E.T.S. de Ingeniería Industrial,
Informática y de Telecomunicación

# Implementación de chatbots con reconocimiento de voz para el control de equipos de consulta en quirófano

Grado en Ingeniería Informática

Trabajo Fin de Grado

Julen Merchán Flores

Mikel Galar Idoate y Mikel Sesma Sara

Pamplona, 11 de junio de 2020

# Resumen

El objetivo de este trabajo fin de grado es el estudio, prueba e implementación de varios motores para el reconocimiento de voz offline en entornos reales como quirófanos o entornos industriales con el objetivo de crear chatbots capaces de realizar un procesamiento del lenguaje natural y de reconocer comandos de voz emitidos por un ser humano en los idiomas español e inglés.

Los chatbots deberán proveer respuestas automáticas a las peticiones realizadas por una o varias personas, realizando el conjunto de acciones asociado a cada petición e informando al usuario en todo momento.

El objetivo final del proyecto será la implementación de un sistema completo en un quirófano real que permita controlar equipos de consulta médicos en quirófano mediante comandos de voz, facilitando así la labor del personal médico durante una operación, aunque también valoraremos la posibilidad de integrar el sistema en un entorno industrial como una fábrica.

# Palabras clave

Machine Learning, redes neuronales, Speech Recognition, motor de reconocimiento de voz, equipos de consulta médicos.

# Índice

1. **Introducción** ... 1

2. **Preliminares** ... 5

2.1. **Machine Learning** ... 5

2.2. **Métodos** ... 10

2.3. **Speech Recognition** ... 19

2.4. **Evaluación de modelos** ... 28

2.5. **Herramientas que vamos a utilizar** ... 32

2.6. **Estado del arte en Speech Recognition** ... 33

3. **Rhasspy** ... 35

3.1. **Introducción a Rhasspy** ... 35

3.2. **Funcionamiento de Rhasspy** ... 36

3.3. **Puesta en marcha** ... 46

3.4. **Pruebas** ... 48

3.4.1. **Configuración previa de los sistemas antes de realizar las pruebas** ... 48

3.4.2. **Pruebas con comandos e intents por defecto y personalizados** ... 49

3.4.3. **Configuración y pruebas con el sistema de wake word** ... 52

3.4.4. **Configuración del sistema de manejo de intents y desarrollo y pruebas de un servidor capaz de gestionar los intents de Rhasspy** ... 53

3.4.5. **Configuración y pruebas del sistema Text to Speech** ... 55

3.4.6. **Configuración y pruebas de Rhasspy y el servidor HTTP para despertar a Rhasspy de forma encadenada** ... 55

3.4.7. **Últimas configuraciones y pruebas de Rhasspy y el servidor HTTP** ... 56

3.5. **Problemas y soluciones** ... 56

4. **Wav2letter++** ... 58

4.1. **Introducción** ... 58

4.2. **Funcionamiento** ... 60

4.3. **Puesta en marcha** ... 66

4.4. **Pruebas** ... 68

4.4.1. **Pruebas con dataset en inglés** ... 68

4.4.2. **Pruebas con dataset en español** ... 71

4.4.2.1. **Creación de un modelo del lenguaje** ... 71

4.4.2.2. **Pre-procesamiento y pruebas utilizando el dataset de Kaggle y la arquitectura de capas convolucionales 2D para el entrenamiento del modelo** ... 72

4.4.2.3. **Entrenamiento utilizando medidas de regularización para el dataset de Kaggle y la arquitectura de capas convolucionales de 2 dimensiones** ... 81

**4.4.2.4. Pre-procesamiento y pruebas utilizando el dataset de Kaggle y el de Common Voice junto con la arquitectura de capas convolucionales 2D para el entrenamiento del modelo** ........ 89

**4.4.2.5. Pruebas utilizando el dataset de Kaggle y diferentes arquitecturas de red para el entrenamiento del modelo** ........ 94

**4.5. Problemas y soluciones** ........ 102

**5. Comparativa experimental** ........ 105

**5.1. Comparativa de herramientas y resultados** ........ 105

**5.2. Integración final** ........ 108

**6. Conclusiones y líneas futuras** ........ 109

**6.1. Conclusiones** ........ 109

**6.2. Líneas futuras** ........ 110

**7. Bibliografía** ........ 113

# 1. Introducción

La Inteligencia artificial, aunque no lo parezca, no es algo nuevo que ha aparecido en nuestras vidas, sino que sus orígenes se remontan a los años 50, cuando Alan Turing publicó un artículo llamado "Computing Machinery and Intelligence" [1], en el que cuestiona si una máquina podría llegar a pensar y aprender para poder interactuar con los humanos, para ello propuso la famosa prueba de Turing con la que determinar si una máquina es inteligente o no. La prueba consta de 3 elementos, 2 de los cuales son personas y el tercero es un computador diseñado para simular respuestas lo más similares posibles a las de un ser humano, 1 de las personas asume el rol de evaluador o interrogador y debe mantener una conversación con los otros 2 elementos del experimento intentando determinar por las respuestas que obtiene cuál de los dos es el computador y cual el ser humano, en caso de que el evaluador no pueda distinguirlos se considera que la máquina ha pasado la prueba. La Figura 1 muestra un sencillo esquema del funcionamiento de la prueba.

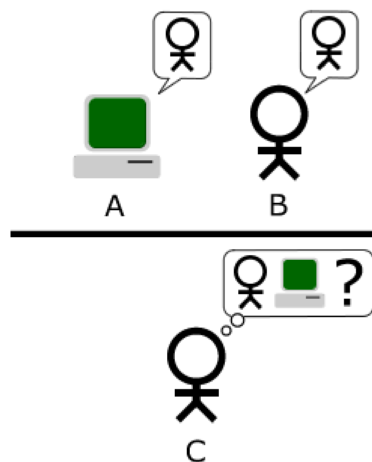



*Figura 1 Test de Turing [https://es.wikipedia.org/wiki/Test_de_Turing]*

Es por esto por lo que Alan Turing es considerado el padre de la inteligencia artificial, además de ser considerado el padre de la computación moderna, pues diseñó una máquina con su mismo nombre que ha resultado ser la precursora de las computadoras que utilizamos hoy en día. La primera aparición de este concepto se da en el artículo que Turing publicó en el año 1936 "On computable numbers, with an application to the Entscheidungsproblem" [2], en el que además estableció las bases teóricas para las ciencias de la computación e introdujo el concepto de algoritmo.

En 1958 John McCarthy introduce LISP, el primer lenguaje de programación pensado para programar software relacionado con la inteligencia artificial, de hecho, en sus inicios fue un lenguaje muy popular en la investigación de la inteligencia artificial, en gran parte porque a diferencia de otros lenguajes que existían en la época en vez de tratar con datos numéricos directamente se trata con representaciones de estructuras simbólicas, lo que facilitaba la interpretación por parte del desarrollador.

Hacia 1980 ya existían alrededor de 3 lenguajes de programación pensados para el desarrollo de inteligencia artificial: LISP, NIAL y PROLOG [3].

En 2011 IBM presenta su ordenador IBM Watson en un programa de televisión de preguntas y respuestas llamado “Jeopardy!” y consigue ganar a los dos mejores concursantes de la historia. Watson representa un primer paso en la era de los sistemas cognitivos, pues combina diferentes habilidades: un procesamiento natural del lenguaje (PNL), la generación y evaluación de hipótesis y el aprendizaje dinámico, lo que le permite dar respuestas basadas en una alta confianza [4].

En el año 2014 finalmente un sistema consigue superar el test de Turing haciendo creer al interrogador del test que estaba hablando con un niño ucraniano de 13 años, esto supone un hito histórico y un gran avance en la inteligencia artificial. El sistema o bot conversacional llamado Eugene Gostman, fue presentado en un concurso de la Royal Society en Londres y consiguió engañar al 33% de los jueces en los test que se habían preparado [5].

En la actualidad, la demanda de trabajos del sector informático ha crecido exponencialmente, más concretamente aquellos trabajos relacionados con la inteligencia artificial como lo demuestra el informe de empleos emergentes en España en 2020 realizado por LinkedIn [6], donde se observa que este tipo de empleos han crecido en aproximadamente un 76% en el último año. Esto mismo está ocurriendo en otros países como Reino Unido donde el incremento de la demanda de este tipo de empleos ha sido de un 74% en los últimos 4 años [7].

Cuando hablamos de IA o Inteligencia Artificial nos referimos a un gran conjunto de disciplinas como: Machine Learning, Deep Learning, Procesamiento del Lenguaje Natural (PLN)... que han ido surgiendo con el paso de los años y que tratan de resolver una gran variedad problemas, algunas de ellas las vemos en la Figura 2. Muchas de estas ramas de la inteligencia artificial están aplicadas no solo al entorno profesional sino también a productos y servicios que vemos y utilizamos en nuestro día a día como, por ejemplo: sistemas de recomendación de páginas web y plataformas de streaming, asistentes de voz virtuales, robots de limpieza... Un claro ejemplo es el sistema de recomendación de la plataforma Netflix, por el que la compañía estaba dispuesta a pagar 1 millón de dólares a quien presentase el mejor sistema[8].

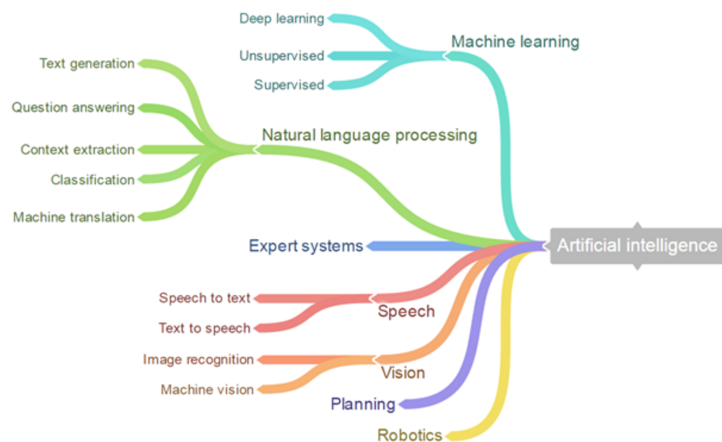



*Figura 2 Ramas de la Inteligencia Artificial [https://es-la.facebook.com/SysoiTechnologies/photos/branches-of-artificial-intelligence-artificial-intelligence-is-not-a-tree-but-a-/1071949122908336/]*

El éxito de la IA en el mercado laboral es debido en mayor parte a que el perfil de especialista en inteligencia artificial se está incorporando de un manera muy rápida a otro ámbitos de la informática como a las Tecnologías de la Información y Comunicación (TIC) o a la programación informática [6]. Como resultado de estos nuevos empleos se están desarrollando productos y servicios innovadores que en muchos casos están sustituyendo a los existentes. La aplicación de

estas nuevas técnicas poco a poco se está extendiendo a muchos sectores como el de las finanzas, el de la educación o incluso el de la salud, donde ya existen diversos sistemas que permiten obtener el diagnóstico de un paciente de una manera más rápida y precisa [9], como es el caso del sistema Watson For Oncology (WFO) desarrollado por IBM que permite detectar de forma precisa casos de cáncer de mama y recomendar tratamientos personalizados dependiendo da cada paciente [10].

No obstante, la aplicación de la IA a este sector no es algo novedoso, entre los años 60 y 70 ya se dejaban ver los primeros sistemas expertos como es el caso de Dendral y Mycin, ambos escritos en LISP. Dendral permitía identificar moléculas orgánicas a partir de los resultados obtenidos de un espectrómetro de masas y de una serie de conocimientos básicos de química principalmente acerca de pesos atómicos y valencias [11]. En cuanto a Mycin, se trataba de un sistema capaz de predecir o diagnosticar enfermedades infecciosas de acuerdo a los síntomas que presentaba el paciente y era capaz de recetar medicamentos personalizados dependiendo de cada persona, estaba basado en Dendral, pues era posterior a este, dependiendo de los síntomas introducidos Mycin mostraba varios diagnósticos y la explicación del porqué de cada uno [12].

Con el paso del tiempo Mycin dejó de utilizarse debido entre otras razones a debilidades del sistema pero también a cuestiones éticas y legales al dejar la responsabilidad de la salud en manos de una máquina, esto es algo que incluso a día de hoy está lastrando la implementación de sistemas similares, otros aspectos que están influyendo en la actualidad son los intereses económicos, que la mayor parte del sector de la salud no cuenta con infraestructuras de datos óptimas para recoger la información que necesitan este tipo de sistemas y temas relacionados con la privacidad de los datos que se manejan [13]. Hemos tenido en cuenta muchos de estos aspectos como se reflejará a lo largo del documento.

A pesar de esto, hoy en día seguimos viendo un incremento constante productos y servicios aplicados al sector de la salud con el fin de facilitar y mejorar la tarea del personal médico. Algo que está en auge en los últimos años es la aplicación de Speech Recognition o reconocimiento del leguaje para la creación de sistemas destinados a temas médicos, como Dragon de la compañía Nuance, una herramienta que permite guardar registros médicos de voz de pacientes en vez de emplear tiempo introduciéndolos por teclado y otros ejemplos como Tedcube, de la empresa Tedcas Medical Systems S.L, que permite controlar por voz y gestos las imágenes e información dentro de un quirófano.

