name: static, type: "costmap_2d::StaticLayer"}
    - {name: obstacles,    type: "costmap_2d::ObstacleLayer"}

obstacle_layer:
    observation_sources: laser_scan_sensor
    laser_scan_sensor: {sensor_frame: laser, data_type: LaserScan,
topic: scan, marking: true, clearing: true, min_obstacle_height:
0.0, max_obstacle_height: 5.0, obstacle_range: 6.0, raytrace_range:
8.5
```

Anexo III. Costmap_local_params.yaml

```
local_costmap:
    global_frame: odom
    robot_base_frame: base_link
    update_frequency: 5
    publish_frequency: 2.5
    static_map: false
    rolling_window: true
    origin_x: -2
    origin_y: -2
    width: 4
    height: 4
    resolution: 0.01
    inflation_radius: 0.10
    cost_scaling_factor: 5.0
    transform_tolerance: 1.0
    plugins:
      - {name: obstacle_layer,     type: "costmap_2d::VoxelLayer"}
      - {name: inflation_layer,    type: 
"costmap_2d::InflationLayer"}
```

Anexo IV. Trayectory_planner.yaml

```
TrajectoryPlannerROS:
    max_vel_x: 0.15
    min_vel_x: 0.05
    max_vel_theta: 4.5
    min_vel_theta: -4.5
    min_in_place_vel_theta: 0.25
    acc_lim_theta: 0.5
    acc_lim_x: 0.5
    acc_lim_Y: 0.5
    holonomic_robot: false
    meter_scoring: true
    xy_goal_tolerance: 0.15
    yaw_goal_tolerance: 0.25
    escape_vel: -0.05
    latch_xy_goal_tolerance: true
```

Anexo V. Hands.py

Código de Python para la prueba del reconocimiento de gestos.

```
import cv2
import mediapipe as mp

mp_drawing = mp.solutions.drawing_utils
mp_drawing_styles = mp.solutions.drawing_styles
mp_hands = mp.solutions.hands

# Definir funciones para los gestos
def gesto_afirmativo():
    print("Gesto afirmativo detectado")
    # Gesto de pulgar hacia arriba detectado

def gesto_negativo():
    print("Gesto negativo detectado")
    # Gesto de pulgar hacia abajo detectado

def gesto_parar():
    print("Gesto parar detectado")
    # Gesto con la mano abierta

# For webcam input:
cap = cv2.VideoCapture(0)
with mp_hands.Hands(
    model_complexity=0,
    min_detection_confidence=0.5,
    min_tracking_confidence=0.5) as hands:
  while cap.isOpened():
    success, image = cap.read()
    if not success:
      print("Ignoring empty camera frame.")
      # If loading a video, use 'break' instead of 'continue'.
      continue

    # To improve performance, optionally mark the image as not writable
to
    # pass by reference.
    image.flags.writeable = False
    image = cv2.cvtColor(image, cv2.COLOR_BGR2RGB)
    results = hands.process(image)

    # Draw the hand annotations on the image.
    image.flags.writeable = True
    image = cv2.cvtColor(image, cv2.COLOR_RGB2BGR)
    if results.multi_hand_landmarks:
```

```
      for hand_landmarks in results.multi_hand_landmarks:
        # Obtener las coordenadas de los puntos clave
        wrist_x = hand_landmarks.landmark[mp_hands.HandLandmark.WRIST].x
        wrist_y = hand_landmarks.landmark[mp_hands.HandLandmark.WRIST].y

        thumb_tip_x =
hand_landmarks.landmark[mp_hands.HandLandmark.THUMB_TIP].x
        thumb_tip_y =
hand_landmarks.landmark[mp_hands.HandLandmark.THUMB_TIP].y

        index_finger_pip_x =
hand_landmarks.landmark[mp_hands.HandLandmark.INDEX_FINGER_PIP].x
        index_finger_pip_y =
hand_landmarks.landmark[mp_hands.HandLandmark.INDEX_FINGER_PIP].y

        index_finger_dip_x =
hand_landmarks.landmark[mp_hands.HandLandmark.INDEX_FINGER_DIP].x
        index_finger_dip_y =
hand_landmarks.landmark[mp_hands.HandLandmark.INDEX_FINGER_DIP].y

        index_finger_tip_x =
hand_landmarks.landmark[mp_hands.HandLandmark.INDEX_FINGER_TIP].x
        index_finger_tip_y =
hand_landmarks.landmark[mp_hands.HandLandmark.INDEX_FINGER_TIP].y

        middle_finger_pip_x =
hand_landmarks.landmark[mp_hands.HandLandmark.MIDDLE_FINGER_PIP].x
        middle_finger_pip_y =
hand_landmarks.landmark[mp_hands.HandLandmark.MIDDLE_FINGER_PIP].y

        middle_finger_dip_x =
hand_landmarks.landmark[mp_hands.HandLandmark.MIDDLE_FINGER_DIP].x
        middle_finger_dip_y =
hand_landmarks.landmark[mp_hands.HandLandmark.MIDDLE_FINGER_DIP].y

        middle_finger_tip_x =
hand_landmarks.landmark[mp_hands.HandLandmark.MIDDLE_FINGER_TIP].x
        middle_finger_tip_y =
hand_landmarks.landmark[mp_hands.HandLandmark.MIDDLE_FINGER_TIP].y

        ring_finger_pip_x =
hand_landmarks.landmark[mp_hands.HandLandmark.RING_FINGER_PIP].x
        ring_finger_pip_y =
hand_landmarks.landmark[mp_hands.HandLandmark.RING_FINGER_PIP].y

        ring_finger_dip_x =
hand_landmarks.landmark[mp_hands.HandLandmark.RING_FINGER_DIP].x
```

