Trabajo de fin de grado

Universidad de Valladolid

Incorporación de un Reconocedor Automático de Voz local sobre plataformas Android

Autor:
Santiago Blasco Arnaiz

Tutor:
Valentín Cardeñoso Payo

24 de septiembre de 2020

Dedicado a mi familia y amigos, por estar y aguantar.

Agradecimientos

Agradezco a mi familia, por el cariño y apoyo,

a los amigos que me ha dado esta etapa universitaria, por ayudarme siempre en lo posible y amenizarme estos años de estudio,

y por último, a los profesores que aman su profesión, por su dedicación y por transmitirme sus ganas de aprender.

Resumen

Hoy en día, todos tenemos al alcance de nuestra mano un smartphone capaz de transcribir nuestras palabras, es habitual que estos reconocedores de voz necesiten una conexión a internet para llevar a cabo esta función ya que no es nuestro dispositivo el que realiza ese reconocimiento, él sólo se encarga de enviar el audio y recibir la transcripción.

Este trabajo tiene como objetivo implementar un reconocedor automático de voz local, es decir, la creación de una aplicación Android capaz de reconocer palabras o frases sin necesitar acceso a internet para llevar a cabo esta función.

Para crear este reconocedor se utilizará el software proporcionado por **Kaldi** ya que proporciona herramientas para trabajar con redes neuronales profundas, que pueden ser entrenadas de forma eficiente mediante procesamiento por GPU, y también con modelos ocultos de Markov, juntos sirven para generar modelos de lenguaje y actuar como reconocedores automáticos del habla. Para utilizar el modelo que generamos con la herramienta ya mencionada utilizaremos la API de VOSK, que nos proporciona métodos para crear y usar dicho modelo.

Palabras clave: Reconocimiento Automático del Habla (ASR), modelo de lenguaje, Modelos Ocultos de Markov (HMM), Autómatas de Estado Finito (FST), Redes Neuroanles Profundas (DNN), receta, Kaldi, API.

Abstract

Today, we all have at our reach a smartphone able of transcribing our words, it is common for these voice recognizers to need an internet connection to carry out this function because it is not our device which performs this recognition, this only takes care of sending the audio and receiving the transcription.

This work aims to implement an automatic local voice recognizer, that is, the creation of an Android application able to recognize words or phrases without needing internet access to carry out this function.

To create this recognizer, the software provided by **Kaldi** will be used as it provides tools to work with deep neural networks, which can be efficiently trained through GPU processing, and also with hidden Markov models, together they serve to generate language models and act as automatic speech recognizers. To use the model that we generate with the previously mentioned tool, we will use the VOSK API, which provides us methods to create and use models.

Keywords: Automatic Speech Recognition(ASR), language model, Hidden Markov Models (HMM), Finite-State Transducers (FST), Deep Neural Networks (DNN), recipe, Kaldi, API.

Índice general

Agradecimientos **III**

Resumen **V**

Abstract **VII**

Lista de figuras **XI**

Lista de tablas **XIII**

1. Introducción **1**
1.1. Propuesta 1
1.2. Objetivos 1
1.3. Estructura 1

2. Marco teórico **3**
2.1. Reconocimiento automático del habla 3
2.1.1. Historia 3
2.1.2. Bases teóricas 4
2.1.3. Aproximación actual 7
2.2. Inteligencia artificial 7
2.2.1. Fundamentos de redes neuronales artificiales 7
2.2.2. Redes neuronales profundas 8

3. Planificación **11**
3.1. Planificación del proyecto 11
3.2. Herramientas utilizadas 12
3.3. Presupuesto 12
3.3.1. Software 13
3.3.2. Hardware 13
3.3.3. Personal 13

4. Desarrollo **15**
4.1. Entorno de desarrollo 15
4.2. Prototipo cero 15
4.3. Análisis 16
4.3.1. Requisitos funcionales 17
4.3.2. Requisitos no funcionales 17
4.4. Diseño 18
4.4.1. Casos de uso 18
4.4.2. Especificación de los casos de uso 18
4.4.3. Diagrama de casos de uso 19

4.4.4. Diagramas de secuencia 20
4.5. Implementación 23
4.5.1. La aplicación 23
4.5.2. El modelo de lenguaje 24
4.5.3. Implementación 25
4.5.4. La interfaz de usuario 27
4.5.5. Creando un modelo de lenguaje 28
4.5.6. Pruebas 38

5. Conclusiones **43**
5.1. Trabajo a realizar 43

Bibliografía **45**

Apéndices

A. Bibliotecas y herramientas **47**

Lista de figuras

2.1. Primera máquina reconocedora del habla, la Shoebox de IBM) 4
2.2. Esquema de un ASR 5
2.3. Espectro de frecuencia de un segmento de audio 5
2.4. Modelo de neurona artificial (McCulloch y Pitts) 8
2.5. Esquema de una red neuronal profunda básica) 8

3.1. Diagrama de Gantt de la planificación 12

4.1. Reconocedor de Google de la aplicación TFG initial demo 16
4.2. Diagrama de casos de uso 19
4.3. Diagrama de secuencia 1 - Inicio de la aplicación 20
4.4. Diagrama de secuencia 2 - Reconocimiento de voz 21
4.5. Diagrama de secuencia 3 - Cambio del modelo de lenguaje 22
4.6. Diagrama de secuencia 4 - Activación de visualización de datos ampliada . 22
4.7. Diagrama de secuencia 5 - Desactivación de visualización de datos ampliada 23
4.8. Estructura del directorio del modelo de lenguaje por defecto de la demo de VOSK 24
4.9. Estructura del directorio del modelo de lenguaje español de VOSK 24
4.10. Modelo de dominio a nivel de implementación 26
4.11. Icono de la aplicación final 27
4.12. Interfaz de la aplicación TFG VOSK demo 28
4.13. Captura de pantalla de la aplicación TFG VOSK demo para el test 1 . . . 39
4.14. Captura de pantalla de la aplicación TFG VOSK demo para el test 2 . . . 40
4.15. Captura de pantalla de la aplicación TFG VOSK demo para el test 3 . . . 41
4.16. Captura de pantalla de la aplicación TFG VOSK demo para el test 4 . . . 42

Lista de tablas

3.1. Coste por hora de los recursos hardware del proyecto 13
3.2. Coste por hora de las personas implicadas en el proyecto 13

4.1. Caso de uso 1 - Reconocimiento de voz 18
4.2. Caso de uso 2 - Elección del modelo de lenguaje 18
4.3. Caso de uso 3 - Activar visualización ampliada de datos 19
4.4. Caso de uso 4 - Desactivar visualización ampliada de datos 19
4.5. Distribución de hablantes del conjunto TIMIT 29

Capítulo 1

Introducción

1.1. Propuesta

Para este trabajo de fin de grado se propone crear una aplicación Android capaz de reconocer y transcribir palabras aisladas o frases. Estas serán dictadas en inglés ya que existen más corpus basados en ese idioma y es más fácil acceder a ellos. La creación de dicha aplicación conlleva el aprendizaje y utilización de las herramientas necesarias para ello, utilizamos estas herramientas por su gran utilidad y bagaje en sus respectivos campos, en nuestro caso utilizaremos Android Studio para el desarrollo de la app, Kaldi para la generación del modelo de lenguaje que utilizará el reconocedor, VOSK para crear el propio reconocedor, Docker para poder utilizar la herramienta Kaldi y Latex para escribir la memoria del trabajo en sí.

1.2. Objetivos

Este proyecto tiene como principal objetivo familiarizarse y aprender los conceptos fundamentales de los reconocedores automáticos del habla y sus bases, así como trabajar de una forma diferente a lo hasta ahora realizado con inteligencia artificial, para ello daré una solución funcional a un aspecto que utilizamos en nuestro día a día como es el dictado de voz. Este último será integrado en una aplicación móvil de forma que podamos acceder a él de rápidamente y sin necesidad de conexión a internet. Los objetivos concretos del proyecto son:

- Aprender a crear una aplicación Android.
- Estudiar teoría sobre los reconocedores automáticos del habla.
- Poner en práctica esos conocimientos con herramientas actuales.
- Realizar un uso diferente de la inteligencia artificial al realizado en el grado.
- Ayudar a otras personas a poder realizar un proyecto semejante.

1.3. Estructura

El documento se divide en cinco capítulos comenzando por este, que es una breve presentación de la motivación del proyecto y la estructura del documento que lo acompaña para ubicar al lector. En el capítulo dos se exponen los principios históricos y teóricos de las ciencias y herramientas utilizadas. Le prosigue el capítulo tres que plantea la planificación

y costes del proyecto. El capítulo cuatro, destinado al desarrollo, está formado por cinco secciones, la primera trata sobre la preparación del entorno de desarrollo y las herramientas software necesarias para el proyecto, la segunda recoge la familiarización con el entorno Android y las tecnologías que se proveen y utilizan actualmente. Las tres secciones que quedan en este capítulo corresponden a las fases de análisis, diseño e implementación del proyecto respectivamente, en cada uno de ellos se lleva a cabo el desarrollo de la fase correspondiente. El último capítulo está destinado a reflexionar sobre lo aprendido en el proyecto, aspectos a mejorar y trabajo a realizar.

Capítulo 2

Marco teórico

El uso de la voz es de vital importancia en nuestro día a día, la utilizamos para expresar nuestras ideas y deseos a otras personas, para comunicarnos con ellas, incluso para comunicarnos con animales, en general la utilizamos para comunicarnos con nuestro entorno. Cada vez es más propenso el uso de dispositivos en nuestra vida cotidiana, estos dispositivos, ya sea para agilizar nuestra interacción con ellos o porque la condición física de las personas les obligue a ello, nos permiten comunicarnos con ellos a través de la voz.

Efectivamente, nuestro entorno ha pasado a estar repleto de dispositivos que nos ofrecen esta posibilidad de interacción mediante la voz, el más cercano y que todos poseemos es el smartphone. Ya sea para realizar una búsqueda rápida en nuestro navegador de internet, para escribir una nota de texto o para cambiar de canción, todos hemos utilizado un reconocedor de voz. El hecho de que todos hayamos utilizado esta tecnología quiere decir que es útil, rápida y funcional.

2.1. Reconocimiento automático del habla

El reconocimiento automático del habla (ASR, por sus siglas en inglés provenientes de Automatic Speech Recognition) o reconocimiento automático de voz, es una disciplina de la inteligencia artificial que tiene como objetivo permitir la comunicación hablada entre seres humanos y computadoras. [1]

2.1.1. Historia

En los inicios del siglo XX ya había prototipos de máquinas construidas a partir de electroimanes y condensadores capaces de transcribir señales acústicas. Estas máquinas eran muy rudimentarias y no fue hasta medio siglo después, en el año 1962, cuando se realizaron importantes avances en este campo. IBM presentó un equipo denominado "shoebox" capaz de reconocer dieciséis palabras habladas entre las que se encuentran los dígitos del cero al nueve y operaciones aritméticas básicas. El progreso en este campo se vio frenado debido a la falta de fondos.

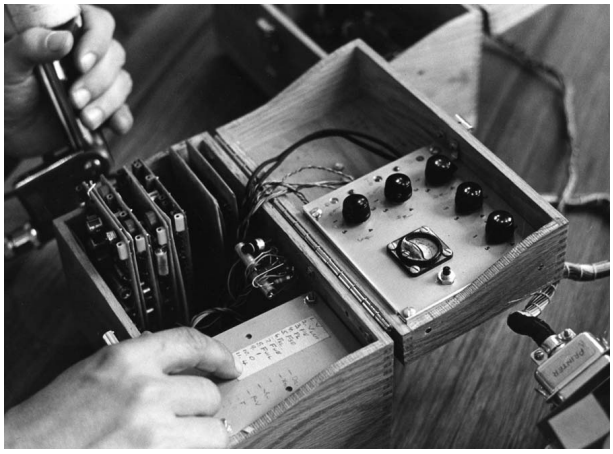

Figura 2.1: Primera máquina reconocedora del habla, la Shoebox de IBM siendo utilizada para realizar una operación matemática[1].

Los fondos se paralizaron durante dos años, en 1971 la Agencia de Proyectos de Investigación Avanzada de Defensa decidió invertir en un proyecto denominado "Speech Understanding Research", que duraría cinco años y tendría como objetivo la creación de nuevos sistemas de ASR. Pese a los intentos por seguir avanzando en este campo, la tecnología no era suficiente para su uso en tareas cotidianas o relativamente abiertas, tan sólo funcionaba en casos concretos dentro de un marco experimental.

A principios de los años 70 surgió un nuevo tipo de modelos estadísticos denominados modelos ocultos de Markov (HMM, por sus siglas en inglés provenientes de Hidden Markov Models) que junto con el avance tecnológico en el campo de la informática, que permitía la implementación de este nuevo modelo, supuso un gran impulso en el campo del reconocimiento del habla, sobre todo para mejorar la precisión de los reconocedores de la época.

2.1.2. Bases teóricas

La comprensión del funcionamiento de un reconocedor automático del habla empieza por el estudio de sus componentes y fundamentos teóricos, comenzando por los del habla, ya que es la información con la que se trabaja. Un locutor habla y, en este caso, se recoge esa voz en un archivo digital de audio para procesar y transcribir lo que se ha dicho.

El locutor emite secuencias de sonidos a las que llamamos palabras y al registro de la forma en que pronuncia cada una de esas palabras se le llama léxico de pronunciación. La forma de indicar la pronunciación de una palabra es descomponiendo esta en los sonidos que la caracterizan, a estas partes se les llama fonemas.

Un fonema se define como: *"Unidad fonológica que no puede descomponerse en unidades sucesivas menores y que es capaz de distinguir significados."* [2] Cada idioma tiene un número reconocido de fonemas, en particular el español consta de veinticuatro y el inglés de cuarenta y cuatro. Por ejemplo, la palabra "casa" se descompone en cuatro fonemas, uno de ellos repetido, su pronunciación sería /k/ /a/ /s/ /a/, puesto que esos cuatro fonemas o sonidos juntos producen la palabra "casa".