El reconocimiento de voz automático o Speech Recognition es una de las disciplinas de la inteligencia artificial que hemos visto más arriba (Figura 2) , permite la transcripción del lenguaje hablado a texto por parte de las computadoras, también es conocido como Speech to Text, en la Figura 3 se muestra un sencillo esquema del funcionamiento. Tiene múltiples aplicaciones como en domótica para el control de elementos del hogar mediante voz, asistentes virtuales, sistemas de traducción en tiempo real...

*Figura 3 Funcionamiento básico del reconocimiento de voz [https://www.authot.com/]*

Es justamente en esta disciplina en la que se va a centrar este proyecto fin de grado que hemos desarrollado y más concretamente en esta misma aplicada al sector de la salud. Este trabajo fin de grado tiene por objeto mejorar y facilitar la tarea del personal médico en quirófano durante una operación, mediante el desarrollo y la integración de un sistema de reconocimiento de voz capaz de reconocer comandos de voz en diferentes idiomas y de llevar a cabo un conjunto de acciones personalizadas dependiendo de cada uno. Para ello proponemos:

1. Probar diferentes herramientas o motores para la construcción de un sistema capaz de realizar un reconocimiento automático del lenguaje tanto en idioma español como inglés.

2. Construir la parte del sistema capaz de reconocer comandos de voz a partir de alguna de las herramientas que vamos a probar.

3. Construir la parte del sistema capaz de procesar los comandos y de realizar las tareas que se especifiquen en estos, como controlar los equipos de consulta en quirófano.

4. Construir la parte del sistema que permita activar el reconocimiento de voz mediante alguna palabra clave.

5. Construir la parte del sistema capaz de enviar una respuesta audible al usuario.

6. Montar todo el sistema juntando cada una de las partes.

7. Probar el sistema en un entorno real y obtener un feedback de los usuarios para mejorar y/o ampliar las funcionalidades de este.

Puesto que en todo momento en un entorno real trataremos con información de un carácter sensible, será de gran importancia que cada una de las partes de este sistema trabaje de manera offline para evitar posibles filtraciones de datos durante el proceso.

La motivación para realizar este trabajo viene de comprender la gran labor que realiza todo el equipo médico durante una operación y de la importancia que tiene cada minuto durante el transcurso de esta. Por ello con este proyecto proponemos un sistema que permita ahorrar tiempo en quirófano al permitir controlar instrumental médico mediante la voz, además de facilitar en la medida de lo posible el trabajo de estos profesionales.

# 2. Preliminares

En esta sección del documento pasamos a hablar de una serie de conocimientos que consideramos necesarios para el entendimiento de las explicaciones posteriores del trabajo que hemos desarrollado, pues constituyen las bases teóricas del funcionamiento de las herramientas que hemos utilizado.

## 2.1. Machine Learning

Comenzamos hablando del Machine Learning una de las ramas que veíamos en la Figura 2. De forma intuitiva podríamos definir el Machine Learning como máquinas inteligentes capaces de aprender a partir de un conjunto de ejemplos con la finalidad de imitar a los humanos, sin embargo no existe una definición concreta, algunas definiciones que se le han dado a lo largo de la historia han sido [14]:

1. En 1959 Arthur Samuel lo definió como: "El área de estudio que da a los computadores la capacidad de aprender sin ser programados de forma explícita".

2. En 1989 Tom Mitchell definió un problema de Machine Learning como: "Un programa se dice que aprende de una experiencia E respecto a una tarea T y alguna medida de rendimiento P, si su rendimiento en T, medido mediante P, mejora con la experiencia E.

3. En 1990 Robert E. Schapire lo definió como: "Aprender a hacer las tareas mejor en el futuro en base a las experiencias (conocimiento) pasadas".

Antes de entrar en detalle, consideramos necesario definir el concepto de algoritmo, pues es esencial para el entendimiento de los siguientes conocimientos. Un algoritmo podríamos definirlo como un conjunto ordenado de reglas o instrucciones finitas predefinidas, cuyo objetivo es realizar un conjunto de tareas o actividades. Generalmente este tipo de algoritmos se representan mediante algoritmos de flujo como el de la Figura 4. El concepto de algoritmo no es exclusivo de la informática sino que los aplicamos incluso en nuestra vida cotidiana como por ejemplo al poner una lavadora.

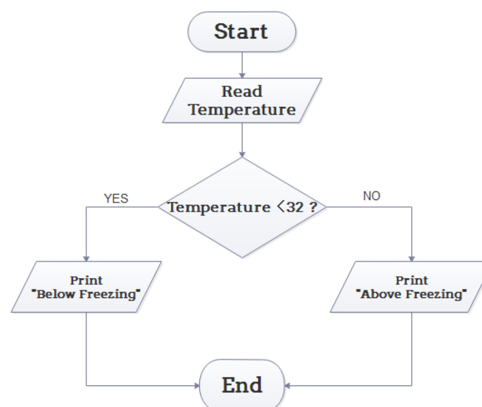



*Figura 4 Diagrama de flujo de un algoritmo [https://www.edrawsoft.com/]*

El Machine Learning surge debido a dos razones importantes:

1. Busca resolver problemas que no pueden resolverse con programas tradicionales en los que se trata con grandes cantidades de datos y de bastante complejidad.

2. Busca generar aplicaciones que serían imposibles de programar de manera tradicional como por ejemplo: Procesamiento del lenguaje natural, reconocimiento de objetos en tiempo real, reconocimiento de texto...

La idea que podemos sacar con las definiciones que hemos dado antes es que el objetivo que buscamos es diseñar algoritmos que sean capaces de aprender patrones a partir de una serie de datos para realizar mejor las tareas de un futuro, al proceso de aprendizaje se le denomina comúnmente entrenamiento, pues el algoritmo "entrena" con una serie de ejemplos para volverse más robusto y realizar sus tareas de una forma óptima. Para aprender, los algoritmos pueden seguir varios tipos de estrategias [15], entre las que destacan:

1. Aprendizaje supervisado: Este tipo de aprendizaje se basa en que los ejemplos o datos con los que entrena el algoritmo, ya se encuentran etiquetados a priori, es decir que para cada uno de los ejemplos de entrada ya conocemos la salida deseada.

   El objetivo final de este tipo de aprendizaje es definir una función que sea capaz de mapear cada ejemplo valido de entrada con su salida correspondiente.

   El algoritmo extrae el conocimiento de los datos de entrada tratando de predecir la salida de cada uno y comparando su predicción con la salida esperada. Con cada error que el algoritmo comete al predecir aprende y trata de ajustar su función de predicción para casos futuros.

   Los datos de salida pueden ser o bien un valor real o bien una clase o etiqueta como: "gato", "perro", "sano", "enfermo" ... Dependiendo del tipo de salida distinguimos dos tipos de problemas:

   1.1. Problemas de clasificación: En este tipo de problemas la salida es una clase o etiqueta (un valor discreto) a la que pertenece el dato de entrada, podríamos definirlo como el "tipo" del dato de entrada.

   Para clasificar cada ejemplo en este tipo de problemas, el algoritmo hace uso de los atributos o características que definen a cada instancia y dependiendo de los valores de estas los clasifica de una manera u otra. Algunos ejemplos de este tipo de problemas que vemos en nuestro día a día son: los sistemas de detección de fraudes, los sistemas para determinar la concesión o no de créditos, los filtros de spam de las aplicaciones de correo...

   Dependiendo de las de posibles salidas diferenciamos varios tipos de problemas de clasificación:

   1.1.1. Problemas binarios: En este tipo de problemas las posibles salidas se reducen a solo dos clases.

   1.1.2. Problemas multi-clase: En este tipo de problemas las posibles salidas se son más de dos clases

1.1.3. Problemas multi-etiqueta: En este tipo de problemas cada ejemplo puede pertenecer a más de una clase.

1.1.4. Problemas multi-instancia: En este tipo de problemas cada ejemplo está formado por varios ejemplos

1.2. Problemas de regresión: En este tipo de problemas la salida es un valor real (un valor numérico). Al igual que en los problemas de clasificación que hemos comentado, el algoritmo utiliza los atributos de cada ejemplo o instancia para determinar el valor de la salida. Algunos ejemplos de este tipo de problemas son: la predicción del valor de un inmueble, la predicción del valor de las acciones, la predicción de la temperatura en una zona...

Dependiendo de las posibles salidas diferenciamos varios tipos de problemas de regresión:

1.2.1. Problemas de regresión con una única variable de salida.

1.2.2. Problemas de regresión con más de una variable de salida.

2. Aprendizaje no supervisado: Este tipo de aprendizaje se basa en que los ejemplos, o datos con los que entrena el algoritmo, no se encuentran etiquetados a priori, es decir que para cada uno de los ejemplos de entrada no conocemos la salida deseada. Por tanto, el algoritmo trata de encontrar una estructura común en los datos que le permita agruparlos.

   Este tipo de aprendizaje es típico de los problemas de clustering en los que se crean subconjuntos de ejemplos, de manera que los ejemplos de un subconjunto son similares entre sí y los elementos de diferentes subconjuntos son diferentes entre sí. Para determinar cuan parecidos son los datos entre si se utilizan medidas de similitud como la distancia euclídea o la distancia de Manhattan [16].

   Encontramos diferentes técnicas de clustering [16] divididas en dos grandes grupos como muestra la Figura 5 :

   2.1. Clustering jerárquico: En clustering jerárquico agrupamos los datos basándonos en la distancia entre cada uno y buscando que los datos que están dentro de un clúster sean los más similares entre sí.

   2.2. Clustering particional: En clustering particional buscamos obtener una partición de los datos en clústeres de forma que todos los ejemplos pertenezcan a alguno de los k clústeres posibles, mientras que por otra parte los clústeres sean disjuntos entre sí.

   Algunas aplicaciones de clustering que vemos en nuestro día a día son: agrupamiento de imágenes, agrupamiento de resultados de búsqueda, segmentación de clientes para el análisis y la creación de publicidad dirigida a grupos de clientes...

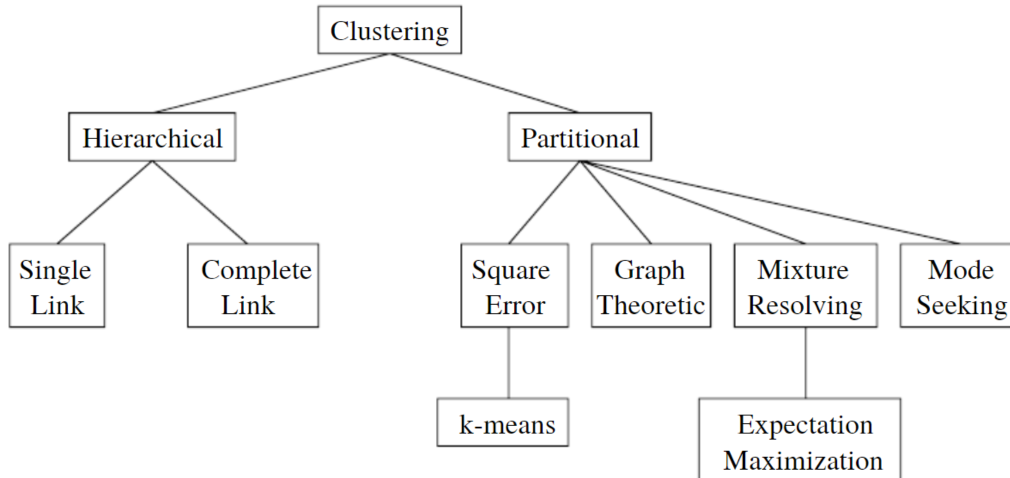



*Figura 5 Tipos de clustering* [16]

No nos centraremos mucho en este tipo de aprendizaje a lo largo de la memoria pues en este proyecto trabajaremos con un aprendizaje supervisado.

3. Aprendizaje semi-supervisado: Este tipo de aprendizaje se basa en que solo algunos de los ejemplos o datos con los que entrena el algoritmo se encuentran etiquetados a priori, es decir que para cada uno de los ejemplos de entrada solo conocemos la salida deseada de unos pocos. Se ha demostrado que en algunos problemas utilizar una pequeña cantidad de datos etiquetados junto a una gran cantidad de datos no etiquetados permite obtener mejores resultados [17].

   Generalmente el proceso asociado a etiquetar datos adquiridos es bastante costoso pues requiere de una persona humana para hacerlo, mientras que adquirir datos sin etiquetar resulta menos costoso. Este hecho hace que en muchos casos etiquetar conjuntos de entrenamiento resulte del todo inviable.

   Algunos ejemplos de este tipo de problemas son: el análisis de conversaciones grabadas en un call center con el objetivo de inferir características, estados de ánimo y motivos de llamada de los interlocutores, el Co-Training que consiste en utilizar el mismo conjunto de datos pero en dos algoritmos diferentes o para un mismo conjunto de datos encontrar dos subconjuntos de atributos independientes y entrenar un modelo con cada uno de ellos.