```
        ring_finger_dip_y =
hand_landmarks.landmark[mp_hands.HandLandmark.RING_FINGER_DIP].y

        ring_finger_tip_x =
hand_landmarks.landmark[mp_hands.HandLandmark.RING_FINGER_TIP].x
        ring_finger_tip_y =
hand_landmarks.landmark[mp_hands.HandLandmark.RING_FINGER_TIP].y

        pinky_pip_x =
hand_landmarks.landmark[mp_hands.HandLandmark.PINKY_PIP].x
        pinky_pip_y =
hand_landmarks.landmark[mp_hands.HandLandmark.PINKY_PIP].y

        pinky_dip_x =
hand_landmarks.landmark[mp_hands.HandLandmark.PINKY_DIP].x
        pinky_dip_y =
hand_landmarks.landmark[mp_hands.HandLandmark.PINKY_DIP].y

        pinky_tip_x =
hand_landmarks.landmark[mp_hands.HandLandmark.PINKY_TIP].x
        pinky_tip_y =
hand_landmarks.landmark[mp_hands.HandLandmark.PINKY_TIP].y


        # Asociar gestos con funciones
        if (thumb_tip_y < middle_finger_tip_y) & (wrist_y >
middle_finger_tip_y) & (index_finger_tip_x < index_finger_pip_x) &
(middle_finger_tip_x < middle_finger_pip_x) & (ring_finger_tip_x <
ring_finger_pip_x) & (pinky_tip_x < pinky_pip_x):
            gesto_afirmativo()
        if (thumb_tip_y > middle_finger_tip_y) & (index_finger_tip_x <
index_finger_pip_x) & (middle_finger_tip_x < middle_finger_pip_x) &
(ring_finger_tip_x < ring_finger_pip_x) & (pinky_tip_x < pinky_pip_x):
            gesto_negativo()
        if (middle_finger_tip_y < pinky_tip_y) & (middle_finger_tip_y <
wrist_y) & (thumb_tip_x > pinky_tip_x) & (middle_finger_tip_y <
middle_finger_dip_y) & (index_finger_tip_y < index_finger_dip_y) &
(ring_finger_tip_y < ring_finger_dip_y) & (pinky_tip_y < pinky_dip_y):
            gesto_parar()


        # Dibujar los puntos clave y las conexiones en la imagen
        mp_drawing.draw_landmarks(
            image,
            hand_landmarks,
            mp_hands.HAND_CONNECTIONS,
            mp_drawing_styles.get_default_hand_landmarks_style(),
            mp_drawing_styles.get_default_hand_connections_style())
```

```
    # Flip the image horizontally for a selfie-view display.
    cv2.imshow('MediaPipe Hands', cv2.flip(image, 1))
    if cv2.waitKey(5) & 0xFF == 27:
      break

cap.release()
cv2.destroyAllWindows()
```

Anexo VI. Clasificación_de_imágenes.py

Código de Python, utilizando la arquitectura Transformer, para la prueba del reconocimiento facial del usuario.

```
import matplotlib.pyplot as plt
import numpy as np
import PIL
import tensorflow as tf

from tensorflow import keras
from tensorflow.keras import layers
from tensorflow.keras.models import Sequential


import pathlib

data_dir = pathlib.Path('/Users/adria/Faces')
print(data_dir)


image_count = len(list(data_dir.glob('*/*.jpg')))
print(image_count)

adrian = list(data_dir.glob('adrian/*'))
PIL.Image.open(str(adrian[0]))
otras = list(data_dir.glob('otras/*'))
PIL.Image.open(str(otras[0]))


batch_size = 170
img_height = 650
img_width = 650


train_ds = tf.keras.utils.image_dataset_from_directory(
  data_dir,
  validation_split=0.2,
  subset="training",
  seed=123,
  image_size=(img_height, img_width),
  batch_size=batch_size)

val_ds = tf.keras.utils.image_dataset_from_directory(
  data_dir,
  validation_split=0.2,
  subset="validation",
  seed=123,
```

```
  image_size=(img_height, img_width),
  batch_size=batch_size)

class_names = train_ds.class_names
print(class_names)

import matplotlib.pyplot as plt

plt.figure(figsize=(10, 10))
for images, labels in train_ds.take(1):
  for i in range(9):
    ax = plt.subplot(3, 3, i + 1)
    plt.imshow(images[i].numpy().astype("uint8"))
    plt.title(class_names[labels[i]])
    plt.axis("off")

for image_batch, labels_batch in train_ds:
  print(image_batch.shape)
  print(labels_batch.shape)
  break

AUTOTUNE = tf.data.AUTOTUNE

train_ds = train_ds.cache().shuffle(1000).prefetch(buffer_size=AUTOTUNE)
val_ds = val_ds.cache().prefetch(buffer_size=AUTOTUNE)

normalization_layer = layers.Rescaling(1./255)

normalized_ds = train_ds.map(lambda x, y: (normalization_layer(x), y))
image_batch, labels_batch = next(iter(normalized_ds))
first_image = image_batch[0]
# Notice the pixel values are now in `[0,1]`.
print(np.min(first_image), np.max(first_image))

num_classes = len(class_names)

model = Sequential([
  layers.Rescaling(1./255, input_shape=(img_height, img_width, 3)),
  layers.Conv2D(16, 3, padding='same', activation='relu'),
  layers.MaxPooling2D(),
  layers.Conv2D(32, 3, padding='same', activation='relu'),
  layers.MaxPooling2D(),
  layers.Conv2D(64, 3, padding='same', activation='relu'),
  layers.MaxPooling2D(),
  layers.Flatten(),
  layers.Dense(128, activation='relu'),
  layers.Dense(num_classes)
])
```

```
model.compile(optimizer='adam',
              loss=tf.keras.losses.SparseCategoricalCrossentropy(from_log
its=True),
              metrics=['accuracy'])

model.summary()

epochs=1
history = model.fit(
  train_ds,
  validation_data=val_ds,
  epochs=epochs
)

acc = history.history['accuracy']
val_acc = history.history['val_accuracy']

loss = history.history['loss']
val_loss = history.history['val_loss']

epochs_range = range(epochs)

plt.figure(figsize=(8, 8))
plt.subplot(1, 2, 1)
plt.plot(epochs_range, acc, label='Training Accuracy')
plt.plot(epochs_range, val_acc, label='Validation Accuracy')
plt.legend(loc='lower right')
plt.title('Training and Validation Accuracy')

plt.subplot(1, 2, 2)
plt.plot(epochs_range, loss, label='Training Loss')
plt.plot(epochs_range, val_loss, label='Validation Loss')
plt.legend(loc='upper right')
plt.title('Training and Validation Loss')
plt.show()

data_augmentation = keras.Sequential(
  [
    layers.RandomFlip("horizontal",
                      input_shape=(img_height,
                                  img_width,
                                  3)),
    layers.RandomRotation(0.1),
    layers.RandomZoom(0.1),
  ]
)
```