Un reconocedor automático del habla se compone principalmente de un extractor de características y de un decodificador que se apoya en un modelo acústico, un léxico de

[1]Imagen de IBM [3]

pronunciación y un modelo de lenguaje como puede verse en la figura 2.2. A continuación, procedo a explicar dichos componentes y su funcionamiento.

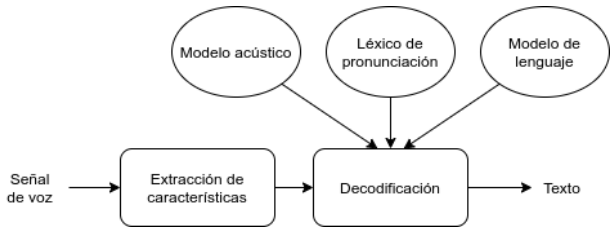

Figura 2.2: Esquema del funcionamiento y componentes de un ASR.

Extracción de características

En el caso del reconocimiento de voz el tipo de datos con el que se trabaja son muestras de voz, al conjunto de estas se le llama corpus de voz. La voz en este caso es capturada en un formato digital y representable por un espectro de frecuencia como el de la figura 2.3. De segmentos de audio como el mencionado se extraen las características, para ello se divide el segmento en marcos de un intervalo de tiempo fijo. La fonética es muy dependiente del contexto, por ello estos marcos se solapan entre sí para poder capturar y representar debidamente esta dependencia entre ellos.

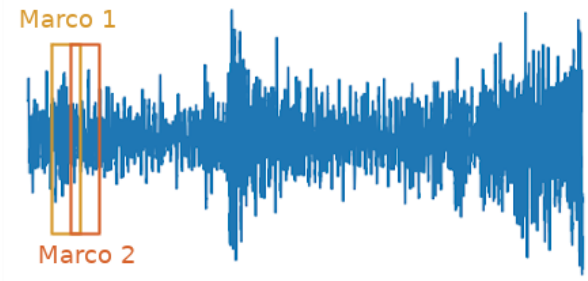

Figura 2.3: Espectro de frecuencia de un segmento de audio.

Uno de los métodos de extracción de características más utilizado son los coeficientes cepstrales de frecuencias de Mel (MFCC, por sus siglas en inglés provenientes de Mel Frequency Cepstral Coefficients) que tienen exactamente treinta y nueve parámetros e intentan representar cómo escucha el oído humano. Este proceso de extracción conlleva una serie de pasos para acondicionar los datos y centrarse en la información relevante y la independencia de las características.

Comienza discretizando la señal de los datos, como ya he explicado el segmento de audio se divide en marcos, estos deben ser suavizados en sus extremos, ya que si no estaremos generando ruido en las muestras, sobre todo en las que comiencen o acaben en frecuencias altas. Se realiza una transformación discreta de Fourier sobre los datos con el fin de extraer información para posteriormente realizar otra, pero esta vez usando los filtros de Mel, con esto consigue imitarse la forma en la que el oído humano funciona y percibe los sonidos.

Finalmente, calcula el logaritmo y mediante la transformada de Fourier discreta inversa (IDFT, por sus siglas en inglés provenientes de Inverse Discrete Fourier Transform) se obtienen los coeficientes cepstrales de las frecuencias de Mel.

Llegado este punto, conocido como la normalización de la varianza y la media cepstral (CMVN, por sus siglas en inglés provenientes de Cesptral Mean and Variance Normalization), las características pueden normalizarse, con esto se logra reducir el impacto del ruido en la extracción de características, aunque hay que mencionar que en segmentos cortos de audio su rendimiento decae.

Modelo acústico

Un modelo acústico es un conjunto de representaciones estadísticas de los sonidos que componen cada palabra. Como ya he explicado, las palabras pueden dividirse en fonemas, estos fonemas pueden ser representados estadísticamente mediante modelos ocultos de Markov).

Los modelos acústicos se entrenan con muestras de voz y sus correspondientes transcripciones, así se consigue un conjunto de HMM. Por lo general, en cuanto más grande sea el corpus de voz y más variedad de sonidos contenga mejor funcionará el modelo acústico. Cada HMM tiene un fonema como etiqueta, de esta forma el reconocedor utiliza el modelo acústico para buscar una coincidencia entre los distintos HMM y la muestra de sonido que esté analizando.

Cuando encuentra una coincidencia toma nota del fonema que le corresponde, así hasta que llega una pausa en el sonido, que indica el final de la palabra. Con la secuencia de fonemas ya identificados puede contrastar con su léxico de pronunciación, si esta secuencia se encuentra en él obtendrá la palabra asociada.

Modelo de lenguaje

Si poseemos información sobre la probabilidad de ocurrencia de cada palabra o de una secuencia de palabras podemos mejorar sustancialmente la precisión del reconocedor. Este es el fin de los modelos de lenguaje, calcular la probabilidad de ocurrencia de las palabras. A los modelos de lenguaje se les nombra en función del número de palabras que tengan en cuenta para calcular la probabilidad de una. Un modelo de lenguaje n-grama es aquel que tiene en cuenta n palabras para calcular la probabilidad de ocurrencia de una, por ejemplo, en un modelo de lenguaje bi-grama la palabra de la que se quiere saber su probabilidad de ocurrencia sólo tiene en cuenta la palabra anterior y a sí misma, en total dos palabras.

Estas probabilidades se calculan en base al conteo de la aparición de las palabra que componen el corpus de voz, por tanto, en cuánto mayor sea el corpus mejor, ya que tendrá más combinaciones posibles de palabras. Pese a utilizar corpus de voz de gran volumen, suele no ser suficiente, por ello se realiza un suavizado del recuento con el que se tiene en cuenta hasta las palabras que no aparecen.

Modelos ocultos de Markov

Un HMM [4] se compone esencialmente de una cadena de Markov, con un pequeño inciso que explicaré más adelante. Una cadena de Markov tiene un número finito de estados en el que el siguiente estado depende únicamente del estado actual y al que pasa con cierta probabilidad, el inciso que la convierte en un HMM es el hecho de que existan estados observables y no observables u ocultos, de ahí su nombre. Su objetivo es intentar definir esos parámetros ocultos en base a los parámetros observables asociando a cada uno de estos un estado, en el caso del ASR cada estado equivale a un fonema y los parámetros a las características.

2.1.3. Aproximación actual

Hoy en día, es común utilizar modelos acústicos híbridos, que mezclan redes neuronales profundas (DNN, por sus siglas en inglés provenientes de Deep Neural Network) con HMM, que parten de un modelo HMM-GMM en el que sustituyen la parte de GMM por una DNN. La entrada de la DNN son las características del audio y la salida son fonemas o palabras, es decir, cada neurona en la capa de entrada está asociada con una característica, y cada neurona en la capa de salida está asociada a un estado del HMM. Gracias a este híbrido es posible aumentar la precisión de los reconocedores, además su entrenamiento es más simple que el de los antiguos modelos, ya que se entrena a la red directamente con el audio y las transcripciones de dicho audio, sin necesidad de realizar un filtrado de ruido.

2.2. Inteligencia artificial

La inteligencia artificial (AI, por sus siglas en inglés provenientes de Artificial Intelligence) se define como: *"Disciplina científica que se ocupa de crear programas informáticos que ejecutan operaciones comparables a las que realiza la mente humana, como el aprendizaje o el razonamiento lógico."* [5]

En esta definición es importante remarcar el término "comparables", ya que existe cierta paradoja alrededor de la AI y de si esta es realmente inteligencia debido a la naturaleza de las máquinas y su incapacidad para pensar, por ello la AI se limita a simular las capacidades cognitivas humanas, por el momento no puede alcanzar una inteligencia propia. El objetivo de la AI es, en definitiva, realizar acciones con base en la información que percibe de su entorno, esta le permite alcanzar el mejor resultado posible en la solución de su tarea.

Con ese objetivo en mente en los años cuarenta surgió esta disciplina de la mano de matemáticos, estadísticos, probabilistas, informáticos y neurobiólogos aunque no se consolidó ni se le dio nombre hasta 1956 en el evento de la Universidad de Dartmouth sobre AI. [6]

Dentro del ámbito de la AI surge una nueva rama conocida como machine learning o aprendizaje automático. Esta disciplina simula el razonamiento y aprendizaje de las personas y la manera en la que estas realizarían tareas tales como el reconocimiento de imágenes, la toma de decisiones, la estimación de parámetros, la clasificación de objetos y, por supuesto, el reconocimiento de voz. Para aprender a realizar estas tareas correctamente se valen del conocimiento que les es transferido mediante ejemplos que no son más que muestras de datos. Su capacidad para realizar la tarea para la que fueron creados va mejorando, debido a que aprenden con el tiempo, las muestras y sus propios resultados.

2.2.1. Fundamentos de redes neuronales artificiales

En el año 1943 el neurofisiólogo Warren McCulloch y el matemático Walter Pitts proponen el primer modelo matemático de una neurona artificial, este pretende imitar el funcionamiento de una neurona celular.

Este modelo de célula tiene una serie de entradas (x1,x2,...,xn) que equivaldrían a las dendritas, el sumatorio ponderado por los pesos de esas entradas ($\sum$) equivaldría al cuerpo celular y por último, tendríamos una función de activación junto con un valor umbral que determinarían la salida de la neurona, la cual sirve de conexión con otras neuronas es equivalente al axón en las células neuronales.

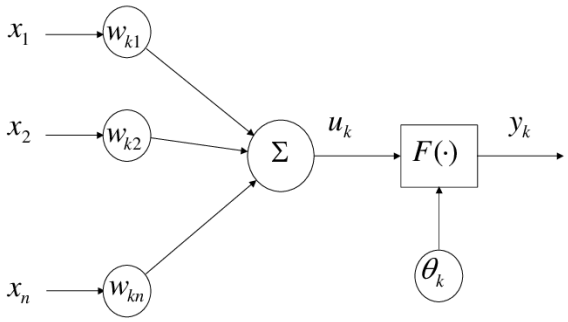

Figura 2.4: Modelo de neurona artificial propuesto por McCulloch y Pitts[2].

Un conjunto interconectado de este objeto, al que llamamos neurona artificial, forma una red neuronal artificial (ANN, por sus siglas en inglés provenientes de Artificial Neural Network). Existen diversos tipos de ANN, pero todas tienen una estructura básica similar. Poseen una capa de entrada formada por un número variable de neuronas que reciben los datos y los propagan a la siguiente capa, que puede ser directamente la capa de salida, la cual también está formada por un número variable de neuronas cuya activación determina la salida de la red y pueden tener entre medias un tipo de capas llamadas ocultas. Estas últimas interconectan los dos tipos de capas ya mencionados y dotan a la red de mayor profundidad y libertad para modelar los problemas.

Hay muchos tipos de redes neuronales, entre las más importantes se encuentran el perceptrón simple, el perceptrón multicapa, las de tipo recurrente, las de tipo convolucional y por último, las redes neuronales profundas.

2.2.2. Redes neuronales profundas

La estructura de este tipo de redes es la misma que la que acabo de definir, con la imposición de que al menos tenga una capa oculta y todas las neuronas de una capa estén conectadas con todas las de la siguiente. Por ejemplo, en la DNN de la figura 2.5 podemos ver una red con una capa de entrada compuesta por dos neuronas, una capa oculta compuesta por cuatro y una capa de salida compuesta por una única neurona, también puede verse la conexión entre capas que mencionaba.

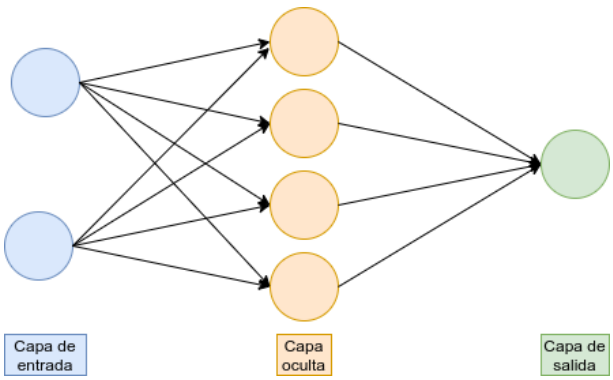

Figura 2.5: Esquema de una red neuronal profunda básica

[2]Imagen de ResearchGate [7]

Esta estructura permite a la red modelar problemas multidimensionales tanto lineales como no lineales. Se denominan profundas porque suelen poseer muchas capas, por tanto estas redes poseen una gran dimensión aunque no una gran complejidad ya que son del tipo "feedforward", donde la información sólo se propaga hacia delante. Cada neurona calcula el peso de sus entradas y en base a su función de activación determina su salida.

Hoy en día, las DNN son muy utilizadas debido a los avances en tecnología. El uso de hardware dedicado, el cual posee una gran cantidad de pequeñas unidades de procesamiento, permite realizar una enorme cantidad de operaciones en paralelo que agilizan el proceso de entrenamiento de las redes.

Capítulo 3

Planificación

Los recursos de los que disponemos en un proyecto son limitados, ya hablemos de tiempo, dinero o personas. Por ello, es de suma importancia organizarlos, tanto para controlar que no excedemos sus límites como para aprovecharlos al máximo. En este capítulo se expone la planificación del proyecto, partiendo de la enumeración de las tareas a realizar y la estimación de su duración a lo largo de las semanas. También se detallan los recursos y herramientas utilizados, esto junto con la estimación de horas nos permite formar un presupuesto para el proyecto.

3.1. Planificación del proyecto

El proyecto, al tratarse de un trabajo de fin de grado, debe suponer 300 horas de trabajo personal, el objetivo es presentar dicho proyecto en Septiembre. Se acuerda con el tutor la realización de una reunión quincenal para seguir los avances y consultar posibles dudas, las reuniones tendrán una duración estimada de dos horas, pero hay que tener en cuenta que estas pueden alargarse o pueden surgir más reuniones de improvisto y también las dudas que se planteen por aplicaciones de mensajería o correo electrónico, por tanto, se calcula un total de treinta horas.