   Estos 3 últimos casos los podemos resumir en la Figura 6 . En esta se habla de modelos, estos son el resultado del proceso de aprendizaje. Estos modelos o programas deben ser capaces de generalizar comportamientos o tareas que hayan aprendido con los datos de entrenamiento para poder tener un rendimiento óptimo ante datos desconocidos. Esta es una de las grandes diferencias entre la programación tradicional y el aprendizaje automático, pues en la primera la computadora trabaja con unos datos y un programa y produce una salida que puede ser un número, una frase, una acción... mientras que en la segunda la computadora trabaja con unos datos y un programa y como salida produce otro programa o como es comúnmente llamado, un modelo.

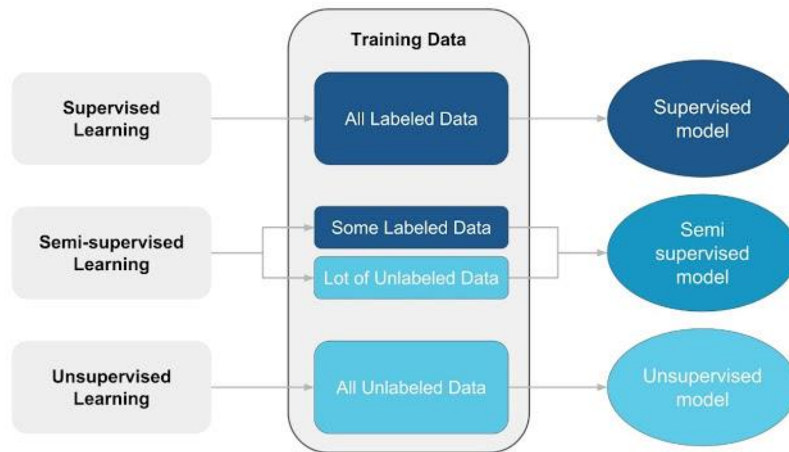



*Figura 6 Métodos de aprendizaje [https://empresas.blogthinkbig.com/semi-supervised-learningel-gran-desconocido/]*

4. Aprendizaje por refuerzo: Este tipo de aprendizaje se basa en ensayo-error, es decir el algoritmo aprende a través del feedback externo que obtiene de su entorno como respuesta a las acciones que ha realizado. A diferencia de otros tipos de aprendizaje en el que se les establece a los algoritmos cuál debe ser su salida, en este tipo de aprendizaje el algoritmo debe aprender cómo comportarse en un entorno mediante prueba y error, en el que recibirá refuerzos en caso de que sus acciones sean correctas o castigos en caso contrario. De esta manera, el algoritmo aprende a comprender el comportamiento de su entorno y a tomar decisiones que le lleven a cumplir una serie de objetivos.

   Algunos ejemplos de este sistema de aprendizaje son: el sistema AlphaGo, un sistema desarrollado por Google DeepMind para jugar el juego de mesa Go y los sistemas de navegación de los robots.

Hasta ahora hemos visto que los algoritmos “entrenan” con una serie de datos o mediante ensayo y error con el objetivo de generar un modelo o programa. Este modelo debe generalizar correctamente para casos futuros (datos o situaciones con las que no ha sido entrenado) con el conocimiento que ha adquirido, para ello algo que resulta esencial es que los datos de entrenamiento sean correctos, por lo que generalmente antes de realizar la fase de entrenamiento es común llevar acabo un preprocesamiento de los datos para eliminar datos inconsistentes o redundantes, dejar los datos en un formato adecuado para el entrenamiento... Además, durante la etapa de entrenamiento se realizan de forma continua análisis de los resultados parciales que se van obteniendo con el objetivo de ajustar el entrenamiento para mejorar el modelo generado.

Una vez obtenido el modelo, pasaríamos a la fase de predicción o test donde este se prueba en un entorno real o con ejemplos con los que no ha sido entrenado para ver su desempeño. Todo eso se puede esquematizar a través de la Figura 7.

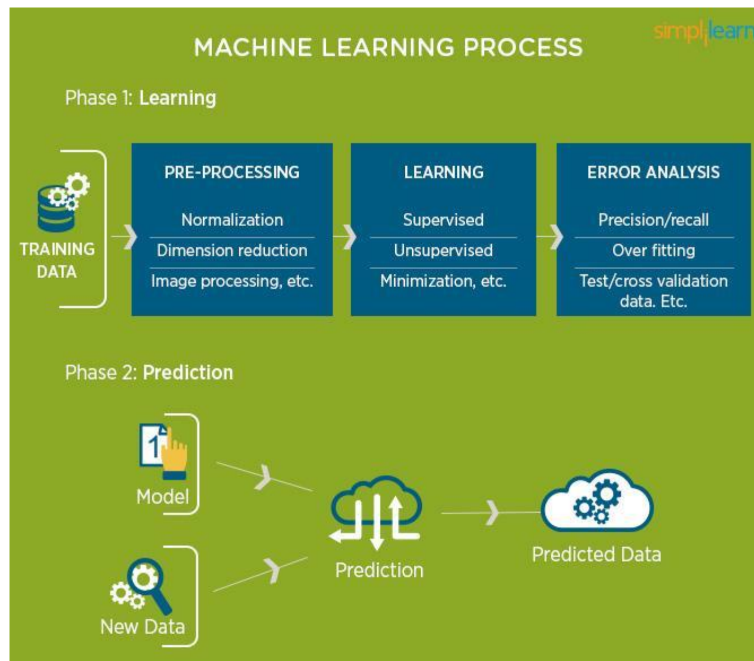



*Figura 7 Proceso del Machine Learning [https://www.simplilearn.com/tutorials/machine-learning-tutorial/what-is-machine-learning?source=sl_frs_nav_playlist_video_clicked]*

## 2.2. Métodos

En esta sección del documento vamos a comentar algunos de los métodos o algoritmos de Machine Learning para la generación de modelos. Antes de explicar estos métodos creemos conveniente introducir dos conceptos que van a aparecer repetidas veces a partir de ahora:

1. Función de coste o error: Es una función matemática que determina de forma numérica cuánto se equivoca un modelo en su predicción respecto al valor esperado de cada muestra del conjunto de datos. En los problemas de optimización trataremos de maximizar o minimizar el coste dependiendo de cada caso, esto nos permitirá ajustar los parámetros que el modelo utiliza para realizar sus predicciones. Además, la definición de esta función variará dependiendo de cada problema.

2. Descenso por gradiente: Se trata de un método iterativo de optimización muy utilizado para ajustar los parámetros de un modelo tratando de minimizar la función de coste que se haya definido buscando el mínimo global de esta, este método no asegura la convergencia a dicho valor, pero si asegura la convergencia a un mínimo local. El descenso por gradiente se basa en encontrar la dirección de más rápida y mayor variación, que viene dada por el vector gradiente, para ello en cada iteración o paso de este método se calculan las derivadas parciales de cada dimensión (variable) considerada en el problema. Distinguimos dos tipos de descenso por gradiente:

    2.1. Descenso por gradiente estocástico (SGD): En este caso en cada iteración del gradiente se trabaja con un único ejemplo, por lo que los parámetros del modelo se actualizan para cada ejemplo. Por norma general suele ser más rápido pero tiene el inconveniente de caer en mínimos locales.

2.2. Descenso por gradiente por lotes: En este caso en cada iteración del gradiente se trabaja con un grupo o lote de ejemplos, por lo que los parámetros del modelo se actualizan para cada conjunto de ejemplos. Por norma general suele ser más lento pero suele dar mejores resultados que el estocástico al tener menor probabilidad de caer en mínimos locales. Cuando el tamaño del lote es igual al tamaño del conjunto de datos, hablamos de un descenso por gradiente tradicional, en el que en cada paso o iteración del método se utilizan todos los ejemplos.

En este algoritmo hablaremos de tasa de aprendizaje para referirnos al hiper parámetro que establece cuánto aprendemos en cada paso del descenso o dicho de otro modo cuánto avanzamos en el proceso de descenso por gradiente, el ajuste de este valor resulta esencial para lograr un buen rendimiento, pues un valor demasiado bajo puede hacer que el descenso necesite demasiadas iteraciones para converger y un valor demasiado alto puede provocar que el modelo acabe divergiendo y nunca llegue a converger como muestra la Figura 8.

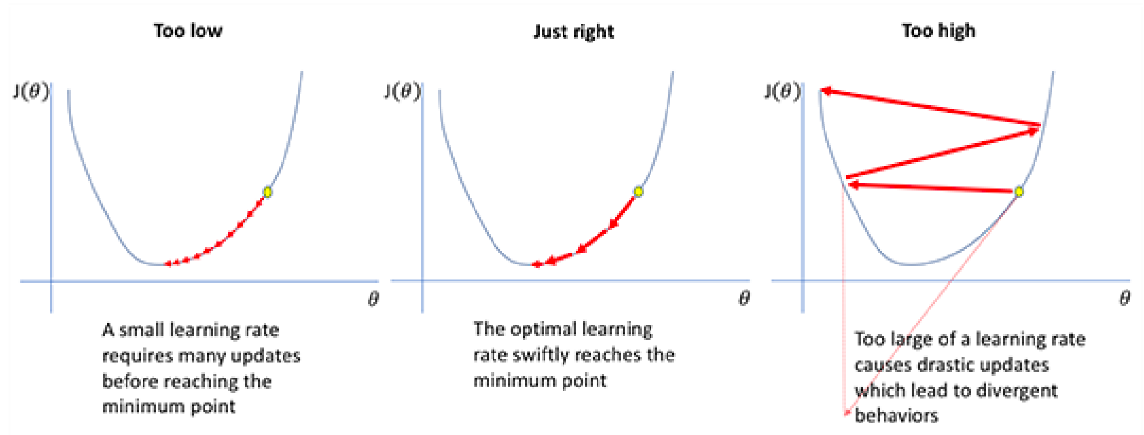



*Figura 8 Importancia de un buen valor de tasa de aprendizaje [https://www.pyimagesearch.com/2019/07/29/cyclical-learning-rates-with-keras-and-deep-learning/]*

Hablaremos de otros como: Momentum, Adam beta 1, Adam beta 2... que básicamente tienen como función acelerar el proceso de descenso por gradiente. La Figura 9 y la Figura 10 muestran un ejemplo del proceso de descenso por gradiente.

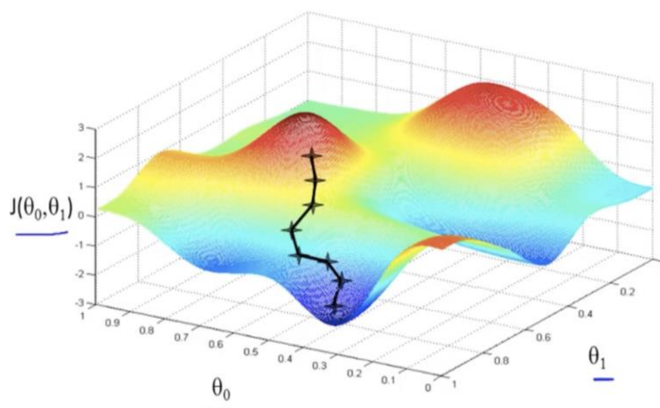



*Figura 9 Proceso de descenso por gradiente en 3D [https://miro.medium.com/max/1098/1*yasmQ5kvlmbYMe8eDkyl6w.png]*

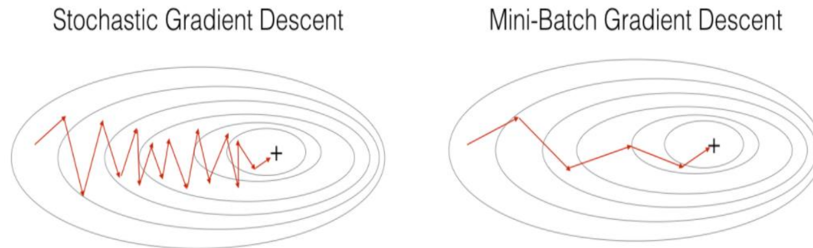



*Figura 10 Descenso estocástico - Descenso por lotes [https://www.i2tutorials.com/wp-content/uploads/2019/09/Neural-network-18-i2tutorials.png]*

- Regresión lineal: En este tipo de método el objetivo es encontrar un modelo matemático que permita representar un conjunto de datos, de tal forma que sea capaz de predecir valores para nuevos datos de entrada, es decir buscará una función matemática que permita obtener cada valor de salida Y mediante la combinación lineal de las variables de los datos de entrada. La salida en regresión lineal es un valor real, por lo que este método está dentro del aprendizaje supervisado y en concreto dentro de los problemas de regresión. Distinguimos dos tipos de problemas de regresión lineal:

    1. Regresión lineal simple: En este tipo de regresión se trabaja con una única variable independiente por lo que el valor de salida depende de esta única variable de entrada, por ejemplo: predecir el valor de una casa en función de los metros cuadrados.