```
plt.figure(figsize=(10, 10))
for images, _ in train_ds.take(1):
  for i in range(9):
    augmented_images = data_augmentation(images)
    ax = plt.subplot(3, 3, i + 1)
    plt.imshow(augmented_images[0].numpy().astype("uint8"))
    plt.axis("off")

model = Sequential([
  data_augmentation,
  layers.Rescaling(1./255),
  layers.Conv2D(16, 3, padding='same', activation='relu'),
  layers.MaxPooling2D(),
  layers.Conv2D(32, 3, padding='same', activation='relu'),
  layers.MaxPooling2D(),
  layers.Conv2D(64, 3, padding='same', activation='relu'),
  layers.MaxPooling2D(),
  layers.Dropout(0.2),
  layers.Flatten(),
  layers.Dense(128, activation='relu'),
  layers.Dense(num_classes, name="outputs")
])

model.compile(optimizer='adam',
              loss=tf.keras.losses.SparseCategoricalCrossentropy(from_log
its=True),
              metrics=['accuracy'])
model.summary()

epochs = 200
history = model.fit(
  train_ds,
  validation_data=val_ds,
  epochs=epochs
)

acc = history.history['accuracy']
val_acc = history.history['val_accuracy']

loss = history.history['loss']
val_loss = history.history['val_loss']

epochs_range = range(epochs)

plt.figure(figsize=(8, 8))
plt.subplot(1, 2, 1)
plt.plot(epochs_range, acc, label='Training Accuracy')
plt.plot(epochs_range, val_acc, label='Validation Accuracy')
```

```
plt.legend(loc='lower right')
plt.title('Training and Validation Accuracy')

plt.subplot(1, 2, 2)
plt.plot(epochs_range, loss, label='Training Loss')
plt.plot(epochs_range, val_loss, label='Validation Loss')
plt.legend(loc='upper right')
plt.title('Training and Validation Loss')
plt.show()

from PIL import Image


# Ruta de la imagen en tu Google Drive
image_path = '/Users/adria/FOTOS/foto1.jpg'

# Cargar la imagen usando Pillow
img = Image.open(image_path)
img = img.resize((img_height, img_width))  # Ajustar al tamaño objetivo

img_array = tf.keras.utils.img_to_array(img)
img_array = tf.expand_dims(img_array, 0) # Crear un lote (batch)

# Resto del código (suponiendo que `model` y `class_names` ya están
definidos)
predictions = model.predict(img_array)
score = tf.nn.softmax(predictions[0])

print(
    "Esta imagen probablemente pertenece a {} con una confianza del
{:.2f} por ciento."
    .format(class_names[np.argmax(score)], 100 * np.max(score))
)
```

Anexo VII. Tokenizer.py

Código del tokenizador del dataset para el preentrenamiento del modelo Transformer.

```
import collections
import os
import pathlib
import re
import string
import sys
import tempfile
import time

import numpy as np
import matplotlib.pyplot as plt

import tensorflow_datasets as tfds
import tensorflow_text as text
import tensorflow as tf

tf.get_logger().setLevel('ERROR')
pwd = pathlib.Path.cwd()

#LOS DATOS
from google.colab import drive
drive.mount('/content/gdrive')

import os
os.chdir('/content/gdrive/MyDrive/lista')

# Download the file
import pathlib

path_to_file = '/content/gdrive/MyDrive/lista/comandos.txt'

with open(path_to_file, "r") as f:
    lines = f.readlines()

pairs = [line.split('\t') for line in lines]
pt_examples = [pair[0] for pair in pairs]
en_examples = [pair[1] for pair in pairs]

import tensorflow as tf
```

```
train_examples = tf.data.Dataset.from_tensor_slices((pt_examples,
en_examples))

for pt, en in train_examples.take(1):
  print("Orden: ", pt.numpy().decode('utf-8'))
  print("Robot:   ", en.numpy().decode('utf-8'))

train_en = train_examples.map(lambda pt, en: en)
train_pt = train_examples.map(lambda pt, en: pt)

from tensorflow_text.tools.wordpiece_vocab import bert_vocab_from_dataset
as bert_vocab

bert_tokenizer_params=dict(lower_case=True)
reserved_tokens=["[PAD]", "[UNK]", "[START]", "[END]"]

bert_vocab_args = dict(
    # The target vocabulary size
    vocab_size = 1000,
    # Reserved tokens that must be included in the vocabulary
    reserved_tokens=reserved_tokens,
    # Arguments for `text.BertTokenizer`
    bert_tokenizer_params=bert_tokenizer_params,
    # Arguments for
`wordpiece_vocab.wordpiece_tokenizer_learner_lib.learn`
    learn_params={},
)



pt_vocab = bert_vocab.bert_vocab_from_dataset(
    train_pt.batch(1000).prefetch(2),
    **bert_vocab_args
)

print(pt_vocab[:10])
print(pt_vocab[100:110])
print(pt_vocab[1000:1010])
print(pt_vocab[-10:])

def write_vocab_file(filepath, vocab):
  with open(filepath, 'w') as f:
    for token in vocab:
      print(token, file=f)

write_vocab_file('orden_vocab.txt', pt_vocab)

en_vocab = bert_vocab.bert_vocab_from_dataset(
    train_en.batch(1000).prefetch(2),
```

```
    **bert_vocab_args
)


print(en_vocab[:10])
print(en_vocab[100:110])
print(en_vocab[1000:1010])
print(en_vocab[-10:])

write_vocab_file('robot_vocab.txt', en_vocab)
!ls *.txt

pt_tokenizer = text.BertTokenizer('orden_vocab.txt',
**bert_tokenizer_params)
en_tokenizer = text.BertTokenizer('robot_vocab.txt',
**bert_tokenizer_params)

for pt_examples, en_examples in train_examples.batch(3).take(1):
  for ex in en_examples:
    print(ex.numpy())