Las primeras semanas se dedicarán a concretar la propuesta del proyecto e instalar lo necesario para este. Una vez realizada la instalación podré comenzar a trabajar con las diversas herramientas y estudiar las bases teóricas que fundamentan el proyecto. Para familiarizarme con el entorno Android y ver qué opciones de implementación se ofrecen hoy en día, crearé una aplicación inicial y posteriormente, habiéndome familiarizado con el entorno Android, crearé la que será la aplicación final. Además, entrenaré un modelo de lenguaje mediante un corpus de voz y lo introduciré en la aplicación para poder utilizarlo. Toda esta planificación y su duración en tiempo viene especificada en el diagrama de Gantt de la figura 3.1.

Figura 3.1: Diagrama de Gantt que muestra la planificación inicial del proyecto.

3.2. Herramientas utilizadas

- Android Studio: El entorno de desarrollo escogido para mi aplicación, ya que esta será una aplicación Android y además, tiene integración con GitHub.
- Kaldi: Herramienta utilizada para reconocimiento del habla y procesamiento de la señal de sonido, con ella entrenaré el modelo que utilizará el reconocedor.
- CUDA: Utilizado para el procesamiento en paralelo cuando se entrenan las redes neuronales.
- Docker: Para aislar la instalación de las herramientas anteriormente nombradas en la máquina Cronos.
- VOSK: Herramienta para trabajar y crear modelos de lenguaje y reconocedores de voz, posee una implementación para Android.
- Latex: La plataforma Overleaf ofrece un compilador y editor online de código sin necesidad de instalación.
  Junto con las siguientes herramientas online para la generación de tablas y diagramas.
  - tablesgenerator.com: Generador de tablas para Latex.
  - draw.io: Permite generar diagramas UML y de propósito general.
  - lucidchart.com: Permite generar diagramas de Gantt.

3.3. Presupuesto

Desgloso el impacto económico del proyecto en tres grupos que abarcan todos los recursos utilizados.

3.3.1. Software

Para el desarrollo del proyecto, así como para la realización de la memoria que acompaña a este se han utilizado las herramientas software ya mencionadas en la sección 3.2.

Todas ellas son de uso gratuito, por tanto, su adquisición y uso no ha supuesto ningún gasto.

3.3.2. Hardware

El servidor al que me conecto de forma remota para trabajar, apodado Cronos, dispone de un procesador de octava generación de la serie i7 fabricado por Intel, concretamente el i7-8700K, que ronda los 300 €. Está equipado con una tarjeta gráfica fabricada por Nvidia basada en la micro arquitectura Pascal, una 1070 ti con seis giga bytes de VRAM, que ronda los 400 €. Por último consta de una memoria RAM total de dieciséis giga bytes DDR4, que junto al resto de componentes como la placa base, la fuente de alimentación y demás suman entre todo lo enumerado un total aproximado de 1500 €. Se estima que el tiempo de utilización de esta máquina sea de unos ocho años, por lo tanto el coste de uso por hora será de 0.021 €. Este hardware se estará utilizando la mayor parte del tiempo pero no todo, así que asumo un uso de unas doscientas cincuenta horas.

Mi ordenador personal fue comprado con el fin de realizar mis estudios universitarios aunque estimo que su vida útil será de al menos cinco años, el coste del ordenador es de 700 €, lo cual nos da un coste de uso por hora de 0.015 €. Este se utilizará para todas las tareas del proyecto, así que se calculan trescientas horas de uso.

El smartphone que se utiliza para el despliegue de la aplicación Android es un Xiaomi Mi 9T con una esperanza de vida de unos tres años, su precio fue de 300 €, por tanto, su coste de uso por hora es de 0.011 €. Sólo se utilizará para probar la aplicación durante el desarrollo, por ello estimo un uso de unas cien horas.

Hardware	Coste	Coste por hora	Coste total
Smartphone	300 €	0.011 €/hora	1.1 €
Ordenador portátil	700 €	0.015 €/hora	4.5 €
Cronos	1500 €	0.021 €/hora	5.25 €

Tabla 3.1: Coste por hora de los recursos hardware del proyecto

3.3.3. Personal

En el proyecto somos dos personas implicadas, yo, que ocupo un puesto de desarrollador con un suelo estimado de unos 18.000 €anuales, y mi tutor, Valentín, cuyo sueldo se estima en 40.000 €anuales.

Teniendo en cuenta que el año 2020 tiene 253 días laborables, el coste por hora de los dos implicados sería el que se muestra en la tabla 3.2.

Persona	Cargo	Coste por hora	Coste total
Santiago	Desarrollador	8.89 €/hora	2700 €
Valentín	Profesor universitario	19.8 €/hora	600 €

Tabla 3.2: Coste por hora de las personas implicadas en el proyecto

Teniendo en cuenta el desglose de horas y coste por hora que he realizado, la suma total del proyecto asciende a un total aproximado de 3310 €.

Capítulo 4

Desarrollo

Como puede verse en el diagrama de Gantt de la figura 3.1, las tareas previas a las fases típicas de desarrollo de la aplicación son la instalación de las herramientas y software necesario, de la que hablaré a continuación, y lo que he llamado una aproximación de aplicación Android, que es un sprint inicial en el que implementaré una aplicación Android para familiarizarme con el entorno y explorar las herramientas actuales que existen para el reconocimiento de voz.

4.1. Entorno de desarrollo

Para comenzar a trabajar con la herramienta Kaldi, y dado que la máquina de la que dispongo no posee una tarjeta gráfica compatible con la plataforma CUDA, necesaria para el procesamiento en paralelo utilizado en los cálculos de las redes neuronales artificiales de Kaldi, trabajaré de forma remota con una máquina de la Escuela de Ingeniería Informática, apodada Cronos, que dispone de unas especificaciones técnicas con las que cubrimos las incompatibilidades que sufríamos, especificadas en el apéndice A, y con la que tenemos potencia y capacidad de procesamiento suficientes.

En esta máquina instalamos Docker y después buscamos la imagen oficial de Kaldi. En la página oficial de Docker [8] indican las imágenes que tienen disponibles y cómo ejecutarlas, a nosotros nos interesa la basada en uso de la GPU. Una vez descargada dicha imagen, podemos proceder a utilizarla en tantos contenedores como queramos, no sin antes actualizar los drivers de CUDA, o instalarlos si no se tienen, ya que los necesitaremos para poder realizar el procesamiento con la GPU.

La aplicación se desarrollará en el entorno Android Studio así que descargo e instalo su última versión. Para ejecutar la aplicación utilizaré mi smartphone personal, así evitamos los problemas de simular dispositivos virtuales en el entorno, como la falta de ciertas características o el consumo elevado de memoria.

La instalación y ejecución de estas herramientas viene explicada en el apéndice A, de esta forma se pueden replicar los pasos a seguir para poder comenzar a trabajar desde cero.

4.2. Prototipo cero

Para iniciarme en el desarrollo de aplicaciones Android destinadas a realizar funciones de captura y reconocimiento de voz comienzo creando una pequeña aplicación [9]. Esta utiliza la API (Application Programming Interfaces) speech de Google, cuya documentación podemos encontrar en la página oficial de Android [10]. Concretamente utilizo la clase

RecognizerIntent, que como su propio nombre indica es un objeto intent, estos permiten invocar componentes, de esta forma puedo crear y configurar el reconocedor del habla e iniciar una actividad que lo ejecute.

Aunque este reconocedor de voz en concreto no es capaz de transcribir archivos de sonido ya grabados, decido implementar un grabador y un reproductor utilizando la API media de Android [11] y sus clases *MediaPlayer* y *MediaRecorder*. De esta manera doy la opción de grabar y reproducir archivos de sonido como prueba concepto para reconocer las palabras de dichos archivos, aunque, como ya he dicho, el reconocedor que usa esta aplicación es sólo en tiempo real y además como puede verse en la figura 4.1 este conecta con Google para realizar la transcripción, por tanto requiere conexión a internet, no es local.

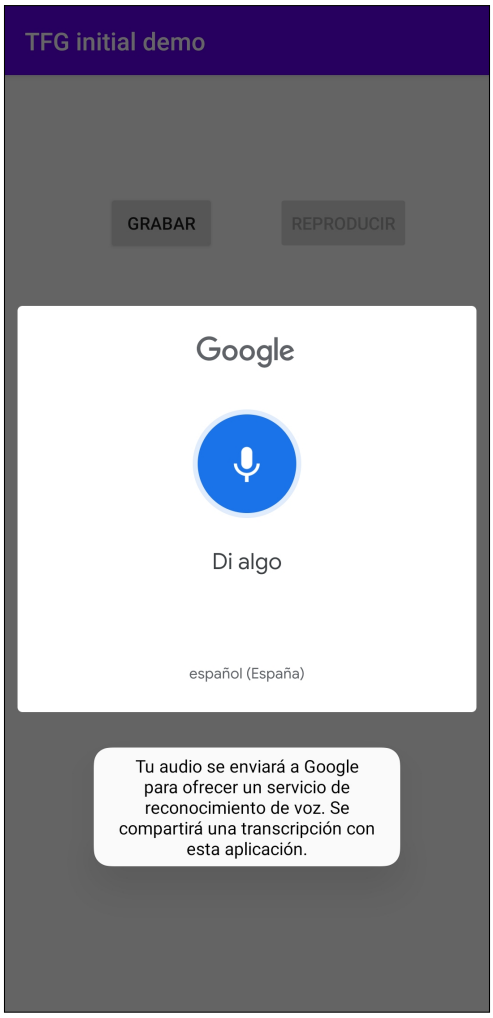

Figura 4.1: Captura de pantalla de la aplicación TFG initial demo usando la API speech de Google.

4.3. Análisis

Teniendo claros los objetivos y las tareas que componen este proyecto, procedo a extraer y enumerar los diferentes requisitos que debe cumplir la aplicación final. En base a estos requisitos se desarrollará la aplicación con el fin de satisfacerlos.

4.3.1. Requisitos funcionales

A continuación, aparece un listado de los requisitos funcionales para la aplicación:

1. La aplicación deberá permitir al usuario iniciar el reconocimiento de voz mediante un botón.
2. La aplicación deberá permitir al usuario detener el reconocimiento de voz mediante un botón.
3. La aplicación deberá permitir al usuario escoger con qué modelo de lenguaje quiere realizar el reconocimiento.
4. La aplicación deberá mostrar al usuario con qué modelo de lenguaje se va a realizar el reconocimiento.
5. La aplicación deberá poder mostrar una visualización extendida de la transcripción, en la que se muestren datos adicionales sobre esta.
6. La aplicación deberá permitir habilitar la visualización extendida de datos mediante un botón.
7. La aplicación deberá permitir deshabilitar la visualización extendida de datos mediante un botón.
8. La aplicación deberá mostrar al usuario si se cambia al modo de visualización extendida de datos.
9. La aplicación deberá seguir mostrando la última transcripción tras haber finalizado el reconocimiento de voz.
10. La aplicación deberá seguir mostrando transcripciones mientras el reconocimiento de voz siga activo.
11. La aplicación deberá impedir que se pulse cualquier botón que no sea el que finaliza el reconocimiento de voz mientras este esté funcionando.
12. La aplicación deberá impedir que se pulse cualquier botón mientras se está cargando el modelo de lenguaje.
13. La aplicación deberá funcionar con todos los modelos de lenguaje introducidos.

4.3.2. Requisitos no funcionales

A continuación, aparece un listado de los requisitos no funcionales para la aplicación:

1. La aplicación deberá iniciarse en menos de 1 segundo.
2. La aplicación deberá ser capaz de funcionar sin conexión a internet.
3. La aplicación deberá funcionar en la versión 10 de Android.
4. La aplicación deberá capturar audio a través de un micrófono.
5. La aplicación debe transcribir el audio capturado y mostrarlo en pantalla en menos de 1 segundo.

6. La aplicación deberá ser capaz de trabajar en dos idiomas, inglés y español.

7. La aplicación debe realizar el cambio de modelo de lenguaje utilizado sin reiniciar la propia aplicación.

4.4. Diseño

4.4.1. Casos de uso

Para esta aplicación sólo se cuenta con un actor, el usuario, todas las personas que utilicen la aplicación tienen el mismo nivel de acceso y permisos, por ende, pueden realizar exactamente las mismas tareas.

4.4.2. Especificación de los casos de uso

Caso de uso 1 - Reconocimiento de voz	
Descripción	El sistema deberá permitir al usuario, una vez lanzada la aplicación y escogido el modelo de lenguaje, utilizar el reconocedor de voz para transcribir lo que dicte.
Secuencia normal	1 - El usuario activa el reconocimiento de voz mediante el botón correspondiente. 2 - El usuario dicta las palabras que desea transcribir. 3 - La aplicación captura las palabras mediante el micrófono. 4 - La aplicación utiliza el reconocedor para transcribir la voz del usuario. 5 - La aplicación muestra la transcripción al usuario. 6 - El usuario desactiva el reconocimiento de voz mediante el botón correspondiente.
Excepciones	1.1 - El reconocedor de voz no es capaz de iniciarse debido a un problema. 1.2 - La aplicación muestra un mensaje de error. 1.3 - La aplicación se cierra.

Tabla 4.1: Caso de uso 1 - Reconocimiento de voz

Caso de uso 2 - Elección del modelo de lenguaje	
Descripción	El sistema deberá permitir al usuario, una vez lanzada la aplicación, escoger el modelo de lenguaje con el que desea que se realice el reconocimiento.
Secuencia normal	1 - El usuario utiliza el botón destinado para cambiar de modelo de lenguaje. 2 - La aplicación muestra al usuario el modelo de lenguaje que se ha seleccionado. 3 - La aplicación crea un nuevo reconocedor con el siguiente modelo de lenguaje que se ha seleccionado. 4 - La aplicación informa al usuario de que el reconocedor ya está listo con el modelo de lenguaje que se ha escogido.
Excepciones	1.1 - El reconocedor de voz no es capaz de iniciarse debido a un problema. 1.2 - La aplicación muestra un mensaje de error. 1.3 - La aplicación se cierra.