    2. Regresión lineal múltiple: En este tipo de regresión se trabaja con varias variables independiente por lo que el valor de salida depende de la combinación de estas variables, por ejemplo: predecir el valor de una casa en función de los metros cuadrados, el número de habitaciones y el número de cuartos de baño.

A medida que crece el número de variables resulta más difícil la representación gráfica de los valores, en la Figura 11 se muestra la aplicación de la regresión lineal simple para la resolución del problema de predicción del valor de una casa en función de los metros cuadrados.

*Figura 11 Regresión logística: predicción del valor de una casa [https://rstudio-pubs-static.s3.amazonaws.com/]*

A la hora de aplicar regresión lineal a un problema trataremos siempre de minimizar la función de coste utilizando para ello el método de descenso por gradiente que veíamos antes.

- Regresión logística: Al igual que en regresión lineal el objetivo de este método es generar un modelo (estadístico) que represente un conjunto de datos y permita hacer predicciones ante la entrada de nuevos ejemplos. Sin embargo, en este caso la salida pasa de ser un valor real a un valor discreto, es decir una clase o etiqueta que permite clasificar al ejemplo de entrada. por lo que este método está dentro del aprendizaje supervisado y en concreto dentro de los problemas de clasificación. Distinguimos dos tipos de problemas de regresión logística:

    1. Problemas de clasificación binaria: Son problemas en los que solo existen dos clases en las que clasificar a los ejemplos, por lo que las posibles salidas se reducen a 2.

    2. Problemas multi-clase: Son problemas en los que existen más de dos clases en las que clasificar a los ejemplos, por lo que las posibles salidas son mayores que 2.

Para la regresión logística, a diferencia de la regresión lineal, es necesaria una función que modele las probabilidades en función de los datos de entrada y de los valores de los parámetros del modelo, es decir, que necesitamos una función que a partir de unos valores reales de entrada permita obtener como salida un valor entre 0 y 1 (una probabilidad), ya que para clasificar trabajaremos con probabilidades de pertenecer a una clase u otra. En este contexto se introduce la función sigmoide, que dado cualquier valor de real de entrada la salida que produce estará entre 0 y 1 como podemos observar en la Figura 12.

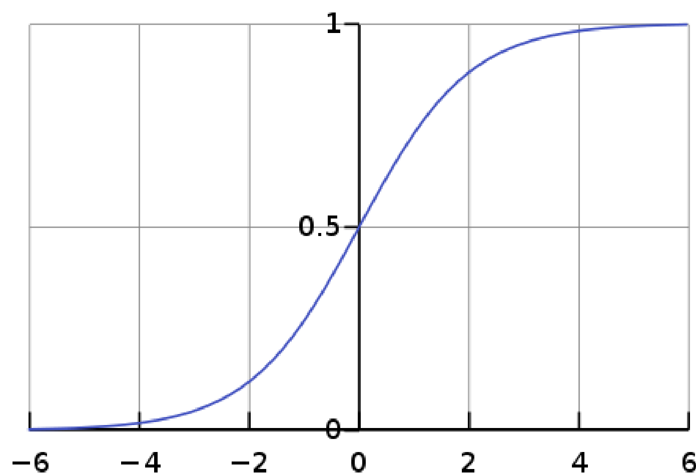



*Figura 12 Función sigmoide [https://en.wikipedia.org/wiki/Logistic_regression]*

Al igual que en regresión lineal trataremos de minimizar la función de coste aplicando la técnica de descenso por gradiente.

- Naïve Bayes [18], [19]: Naïve Bayes es un método de clasificación probabilista, que se basa fundamentalmente en el teorema de Bayes, a partir de las probabilidades a priori, aplicando el teorema de Bayes permite calcular las probabilidades a posteriori. Es decir, a partir de las probabilidades a priori calculadas previamente a la clasificación de cada clase del problema y de tener una instancia determinada, este clasificador permite calcular la probabilidad de tener una instancia determinada dada una clase del problema, esto se muestra en la Figura 13. A partir del cálculo de todas las probabilidades a posteriori para cada ejemplo, se calcula mediante la regla MAP (Maximum a Posteriori) cuál de ellas es más alta y se asigna al ejemplo la clase asociada a la probabilidad más alta.

  Algo importante de Naïve Bayes es que asume una independencia entre las variables, es decir que considera que cada una de las variables que componen a un ejemplo contribuyen de una forma independiente a la probabilidad de clasificar al mismo en una de las clases del problema, independientemente de si alguna de estas está presente o no.

Likelihood

Class Prior Probability

$$P(c|x) = \frac{P(x|c)P(c)}{P(x)}$$

Posterior Probability

Predictor Prior Probability

$$P(c|X) = P(x_1|c) \times P(x_2|c) \times \cdots \times P(x_n|c) \times P(c)$$

*Figura 13 Cálculo de probabilidades a posteriori [https://i.pinimg.com/originals/be/ae/d7/beaed77528e2a23adaa93c1b32baa3c5.png]*

Naïve Bayes se considera un clasificador probabilista generativo pues a diferencia de otros clasificadores como por ejemplo la regresión logística, no busca generar una frontera entre los ejemplos que permita separarlos en diferentes clases sino que, más bien se centra en una clase del problema y con el cálculo de las probabilidades a posteriori genera un modelo para cada clase que permite clasificar a cada uno de los ejemplos o datos del conjunto.

Hemos visto que Naïve Bayes es un clasificador probabilista generativo, el otro tipo de clasificadores son los discriminativos, este tipo de clasificadores buscan a partir del ajuste de los parámetros de sus modelos crear una frontera (frontera de decisión) que les permita separar los datos en sus respectivas clases como podemos ver en la Figura 14.

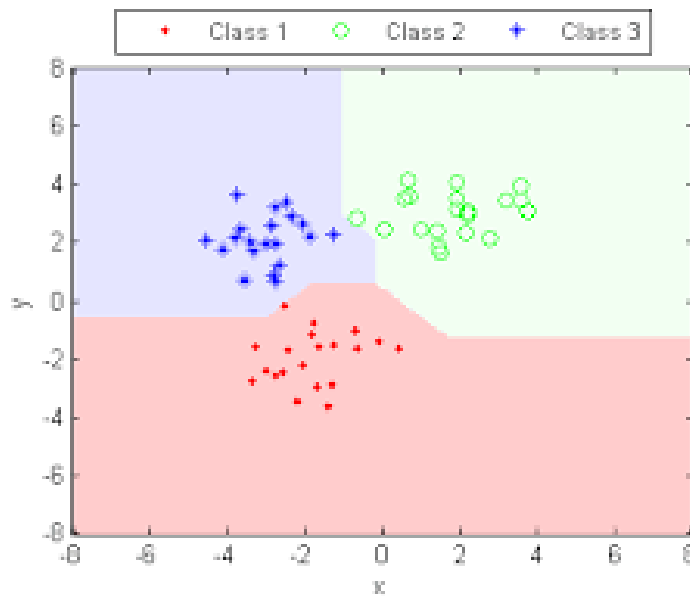



*Figura 14 Fronteras de decisión [www.peteryu.ca]*

Este clasificador rinde muy bien en entornos donde el aprendizaje es supervisado, pues requiere de una cantidad muy pequeña de datos para poder entrenar y generar los parámetros necesarios para clasificar de cada modelo. Una aplicación típica de este clasificador es la creación de un filtro de spam para aplicaciones de correo electrónico [20].

Hasta ahora hemos visto el clasificador Naïve Bayes binomial donde tenemos múltiples ejemplos o muestras y un par de categorías, pasaremos a ver Naïve Bayes multinomial que se basa en la distribución con el mismo nombre, es un caso generalizado de la distribución binomial para un número de clases o categorías mayor que 2 [21]. Esta versión de Naïve Bayes es muy utilizada en campos del procesamiento del lenguaje natural, un ejemplo típico es la relación ente Naïve Bayes multinomial y los modelos de lenguaje que veremos más adelante. El clasificador Naïve Bayes que habíamos visto hasta ahora considera que la probabilidad de aparición de cada ejemplo dada una clase de forma binaria, es decir el ejemplo podía aparecer o no.

El clasificador Naïve Bayes multinomial mejora en rendimiento pues considera el número de ocurrencias de un ejemplo para determinar la contribución de la probabilidad condicional dada una clase determinada del problema, es decir, el modelo Naïve Bayes multinomial considera la frecuencia de aparición de cada ejemplo en el conjunto de entrenamiento en vez de una ocurrencia binaria [22]. La aplicación más típica de este modelo suele ser la clasificación de texto [22], [23]

- Redes neuronales: Este tipo de métodos se ven inspirado por la naturaleza biológica del cerebro humano, de forma básica una red neuronal está constituida por un conjunto de neuronas artificiales agrupadas en varias capas y conectadas entre sí, como muestra la Figura 15.

  Por tanto la unidad básica de toda red neuronal es la neurona, dentro de una red neuronal cada neurona recibe una serie de entradas a partir de los enlaces que la conectan con la capa anterior, estos enlaces tienen unos valores asignados que

llamaremos a partir de ahora pesos. Los pesos de las conexiones de una neurona con el resto de las neuronas de la siguiente capa se pueden representar de forma vectorial como un vector de una única fila y con tantas columnas como conexiones tiene una neurona con la siguiente capa. Cada posición del vector contiene el valor del peso de la conexión, la agrupación de todos los vectores de una capa, o dicho de otro modo, la agrupación de todas las conexiones de todas las neuronas de una capa con la capa siguiente nos da la matriz de pesos W que vemos representada en la Figura 15.

Toda red neuronal siempre sonta de 3 capas:

1. Capa de entrada: Esta es la primera capa de la red, se encarga de recibir los datos de entrada del exterior y de propagarlos a lo largo de la red. En la Figura 15 está representada por la capa con los elementos de color rojo.

2. Capa(s) oculta(s): Pueden ser una o varias capas, se encargan de tomar los datos de la entrada, procesarlos y propagar las salidas a lo largo de la red hasta llegar a la capa de salida. En la Figura 15 está representada por la capa con los elementos de color azul.

3. Capa de salida: Esta capa es la última de la red, toma como entrada las salidas de la última capa oculta y devuelve la predicción de la red neuronal para las entradas que se habían introducido. En la Figura 15 está representada por la capa con los elementos de color verde.

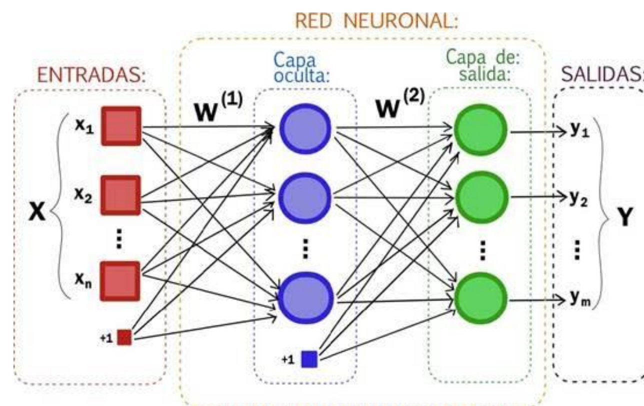



*Figura 15 Estructura de una red neuronal clásica [https://sites.google.com/site/mayinteligenciartificial/unidad-4-redes-neuronales]*

Cada neurona dentro de una red actúa como un clasificador que genera su propio modelo, pues recibe unos datos de entrada, los multiplica por los pesos que tiene asociados y a la salida aplica una función que llamaremos función de activación sobre este resultado, esta función transforma los datos de salida y los deja en un rango determinado de valores (generalmente entre [0,1]) antes de pasarlos a la siguiente capa. Por tanto podemos ver una capa de la red como una pila de modelos donde la salida que proporcionan corresponde con la entrada de capa posterior.

De hecho una red de una única neurona se comporta igual que una regresión logística con tantas características como entradas tenga la neurona en cuestión. Algunas de las

funciones de activación más típicas son: la función sigmoide (Figura 12), la función ReLu (Figura 16) y la función tangente hiperbólica (Figura 17).