# Tokenize the examples -> (batch, word, word-piece)
token_batch = en_tokenizer.tokenize(en_examples)
# Merge the word and word-piece axes -> (batch, tokens)
token_batch = token_batch.merge_dims(-2,-1)

for ex in token_batch.to_list():
  print(ex)

# Lookup each token id in the vocabulary.
txt_tokens = tf.gather(en_vocab, token_batch)
# Join with spaces.
tf.strings.reduce_join(txt_tokens, separator=' ', axis=-1)

words = en_tokenizer.detokenize(token_batch)
tf.strings.reduce_join(words, separator=' ', axis=-1)

START = tf.argmax(tf.constant(reserved_tokens) == "[START]")
END = tf.argmax(tf.constant(reserved_tokens) == "[END]")

def add_start_end(ragged):
  count = ragged.bounding_shape()[0]
  starts = tf.fill([count,1], START)
  ends = tf.fill([count,1], END)
  return tf.concat([starts, ragged, ends], axis=1)

words = en_tokenizer.detokenize(add_start_end(token_batch))
tf.strings.reduce_join(words, separator=' ', axis=-1)
```

```
def cleanup_text(reserved_tokens, token_txt):
  # Drop the reserved tokens, except for "[UNK]".
  bad_tokens = [re.escape(tok) for tok in reserved_tokens if tok !=
"[UNK]"]
  bad_token_re = "|".join(bad_tokens)

  bad_cells = tf.strings.regex_full_match(token_txt, bad_token_re)
  result = tf.ragged.boolean_mask(token_txt, ~bad_cells)

  # Join them into strings.
  result = tf.strings.reduce_join(result, separator=' ', axis=-1)

  return result

en_examples.numpy()

token_batch = en_tokenizer.tokenize(en_examples).merge_dims(-2,-1)
words = en_tokenizer.detokenize(token_batch)
words

cleanup_text(reserved_tokens, words).numpy()

class CustomTokenizer(tf.Module):
  def __init__(self, reserved_tokens, vocab_path):
    self.tokenizer = text.BertTokenizer(vocab_path, lower_case=True)
    self._reserved_tokens = reserved_tokens
    self._vocab_path = tf.saved_model.Asset(vocab_path)

    vocab = pathlib.Path(vocab_path).read_text().splitlines()
    self.vocab = tf.Variable(vocab)

    ## Create the signatures for export:

    # Include a tokenize signature for a batch of strings.
    self.tokenize.get_concrete_function(
        tf.TensorSpec(shape=[None], dtype=tf.string))

    # Include `detokenize` and `lookup` signatures for:
    #   * `Tensors` with shapes [tokens] and [batch, tokens]
    #   * `RaggedTensors` with shape [batch, tokens]
    self.detokenize.get_concrete_function(
        tf.TensorSpec(shape=[None, None], dtype=tf.int64))
    self.detokenize.get_concrete_function(
          tf.RaggedTensorSpec(shape=[None, None], dtype=tf.int64))

    self.lookup.get_concrete_function(
        tf.TensorSpec(shape=[None, None], dtype=tf.int64))
```

```
    self.lookup.get_concrete_function(
          tf.RaggedTensorSpec(shape=[None, None], dtype=tf.int64))

    # These `get_*` methods take no arguments
    self.get_vocab_size.get_concrete_function()
    self.get_vocab_path.get_concrete_function()
    self.get_reserved_tokens.get_concrete_function()

  @tf.function
  def tokenize(self, strings):
    enc = self.tokenizer.tokenize(strings)
    # Merge the `word` and `word-piece` axes.
    enc = enc.merge_dims(-2,-1)
    enc = add_start_end(enc)
    return enc

  @tf.function
  def detokenize(self, tokenized):
    words = self.tokenizer.detokenize(tokenized)
    return cleanup_text(self._reserved_tokens, words)

  @tf.function
  def lookup(self, token_ids):
    return tf.gather(self.vocab, token_ids)

  @tf.function
  def get_vocab_size(self):
    return tf.shape(self.vocab)[0]

  @tf.function
  def get_vocab_path(self):
    return self._vocab_path

  @tf.function
  def get_reserved_tokens(self):
    return tf.constant(self._reserved_tokens)

tokenizers = tf.Module()
tokenizers.pt = CustomTokenizer(reserved_tokens, 'orden_vocab.txt')
tokenizers.en = CustomTokenizer(reserved_tokens, 'robot_vocab.txt')

model_path = '/content/gdrive/MyDrive/lista/SavedModel'
tf.saved_model.save(tokenizers, model_path)

reloaded_tokenizers = tf.saved_model.load(model_path)
reloaded_tokenizers.en.get_vocab_size().numpy()

tokens = reloaded_tokenizers.en.tokenize(['Hello TensorFlow!'])
```

```
tokens.numpy()

text_tokens = reloaded_tokenizers.en.lookup(tokens)
text_tokens

round_trip = reloaded_tokenizers.en.detokenize(tokens)

print(round_trip.numpy()[0].decode('utf-8'))
```

Anexo VIII. Transformer.py

Código de la estructura Transformer para el procesamiento del lenguaje natural.

```
import logging
import time

import numpy as np
import matplotlib.pyplot as plt

import tensorflow_datasets as tfds
import tensorflow as tf

import tensorflow_text

# Download the file
import pathlib

path_to_file = 'lista/comandos.txt'

with open(path_to_file, "r") as f:
    lines = f.readlines()

import tensorflow as tf
from sklearn.model_selection import train_test_split

pairs = [line.split('\t') for line in lines]
pt_examples = [pair[0] for pair in pairs]
en_examples = [pair[1] for pair in pairs]

# Dividir los datos en entrenamiento, validación y prueba
pt_train, pt_temp, en_train, en_temp = train_test_split(pt_examples,
en_examples, test_size=0.3, random_state=42) #0.3 porque se queda con el
70% de datos de entrenamiento / La variable x_temp se utiliza para
guardar temporalmente una parte de los datos
pt_val, pt_test, en_val, en_test = train_test_split(pt_temp, en_temp,
test_size=0.5, random_state=42) #0.5 porque del 30% restante se reparten
15% cada uno