Tabla 4.2: Caso de uso 2 - Elección del modelo de lenguaje

Caso de uso 3 - Activación de visualización ampliada de los datos	
Descripción	El sistema deberá permitir al usuario, una vez lanzada la aplicación, activar la visualización ampliada de los datos de la transcripción.
Secuencia normal	1 - El usuario utiliza el botón destinado a la activación de la visualización ampliada de datos. 2 - La aplicación activa la visualización de datos ampliada. 3 - La aplicación informa al usuario de que la visualización de datos ampliada ha sido activada.

Tabla 4.3: Caso de uso 3 - Activar visualización ampliada de datos

Caso de uso 4 - Desactivación de visualización ampliada de los datos	
Descripción	El sistema deberá permitir al usuario, una vez lanzada la aplicación, desactivar la visualización ampliada de los datos de la transcripción.
Secuencia normal	1 - El usuario utiliza el botón destinado a la activación de la visualización ampliada de datos. 2 - La aplicación activa la visualización de datos ampliada. 3 - La aplicación informa al usuario de que la visualización de datos ampliada ha sido activada.

Tabla 4.4: Caso de uso 4 - Desactivar visualización ampliada de datos

4.4.3. Diagrama de casos de uso

Diagrama de los casos de uso realizables por el actor usuario.

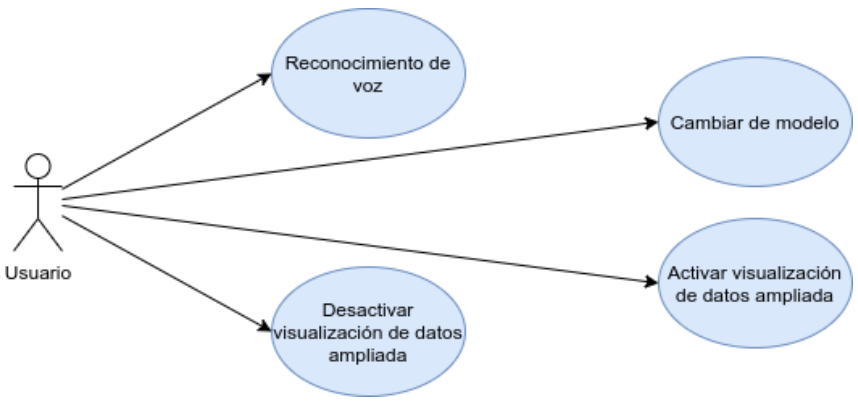

Figura 4.2: Diagrama de casos de uso.

4.4.4. Diagramas de secuencia

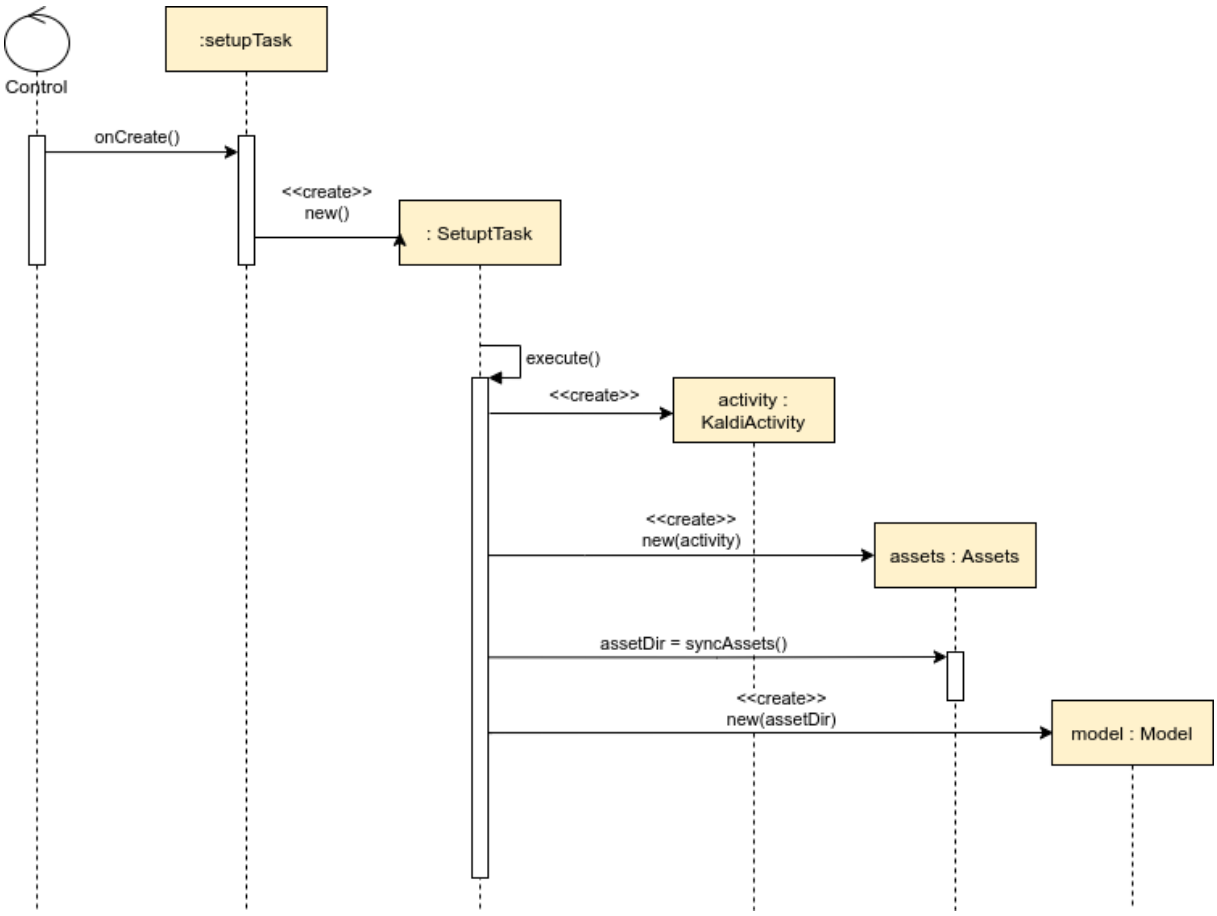

Figura 4.3: Diagrama de secuencia 1 - Inicio de la aplicación.

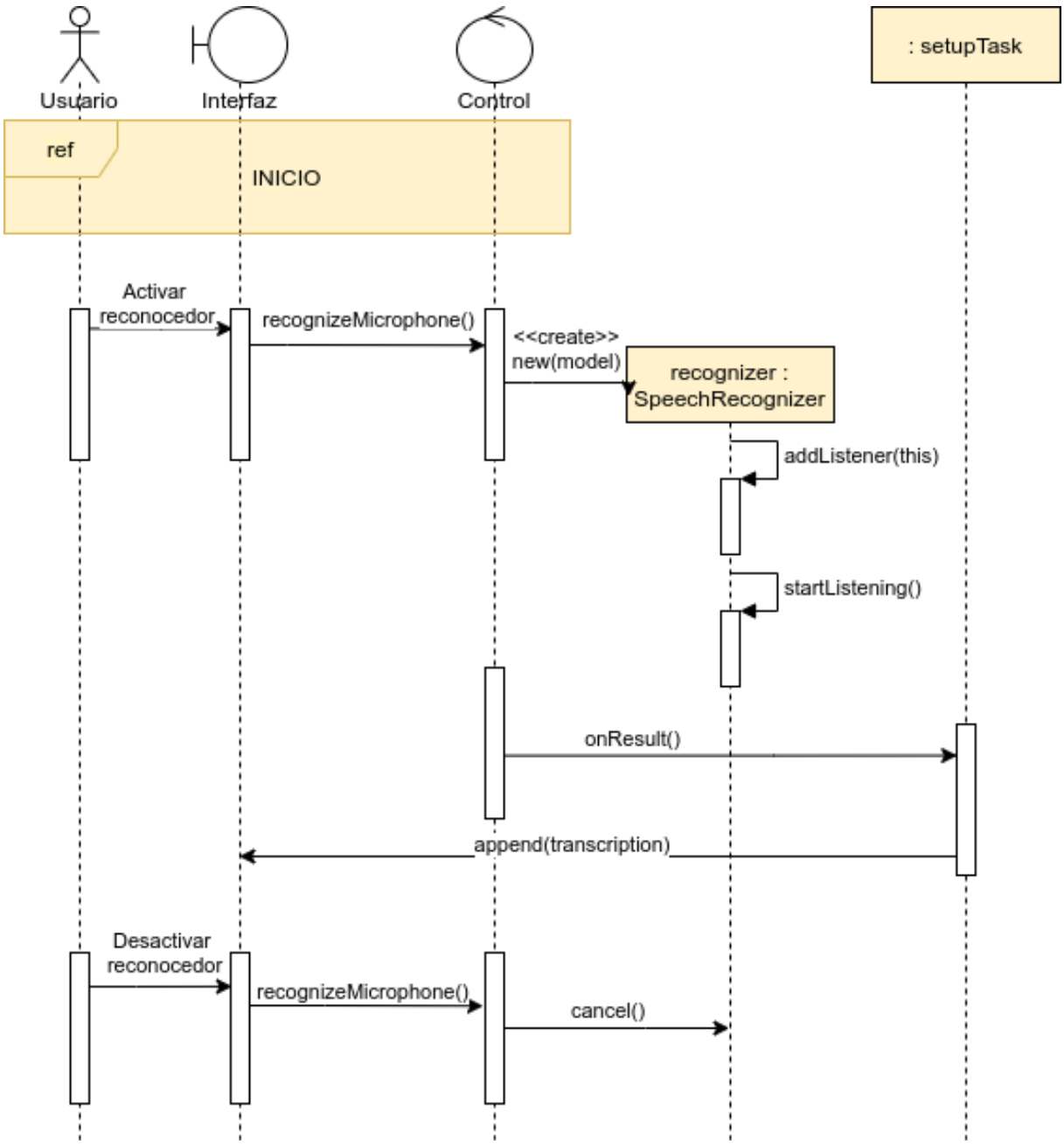

Figura 4.4: Diagrama de secuencia 2 - Reconocimiento de voz.

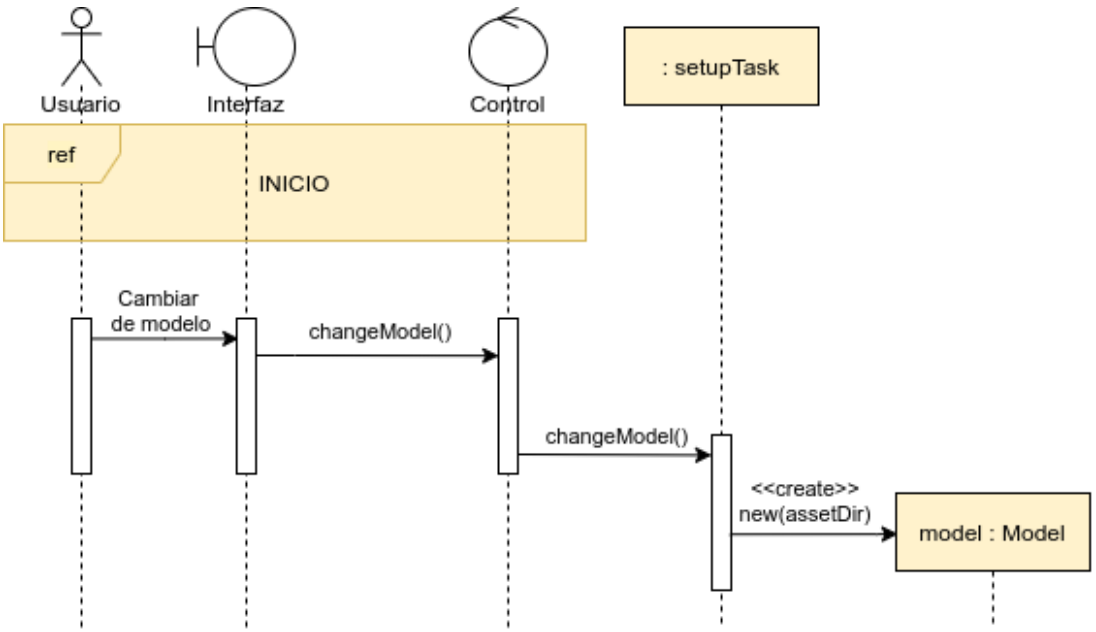

Figura 4.5: Diagrama de secuencia 3 - Cambio del modelo de lenguaje.

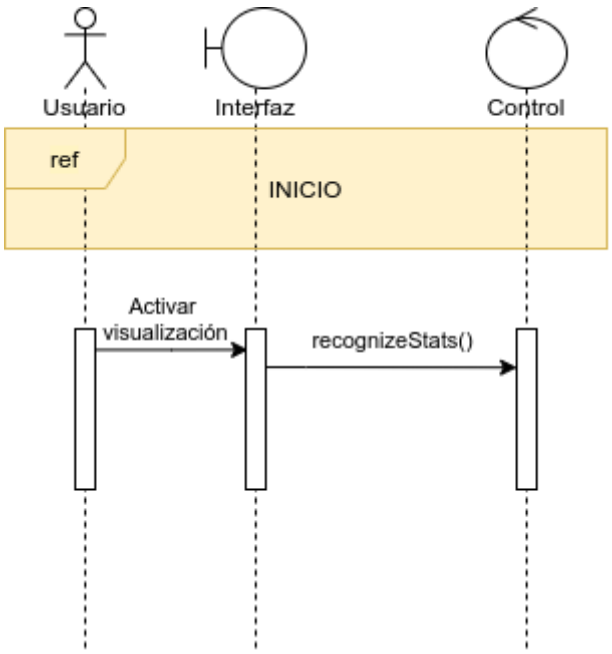

Figura 4.6: Diagrama de secuencia 4 - Activación de visualización de datos ampliada.