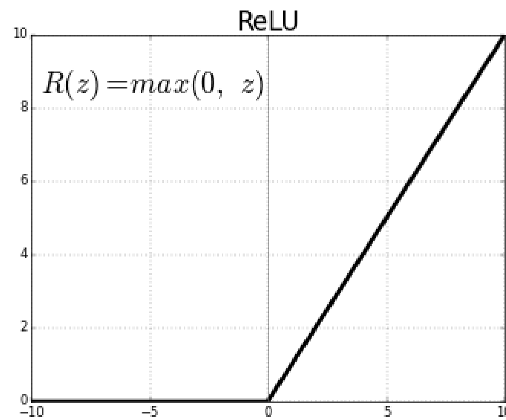



*Figura 16 Función de activación ReLu [https://ml4a.github.io/images/figures/relu.png]*

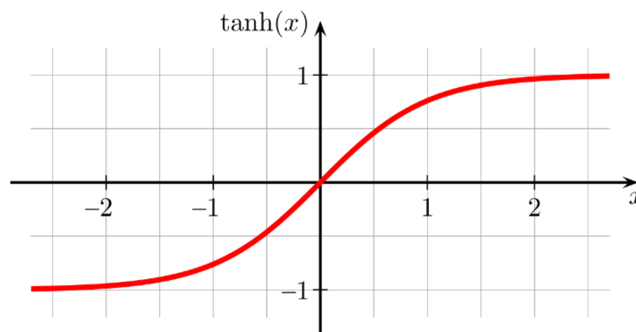



*Figura 17 Función tangente hiperbólica [https://upload.wikimedia.org/wikipedia/commons/thumb/8/87/Hyperbolic_Tangent.svg/1200px-Hyperbolic_Tangent.svg.png]*

Para entrenar una red neuronal, al igual que hemos visto en los métodos anteriores, es necesario definir previamente una función de coste a optimizar, después debemos aplicar descenso por gradiente para minimizarla. Para este caso resulta más complicado realizar el proceso debido a la complejidad que añaden las redes neuronales, en este contexto se introducen dos nuevos conceptos:

1. Forward propagation: Se trata del proceso de propagar los datos de entrada a través de la red neuronal, desde la capa de entrada hasta la capa de salida en la que la red realiza su predicción. En este proceso los datos pasan por las capas ocultas de la red donde las neuronas de cada capa van devolviendo resultados de las activaciones que obtienen y que son utilizados por las capas posteriores. Podemos ver una descripción de este proceso en la Figura 18.

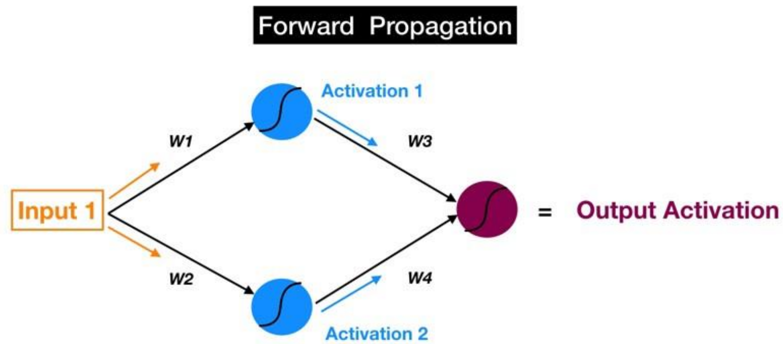



*Figura 18 Forward propagation [https://towardsdatascience.com/understanding-neural-networks-19020b758230]*

2. Backward propagation: Se trata del proceso opuesto al forward propagation, en este caso en vez de recorrer la red hacia delante la recorremos hacia atrás. Una vez obtenida la predicción a través de la última capa de la red se compara con la salida esperada y se utiliza la diferencia entre ambos valores (el error cometido) para actualizar los pesos de las conexiones que veíamos, actualizando estos valores conseguimos minimizar la función de coste haciendo que las predicciones sean cada vez más acertadas.

Lo que hacemos al propagar el error hacia atrás es ver o atribuir a cada neurona de la red la influencia que tiene de la salida que se ha obtenido, es decir dividimos el error que se ha obtenido entre todas las neuronas de la red, este proceso comienza desde la capa anterior a la de output hasta la primera capa de la red. Esto lo hacemos porque la magnitud del error cometido por una neurona específica es directamente proporcional al impacto de la salida de dicha neurona en la función de coste [24].

Podemos ver el proceso de Backward Propagation en la Figura 19.

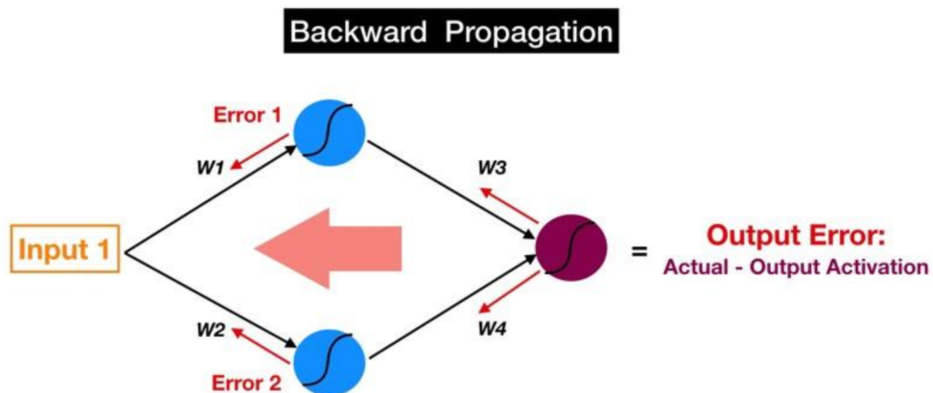



*Figura 19 Backward Propagation [https://towardsdatascience.com/understanding-neural-networks-19020b758230]*

Las redes neuronales generalmente se utilizan en diversas tareas, generalmente de una alta complejidad como visión por computador, reconocimiento de voz... Algo a tener en cuenta es que pequeños cambios en los datos de una red neuronal, pueden producir grandes cambios en la salida.

## 2.3. Speech Recognition

En esta sección vamos a tratar la rama de la IA relacionada con el reconocimiento del habla, ya habíamos hablado algo de ella en la sección 1 del documento, no obstante, ahora entraremos más en detalle y explicaremos algunas de las redes neuronales típicas utilizadas para realizar estas tareas.

Speech Recognition es una de las ramas de la IA que tiene como objetivo dotar a las computadoras de la capacidad de reconocer el lenguaje natural, para ello este tipo de sistemas procesan las señales de voz emitidas por los seres humano y reconocen y extraen la información contenida en las mismas convirtiéndola a texto (Speech to Text) o clasificándola.

Distinguimos dos tipos de aprendizaje de aprendizaje para estos sistemas:

1. Aprendizaje deductivo: En este tipo de aprendizaje los conocimientos de un experto humano son transmitidos al sistema, los sistemas expertos o los sistemas basados en el conocimiento suelen ser un ejemplo claro.

2. Aprendizaje inductivo: En este tipo de aprendizaje el sistema debe extraer automáticamente los conocimientos necesarios a partir de los ejemplos que tiene disponibles sobre la tarea que debe realizar.

Algunos de los métodos más utilizados para estas tareas son:

- Hidden Markov Model (HMM): Muchos de los sistemas actuales están basados en este tipo de modelos, se trata de modelos estadísticos cuya salida es un conjunto de símbolos. Se fundamentan en la idea de que una señal de voz puede verse como una señal estacionaria que puede ser dividida en varias partes a lo largo del tiempo. Se han vuelto muy populares en los últimos años pues pueden ser entrenados automáticamente de una forma sencilla y factible desde el punto de vista computacional. Conseguimos obtener como salida una secuencia de palabras o fonemas concatenando varios modelos de Markov que han sido entrenados para palabras o fonemas por separado.

- Redes neuronales: Se ha comprobado que las redes neuronales tradicionales tienen un buen rendimiento para detectar palabras aisladas o fonemas, es decir cosas que requieren de poco contexto, sin embargo, para tareas donde es necesario un mayor contexto o un reconocimiento del habla continuo no tienen muy buen rendimiento.

- Sistemas de reconocimiento del habla end-to-end: Este tipo de sistemas tienen por objetivo simplificar los sistemas de aprendizaje automático abstrayendo al usuario de todos los pasos intermedios que hay desde que este introduce una muestra de audio hasta que obtiene la predicción o transcripción de la esta. Generalmente este tipo de sistemas se utilizan cuando se tiene un conjunto de datos suficientemente grande como vemos en la Figura 20.

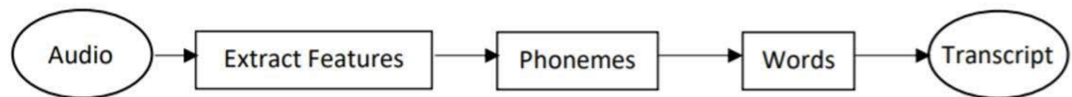


*Figura 20 Sistemas de aprendizaje tradicionales vs end-to-end [https://harangdev.github.io/deep-learning/structuring-machine-learning-projects/21/]*

Algunas de las ventajas de este tipo de sistemas son:

- Permitimos a los datos "hablar" por si mismos, es decir no forzamos a nuestros algoritmos a aprender representaciones de los datos, sino que dejamos que aprendan la representación que más les convenga.

- No es necesario diseñar las características de los datos a mano, por lo que perdemos menos tiempo en diseñar.

- Deep Neural Networks (DNN): Se trata de redes neuronales pero que cuentan con una mayor cantidad de capas y unidades ocultas entre la capa de input y la capa de output que las redes neuronales tradicionales. En Speech Recognition los dos tipos de redes más utilizadas son:

  - Convolutional Neural Networks (CNN): Generalmente este tipo de redes neuronales son utilizadas para problemas de visión por computador, no obstante se ha demostrado que en muchas ocasiones también funcionan bien para resolver problemas de Speech Recognition [25].

    La idea básica de este tipo de redes neuronales es aplicar una operación de convolución sobre los datos de entrada de la red (Figura 21) que tendrán una forma matricial con unas dimensiones: (número de imágenes) x (anchura de la imagen) x (altura de la imagen) x (profundidad de la imagen). Sobre cada una de las imágenes del conjunto se aplica la operación de convolución utilizando una serie de filtros o kernels en cada capa, los valores de estos filtros se irán modificando con el paso de las iteraciones durante el proceso de aprendizaje en la fase de Backward Propagation, es decir, los parámetros que se ajustarán durante la fase de entrenamiento serán los filtros de la red.

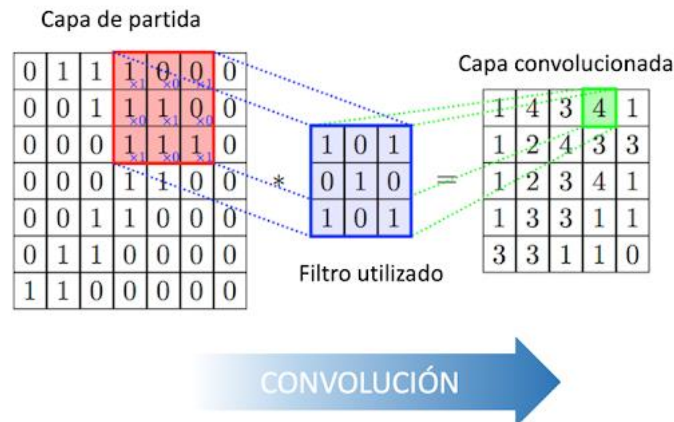



*Figura 21 Operación de convolución [http://www.diegocalvo.es/red-neuronal-convolucional/]*

Creemos conveniente explicar un par de conceptos asociados a la operación de convolución:

1. Padding: Uno de los grandes problemas de aplicar convolución es que se pierde información pues reduce el tamaño de los datos, en este contexto se introduce la operación padding que permite añadir filas y columnas recuperando el tamaño original de los datos.

2. Stride: Se trata de una operación que permite saltar filas y columnas en el proceso de convolución que se aplica sobre una imagen.

Teniendo en cuenta estos dos términos diferenciamos dos tipos de operaciones de convolución:

1. Valid convolution: Son aquellas operaciones de convolución que no utilizan padding por lo que el tamaño de la imagen obtenido después de realizar convolución es menor que la imagen original.

2. Same convolution: Son aquellas operaciones de convolución que si utilizan padding por lo que el tamaño de la imagen obtenido después de realizar convolución es igual que la imagen original.

3. Strided convolution: Son aquellas operaciones de convolución que no utilizan stride para saltar filas y columnas.

Dentro de las redes neuronales convolucionales distinguimos 3 tipos de capas que conviene, pues las mencionaremos en alguna ocasión a lo largo de la memoria:

1. Capa de convolución: Este tipo de capas son las que hemos visto hasta ahora, aplican un filtro de convolución que desplazan por toda la matriz de los datos.

2. Capas de Pooling: Este tipo de capas permiten reducir el tamaño de la representación de los datos, en vez de desplazar el filtro por la matriz de los datos y aplicar una operación de multiplicación, en cada sección

se aplica una operación de máximo, mínimo o media, generalmente esto permite que las características detectadas sean mucho más robustas.

3. Capas Fully-Conneted: Este tipo de capas es similar a la de las redes neuronales tradicionales, donde cada una de las neuronas de una capa está conectada a todas las neuronas de la capa siguiente.

Las redes neuronales convolucionales mejoran en eficiencia a las tradicionales pues permiten reducir el número de parámetros con los que trabaja la red durante el entrenamiento, esto es debido a dos razones:

1. Uso compartido de parámetros: Se basa en la idea de que aplicar un filtro a una zona de la matriz de los datos es igual de útil que aplicarlo en otra zona, pues existen zonas muy similares entre sí, por tanto los filtros tienden a compartir parámetros.