# Crear conjuntos de datos tf.data.Dataset
train_examples = tf.data.Dataset.from_tensor_slices((pt_train, en_train))
val_examples = tf.data.Dataset.from_tensor_slices((pt_val, en_val))
test_examples = tf.data.Dataset.from_tensor_slices((pt_test, en_test))

# Ahora, tenemos tres conjuntos de datos
```

```
for pt_examples, en_examples in train_examples.batch(3).take(1):
  for pt in pt_examples.numpy():
    print(pt.decode('utf-8'))

  print()

  for en in en_examples.numpy():
    print(en.decode('utf-8'))

model_path = 'lista/SavedModel'
tokenizers = tf.saved_model.load(model_path)

[item for item in dir(tokenizers.en) if not item.startswith('_')]

print('> This is a batch of strings:')
for en in en_examples.numpy():
  print(en.decode('utf-8'))

encoded = tokenizers.en.tokenize(en_examples)

print('> This is a padded-batch of token IDs:')
for row in encoded.to_list():
  print(row)

round_trip = tokenizers.en.detokenize(encoded)

print('> This is human-readable text:')
for line in round_trip.numpy():
  print(line.decode('utf-8'))

print('> This is the text split into tokens:')
tokens = tokenizers.en.lookup(encoded)
tokens

lengths = []

for pt_examples, en_examples in train_examples.batch(211):
  pt_tokens = tokenizers.pt.tokenize(pt_examples)
  lengths.append(pt_tokens.row_lengths())

  en_tokens = tokenizers.en.tokenize(en_examples)
  lengths.append(en_tokens.row_lengths())
  print('.', end='', flush=True)

all_lengths = np.concatenate(lengths)

plt.hist(all_lengths, np.linspace(0, 500, 101))
```

```
plt.ylim(plt.ylim())
max_length = max(all_lengths)
plt.plot([max_length, max_length], plt.ylim())
plt.title(f'Maximum tokens per example: {max_length}');

MAX_TOKENS=128
def prepare_batch(pt, en):
    pt = tokenizers.pt.tokenize(pt)      # Output is ragged.
    pt = pt[:, :MAX_TOKENS]    # Trim to MAX_TOKENS.
    pt = pt.to_tensor()  # Convert to 0-padded dense Tensor

    en = tokenizers.en.tokenize(en)
    en = en[:, :(MAX_TOKENS+1)]
    en_inputs = en[:, :-1].to_tensor()  # Drop the [END] tokens
    en_labels = en[:, 1:].to_tensor()   # Drop the [START] tokens

    return (pt, en_inputs), en_labels

BUFFER_SIZE = 211
BATCH_SIZE = 64

def make_batches(ds):
  return (
      ds
      .shuffle(BUFFER_SIZE)
      .batch(BATCH_SIZE)
      .map(prepare_batch, tf.data.AUTOTUNE)
      .prefetch(buffer_size=tf.data.AUTOTUNE))

# Create training and validation set batches.
train_batches = make_batches(train_examples)
val_batches = make_batches(val_examples)

for (pt, en), en_labels in train_batches.take(1):
  break

print(pt.shape)
print(en.shape)
print(en_labels.shape)

print(en[0][:10])
print(en_labels[0][:10])

def positional_encoding(length, depth):
  depth = depth/2

  positions = np.arange(length)[:, np.newaxis]     # (seq, 1)
  depths = np.arange(depth)[np.newaxis, :]/depth   # (1, depth)
```

```
  angle_rates = 1 / (10000**depths)         # (1, depth)
  angle_rads = positions * angle_rates      # (pos, depth)

  pos_encoding = np.concatenate(
      [np.sin(angle_rads), np.cos(angle_rads)],
      axis=-1)

  return tf.cast(pos_encoding, dtype=tf.float32)


#@title
pos_encoding = positional_encoding(length=2048, depth=512)

# Check the shape.
print(pos_encoding.shape)

# Plot the dimensions.
plt.pcolormesh(pos_encoding.numpy().T, cmap='RdBu')
plt.ylabel('Depth')
plt.xlabel('Position')
plt.colorbar()
plt.show()

#@title
pos_encoding/=tf.norm(pos_encoding, axis=1, keepdims=True)
p = pos_encoding[1000]
dots = tf.einsum('pd,d -> p', pos_encoding, p)
plt.subplot(2,1,1)
plt.plot(dots)
plt.ylim([0,1])
plt.plot([950, 950, float('nan'), 1050, 1050],
         [0,1,float('nan'),0,1], color='k', label='Zoom')
plt.legend()
plt.subplot(2,1,2)
plt.plot(dots)
plt.xlim([950, 1050])
plt.ylim([0,1])

class PositionalEmbedding(tf.keras.layers.Layer):
  def __init__(self, vocab_size, d_model):
    super().__init__()
    self.d_model = d_model
    self.embedding = tf.keras.layers.Embedding(vocab_size, d_model,
mask_zero=True)
    self.pos_encoding = positional_encoding(length=2048, depth=d_model)

  def compute_mask(self, *args, **kwargs):
    return self.embedding.compute_mask(*args, **kwargs)
```

```
  def call(self, x):
    length = tf.shape(x)[1]
    x = self.embedding(x)
    # This factor sets the relative scale of the embedding and
positonal_encoding.
    x *= tf.math.sqrt(tf.cast(self.d_model, tf.float32))
    x = x + self.pos_encoding[tf.newaxis, :length, :]
    return x

embed_pt = PositionalEmbedding(vocab_size=tokenizers.pt.get_vocab_size(),
d_model=64)
embed_en = PositionalEmbedding(vocab_size=tokenizers.en.get_vocab_size(),
d_model=64)

pt_emb = embed_pt(pt)
en_emb = embed_en(en)

en_emb._keras_mask

class BaseAttention(tf.keras.layers.Layer):
  def __init__(self, **kwargs):
    super().__init__()
    self.mha = tf.keras.layers.MultiHeadAttention(**kwargs)
    self.layernorm = tf.keras.layers.LayerNormalization()
    self.add = tf.keras.layers.Add()

class CrossAttention(BaseAttention):
  def call(self, x, context):
    attn_output, attn_scores = self.mha(
        query=x,
        key=context,
        value=context,
        return_attention_scores=True)