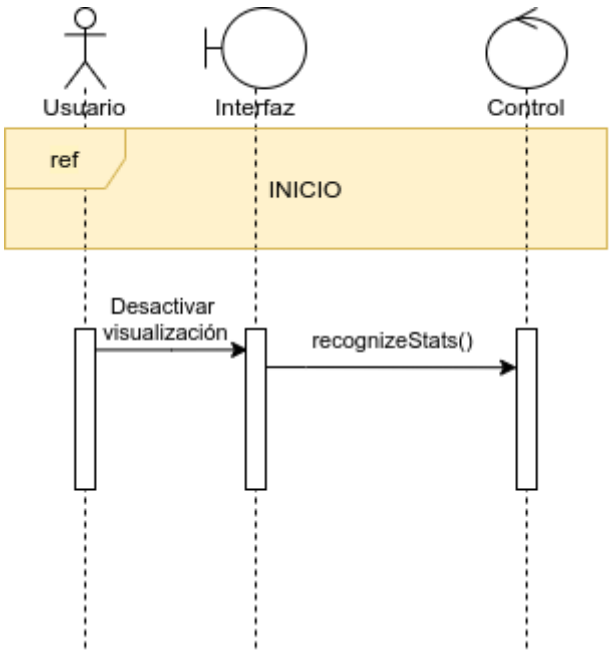

Figura 4.7: Diagrama de secuencia 5 - Desactivación de visualización de datos ampliada.

4.5. Implementación

Para crear la aplicación principal del proyecto utilizo la API de VOSK, esta permite a través de sus clases crear reconocedores de voz basándose en los modelos de lenguaje generados con la herramienta Kaldi, de los que hablaremos más adelante. En el perfil de GitHub de Alpha Cephei, corporación encargada del proyecto VOSK, podemos encontrar un repositorio dedicado a su API [12] para realizar reconocimiento de voz offline en diferentes dispositivos, entre ellos Android. El proyecto vosk-android-demo [13] utiliza las librerías de VOSK y Kaldi para funcionar, partiré de dicho ejemplo para desarrollar mi aplicación.

Para comenzar realizo un fork del proyecto de GitHub, de esta forma podré realizar una copia del proyecto en mi máquina y trabajar en él realizando modificaciones respaldadas por un control de versiones online con el que asegurar mi trabajo. Comienzo importando el proyecto a Android Studio, para ello debo acceder a los ajustes de control de versiones e iniciar sesión en la plataforma escogida, en este caso GitHub, una vez he iniciado sesión ya puedo importar el proyecto desde el control de versiones.

Para comprobar que el proyecto se ha importado correctamente lo ejecuto, si se produce un error puede deberse a las licencias del SDK (Software Development Kit), deben aceptarse a través del administrador de SDK de Android Studio, este se encargará de instalar cualquier componente que falte y permitirá aceptar los acuerdos de las licencias.

4.5.1. La aplicación

El proyecto posee un directorio *models*, dentro de él se ubican los modelos de lenguaje, mejor dicho, se encuentran los directorios que contienen los archivos que definen cada modelo de lenguaje, en la figura 4.8 pueden verse los archivos y directorios que contiene el modelo de lenguaje por defecto de la aplicación.

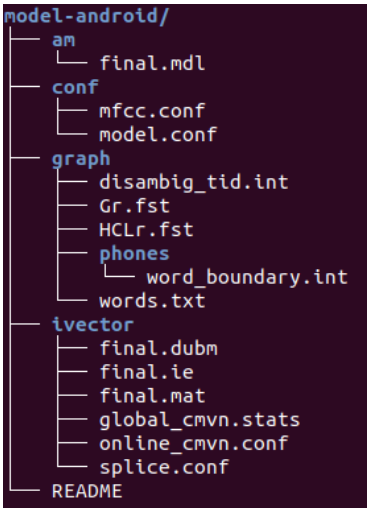

Figura 4.8: Estructura del directorio del modelo de lenguaje por defecto de la demo de VOSK.

Además de este modelo de lenguaje que viene por defecto decido probar a introducir uno de los modelos de lenguaje que provee VOSK en su web oficial [14], concretamente el español, según dicen en la propia página la estructura de directorios no tiene por qué ser como la del modelo de lenguaje por defecto, también es válida la de los modelos de lenguaje que proporcionan, como la del modelo de lenguaje español, que puede verse en la figura 4.9.

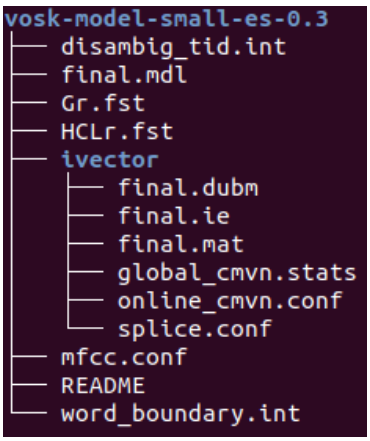

Figura 4.9: Estructura del directorio del modelo de lenguaje español de VOSK.

Comparando ambos directorios podemos notar que faltan algunos archivos, a continuación voy a explicar la función de cada archivo del modelo de lenguaje por defecto de VOSK.

4.5.2. El modelo de lenguaje

Voy a definir la estructura del modelo de lenguaje que hemos visto en la figura 4.8, pese a que tiene más archivos que el mínimo necesario para funcionar me parece interesante mencionar estos y explicar su función y características.

En el directorio *am* se encuentra el archivo *final.mdl* que contiene el modelo acústico, en la sección 2.1.2 se encuentra la definición y explicación en detalle de lo que es un modelo acústico.

Los archivos *mfcc.conf* y *model.conf* se encuentran en el directorio *conf*, y contienen parámetros de configuración, como su propio nombre indica. El primero contiene los parámetros referentes a las características MFCC, estos coeficientes son necesarios para diferenciar entre los componentes de las señales de audio que aportan información útil y los que no. El segundo archivo contiene información sobre el modelo de lenguaje, sobre las pautas de inicio y fin de los audios y los fonemas de silencio .

En caso de que el modelo haya sido entrenado con ivectors deberá existir este directorio que contendrá los archivos generados por el extractor de ivectors. En este caso utilizo el entrenamiento por DNN así que sí dispondremos de ivectors.

Dentro del directorio *graph* encontramos el fichero *words.txt* que contiene un listado de símbolos, palabras con su equivalente numérico que se utilizan para pasar de la forma textual a la numérica y viceversa. Dentro de este directorio, se encuentra otro llamado *phones* que contiene el archivo *word_boundary.int*, este define cómo se relacionan los fonemas y las posiciones de las palabras, esto es necesario para definir los límites de las palabras. Los símbolos de desambiguación se encuentran en *disambig_tid.int*. Y por último, los dos más importantes *Gr.fst* y *HCLr.fst*, en este caso se utilizan estos dos grafos en vez de un HCLG porque se recalcula el grafo, a continuación explico lo que es un grafo HCLG.

Grafos HCLG

Un grafo HCLG no es más que un autómata de estado finito que ofrece la probabilidad de sus secuencias de estados basadas en el modelo de lenguaje y el léxico, para construir dicho grafo se tienen en cuenta las secuencias de fonemas [15]. HCLG se compone de:

- H son las definiciones de los HMM, sus símbolos de salida son fonemas dependientes del contexto y sus símbolos de entrada son identificadores de funciones de distribución de probabilidad.
- C es la dependencia del contexto, sus símbolos de entrada son fonemas, y sus símbolos de salida son fonemas dependientes del contexto.
- L es el léxico, sus símbolos de salida son palabras y sus símbolos de entrada son fonemas.
- G es el codificador de la gramática, sus símbolos de entrada y salida son los mismos.

La salida de este autómata, formado por la composición de los cuatro, debe estar determinada y minimizada, para que se cumpla la primera característica deben introducirse símbolos de desambiguación. Estos símbolos se indican en el léxico, hay secuencias de fonemas que se repiten, aparecen en más de una palabra como parte de esta, para diferenciar estas secuencias se necesita añadir un símbolo de desambiguación al final de cada una. Posteriormente, debe minimizarse el autómata reduciendo sus posibles estados de transición.

4.5.3. Implementación

A continuación, adjunto un modelo de dominio a nivel de implementación directamente, saltando las fases conceptual y de especificación.

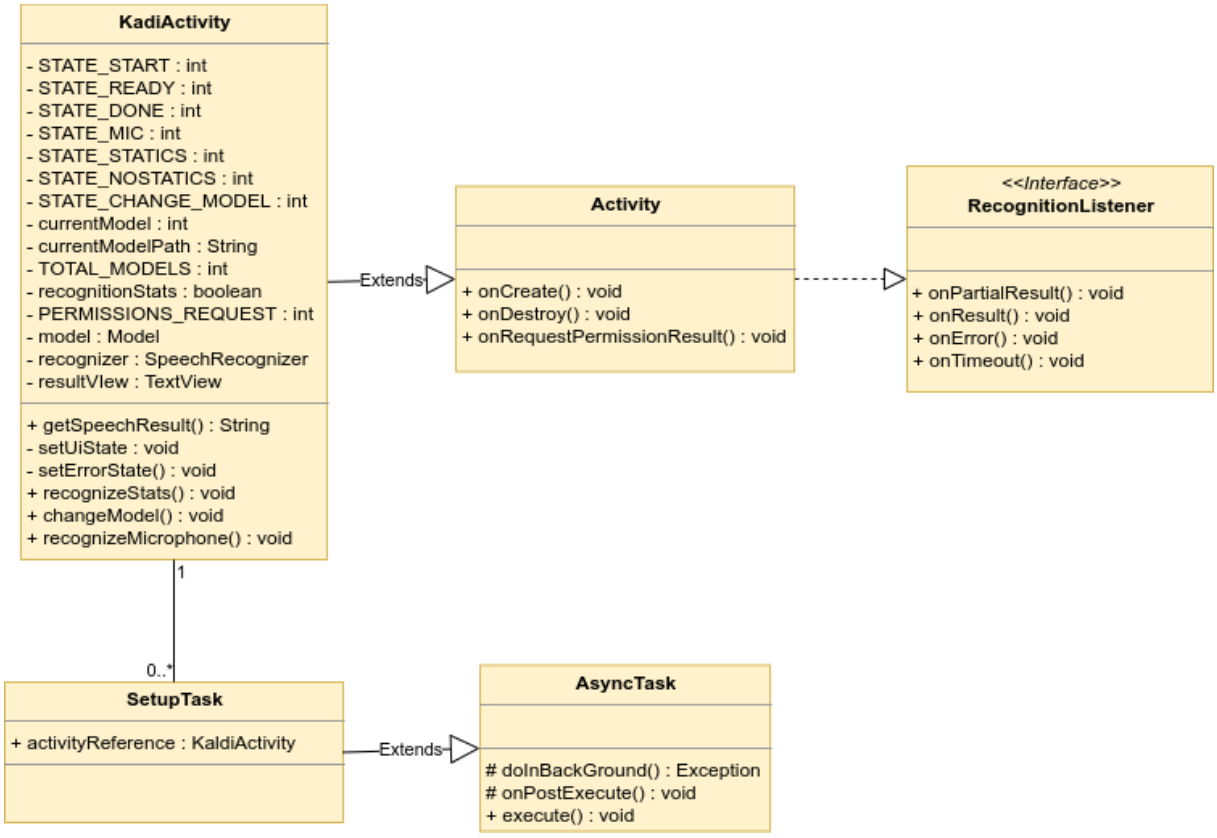

Figura 4.10: Modelo de dominio a nivel de implementación.

La información concreta de las clases y métodos de la actividad utilizada se encuentra en el repositorio de GitHub[1]en formato Javadoc, aquí comentaré los métodos y aspectos más importantes.

La aplicación se basa en la API de VOSK, concretamente en su interfaz *RecognitionListener*. Como estoy construyendo un reconocedor automático del habla, lo primero debe ser garantizar que la aplicación tiene permisos para grabar audio, si no la aplicación no tendrá acceso al micrófono y no podrá conseguir los datos de entrada.

La inicialización del modelo de lenguaje se realiza de forma asíncrona ya que lleva cierto tiempo, así la aplicación puede arrancar a la vez que se realiza dicha tarea, al ser una tarea en segundo plano cuando se realiza un cambio de modelo de lenguaje habrá que lanzarla de nuevo para que se cargué el nuevo modelo de lenguaje, es importante también lanzar esta tarea de crear y cargar el modelo de lenguaje si la aplicación no tenía permisos para acceder al micrófono y hemos tenido que concedérselos, si no, no se lanzará y no podremos utilizar la aplicación.

Con el método *setUiState* controlo el estado y funcionamiento de la interfaz de usuario, dependiendo del código que reciba el método llevará a cabo los cambios en la interfaz que sean necesarios para la acción realizada.

Un método *recognizeMicrophone* al que se llame cuando el botón con el símbolo del micrófono sea pulsado, se encargará de crear el reconocedor de voz y de destruirlo una vez finalizado el reconocimiento y pulsado de nuevo el botón. Por último, un método que en base al resultado final, obtenido por este reconocedor, pase la transcripción a un formato legible.

[1]Repositorio GitHub del proyecto android-vosk-demo [16]

4.5.4. La interfaz de usuario

Funcionamiento

Los cambios en la interfaz se realizan mediante un único método llamado *setUiState* que tiene diferentes estados. Estos estados están definidos desde un inicio y son un total de siete, los diferentes métodos de la aplicación necesitan realizar cambios sobre la interfaz para limitar las acciones que el usuario puede realizar o bien para mostrar información al usuario, ya sea la transcripción de voz que han solicitado o el cambio de modelo de lenguaje que han realizado. Para ello, cada método que quiera realizar un cambio deberá llamar a esta especie de controlador indicando al estado que quiere que cambie la interfaz. Entonces, el controlador se encargará de modificar la interfaz, tal y como tiene establecido para el estado indicado.