2. Conexiones dispersas: Se basa en la idea de que en cada capa cada valor de salida solo depende una un pequeño conjunto de los datos de entrada y no de todos ellos, es decir, que un valor obtenido a través de la operación de convolución solo depende del conjunto de número sobre los que se aplicó el filtro.

La idea de aplicar redes convolucionales para Speech Recognition reside en extraer las características del audio en crudo o bruto y tratar dichas características de forma matricial como si se tratasen de imágenes, como se observa en la Figura 22.

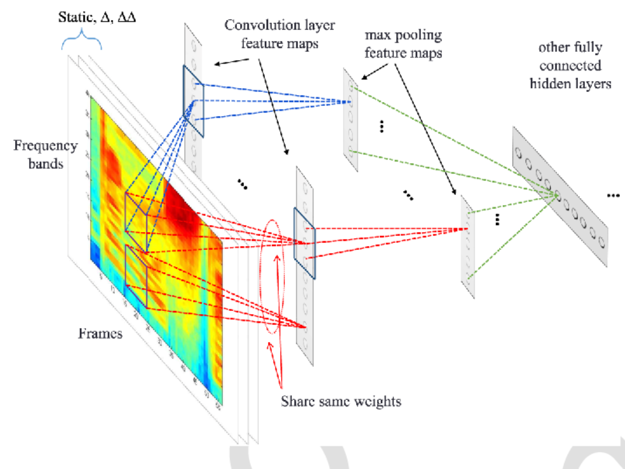



*Figura 22 Redes neuronales convolucionales para Speech Recognition [https://d3i71xaburhd42.cloudfront.net/86efe7769f2b8a0e15ca213ab09881e6705caeb0/5-Figure3-1.png]*

- Recurrent Neural Networks (RNN): Este tipo de redes neuronales buscan resolver problemas en los que las entras y/o las salidas que debe manejar la red son secuenciales, por ejemplo en Speech Recognition la entrada es una secuencia de audio y la salida es una secuencia de texto transcrita a partir del audio, en problemas de music generation la entrada es un numero o un identificador del tipo de música mientras que la salida es una secuencia de audio.

  Mejoran a las redes neuronales tradicionales, pues permiten que las entradas y salidas de la red sean de diferente tamaño, además al igual que en el caso de las convolucionales, este tipo de redes permiten compartir características aprendidas, también resultan una mejor representación del problema a tratar lo que ayuda a reducir el número de parámetros necesarios.

  Las RNN añaden bucles de retroalimentación dentro de una misma capa de neuronas, de manera que cada neurona de una capa trabaja con información previa de las anteriores neuronas, es decir pasamos de tener la estructura clásica de la red neuronal donde la salida de cada neurona de una capa alimenta a la siguiente capa a tener una nueva estructura donde la salida de cada neurona además de alimentar a la siguiente capa alimenta a la neurona siguiente de su misma capa. Una manera más sencilla de entender lo que hemos explicado es mediante la Figura 23, a la izquierda podemos ver una RNN de dos capas ocultas, mientras que a la derecha vemos la misma red desenrollada.

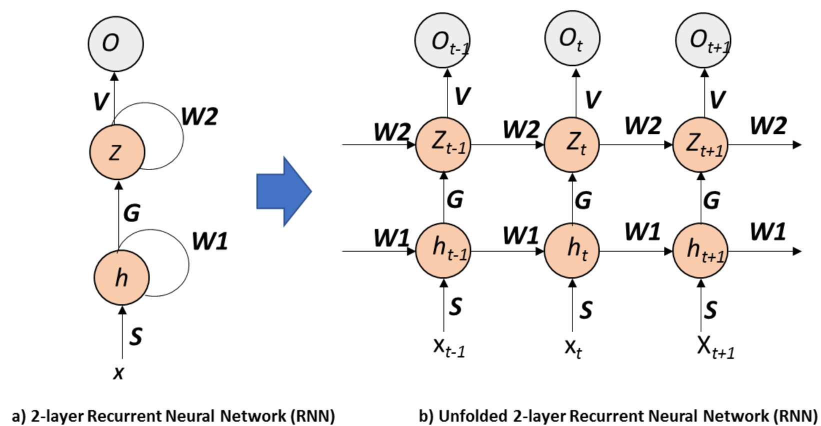



*Figura 23 Ejemplo de RNN [https://d3i71xaburhd42.cloudfront.net/86efe7769f2b8a0e15ca213ab09881e6705caeb0/5-Figure3-1.png]*

  Esta retroalimentación con la que cuentan las RNN permite mantener información del contexto de la actividad que se esté tratando durante periodos prolongados de tiempo les permite tener "memoria", esto resulta ideal en tareas de Speech Recognition donde a la hora de transcribir texto resulta imprescindible tener un contexto previo de lo que se ha transcrito.

  Además puede darse el caso en el que no solo necesitemos un contexto previo sino que también necesitemos información posterior a la que estamos tratando, en este contexto surgen las RNN bidireccionales que tratan de solventar este problema. Por tanto, además del flujo de información hacia adelante (que va de

una neurona a la siguiente), se añade un nuevo flujo de información hacia atrás, esto permite que en todo momento una neurona pueda trabajar con información pasada, presente y futura. Mostramos un ejemplo gráfico de esta arquitectura en la Figura 24.

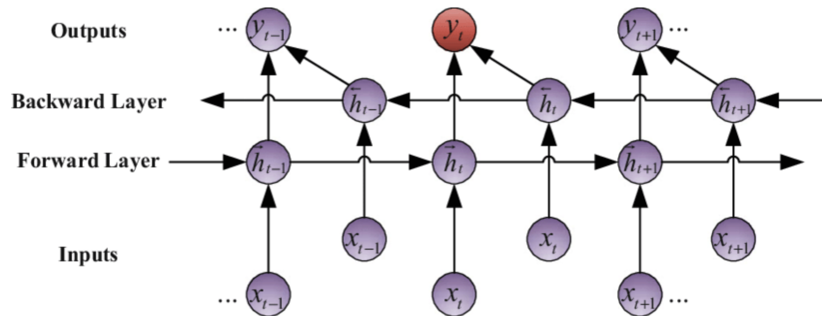



*Figura 24 RNN bidireccional [Redes neuronales convolucionales para Speech Recognition [https://d3i71xaburhd42.cloudfront.net/86efe7769f2b8a0e15ca213ab09881e6705caeb0/5-Figure3-1.png]*

Un problema evidente de las RNN bidireccionales es que requieren de la secuencia completa antes de realizar predicciones, otro problema que afecta generalmente a las DNN muy profundas es el conocido problema de "vanishing gradients" [26] (desvanecimiento de gradientes). El problema de desvanecimiento de gradientes consiste en que durante el proceso de entrenamiento los gradientes tienden a disminuir su valor y a hacerse muy pequeños y puede llegar a darse el caso en el que los gradientes valgan 0. En esta situación tendremos divisiones entre 0, lo que provocará que se produzcan errores durante la fase de entrenamiento.

Este problema es bastante común en las RNN tradicionales puesto que les es complicado mantener el contexto o las dependencias durante periodos de tiempo muy elevados, esto es debido a que en una RNN las salidas en cada paso están influenciadas por las entradas cercanas, es decir, resulta complicado que salidas en las zonas más tardías de la secuencia se vean influenciadas por zonas más cercanas. Para resolver se desarrollaron una serie de neuronas o unidades de red:

1. GRU (Gated Recurrent Unit): Este tipo de unidades cuentan con una celda de memoria que provee memoria a la red. La clave en las versiones sencillas de GRU es que poseen una puerta lógica que determina cuando se debe actualizar y cuando no la información contenida en la célula de memoria. De esta manera según avanza la secuencia la red puede decidir en cada momento si mantener cerrada la puerta lógica para evitar actualizar la información del contexto o si por el contrario la información almacenada ya no resulta necesaria y es necesario actualizar.

   La ventaja de contar con la puerta lógica es que cuando los valores que se le pasan a la función de activación de la GRU (la función sigmoide) son negativos el valor devuelto por esta será cero o muy próximo a cero y esto permite que el valor de la celda de memoria se mantenga en muchos pasos en el tiempo, ya que con valor 0 la puerta se mantendrá

cerrada y por tanto, no se actualizará la información, evitando así valores muy pequeños que hagan que se produzca el problema de vanishing gradients.

Las últimas versiones de GRU y más complejas poseen dos puertas lógicas, una que decide cuando actualizar la información pasada (Update gate), mientras que la otra decide qué porcentaje de información pasada es necesario olvidar (Reset gate) [27].

2. LSTM (Long Short Term Memory): Este tipo de unidad es una versión algo más compleja que la GRU, en vez de contar con dos puertas lógicas, cuenta con 3 puertas que vamos a explicar a continuación. Al igual que antes contamos con una celda de memoria que habitualmente se le llama estado de la celda.

   En primer lugar, en estas unidades nos encontramos con la puerta que determina que información previa de las predicciones de la red o hidden state es necesario mantener (Forget gate), para ello se combina la información previa con los datos de entrada de la neurona y se pasa a través de una función sigmoide esta función, devuelve valores entre 0 y 1 que determinan que es necesario mantener y que no.

   En segundo lugar, en estas unidades nos encontramos con la puerta que permite actualizar la información contenida en la célula de memoria (Input gate), esta puerta recibe como entrada el hidden state junto con los datos de entrada actuales y por un lado mediante la función sigmoide los combina y decide qué información es más o menos importante actualizar (son valores entre 0 y 1), por otro lado mediante la función tangente hiperbólica transforma la información en valores entre el rango [-1,1] lo que permite regularizar la red, por último multiplica la salida de ambas funciones para obtener la información a actualizar del estado de la celda.

   La salida de estas dos primeras puertas se combina para actualizar el estado anterior o la memoria de la celda, esto devuelve el nuevo estado de la celda.

   Por último, en la última puerta se decide la información del hidden state que se utilizará posteriormente en predicciones. Esta puerta recibe los datos de entrada actuales y el hidden state, los combina e introduce en una función sigmoide, la salida de esta función se multiplica por el resultado de aplicar la función tangente hiperbólica al nuevo estado de la celda y con esto se decide qué información debe mantenerse del hidden state.

En la Figura 25 podemos ver una descripción gráfica de las tres unidades que hemos explicado.

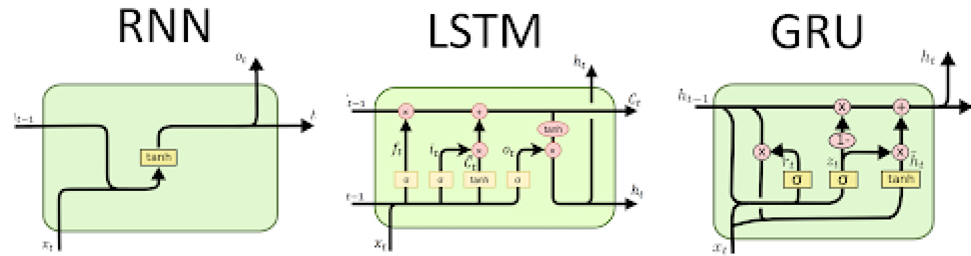



*Figura 25 Los tres tipos de unidades de las RNN [http://dprogrammer.org/rnn-lstm-gru]*

Hemos visto los métodos más comunes para la resolución de problemas relacionados con Speech Recognition, vamos a comentar los 4 elementos que suelen utilizarse para llevar a cabo estas tareas:

1. Modelo acústico: Este modelo representa la relación entre las señales de audio y los fonemas o las unidades básicas del lenguaje considerado, dicho de otra forma es un conjunto de representaciones estadísticas de los distintos sonidos que componen una palabra (HMM), cada fonema tiene asociado una representación estadística [28]. Un modelo acústico se obtiene como resultado de realizar un entrenamiento con uno de los métodos que acabamos de ver utilizando como conjunto de datos una serie de audios junto con sus transcripciones (las salidas esperadas).

2. Alfabeto: Generalmente se trata de un fichero que contiene todos los caracteres permitidos en el lenguaje con el que se está tratando. Normalmente se añaden las letras del alfabeto considerado pero también se admiten otros símbolos como: -, ‘, “...

3. Diccionario de fonemas o léxico: Generalmente se trata de un fichero que contiene todas las palabras permitidas para un lenguaje y por cada palabra el conjunto de fonemas que la constituye, normalmente se utilizan junto con modelos acústicos basados en léxico, de esta manera se limita el número de palabras permitidas en un lenguaje, evitando así que durante las transcripciones aparezcan palabras malformadas que no estén contempladas en el lenguaje, además facilita la tarea del sistema a la hora de reconocer palabras.

4. Modelo del lenguaje: Resulta un elemento clave en los sistemas de reconocimiento del habla más modernos. Un modelo del lenguaje nos permite obtener la probabilidad de que después de una secuencia de palabras venga otra, esto a efectos prácticos nos permite que si por ejemplo tenemos dos frases que suenan muy similar, el modelo de lenguaje nos da un contexto para seleccionar una u otra. Dicho de una manera formal el LM, estima la probabilidad de que ocurra una frase concreta.