    # Cache the attention scores for plotting later.
    self.last_attn_scores = attn_scores

    x = self.add([x, attn_output])
    x = self.layernorm(x)

    return x

sample_ca = CrossAttention(num_heads=4, key_dim=32)

print(pt_emb.shape)
print(en_emb.shape)
print(sample_ca(en_emb, pt_emb).shape)
```

```
class GlobalSelfAttention(BaseAttention):
  def call(self, x):
    attn_output = self.mha(
        query=x,
        value=x,
        key=x)
    x = self.add([x, attn_output])
    x = self.layernorm(x)
    return x

sample_gsa = GlobalSelfAttention(num_heads=4, key_dim=32)

print(pt_emb.shape)
print(sample_gsa(pt_emb).shape)

class CausalSelfAttention(BaseAttention):
  def call(self, x):
    attn_output = self.mha(
        query=x,
        value=x,
        key=x,
        use_causal_mask = True)
    x = self.add([x, attn_output])
    x = self.layernorm(x)
    return x

sample_csa = CausalSelfAttention(num_heads=4, key_dim=32)

print(en_emb.shape)
print(sample_csa(en_emb).shape)

out1 = sample_csa(embed_en(en[:, :3]))
out2 = sample_csa(embed_en(en))[:, :3]

tf.reduce_max(abs(out1 - out2)).numpy()

class FeedForward(tf.keras.layers.Layer):
  def __init__(self, d_model, dff, dropout_rate=0.3):
    super().__init__()
    self.seq = tf.keras.Sequential([
      tf.keras.layers.Dense(dff, activation='relu'),
      tf.keras.layers.Dense(d_model),
      tf.keras.layers.Dropout(dropout_rate)
    ])
    self.add = tf.keras.layers.Add()
    self.layer_norm = tf.keras.layers.LayerNormalization()
```

```
  def call(self, x):
    x = self.add([x, self.seq(x)])
    x = self.layer_norm(x)
    return x

sample_ffn = FeedForward(64, 128)

print(en_emb.shape)
print(sample_ffn(en_emb).shape)

class EncoderLayer(tf.keras.layers.Layer):
  def __init__(self,*, d_model, num_heads, dff, dropout_rate=0.3):
    super().__init__()

    self.self_attention = GlobalSelfAttention(
        num_heads=num_heads,
        key_dim=d_model,
        dropout=dropout_rate)

    self.ffn = FeedForward(d_model, dff)

  def call(self, x):
    x = self.self_attention(x)
    x = self.ffn(x)
    return x

sample_encoder_layer = EncoderLayer(d_model=64, num_heads=2, dff=128)

print(pt_emb.shape)
print(sample_encoder_layer(pt_emb).shape)

class Encoder(tf.keras.layers.Layer):
  def __init__(self, *, num_layers, d_model, num_heads,
               dff, vocab_size, dropout_rate=0.1):
    super().__init__()

    self.d_model = d_model
    self.num_layers = num_layers

    self.pos_embedding = PositionalEmbedding(
        vocab_size=vocab_size, d_model=d_model)

    self.enc_layers = [
        EncoderLayer(d_model=d_model,
                     num_heads=num_heads,
                     dff=dff,
                     dropout_rate=dropout_rate)
        for _ in range(num_layers)]
```

```
    self.dropout = tf.keras.layers.Dropout(dropout_rate)

  def call(self, x):
    # `x` is token-IDs shape: (batch, seq_len)
    x = self.pos_embedding(x)  # Shape `(batch_size, seq_len, d_model)`.

    # Add dropout.
    x = self.dropout(x)

    for i in range(self.num_layers):
      x = self.enc_layers[i](x)

    return x  # Shape `(batch_size, seq_len, d_model)`.

# Instantiate the encoder.
sample_encoder = Encoder(num_layers=2,
                         d_model=64,
                         num_heads=2,
                         dff=128,
                         vocab_size=2000)

sample_encoder_output = sample_encoder(pt, training=False)

# Print the shape.
print(pt.shape)
print(sample_encoder_output.shape)  # Shape `(batch_size, input_seq_len,
d_model)`.

class DecoderLayer(tf.keras.layers.Layer):
  def __init__(self,
               *,
               d_model,
               num_heads,
               dff,
               dropout_rate=0.1):
    super(DecoderLayer, self).__init__()

    self.causal_self_attention = CausalSelfAttention(
        num_heads=num_heads,
        key_dim=d_model,
        dropout=dropout_rate)

    self.cross_attention = CrossAttention(
        num_heads=num_heads,
        key_dim=d_model,
        dropout=dropout_rate)

    self.ffn = FeedForward(d_model, dff)
```

```
  def call(self, x, context):
    x = self.causal_self_attention(x=x)
    x = self.cross_attention(x=x, context=context)

    # Cache the last attention scores for plotting later
    self.last_attn_scores = self.cross_attention.last_attn_scores

    x = self.ffn(x)  # Shape `(batch_size, seq_len, d_model)`.
    return x

sample_decoder_layer = DecoderLayer(d_model=64, num_heads=2, dff=128)

sample_decoder_layer_output = sample_decoder_layer(
    x=en_emb, context=pt_emb)

print(en_emb.shape)
print(pt_emb.shape)
print(sample_decoder_layer_output.shape)  # `(batch_size, seq_len,
d_model)`

class Decoder(tf.keras.layers.Layer):
  def __init__(self, *, num_layers, d_model, num_heads, dff, vocab_size,
               dropout_rate=0.3):
    super(Decoder, self).__init__()

    self.d_model = d_model
    self.num_layers = num_layers

    self.pos_embedding = PositionalEmbedding(vocab_size=vocab_size,
                                             d_model=d_model)
    self.dropout = tf.keras.layers.Dropout(dropout_rate)
    self.dec_layers = [
        DecoderLayer(d_model=d_model, num_heads=num_heads,
                     dff=dff, dropout_rate=dropout_rate)
        for _ in range(num_layers)]

    self.last_attn_scores = None

  def call(self, x, context):
    # `x` is token-IDs shape (batch, target_seq_len)
    x = self.pos_embedding(x)  # (batch_size, target_seq_len, d_model)

    x = self.dropout(x)

    for i in range(self.num_layers):
      x  = self.dec_layers[i](x, context)
```

```
    self.last_attn_scores = self.dec_layers[-1].last_attn_scores

    # The shape of x is (batch_size, target_seq_len, d_model).
    return x

# Instantiate the decoder.
sample_decoder = Decoder(num_layers=2,
                         d_model=64,
                         num_heads=2,
                         dff=128,
                         vocab_size=3000)

output = sample_decoder(
    x=en,
    context=pt_emb)