Diseño

La aplicación ya tenía un diseño preestablecido, lo que he hecho ha sido modificar su estructura básica para darle un aspecto más estético y actual, siempre manteniendo la línea de minimalismo y sencillez para no abrumar al usuario y así facilitar el uso de la aplicación. Como se trata de una aplicación Android, he diseñado un icono para que esta sea fácilmente reconocible en nuestro cajón de aplicaciones, este puede verse en la figura 4.11

Figura 4.11: Icono de la aplicación final TFG VOSK demo.

El diseño de la interfaz de la aplicación puede verse en la figura 4.12, he añadido a la captura unas etiquetas visuales para poder identificar mejor los elementos de la interfaz, que son los siguientes:

1. Este botón permite alternar entre el modo de visualización de datos ampliada y el normal, en el normal tan sólo aparecerá la transcripción, y en el modo de visualización de datos ampliada aparecerá la confianza con la que el reconocedor decide cuál es la palabra que ha escuchado, los instantes de tiempo en los que comienza y termina la palabra desde que se inició el reconocimiento, y la propia palabra reconocida, esto para cada una de las palabras dichas, y finalmente la frase completa.

2. Al pulsar este botón la aplicación alterna entre los diferentes modelos de lenguaje que posee, siempre informando al usuario del modelo de lenguaje seleccionado.

3. Este botón con el símbolo de un micrófono sirve para activar y desactivar el reconocedor de voz, cuando el reconocedor está activo, el símbolo pasa a tener un color rojo para que el usuario tenga una respuesta visual de la acción que ha realizado.

4. Por último, la superficie dónde se muestra toda la información que emite la aplicación, desde las transcripciones hasta los avisos de esta.

Figura 4.12: Captura de pantalla de la aplicación TFG VOSK para mostrar su interfaz.

4.5.5. Creando un modelo de lenguaje

Ahora que la aplicación ya funciona con dos modelos de lenguaje diferentes, voy a proceder a introducir un modelo de lenguaje creado por mí, para ello utilizaré el corpus TIMIT, del cual el grupo de investigación ECA-SIMM posee una copia. Para generar los archivos necesarios del modelo de lenguaje haré uso de Docker y la imagen de Kaldi para GPUs que descargué, utilizaré la receta de Kaldi que propone VOSK en su web [14] para el entrenamiento con redes neuronales y la extracción de ivectors.

Conjunto de datos TIMIT

Como ya he dicho, utilizo el corpus TIMIT para generar el modelo de lenguaje, este corpus fue desarrollado conjuntamente por el Stanford Research Institute (SRI), la empresa Texas Instruments (TI), dónde se grabaron los archivos de audio, y el Massachusetts Institute of Technology (MIT), dónde fueron transcritos dichos archivos, es por ello que la concatenación de las siglas de estas dos últimas entidades da nombre al corpus.

Este conjunto de datos está formado por seis mil trescientas frases transcritas fonética y léxicamente, pertenecen a seiscientos treinta hablantes de inglés americano originarios de ocho regiones dialécticas diferentes con la distribución que se muestra en la figura 4.5. De

cada locutor se tienen diez frases que fueron grabadas por teléfono.

Dialec Region	Male	Female	Total
New England	31 (63 %)	18 (27 %)	49 (8 %)
Northern	71 (70 %)	31 (30 %)	102 (16 %)
North Midland	79 (67 %)	23 (23 %)	102 (16 %)
South Midland	69 (69 %)	31 (31 %)	100 (16 %)
Southern	62 (63 %)	36 (37 %)	98 (16 %)
New York City	30 (65 %)	16 (35 %)	46 (7 %)
Western	74 (74 %)	26 (26 %)	100 (16 %)
Army Brat (moed around)	22 (67 %)	11 (33 %)	33 (5 %)
Total	438 (70 %)	192 (30 %)	630 (100 %)

Tabla 4.5: Distribución de hablantes del conjunto TIMIT

Entrenamiento del modelo de lenguaje

Explicaré paso a paso la receta utilizada para entrenar el modelo de lenguaje y adjuntaré el código de dicha receta para que resulte más ilustrativo, ya que no puedo facilitar todos los archivos y directorios que genera la receta de Kaldi puesto que algunos archivos son modificaciones del corpus y este tiene una licencia que lo protege.

Antes de empezar, como estoy utilizando un contenedor con la imagen de Kaldi comparto el directorio que contiene el corpus de TIMIT así como mi carpeta personal para guardar en ella los archivos, al ejecutar la instancia de la imagen es importante pasarle la variable de entorno USER con nuestro usuario para que todo funcione correctamente.

Script run.sh

```
. ./cmd.sh
[ -f path.sh ] && . ./path.sh
set -e

numLeavesTri1=2500
numGaussTri1=15000
numLeavesMLLT=2500
numGaussMLLT=15000
numLeavesSAT=2500
numGaussSAT=15000
numGaussUBM=400
numLeavesSGMM=7000
numGaussSGMM=9000

feats_nj=10
train_nj=30
decode_nj=5
```

Todo el código que se ve de ahora en adelante ha sido extraído del script *run.sh* del ejemplo TIMIT de Kaldi. Para comenzar, se ejecutan un par de scripts que determinan la ejecución del código de forma exclusiva o no exclusiva y las rutas que va a utilizar, además, se inicializan los parámetros del modelo acústico y el número de trabajos en paralelo que se realizaran en las distintas etapas.

```
timit=/home/corpus/timit/timit
```

```
local/timit_data_prep.sh $timit || exit 1

local/timit_prepare_dict.sh

utils/prepare_lang.sh --sil-prob 0.0 --position-dependent-phones false --
    num-sil-states 3 data/local/dict "sil" data/local/lang_tmp data/lang

local/timit_format_data.sh
```

Kaldi necesita una serie de ficheros y directorios que los contengan para funcionar, estos scripts se encargan de preparar los datos para poder ser utilizados, el primero de estos genera ficheros como los que se encuentran en el directorio *data/train* y se encargan por ejemplo de asignar a cada enunciado su locutor y viceversa, asignar a cada grabación su transcripción y también de recopilar las rutas de los archivos de audio tipo wav.

El segundo script se encarga de preparar el diccionario, crea el léxico con los fonemas y construye el modelo de lenguaje con fonemas de dos letras o bi-grama gracias al la herramienta IRSTLM. El último script genera los transductores de estado finito del léxico con y sin símbolos de desambiguación, define la topología de los modelos ocultos de Markov y en general, trata los fonemas.

```
mfccdir=mfcc

for x in train dev test; do
  steps/make_mfcc.sh --cmd "$train_cmd" --nj $feats_nj data/$x exp/
      make_mfcc/$x $mfccdir
  steps/compute_cmvn_stats.sh data/$x exp/make_mfcc/$x $mfccdir
done
```

Calcula para los datos de entrenamiento, test y validación los MFCC y la normalización de la media y la varianza cepstral (CMVN), estas características del habla son almacenadas en el directorio mfcc.

```
steps/train_mono.sh  --nj "$train_nj" --cmd "$train_cmd" data/train data/
    lang exp/mono

utils/mkgraph.sh data/lang_test_bg exp/mono exp/mono/graph

steps/decode.sh --nj "$decode_nj" --cmd "$decode_cmd" exp/mono/graph data/
    dev exp/mono/decode_dev

steps/decode.sh --nj "$decode_nj" --cmd "$decode_cmd" exp/mono/graph data/
    test exp/mono/decode_test
```

Realiza un entrenamiento monofónico con el modelo de lenguaje que se acaba de entrenar y, a continuación, genera el grafo de decodificación expandido, un HCLG que representa todo el modelo de lenguaje, el léxico, la dependencia de contexto y la estructura de los modelos ocultos de Markov. El script *decode.sh* sirve para obtener la secuencia de fonemas que el reconocedor asocia a una muestra de voz, esa secuencia está alineada con la señal de voz y proporciona un etiquetado de la señal usando el modelo. Esta función se realiza de forma iterativa, es decir, estas secuencias se utilizarán para entrenar el siguiente modelo.

```
steps/align_si.sh --boost-silence 1.25 --nj "$train_nj" --cmd "$train_cmd"
    data/train data/lang exp/mono exp/mono_ali

steps/train_deltas.sh --cmd "$train_cmd" $numLeavesTri1 $numGaussTri1 data/
    train data/lang exp/mono_ali exp/tri1

utils/mkgraph.sh data/lang_test_bg exp/tri1 exp/tri1/graph
```

```
steps/decode.sh --nj "$decode_nj" --cmd "$decode_cmd" exp/tri1/graph data/
    dev exp/tri1/decode_dev

steps/decode.sh --nj "$decode_nj" --cmd "$decode_cmd" exp/tri1/graph data/
    test exp/tri1/decode_test
```

Primero, se calculan las alineaciones de entrenamiento utilizando el modelo monofónico que acaba de entrenar, a continuación, hace un entrenamiento trifónico deltas + delta-deltas, que tiene en cuenta las características delta y delta-delta como su nombre indica, estas son la velocidad y la aceleración, se vuelve a generar el grafo completo y se realiza la decodificación.

```
steps/align_si.sh --nj "$train_nj" --cmd "$train_cmd"  data/train data/lang
     exp/tri1 exp/tri1_ali

steps/train_lda_mllt.sh --cmd "$train_cmd" --splice-opts "--left-context=3
    --right-context=3" $numLeavesMLLT $numGaussMLLT data/train data/lang exp
    /tri1_ali exp/tri2

utils/mkgraph.sh data/lang_test_bg exp/tri2 exp/tri2/graph

steps/decode.sh --nj "$decode_nj" --cmd "$decode_cmd" exp/tri2/graph data/
    dev exp/tri2/decode_dev

steps/decode.sh --nj "$decode_nj" --cmd "$decode_cmd" exp/tri2/graph data/
    test exp/tri2/decode_test
```

Como ya he dicho, esto es un sistema de entrenamiento iterativo, por lo tanto, se basa en repetir el mismo procedimiento pero con el nuevo modelo trifónico que se ha entrenado. Vuelven a calcularse las alineaciones de entrenamiento, esta vez con el modelo anterior y después se realiza un entrenamiento LDA + MLLT con su posterior decodificación para conseguir un modelo trifónico HMM-GMM. Con entrenamiento LDA (Linear Discriminant Analysis) y MLLT (Maximum Likelihood Linear Transforms) se refiere a la forma en que transforma las características para conseguir mayor independencia.

```
steps/align_si.sh --nj "$train_nj" --cmd "$train_cmd" --use-graphs true
    data/train data/lang exp/tri2 exp/tri2_ali

steps/train_sat.sh --cmd "$train_cmd" $numLeavesSAT $numGaussSAT data/train
     data/lang exp/tri2_ali exp/tri3

utils/mkgraph.sh data/lang_test_bg exp/tri3 exp/tri3/graph

steps/decode_fmllr.sh --nj "$decode_nj" --cmd "$decode_cmd" exp/tri3/graph
    data/dev exp/tri3/decode_dev

steps/decode_fmllr.sh --nj "$decode_nj" --cmd "$decode_cmd" exp/tri3/graph
    data/test exp/tri3/decode_test

steps/align_fmllr.sh --nj "$train_nj" --cmd "$train_cmd" data/train data/
    lang exp/tri3 exp/tri3_ali
```

Para terminar con el script *run.sh*, ya que la parte de redes neuronales se realizará con el script que sugiere la página de VOSK, tenemos otra vez el mismo proceso pero esta vez al entrenamiento se le suma el SAT (Speaker Adapted Training), que se utiliza para discriminar las variantes fonéticas del locutor. Es decir, realiza un entrenamiento LDA + MLLT + SAT, y como anteriormente, después genera el grafo y realiza la decodificación.

Script run_tdnn_1j.sh

Ahora toca el entrenamiento con redes neuronales profundas, este se realiza a través del script *run_tdnn_1j.sh*, concretamente entrenará un modelo de cadena, que es un tipo de modelo que mezcla redes neuronales profundas y modelos ocultos de Markov.

Antes de ejecutarlo dentro del directorio local debemos añadir un enlace al directorio *nnet3* del ejemplo Wall Street Journal si este no existe ya con sus respectivos scripts, además deberemos añadir un par de archivos de configuración en la carpeta *conf*, estos son *mfcc_hires.conf* y *online_cmvn.conf* que sirven para indicar la configuración del entrenamiento de la red neuronal y la aplicación de la CMVN, tras estas modificaciones el script se lanza con éxito y realiza las tareas que comento a continuación.

```
set -euo pipefail

stage=0
decode_nj=10
train_set=train
test_sets=dev_clean_2
gmm=tri3
nnet3_affix=

affix=1j
tree_affix=
train_stage=-10
get_egs_stage=-10
decode_iter=

chunk_width=140,100,160
common_egs_dir=
xent_regularize=0.1

srand=0
remove_egs=true
reporting_email=

gmm_dir=exp/$gmm
ali_dir=exp/${gmm}_ali_${train_set}_sp
tree_dir=exp/chain${nnet3_affix}/tree_sp${tree_affix:+_$tree_affix}
lang=data/lang_chain
lat_dir=exp/chain${nnet3_affix}/${gmm}_${train_set}_sp_lats
dir=exp/chain${nnet3_affix}/tdnn${affix}_sp
train_data_dir=data/${train_set}_sp_hires
lores_train_data_dir=data/${train_set}_sp
train_ivector_dir=exp/nnet3${nnet3_affix}/ivectors_${train_set}_sp_hires

test_online_decoding=true

. ./cmd.sh
. ./path.sh
. ./utils/parse_options.sh

if ! cuda-compiled; then
  cat <<EOF && exit 1
This script is intended to be used with GPUs but you have not compiled
   Kaldi with CUDA
If you want to use GPUs (and have them), go to src/, and configure and make
    on a machine
where "nvcc" is installed.
EOF
fi
```

Nada más comenzar especifica los parámetros y rutas que serán utilizados por este y otros scripts para el entrenamiento, también comprueba si Kaldi está compilado con CUDA, ya que necesitará una GPU para realizar el entrenamiento de la red neuronal.

```
local/nnet3/run_ivector_common.sh --stage $stage \
                                  --train-set $train_set \
                                  --gmm $gmm \
                                  --nnet3-affix "$nnet3_affix" || exit 1;
```

Dentro de este script se ejecuta el *run_ivector_common.sh* que realiza la extracción de ivectors, es una parte importante de la receta, por ello de aquí en adelante explicaré los pasos que sigue dicho script.