   Para generar un modelo del lenguaje lo primero que se necesitará será un gran conjunto de datos de un idioma concreto llamado corpus, generalmente se trata de un gran conjunto de textos sobre un idioma y una temática concreta, aunque como veremos más adelante en este proyecto se ha utilizado un corpus con textos bastante variados todos ellos seleccionados de página Wikipedia.

Los modelos del lenguaje son N-gramas, un N-grama es una subsecuencia de una secuencia de N elementos consecutivos, en procesamiento del lenguaje natural (PNL) se construyen N-gramas utilizando como bases fonemas, sílabas o letras. Es decir si tomamos letras como base, por ejemplo en la frase: “Platero y yo” los posibles trigramas (3-gramas) serían: "Pla", "lat", "ate", "ter", "ero", "ro ", "o y", " y ", "y" y finalmente " yo", en definitiva combinaciones de 3 letras (para este caso).

Los modelos N-grama son en definitiva, modelos probabilísticos que intentan predecir con una cierta probabilidad cuál será el siguiente elemento dada una secuencia de n-1 elementos. En el reconocimiento de voz los fonemas se modelan a partir de distribuciones de N-gramas.

Cuanto mayor es la N mayor es el número de combinaciones posibles por así decirlo, por ejemplo teniendo como base el diccionario de la lengua española para un 3-grama basado en letras, se tendrían un total de $27^3$ combinaciones de 3 letras (ya que como base se estaría utilizando el diccionario de la lengua española, es decir los elementos serían letras).

Existe una relación muy cercana en entre Naïve Bayes multinomial y los modelos del lenguaje. Si consideramos cada n-grama como una posible clase podemos calcular la probabilidad de una palabra determinada dada una clase y multiplicando cada una de estas probabilidades podemos obtener la probabilidad de una frase dada una clase determinada, de manera que para cada clase (n-grama) del modelo de lenguaje calculamos la probabilidad de una serie de palabras y obtenemos la probabilidad de un conjunto de frases, de manera que la frase seleccionada será la que tenga un mayor valor de probabilidad. Un ejemplo de lo que acabamos de explicar es la Figura 26.

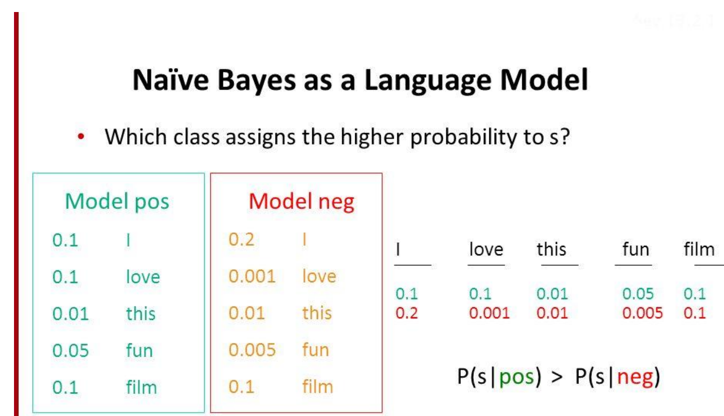



*Figura 26 Relación entre Naïve Bayes y modelos del lenguaje [https://slideplayer.com/slide/7073400/24/images/38/Na%C3%AFve+Bayes+as+a+Language+Model.jpg]*

Resumimos el proceso de Speech Recognition con la Figura 27, en esta podemos ver la integración de los elementos que hemos visto arriba para la fase de decodificación que se lleva a cabo en este tipo de sistemas.

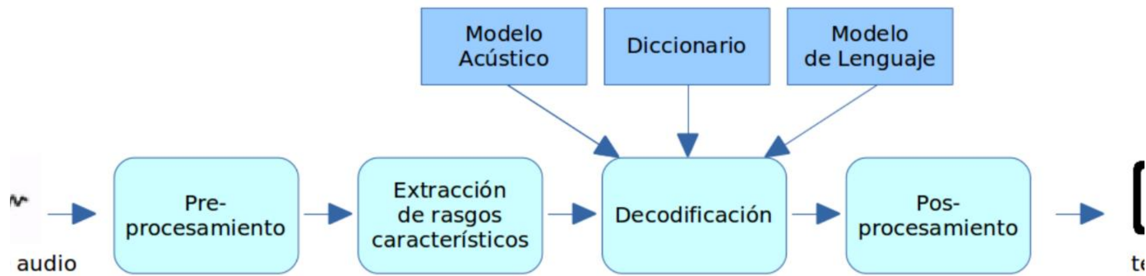



*Figura 27 Proceso de Speech Recognition [https://miro.medium.com/fit/c/1838/551/1*TOss9KmH8e8asYbJX_GW0Q.png]*

## 2.4. Evaluación de modelos

En este apartado vamos a explicar las técnicas más comunes que nos permitirán evaluar los modelos que generaremos con las herramientas que vamos a utilizar. Comenzaremos hablando del particionamiento de los datos.

Hasta ahora hemos visto que para generar un modelo es necesario un entrenamiento previo utilizando un conjunto de datos o ejemplos y un método o algoritmo, este conjunto de datos comúnmente recibe el nombre de dataset. Para realizar la fase de entrenamiento se utiliza parte de este conjunto y no todo como cabría esperar, esto es debido a que el resto es utilizado para otras dos fases que llamaremos validación y test.

La fase de validación como su nombre indica permite validar el modelo generado durante el entrenamiento, en esta etapa se utiliza un conjunto de datos diferente al que se ha utilizado durante el entrenamiento, esto entre otras cosas nos permite determinar si el modelo está generalizando bien o no, si la configuración de hiper parámetros que estamos utilizando para el entrenamiento es correcta o si por el contrario es necesario realizar algún ajuste y también permite determinar si la arquitectura que estamos utilizando funciona de manera correcta.

En cuanto a la fase de test, permite simular el funcionamiento del modelo en un entorno “real”, comprobando el rendimiento del sistema frente a ejemplos que no se han utilizado en ninguna de las dos fases anteriores.

Generalmente se utiliza un conjunto de entrenamiento mucho mayor al de validación y test puesto que el entrenamiento es la fase más importante del proceso, un conjunto de datos para entrenamiento de calidad, amplio y con gran diversidad permite obtener un buen modelo con el que generalizar correctamente para nuevos casos.

Para hacer el particionamiento del conjunto de datos distinguimos varios métodos entre los que destacan:

1. Hold-out: Consiste en dividir el dataset en conjuntos independientes de datos, normalmente cuando el dataset es pequeño se aplica la regla del 60/20/20, es decir, el 60% de los datos se utilizan para entrenamiento, un 20% de los restantes para validación y el último 20% para test, cuando el dataset es grande se pueden utilizar otras proporciones como 80/10/10 o 98/1/1... Toda la selección de los ejemplos que formarán parte de cada conjunto se hace de forma aleatoria.

Este tipo de particionamiento tiene un problema y es que puede producir conjuntos donde los ejemplos de cada clase no sean representativos, una posible solución es aplicar estratificación, es decir distribuir los ejemplos en los conjuntos de una manera similar al conjunto original. Aplicar esta técnica puede provocar que el conjunto de entrenamiento sea de menor tamaño con lo que el entrenamiento puede ser de menor calidad generando por tanto un peor modelo.

Generalmente esta técnica se aplica de forma iterativa pues tiende a ser más fiable, sin embargo, puede darse el caso en el que los diferentes conjuntos generados en varias iteraciones tengan intersección no vacía entre ellos e incluso puede haber ejemplos que no aparezcan nunca para aprender el modelo, una solución a esto es usar la técnica de validación cruzada.

2. Validación cruzada: Este método resuelve el problema de intersección no vacía de Hold-out, generalmente se aplica para generar dos particiones o bien la de entrenamiento y validación o bien la de entrenamiento y test. El método consiste en dividir el dataset en k subconjuntos aleatorios de forma que cada subconjunto tiene el mismo número de ejemplos y cada ejemplo está asignado a un solo subconjunto.

   Los k-1 primeros subconjuntos se toman como conjunto de entrenamiento, mientras que el restante se toma como conjunto de test o de validación. Al igual que en Hold-out suele aplicarse de forma iterativa, de esta manera se puede encontrar la mejor distribución del dataset en los k subconjuntos. El procedimiento que se sigue es el siguiente:

   2.1. Repetir k veces:

   2.1.1. Se escoge un subconjunto como conjunto de test y el resto como conjunto de entrenamiento.

   2.1.2. Se entrena el modelo con el conjunto de entrenamiento y se evalúa con el conjunto de test.

   2.2. De entre todos los resultados seleccionar aquel cuya evaluación sea la mejor.

Una vez explicado el particionamiento consideramos importante explicar algunas de las medidas que nos permitirán evaluar el rendimiento de un sistema:

- Valor de coste o loss: Este valor nos determina cual es el error cometido en cada iteración del proceso de entrenamiento, se obtiene evaluando la función de coste. El objetivo será que el valor vaya disminuyendo conforme pasen las iteraciones, pues nuestro objetivo es minimizar la función de coste. El valor de coste se puede calcular tanto para la fase de entrenamiento, como para la de validación o test.

- Accuracy o tasa de acierto: Se trata de una medida bastante sencilla que devuelve el porcentaje de ejemplos bien clasificados, es decir el porcentaje de ejemplos cuya salida esperada coincide con la predicción del modelo. Se obtiene al dividir el número de ejemplos bien clasificados entre el número total de ejemplos. Podemos obtener los valores de tasa de acierto tanto para la fase de entrenamiento, como para test y validación.

- Error: Se trata del porcentaje de ejemplos mal clasificados. Su valor se obtiene como 1-Accuracy.

- WER (Word Error Rate): Es una medida de rendimiento muy utilizada en los sistemas de Speech Recognition. Determina el porcentaje de palabras que han sido incorrectamente transcritas. El cálculo de la tasa de WER lo encontramos en la Figura 28.

$$WER = \frac{S + D + I}{N}$$

where...
S = number of substitutions
D = number of deletions
I = number of insertions
N = number of words in the reference

*Figura 28 Cálculo de WER [https://sonix.ai/packs/media/images/corp/articles/word-error-rate-2017-c5aba7282b39531154f5676a184c7ec4.png]*

- LER (Letter Error Rate): También es una medida de rendimiento muy utilizada en los sistemas de Speech Recognition. Determina el porcentaje de letras que han sido incorrectamente transcritas. El cálculo de LER es igual al de WER solo que se realiza a nivel de letras.

Las medidas como accuracy, loss o el error nos permiten identificar dos tipos de problemas que resultan comunes en modelos de Machine Learning y Deep Learning:

- Bias alto o under-fitting (Bajo-aprendizaje): Decimos que tenemos un problema de bias cuando nuestro modelo es demasiado sencillo y no se ajusta (o lo hace muy mal) ni a los datos de entrenamiento ni a los de validación y test, es decir generaliza en exceso. Una manera de identificar este problema es cuando tenemos un valor de loss y de error demasiado alto (por tanto un valor bajo de accuracy) en todas las fases y observamos un rendimiento pésimo del sistema. Como posibles soluciones podemos tratar de utilizar una red más grande (si estamos tratando con redes neuronales) o con una arquitectura diferente, utilizar un algoritmo de optimización diferente...

- Varianza alta o overfitting (Sobre-aprendizaje): Decimos que tenemos un problema de varianza alta cuando nuestro modelo se ajusta muy bien a los ejemplos del conjunto de entrenamiento, sin embargo lo hace muy mal para los de validación y test, es decir que el modelo generaliza muy poco y recordemos que el objetivo es que generalice bien para ejemplos nuevos con el conocimiento que ha adquirido en el entrenamiento. Una manera de identificar este problema es cuando tenemos un valor de loss y de error demasiado bajo (por tanto un valor alto de accuracy) durante el entrenamiento, mientras que en validación y test ocurre lo contrario. Como posibles soluciones podemos:

- Utilizar un conjunto de entrenamiento mayor y/o de mayor calidad.

- Aplicar técnicas de regularización: La idea básica de la regularización es reducir el valor de los coeficientes del modelo haciéndolos próximos a cero, lo que provoca que el modelo sea menos flexible y por tanto tienda menos a ajustarse, esto es debido a que cuanto menores son los valores de los parámetros del modelo, más simples resultan las hipótesis que realiza, por lo que tiende menos a sobre-aprender.

  En Machine Learning los dos tipos de regularización más habituales son la regularización l1 y l2. Ambas regularizaciones introducen un nuevo hiper parámetro λ a la hora de entrenar, sin embargo funcionan de una manera diferente, en el caso de la regularización l2 establece el porcentaje de regularización aplicar, es decir, cuánto se deben reducir los coeficientes del modelo, mientras que en el caso de l1 determina que porcentaje de coeficientes deben valer 0. En ambos casos se debe seleccionar un valor adecuado de λ pues un valor demasiado bajo puede hacer que el modelo siga sobre-aprendiendo, mientras que un valor demasiado alto puede hacer que generalice en exceso y ni siquiera se ajuste a los ejemplos de entrenamiento.