# Print the shapes.
print(en.shape)
print(pt_emb.shape)
print(output.shape)

sample_decoder.last_attn_scores.shape  # (batch, heads, target_seq,
input_seq)

class Transformer(tf.keras.Model):
  def __init__(self, *, num_layers, d_model, num_heads, dff,
               input_vocab_size, target_vocab_size, dropout_rate=0.3):
    super().__init__()
    self.encoder = Encoder(num_layers=num_layers, d_model=d_model,
                           num_heads=num_heads, dff=dff,
                           vocab_size=input_vocab_size,
                           dropout_rate=dropout_rate)

    self.decoder = Decoder(num_layers=num_layers, d_model=d_model,
                           num_heads=num_heads, dff=dff,
                           vocab_size=target_vocab_size,
                           dropout_rate=dropout_rate)

    self.final_layer = tf.keras.layers.Dense(target_vocab_size)

  def call(self, inputs):
    # To use a Keras model with `.fit` you must pass all your inputs in
the
    # first argument.
    context, x  = inputs

    context = self.encoder(context)  # (batch_size, context_len, d_model)
```

```
    x = self.decoder(x, context)  # (batch_size, target_len, d_model)

    # Final linear layer output.
    logits = self.final_layer(x)  # (batch_size, target_len,
target_vocab_size)

    try:
      # Drop the keras mask, so it doesn't scale the losses/metrics.
      # b/250038731
      del logits._keras_mask
    except AttributeError:
      pass
    # Return the final output and the attention weights.
    return logits

  def get_config(self):
    config = super().get_config()
    config.update({
        "num_layers": self.num_layers,
        "d_model": self.d_model,
        "num_heads": self.num_heads,
        "dff": self.dff,
        # etc...
    })
    return config


num_layers = 3
d_model = 64
dff = 128
num_heads = 4
dropout_rate = 0.3

transformer = Transformer(
    num_layers=num_layers,
    d_model=d_model,
    num_heads=num_heads,
    dff=dff,
    input_vocab_size=tokenizers.pt.get_vocab_size().numpy(),
    target_vocab_size=tokenizers.en.get_vocab_size().numpy(),
    dropout_rate=dropout_rate)

output = transformer((pt, en))

print(en.shape)
print(pt.shape)
print(output.shape)
```

```
attn_scores = transformer.decoder.dec_layers[-1].last_attn_scores
print(attn_scores.shape)  # (batch, heads, target_seq, input_seq)

transformer.summary()

class CustomSchedule(tf.keras.optimizers.schedules.LearningRateSchedule):
  def __init__(self, d_model, warmup_steps=4000):
    super().__init__()

    self.d_model = d_model
    self.d_model = tf.cast(self.d_model, tf.float32)

    self.warmup_steps = warmup_steps

  def __call__(self, step):
    step = tf.cast(step, dtype=tf.float32)
    arg1 = tf.math.rsqrt(step)
    arg2 = step * (self.warmup_steps ** -1.5)

    return tf.math.rsqrt(self.d_model) * tf.math.minimum(arg1, arg2)

learning_rate = CustomSchedule(d_model)

optimizer = tf.keras.optimizers.Adam(learning_rate, beta_1=0.9,
beta_2=0.98,
                                     epsilon=1e-9)

plt.plot(learning_rate(tf.range(40000, dtype=tf.float32)))
plt.ylabel('Learning Rate')
plt.xlabel('Train Step')

def masked_loss(label, pred):
  mask = label != 0
  loss_object = tf.keras.losses.SparseCategoricalCrossentropy(
    from_logits=True, reduction='none')
  loss = loss_object(label, pred)

  mask = tf.cast(mask, dtype=loss.dtype)
  loss *= mask

  loss = tf.reduce_sum(loss)/tf.reduce_sum(mask)
  return loss


def masked_accuracy(label, pred):
  pred = tf.argmax(pred, axis=2)
  label = tf.cast(label, pred.dtype)
  match = label == pred
```

```
    mask = label != 0

    match = match & mask

    match = tf.cast(match, dtype=tf.float32)
    mask = tf.cast(mask, dtype=tf.float32)
    return tf.reduce_sum(match)/tf.reduce_sum(mask)

transformer.compile(
    loss=masked_loss,
    optimizer=optimizer,
    metrics=[masked_accuracy])

transformer.fit(train_batches,
                epochs=220,
                validation_data=val_batches)


import actionlib
from move_base_msgs.msg import MoveBaseAction, MoveBaseGoal

# Diccionario que asigna nombres de zonas a coordenadas (x, y)
zonas = {
    "ir zona uno": (0.0, 0.0),
    "ir zona dos": (5.94, -2.52),
    "ir zona tres": (5.84, -4.93),
    "ir zona cuatro": (-8.26, -3.44),
    "ir zona cinco": (-5.92, -5.8),
    "ir zona seis": (6.18, -8.6),
    "ir zona siete": (-12.2, -6.32)
}

def move_to_zone(texto_zon):
    client = actionlib.SimpleActionClient('move_base', MoveBaseAction)
    client.wait_for_server()

    # Obtener las coordenadas asociadas con el nombre de la zona
    x, y = zonas.get(texto_zon, (0.0, 0.0))


    goal = MoveBaseGoal()
    goal.target_pose.header.frame_id = "map"
    goal.target_pose.pose.position.x = x
    goal.target_pose.pose.position.y = y
    goal.target_pose.pose.orientation.w = 1.0
    print("Sending goal to x=%.2f, y=%.2f" % (x, y))
    print("Sending goal to %s" % goal.target_pose.pose.position)
```