Script run_ivector_common.sh

```
set -e -o pipefail

stage=0
nj=30
train_set=train_si284
test_sets="test_dev93 test_eval92"
gmm=tri4b

num_threads_ubm=32
nnet3_affix=

. ./cmd.sh
. ./path.sh
. utils/parse_options.sh

gmm_dir=exp/${gmm}
ali_dir=exp/${gmm}_ali_${train_set}_sp

for f in data/${train_set}/feats.scp ${gmm_dir}/final.mdl; do
  if [ ! -f $f ]; then
    echo "$0: expected file $f to exist"
    exit 1
  fi
done
```

Como es costumbre, al inicio de su ejecución inicializa parámetros que necesitará para su ejecución y determina las rutas que utilizará. Además, comprueba que los directorios que va a utilizar contengan los archivos *feats.scp* y *final.mdl* que requiere.

```
utils/data/perturb_data_dir_speed_3way.sh data/${train_set} data/${
    train_set}_sp
```

A continuación, acondiciona los datos para conseguir unos de "alta resolución", para ello utiliza el script *perturb_data_dir_speed_3way.sh* que genera a partir de los datos originales una copia de estos con modificaciones en la velocidad con factor 0.9 y 1.1, junto con los originales contamos con tres muestras de datos que se guardan en *datatrain_clean_5_sp*. Para realizar esta perturbación en la velocidad se necesitan los archivos *utt2dur* y *reco2dur*. Estos ficheros generan el mapeo de duración en segundos de las voces y grabaciones.

```
mfccdir=data/${train_set}_sp_hires/data
  if [[ $(hostname -f) == *.clsp.jhu.edu ]] && [ ! -d $mfccdir/storage ];
     then
```

```
    utils/create_split_dir.pl /export/b0{5,6,7,8}/$USER/kaldi-data/mfcc/wsj
      -$(date +'%m_%d_%H_%M')/s5/$mfccdir/storage $mfccdir/storage
  fi

  for datadir in ${train_set}_sp ${test_sets}; do
    utils/copy_data_dir.sh data/$datadir data/${datadir}_hires
  done

  utils/data/perturb_data_dir_volume.sh data/${train_set}_sp_hires

  for datadir in ${train_set}_sp ${test_sets}; do
    steps/make_mfcc.sh --nj $nj --mfcc-config conf/mfcc_hires.conf \
      --cmd "$train_cmd" data/${datadir}_hires
    steps/compute_cmvn_stats.sh data/${datadir}_hires
    utils/fix_data_dir.sh data/${datadir}_hires
  done
```

Prepara los directorios, así como la información que necesita en ellos. Para extraer características de alta resolución y los ivector comienza aumentando la cantidad de datos de la que disponemos, para ello realiza una perturbación en los datos de entrenamiento antes utilizados, reduciendo y aumentando su volumen consiguiendo así datos con calidad variada. Sobre estos nuevos datos calcula los MFCC y la CMVN y continua extrayendo las características de alta resolución de los MFCC, esto consigue que las redes neuronales una vez entrenadas sean menos variantes con el conjunto de prueba. Al finalizar, se comprueba que estén presentes todos los segmentos necesarios.

```
mkdir -p exp/nnet3${nnet3_affix}/diag_ubm
temp_data_root=exp/nnet3${nnet3_affix}/diag_ubm
num_utts_total=$(wc -l <data/${train_set}_sp_hires/utt2spk)
num_utts=$[$num_utts_total/4]
utils/data/subset_data_dir.sh data/${train_set}_sp_hires $num_utts ${
    temp_data_root}/${train_set}_sp_hires_subset

steps/online/nnet2/get_pca_transform.sh --cmd "$train_cmd" --splice-opts "
    --left-context=3 --right-context=3" --max-utts 10000 --subsample 2 ${
    temp_data_root}/${train_set}_sp_hires_subset exp/nnet3${nnet3_affix}/
    pca_transform

steps/online/nnet2/train_diag_ubm.sh --cmd "$train_cmd" --nj 30 --num-
    frames 700000 --num-threads $num_threads_ubm      ${temp_data_root}/${
    train_set}_sp_hires_subset 512 exp/nnet3${nnet3_affix}/pca_transform exp
    /nnet3${nnet3_affix}/diag_ubm
```

Se crea un subconjunto de entrenamiento con aproximadamente una cuarta parte de los datos para entrenar el UBM (Universal Backgroud Model), este modelo se utiliza para extraer características generales del habla. Este modelo será usado para la estimación de los ivector, y esta estimación será usada como entrada en la red neuronal profunda de un solo paso. También calcula un análisis de los componentes principales (PCA), para realizar una transformación lineal ortogonal de los datos de alta resolución, reduciendo así sus dimensiones y aumentando el rendimiento del reconocedor. Y finalmente, entrena el UBM con el subconjunto generado.

```
steps/online/nnet2/train_ivector_extractor.sh --cmd "$train_cmd" --nj 10
    data/${train_set}_sp_hires exp/nnet3${nnet3_affix}/diag_ubm exp/nnet3${
    nnet3_affix}/extractor || exit 1;
```

Entrena el extractor de ivectors con los datos de alta resolución cuya velocidad ha sido modificada.

```
ivectordir=exp/nnet3${nnet3_affix}/ivectors_${train_set}_sp_hires
  if [[ $(hostname -f) == *.clsp.jhu.edu ]] && [ ! -d $ivectordir/storage
      ]; then
    utils/create_split_dir.pl /export/b0{5,6,7,8}/$USER/kaldi-data/ivectors
       /wsj-$(date +'%m_%d_%H_%M')/s5/$ivectordir/storage $ivectordir/
       storage
  fi
emp_data_root=${ivectordir}
utils/data/modify_speaker_info.sh --utts-per-spk-max 2 data/${train_set}
    _sp_hires ${temp_data_root}/${train_set}_sp_hires_max2

  steps/online/nnet2/extract_ivectors_online.sh --cmd "$train_cmd" --nj $nj
      ${temp_data_root}/${train_set}_sp_hires_max2 exp/nnet3${nnet3_affix}/
     extractor $ivectordir
for data in ${test_sets}; do
    nspk=$(wc -l <data/${data}_hires/spk2utt)
    steps/online/nnet2/extract_ivectors_online.sh --cmd "$train_cmd" --nj "
       ${nspk}" data/${data}_hires exp/nnet3${nnet3_affix}/extractor exp/
       nnet3${nnet3_affix}/ivectors_${data}_hires
  done
```

Para extraer los ivectors primero se agrupan las voces en grupos de dos y tres mezclando hablantes, de esta forma se le da más diversidad a los ivectors ya que generalizan mejor. Se extraen los ivector sobre el conjunto de entrenamiento y también sobre el conjunto de prueba, este sin modificaciones de velocidad.

```
steps/make_mfcc.sh --nj $nj --cmd "$train_cmd" data/${train_set}_sp
steps/compute_cmvn_stats.sh data/${train_set}_sp
utils/fix_data_dir.sh data/${train_set}_sp

steps/align_fmllr.sh --nj $nj --cmd "$train_cmd" data/${train_set}_sp data/
    lang $gmm_dir $ali_dir
```

Para realizar la alineación de datos primero deben sacarse los MFCC y la CMVN de los datos con velocidad modificada, tras estos cálculos comprueba que todos los segmentos necesarios estén presentes. Alinea los datos modificados gracias al archivo *final.mdl*, la alineación se realiza entre el audio y el texto para saber cuándo empieza y cuándo acaba una palabra y así poder determinar de cuál se trata.

Después de extraer los ivectors volvemos al script *run_tdnn_1j.sh*.

De vuelta al script run_tdnn_1j.sh

```
cp -r data/lang $lang
silphonelist=$(cat $lang/phones/silence.csl) || exit 1;
nonsilphonelist=$(cat $lang/phones/nonsilence.csl) || exit 1;

steps/nnet3/chain/gen_topo.py $nonsilphonelist $silphonelist >$lang/topo
```

Crea un directorio lang con una topología de tipo cadena y genera el archivo de topología que permite controlar el número de estados en los modelos ocultos de Markov.

```
steps/align_fmllr_lats.sh --nj 75 --cmd "$train_cmd" ${lores_train_data_dir
    } data/lang $gmm_dir $lat_dir
rm $lat_dir/fsts.*.gz

steps/nnet3/chain/build_tree.sh --frame-subsampling-factor 3 --context-opts
     "--context-width=2 --central-position=1" --cmd "$train_cmd" 3500 ${
    lores_train_data_dir} $lang $ali_dir $tree_dir
```

Genera representaciones de secuencias de palabras alternativas que son suficientemente parecidas para una pronunciación, esto lo hace a partir de los MFCC de baja resolución y le dan el nombre de entramado. Con esta nueva topología construye un árbol.

```
mkdir -p $dir

num_targets=$(tree-info $tree_dir/tree |grep num-pdfs|awk '{print $2}')
learning_rate_factor=$(echo "print (0.5/$xent_regularize)" | python)

tdnn_opts="l2-regularize=0.03"
tdnnf_opts="l2-regularize=0.03 bypass-scale=0.66"
linear_opts="l2-regularize=0.03 orthonormal-constraint=-1.0"
prefinal_opts="l2-regularize=0.03"
output_opts="l2-regularize=0.015"

mkdir -p $dir/configs
cat <<EOF > $dir/configs/network.xconfig

steps/nnet3/xconfig_to_configs.py --xconfig-file $dir/configs/network.
    xconfig --config-dir $dir/configs/

if [[ $(hostname -f) == *.clsp.jhu.edu ]] && [ ! -d $dir/egs/storage ];
    then
     utils/create_split_dir.pl /export/b0{3,4,5,6}/$USER/kaldi-data/egs/
         mini_librispeech-$(date +'%m_%d_%H_%M')/s5/$dir/egs/storage $dir/egs
         /storage
  fi
```

Prepara el archivo de configuración para la estructura de la red neuronal.

```
steps/nnet3/chain/train.py --stage=$train_stage \
    --cmd="run.pl --mem 6G" \
    --feat.online-ivector-dir=$train_ivector_dir \
    --feat.cmvn-opts="--norm-means=false --norm-vars=false" \
    --chain.xent-regularize $xent_regularize \
    --chain.leaky-hmm-coefficient=0.1 \
    --chain.l2-regularize=0.0 \
    --chain.apply-deriv-weights=false \
    --chain.lm-opts="--num-extra-lm-states=2000" \
    --trainer.add-option="--optimization.memory-compression-level=2" \
    --trainer.srand=$srand \
    --trainer.max-param-change=2.0 \
    --trainer.num-epochs=20 \
    --trainer.frames-per-iter=3000000 \
    --trainer.optimization.num-jobs-initial=2 \
    --trainer.optimization.num-jobs-final=5 \
    --trainer.optimization.initial-effective-lrate=0.002 \
    --trainer.optimization.final-effective-lrate=0.0002 \
    --trainer.num-chunk-per-minibatch=128,64 \
    --egs.chunk-width=$chunk_width \
    --egs.dir="$common_egs_dir" \
    --egs.opts="--frames-overlap-per-eg 0" \
    --cleanup.remove-egs=$remove_egs \
    --use-gpu=wait \
    --reporting.email="$reporting_email" \
    --feat-dir=$train_data_dir \
    --tree-dir=$tree_dir \
    --lat-dir=$lat_dir \
    --dir=$dir  || exit 1;

utils/mkgraph.sh --self-loop-scale 1.0 data/lang_test_bg $tree_dir
```

```
    $tree_dir/graph_bg || exit 1;
```

Se entrena la red neuronal con los MFCC de alta resolución, el árbol de decisión y los ivectors extraídos anteriormente. Esta parte del entrenamiento tuvo que ser ejecutada forzando el modo exclusivo de la GPU ya que sino surgían problemas con la memoria compartida de esta.

```
utils/mkgraph.sh --self-loop-scale 1.0 data/lang_test_bg $tree_dir
    $tree_dir/graph_bg || exit 1;
rames_per_chunk=$(echo $chunk_width | cut -d, -f1)
  rm $dir/.error 2>/dev/null || true
for data in $test_sets; do
    (
    nspk=$(wc -l <data/dev/spk2utt)
    steps/nnet3/decode.sh --acwt 1.0 --post-decode-acwt 10.0 --frames-per-
        chunk $frames_per_chunk --nj $nspk --cmd "$decode_cmd"  --num-
        threads 4 --online-ivector-dir exp/nnet3${nnet3_affix}/
        ivectors_test_hires $tree_dir/graph_bg data/dev ${dir} || exit 1
    steps/lmrescore_const_arpa.sh --cmd "$decode_cmd" data/lang_test_{
        tgsmall,tglarge} data/${data}_hires ${dir}/decode_{tgsmall,tglarge}
        _${data} || exit 1
    ) || touch $dir/.error &
  done
```

Por último se realiza la decodificación como ya ha pasado anteriormente, pero esta vez utilizando una red neuronal y se vuelven a evaluar los entramados, esta vez con el modelo de lenguaje ConstArpa.

Introducción del modelo de lenguaje propio

Tras el entrenamiento, ya tenemos todos los archivos necesarios para incluirlos en el directorio de la aplicación Android y utilizar este nuevo modelo de lenguaje. En el directorio */exp/nnet3/extractor* se encuentran todos los archivos referentes a la extracción de ivectors que necesitamos. En */exp/chain/tree_sp/graph_bg* encontramos todos los archivos referentes a grafos. Por último, el archivo *final.mdl* se encuentra en */exp/chain/tdnn1j_sp* y los archivos de configuración deberemos generarlos nosotros, en este caso con los valores por defecto para que el reconocedor funcione, este es el contenido del archivo *mfcc.conf*:

```
--sample-frequency=16000
--use-energy=true
--num-mel-bins=40
--num-ceps=40
--low-freq=40
--high-freq=7600
--allow-upsample=true
--allow-downsample=true
```

y este el del archivo model.conf:

```
--min-active=200
--max-active=3000
--beam=10.0
--lattice-beam=2.0
--acoustic-scale=1.0
--frame-subsampling-factor=3
```

```
--endpoint.silence-phones=1:2:3:4:5:6:7:8:9:10
--endpoint.rule2.min-trailing-silence=0.5
--endpoint.rule3.min-trailing-silence=1.0
--endpoint.rule4.min-trailing-silence=2.0
```

4.5.6. Pruebas

Dado que el número de requisitos no es muy grande y se requiere una interacción con la aplicación que sería más costosa de automatizar que de realizar directamente, las pruebas serán realizadas por mí de forma manual. Estas pruebas tratarán de garantizar que se cumplen todos los requisitos y pese a ser pocos se itentará cubrir el máximo número de requisitos por test.

Test 1

Con este test doy cobertura a los requisitos funcionales 1, 2, 9, 10 y 11. La prueba consiste en activar el reconocedor de voz, confirmar que todos los botones se deshabilitan exceptuando el de desactivación del reconocedor, dictar pausadamente los números del uno al quince y detener el reconocedor de voz, de esta forma poder comprobar que se activa y desactiva correctamente, y que muestra transcripciones indefinidamente hasta que este se desactive, habilitando una barra de desplazamiento lateral que permita consultar el total de lo transcrito tras haber desactivado el reconocedor.

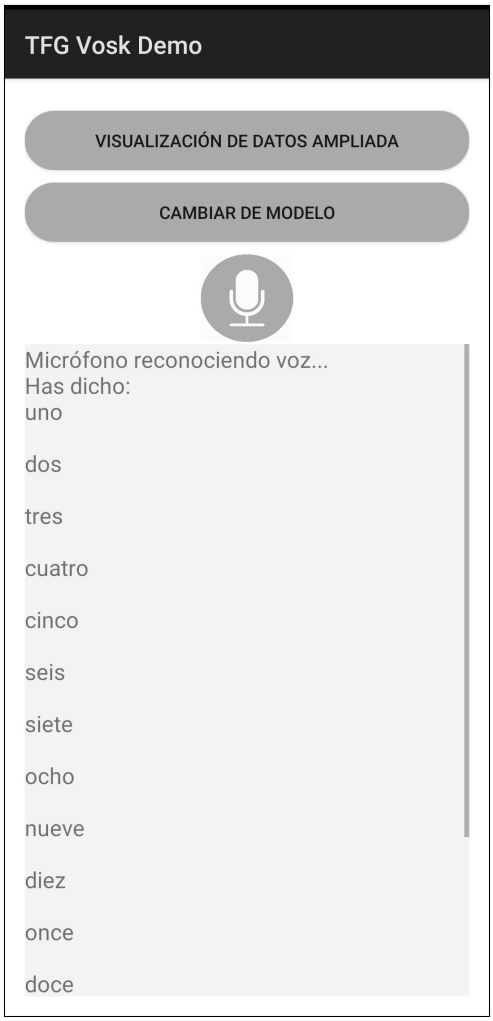

Figura 4.13: Captura de pantalla de la aplicación TFG VOSK demo para el test 1.

Test 2

Con este test doy cobertura a los requisitos funcionales 3, 4 y 12. La prueba consiste en hacer uso del botón destinado a cambiar de modelo de lenguaje, observar cómo se bloqueen los botones mientras se prepara el reconocedor y esperar la respuesta de la aplicación para comprobar que se ha cambiado con éxito.

Figura 4.14: Captura de pantalla de la aplicación TFG VOSK demo para el test 2.

Test 3

Con este test doy cobertura a los requisitos funcionales 5, 6, 7 y 8. La prueba consiste en activar y desactivar la visualización de datos extendida y probar que esta muestra los datos correctamente.

Figura 4.15: Captura de pantalla de la aplicación TFG VOSK demo para el test 3.

Test 4

Con este test doy cobertura al requisito funcional 13. La prueba consiste en probar los tres modelos de lenguaje de los que dispone la aplicación, para ello se dictará una serie de palabras con cada uno de los test y se comprobará que coincida con la transcripción.

Figura 4.16: Captura de pantalla de la aplicación TFG VOSK demo para el test 4.

Resultados

Todos los test fueron superados con éxito a excepción del último, en cada uno de los test he ido incorporando imágenes que muestran de alguna forma cómo se realizaba. El cuarto test no se completó con éxito, debido a que el modelo de lenguaje que he creado e introducido en la aplicación no funciona, como puede verse en la figura 4.16 este sólo devuelve "sil", que representa el silencio, por tanto, no está identificando nada, lo más parecido es el silencio y es lo que nos devuelve.

Al lanzar la aplicación y escoger el modelo de lenguaje creado por mí, la aplicación crea el reconocedor con éxito, por tanto, el modelo de lenguaje es válido. El problema se encuentra en el propio modelo de lenguaje entonces, este no funciona correctamente pese a tener todo lo necesario para funcionar. El problema puede provenir del léxico del modelo de lenguaje, se genera un léxico fonológico, de hecho se trabaja siempre a nivel de fonema, esto es muy problemático, partiendo de un léxico formado por palabras puede descomponer este en fonemas y el modelo acústico puede comparar estos con las secuencias fonéticas del léxico, partiendo de uno fonológico no.

Capítulo 5

Conclusiones

En este proyecto se han tratado las bases y principios del reconocimiento automático del habla, me ha ayudado a adquirir conocimientos sobre este campo de la inteligencia artificial y a iniciarme en él, así como en el desarrollo de aplicaciones Android. A su vez, para el desarrollo y correspondiente documentación de esta aplicación, he puesto a prueba los conocimientos adquiridos durante el grado.

Pese a que no se han cumplido todos los objetivos la funcionalidad de la aplicación es completa. En la sección 4.5.6 comentaba los problemas que surgieron, el propio creador de la herramienta Kaldi, Dan Povey, [1]desaconsejaba rotundamente el uso del corpus de voz que he utilizado.

Referente a las herramientas utilizadas, la API de VOSK es tremendamente útil y sencilla de utilizar, está bien documentada y, pese a no contar con una comunidad tan grande como otras, se encuentra solución a los problemas que puedan surgir sin realizar una gran inversión de tiempo. No puedo decir lo mismo de Kaldi, pese a ser una herramienta muy potente, la curva de aprendizaje que hay que afrontar en un inicio es muy grande, no puede exprimirse todo su potencial si no se tiene una amplia experiencia en el campo del reconocimiento automático del habla, y pese a tenerlo es laborioso.

Por otra parte nunca había utilizado Docker y me ha parecido a parte de cómodo, útil y sencillo. Tanto el despliegue de los contenedores como la adquisición de las imágenes se realiza de forma rápida y sencilla, resuelve un sin fin de quebraderos de cabeza y creo que se debería animar más a utilizar esta herramienta en los estudios.

5.1. Trabajo a realizar

Queda pendiente la incorporación de un modelo de lenguaje creado por cuenta propia, tal vez utilizando otro corpus de voz pueda conseguirse esta hazaña. También como mejora directa de este proyecto podría añadirse la funcionalidad de reconocer archivos de voz grabados directamente desde la aplicación, esta tarea es sencilla de realizar ya que se implementó en la aplicación inicial.

Como idea de proyecto, se podría implementar un editor de transcripciones incorporando este reconocedor en otra aplicación que permitiese crear diferentes documentos de texto, que se escribiesen a través del reconocedor y que posteriormente permitiese editarlos y guardarlos. La aplicación podría tener un sistema de usuarios, de esta forma cada usuario podría tener seleccionado por defecto un reconocedor de voz con el modelo de lenguaje de su idioma y también acceso únicamente a sus transcripciones, que son las que le interesan.

[1]Mensaje de Dan Povey sobre el corpus TIMIT [17]

Bibliografía

[1] Definición de Reconocimiento Automático del Habla. Recuperado de `https://es.wikipedia.org/wiki/Reconocimiento_del_habla` a Septiembre de 2020.

[2] Definición de fonema. Recuperado de `https://dle.rae.es/fonema` a Septiembre de 2020.

[3] Fotografía de la máquina Shoebox de IBM para reconocimiento del habla. Recuperado de `https://www.ibm.com/ibm/history/ibm100/us/en/icons/speechreco/transform/` a Septiembre de 2020.

[4] Teoría sobre modelos ocultos de Markov. Recuperado de `https://es.wikipedia.org/wiki/Modelo_oculto_de_M%C3%A1rkov` a Septiembre de 2020.

[5] Definición de Inteligencia artificial por la Real Academia Española. Recuperado de `https://dle.rae.es/inteligencia#2DxmhCT` a Septiembre de 2020.

[6] Proyecto de investigación de la Universidad de Dartmouth sobre Inteligencia Artificial. Recuperado de `https://en.wikipedia.org/wiki/Dartmouth_workshop` a Septiembre de 2020.

[7] Imagen del modelo de neurona artificial de McCulloch y Pitts. Recuperado de `https://www.researchgate.net/figure/neurona-artificial-McCulloch-Pitts_fig3_39697458` a Septiembre de 2020.

[8] Docker Hub oficial del proyecto Kaldi. Recuperado de `https://hub.docker.com/r/kaldiasr/kaldi` a Septiembre de 2020.

[9] Proyecto GitHub de la aplicación Android *TFG initial demo*. Recuperado de `https://github.com/SantiagoBlascoArnaiz/TFG-initial-demo` a Septiembre de 2020.

[10] Documentación sobre la API speech de Google. Recuperado de `https://developer.android.com/reference/android/speech/package-summary` a Septiembre de 2020.

[11] Documentación sobre la API media de Android. Recuperado de `https://developer.android.com/reference/android/media/package-summary` a Septiembre de 2020.

[12] Proyecto GitHub de la API de VOSK. Recuperado de `https://github.com/alphacep/vosk-api` a Septiembre de 2020.

[13] Proyecto GitHub de aplicación Android utilizando las librerías de VOSK y Kaldi. Recuperado de `https://github.com/alphacep/vosk-android-demo` a Septiembre de 2020.

[14] Página oficial de VOSK con el listado de modelos de lenguaje compatibles con su API. Recuperado de `https://alphacephei.com/vosk/models` a Septiembre de 2020.

[15] Información sobre los grafos HCLG que utiliza Kaldi. Recuperado de `https://kaldi-asr.org/doc/graph.html` a Septiembre de 2020.

[16] Proyecto GitHub de la aplicación Android *VOSK android demo*. Recuperado de `https://github.com/SantiagoBlascoArnaiz/vosk-android-demo` a Septiembre de 2020.

[17] Mensaje de Dan Povey sobre el corpus TIMIT. Recuperado de `https://groups.google.com/g/kaldi-help/c/YUbX_XUkFCw?pli=1` a Septiembre de 2020.

[18] Página oficial de Nvidia con el listado de GPU's compatibles con CUDA. Recuperado de `https://developer.nvidia.com/cuda-gpus` a Septiembre de 2020.

[19] Página oficial de Nvidia para la descarga de CUDA. Recuperado de `https://developer.nvidia.com/cuda-downloads` a Septiembre de 2020.

[20] Página oficial de Android para descargar su entorno de desarrollo integrado oficial. Recuperado de `https://developer.android.com/studio?hl=es` a Septiembre de 2020.

[21] Página oficial de Docker para descargar su herramienta e instalarla. Recuperado de `https://docs.docker.com/engine/install/ubuntu/` a Septimebre de 2020.

[22] Información sobre reconocimiento del habla y su implementación con Kaldi. Recuperado de `https://medium.com/@jonathan_hui/speech-recognition-series-71fd6784551a` a Septiembre de 2020.

Apéndice A

Bibliotecas y herramientas

CUDA

CUDA es un conjunto de herramientas que permiten el computo en paralelo en las gráficas de Nvidia.

En un principio pensé en utilizar mi ordenador personal, ya que dispone de una gráfica dedicada fabricada por Nvidia, pero debido a que es un ordenador portátil la GPU de la que dispone no es la versión por defecto, es una versión notebook, y no es compatible con los drivers de CUDA necesarios para el procesamiento en paralelo, como puede verse en este listado de la página oficial de Nvidia [18].

Nvidia provee en su página oficial una sección dedicada a la descarga de CUDA [19]. En ella que explica con detalle cómo instalar la herramienta en función del sistema, arquitectura, distribución y versión de esta.

Android Studio

Instalé la última última versión hasta la fecha, versión 4.0.1, del entorno de desarrollo Android Studio en mi ordenador personal, ya que la aplicación es para uso en dispositivos Android. Este entorno de desarrollo integrado es el oficial para Android, puede descargarse desde su web oficial [20].

Docker

Docker es una herramienta que permite la creación de contenedores de software, en dichos contenedores se pueden ejecutar imágenes de forma aislada. Estas imágenes pueden ejecutarse en cualquier máquina con Docker instalado, evitando así lo problemas de versión y distribución de software.

Se instaló su última versión, la 19.03.12. Docker indica en su página web los pasos a seguir para la instalación de su producto [21]

Kaldi

Kaldi es una herramienta de código abierto utilizada para el reconocimiento de voz y el procesamiento de señales. Será usada e instalada a través de un contenedor Docker, para ello descargo la imagen oficial de Kaldi para Docker desde la página de Docker [8], en ella explican las distintas imágenes disponibles y cómo ejecutarlas.

VOSK

VOSK es una herramienta de código abierto destinada al reconocimiento de voz sin conexión a internet, que utiliza de forma subyacente la herramienta Kaldi para aprovisionarse de modelos y trabajar con los reconocedores de voz. Puede accederse al código completo de su API en su repositorio de GitHub [12].