```
    client.send_goal(goal)
    rospy.sleep(0.1)
    client.wait_for_result()


class Translator(tf.Module):
  def __init__(self, tokenizers, transformer):
    self.tokenizers = tokenizers
    self.transformer = transformer

  def __call__(self, sentence, max_length=MAX_TOKENS):
    # The input sentence is Portuguese, hence adding the `[START]` and
`[END]` tokens.
    assert isinstance(sentence, tf.Tensor)
    if len(sentence.shape) == 0:
      sentence = sentence[tf.newaxis]

    sentence = self.tokenizers.pt.tokenize(sentence).to_tensor()

    encoder_input = sentence

    # As the output language is English, initialize the output with the
    # English `[START]` token.
    start_end = self.tokenizers.en.tokenize([''])[0]
    start = start_end[0][tf.newaxis]
    end = start_end[1][tf.newaxis]

    # `tf.TensorArray` is required here (instead of a Python list), so
that the
    # dynamic-loop can be traced by `tf.function`.
    output_array = tf.TensorArray(dtype=tf.int64, size=0,
dynamic_size=True)
    output_array = output_array.write(0, start)

    for i in tf.range(max_length):
      output = tf.transpose(output_array.stack())
      predictions = self.transformer([encoder_input, output],
training=False)

      # Select the last token from the `seq_len` dimension.
      predictions = predictions[:, -1:, :]  # Shape `(batch_size, 1,
vocab_size)`.

      predicted_id = tf.argmax(predictions, axis=-1)

      # Concatenate the `predicted_id` to the output which is given to
the
      # decoder as its input.
      output_array = output_array.write(i+1, predicted_id[0])
```

```
      if predicted_id == end:
        break

    output = tf.transpose(output_array.stack())
    # The output shape is `(1, tokens)`.
    text = tokenizers.en.detokenize(output)[0]  # Shape: `()`.

    tokens = tokenizers.en.lookup(output)[0]

    # `tf.function` prevents us from using the attention_weights that
were
    # calculated on the last iteration of the loop.
    # So, recalculate them outside the loop.
    self.transformer([encoder_input, output[:,:-1]], training=False)
    attention_weights = self.transformer.decoder.last_attn_scores
    #move_to_zone(translated_text)

    return text, tokens, attention_weights

translator = Translator(tokenizers, transformer)

def print_translation(sentence, tokens, ground_truth):
  print(f'{"Input:":15s}: {sentence}')
  print(f'{"Prediction":15s}: {tokens.numpy().decode("utf-8")}')
  print(f'{"Ground truth":15s}: {ground_truth}')

import rospy
from std_msgs.msg import String

sentence=""
last_sentence=""


def comandosvoz_callback(data):
    global sentence
    sentence = data.data.lower()

terminar = False

rospy.init_node('listener', anonymous=True)
# Suscribirse al topic "comandosvoz"
rospy.Subscriber("comandosvoz", String, comandosvoz_callback)

while not terminar:
   print("Comando voz: ", sentence)
   if sentence == "":
       continue
```

```
    if sentence.lower() == 'puedes salir':
        terminar = True
        sentence = last_sentence
    else:
        last_sentence = sentence
        ground_truth = input("Introduzca la verdad: ")
        translated_text, translated_tokens, attention_weights =
translator(
        tf.constant(sentence))
        print_translation(sentence, translated_text, ground_truth)
        texto_zon = translated_text.numpy().decode("utf-8")
        print(texto_zon)
        move_to_zone(texto_zon)
print("Saliendo...")




def plot_attention_head(in_tokens, translated_tokens, attention):
  # The model didn't generate `<START>` in the output. Skip it.
  translated_tokens = translated_tokens[1:]

  ax = plt.gca()
  ax.matshow(attention)
  ax.set_xticks(range(len(in_tokens)))
  ax.set_yticks(range(len(translated_tokens)))

  labels = [label.decode('utf-8') for label in in_tokens.numpy()]
  ax.set_xticklabels(
      labels, rotation=90)

  labels = [label.decode('utf-8') for label in translated_tokens.numpy()]
  ax.set_yticklabels(labels)

head = 0
# Shape: `(batch=1, num_heads, seq_len_q, seq_len_k)`.
attention_heads = tf.squeeze(attention_weights, 0)
attention = attention_heads[head]
attention.shape

in_tokens = tf.convert_to_tensor([sentence])
in_tokens = tokenizers.pt.tokenize(in_tokens).to_tensor()
in_tokens = tokenizers.pt.lookup(in_tokens)[0]
in_tokens

translated_tokens

plot_attention_head(in_tokens, translated_tokens, attention)
```

```
def plot_attention_weights(sentence, translated_tokens, attention_heads):
  in_tokens = tf.convert_to_tensor([sentence])
  in_tokens = tokenizers.pt.tokenize(in_tokens).to_tensor()
  in_tokens = tokenizers.pt.lookup(in_tokens)[0]

  fig = plt.figure(figsize=(16, 8))

  for h, head in enumerate(attention_heads):
    ax = fig.add_subplot(2, 4, h+1)

    plot_attention_head(in_tokens, translated_tokens, head)

    ax.set_xlabel(f'Head {h+1}')

  plt.tight_layout()
  plt.show()

plot_attention_weights(sentence,
                       translated_tokens,
                       attention_weights[0])

sentence = 'Por favor ve a la zona cuatro.'
ground_truth = 'Ir zona cuatro.'

translated_text, translated_tokens, attention_weights = translator(
    tf.constant(sentence))
print_translation(sentence, translated_text, ground_truth)

plot_attention_weights(sentence, translated_tokens, attention_weights[0])

class ExportTranslator(tf.Module):
  def __init__(self, translator):
    self.translator = translator

  @tf.function(input_signature=[tf.TensorSpec(shape=[],
dtype=tf.string)])
  def __call__(self, sentence):
    (result,
     tokens,
     attention_weights) = self.translator(sentence,
max_length=MAX_TOKENS)

    return result


translator = ExportTranslator(translator)
```

```
translator('Ve a la zona cinco.').numpy()
tf.saved_model.save(translator, export_dir='translator')
reloaded = tf.saved_model.load('translator')
reloaded('Ve a la zona cinco.').numpy()
```