  En Deep Learning, generalmente se utiliza la técnica de dropout con las redes neuronales para aplicar un efecto de regularización al modelo. Con dropout lo que hacemos es recorrer cada capa de la red neuronal y asignar una probabilidad de eliminar un nodo de la red, además de eliminar el nodo, se eliminan todas la conexiones entrantes y salientes de este, lo que hace que se genera una red más pequeña y simple, por lo que tiene menos probabilidad de sobre-aprender. La idea de dropout es que como cualquier nodo puede desaparecer en cualquier momento, ningún nodo puede confiar completamente en una característica de entrada, ya que en cualquier momento podría desaparecer, por ello los pesos de la red son reacios a poner demasiado valor en una única característica y por tanto los valores se distribuyen o esparcen lo que genera un efecto similar a la regularización L2.

En la Figura 29 podemos ver un caso típico de underfitting (a la izquierda) donde un clasificador se ajusta incorrectamente a los datos, pues generaliza en exceso, por otro lado vemos (en el centro) el caso ideal, mientras que observamos (a la derecha) un caso donde se produce overfitting pues el clasificador se ajusta demasiado a los datos de entrenamiento y adquiere esa forma tan compleja.

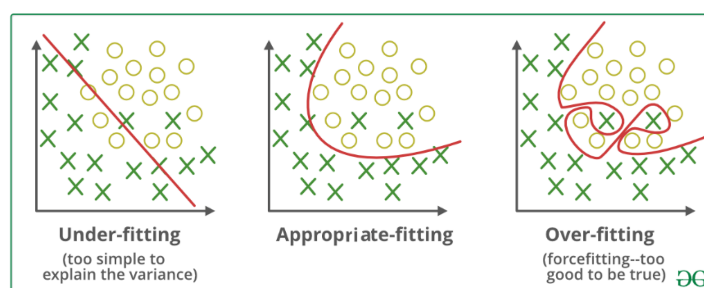



*Figura 29 Casos de underfitting, situación ideal y overfitting [https://d31dn7nfpuwjnm.cloudfront.net/images/valoraciones/0037/4082/overfitting_Underfiting.png?1586246003]*

Para terminar de hablar acerca de la evaluación de modelos debemos comentar que aunque el accuracy resulta una buena medida de rendimiento para algunos problemas aporta poca información, pues solo nos aporta una visión global del rendimiento del sistema. El accuracy no permite dar diferente peso a los ejemplos de las clases, además no tiene en cuenta el número o de ejemplos de cada clase y en problemas en los que el número de ejemplos por clase es distinto esto puede suponer un gran inconveniente, un ejemplo de esto sería un clasificador de casos de cáncer en el que solo un 0.5% de los pacientes tienen cáncer, si predecimos siempre que los pacientes no tienen cáncer la tasa de acierto es del 99.5% sin embargo el clasificador lo está haciendo realmente mal, además no estamos concretando si damos más peso a detectar casos de cáncer (aunque el número de falsos positivos sea mayor) o si por el contrario buscamos detectar con precisión casos de cáncer (aunque pasemos por alto casos positivos).

En este contexto resulta necesario añadir algunas medidas de rendimiento que permitan resolver estos problemas, estas medidas al igual que el accuracy están basadas en la matriz de confusión. La matriz de confusión es una matriz cuadrada con tantas filas y columnas como clases tiene el problema, donde cada posición $c_{i,j}$ indica el número de instancias clasificadas como $c_i$ cuando su clase real es $c_j$ , la matriz de confusión tiene la apariencia de la Figura 30.

| | | **Predicción** | |
|---|---|---|---|
| | | **Positivos** | **Negativos** |
| **Observación** | **Positivos** | Verdaderos Positivos (VP) | Falsos Negativos (FN) |
| | **Negativos** | Falsos Positivos (FP) | Verdaderos Negativos (VN) |

*Figura 30 Matriz de confusión [https://rpubs.com/chzelada/275494]*

Las nuevas medidas de rendimiento son las siguientes:

- Precisión: Es la proporción de ejemplos que han sido clasificados como clase positiva y realmente pertenecen a la clase positiva. Se calcula como. VP/(VP+FP).

- Recall o True Positive Rate (TPR): Es la proporción de ejemplos de la clase positiva que han sido clasificados correcta. Se calcula como: VP/(VP+FN).

- Especificidad o True Negative Rate (TNR): Es la proporción de ejemplos de la clase negativa clasificados correctamente. Se calcula como: VN/(VN+FP).

- F-Score: Es una medida balanceada entre la precisión y el recall, se calcula mediante la media armónica de las dos: 2*(precisión*recall)/(precisión+recall).

## 2.5. Herramientas que vamos a utilizar

Para el desarrollo de este proyecto vamos a utilizar una serie de herramientas y en esta sección las mencionaremos y explicaremos de forma breve. Estas herramientas son:

- Python 3.6: Para la mayoría de herramientas que hemos testeado ha sido imprescindible el uso del lenguaje de programación Python a partir de su versión 3, pues hemos tenido que escribir código en este lenguaje, tratar con diferentes librerías y manejar entornos virtuales entre otras cosas.

- C++: Al igual que Python hemos necesitado el lenguaje C++, para la instalación y manejo de ciertas herramientas y librerías.

- Editores de texto: Se ha necesitado de editores de texto como: Kate, Thony, Gedit... para poder escribir y modificar código, modificar y crear ficheros de texto plano...

- Docker: Docker es una herramienta de código libre que permite la ejecución de aplicaciones dentro de contenedores software, de manera que estos se ejecutan de forma aislada, segura e independiente algo similar a lo que hacen las máquinas virtuales, solo que Docker evita la sobrecarga de iniciar y mantener las máquinas, además los contenedores de Docker resultan más portables y eficientes pues virtualizan el sistema operativo en vez del hardware.

- Google Colab: Se trata de un entorno de máquinas virtuales en la nube basado en Jupyter notebook, en definitiva Python. Permite el uso de recursos tanto CPU como GPU de máquinas remotas de Google para la ejecución de notebooks de Python. Las máquinas funcionan durante un periodo máximo de 12 horas tras el cual estas se reinician y eliminan todo su contenido.

- Google Drive: Se trata del servicio de almacenamiento en la nube de Google, como explicaremos más adelante fue necesario para trabajar con Google Colab.

- Audacity: Se trata de un programa que provee múltiples herramientas para la grabación y edición de audio, además permite utilizar diferentes códecs de audio e incluso guardar los datos en formato WAV.

## 2.6. Estado del arte en Speech Recognition

En esta sección estudiamos varias opciones disponibles en el mercado a la hora de tratar con problemas de Speech Recognition. Para el estudio de las diferentes alternativas exigimos 2 requisitos mínimos, en primer lugar las herramientas, librerías o sistemas que vayamos a utilizar deben ser open source y totalmente gratuitas, en segundo lugar deben trabajar de manera offline todo el tiempo pues como ya hemos comentado el sistema que deseamos construir trabajará con datos sensibles y deseamos evitar filtraciones de datos a toda costa.

Entre las diferentes alternativas destacamos 4:

1. Rhasspy: Rhasspy es una herramienta open source y que funciona completamente offline, está pensada especialmente para construir asistentes de voz virtuales, pues trabaja bien con software externo de automatización como Home Assistant o Node-Red Flow. Además Rhasspy soporta múltiples lenguajes y funciona en muchos tipos de hardware, destacando la Raspberry desde el modelo “Zero”, esto resulta ideal pues en

primer lugar podría abaratar los costes de hardware del sistema planteado y en segundo lugar nos facilitaría las pruebas en un entorno real pues solo habría que introducir en quirófano un pequeño dispositivo como es una Raspberry.

Rhasspy integra múltiples herramientas y librerías que no solo permiten realizar un reconocimiento del habla, sino también transformar texto a voz y configurar trigger words o palabras para despertar al asistente. Por lo que a priori parece cumplir casi todos los requisitos que nos habíamos planteado.

2. Wav2letter++: Wav2letter++ es una herramienta open source completamente escrita en C++ y desarrollada por el equipo de investigadores de Facebook AI Research, que trata de facilitar la creación de modelos end-to-end. Wav2letter es un motor relativamente nuevo que presume de un mejor y mayor rendimiento frente a las alternativas de sus competidores.

A priori resulta más complejo que Rhasspy pues a diferencia de este, permite definir arquitecturas de red para los modelos acústicos que genera, permite personalizar los hiper parámetros del entrenamiento y además requiere de un modelo de lenguaje para la decodificación de audio. Algo que hemos comprobado es que requiere de una gran cantidad de dependencias (librerías y aplicaciones software de terceros) y que al igual que Rhasspy se ejecuta en contenedores Docker.

3. DeepSpeech: Es una herramienta open source completamente offline y escrita en Python su arquitectura es similar a la descrita en el paper del sistema de Speech Recognition de Baidu [29], por lo que utiliza redes RNN, lo que en principio debería buenos resultados, ya que como hemos comentado este tipo de redes neuronales tienden a funcionar muy bien en problemas de reconocimiento del habla.

   Al igual que Wav2letter++ requiere de una personalización de los parámetros para el entrenamiento y de un modelo de lenguaje entre otros elementos, sin embargo no requiere de una definición de arquitectura concreta pues como comentamos ya está implementada. Además proveen de modelos pre-entrenados en inglés por lo que supone una ventaja a la hora de tratar con este idioma. Observamos que al igual que Wav2letter++ da opción a realizar ejecuciones con soporte de GPU lo que permitirá acelerar la fase de entrenamiento. A diferencia de los dos sistemas anteriores no se ejecuta en un contenedor de Docker sino que requiere de un entorno virtual de Python.

4. Frameworks en Python: Hemos valorado como última opción utilizar frameworks de Python como: PyTorch, Keras, Tensorflow... para construir redes neuronales que nos permitan realizar un reconocimiento del habla, sin embargo, consideramos esto como una última alternativa pues implica la construcción del sistema desde 0, algo que se sale del objetivo inicial además requiere de una gran documentación previa para saber utilizar el framework seleccionado.

   A priori frameworks como Keras permiten construir RNN e incluso descargar modelos pre-entrenados, sin embargo con un estudio superficial no sabemos si permiten el uso de modelos del lenguaje que es algo que mejorar sustancialmente el funcionamiento de este tipo de sistemas.

# 3. Rhasspy

En este apartado del documento, comenzamos hablando de la primera herramienta que hemos utilizado para la construcción del sistema. Por ello, esta sección estructuramos de la siguiente manera:

1. Comenzamos dando una pequeña introducción sobre la herramienta en la que veremos qué es y por qué destaca.

2. Estudiamos el funcionamiento de Rhasspy, los componentes que lo forman y cómo interactúan para poder generar un asistente de voz.

3. Vemos de una forma simplificada el proceso de instalación o puesta en marcha de la herramienta.

4. Realizamos pruebas con Rhasspy para probar su funcionamiento con el idioma español, probamos sus sistemas de trigger word y desarrollamos un servidor encargado de manejar las peticiones del usuario.

5. Hablamos de los problemas que hemos tenido durante las fases de instalación y pruebas y de las soluciones que hemos encontrado.

## 3.1. Introducción a Rhasspy

Rhasspy es una herramienta open source que trabaja completamente offline y está escrita en Python. Este motor funciona en varios idiomas entre los que están incluidos inglés y español, para ello utiliza modelos acústicos pre-entrenados, modelos del lenguaje ya generados, así como diccionarios de palabras y léxico predeterminados para cada idioma.

Esta herramienta está optimizada y pensada para detectar comandos predefinidos por el usuario en la gramática del sistema, para comunicarse con servicios externos vía sockets web y para trabajar con software externo de automatización como Home Assistant o Node-Red Flow por lo que resulta idóneo para para construir asistentes de voz.

Algo que diferencia a Rhasspy del resto de competidores es que integra múltiples herramientas y librerías ya desarrolladas que no solo permiten realizar un reconocimiento del habla, sino también configurar trigger words o palabras para despertar al asistente, con la posibilidad además de funcionar en un hardware tan sencillo como el de una Raspberry Pi.

## 3.2. Funcionamiento de Rhasspy

Para explicar el funcionamiento de Rhasspy es necesario que veamos el motor como una herramienta que se encarga de integrar otras en una sola, con el objetivo final de crear un sistema que permita facilitar la tarea de crear asistentes de voz a los usuarios. Por ello diferenciamos varias partes dentro del sistema:

1. Sistema para recoger el audio externo: Rhasspy provee de varios sistemas para recoger el audio del exterior y procesarlo, en este caso permite que el audio venga de un micrófono local, es decir directamente enchufado a la máquina que ejecuta Rhasspy o mediante un stream de audio remoto. Entre los sistemas que incluye para llevar a cabo esta tarea destacan:

    - ALSA: Es una parte del núcleo de los sistemas operativos GNU/Linux, su objetivo es proveer de una configuración automática a las tarjetas de sonido y manejar varios dispositivos de sonido dentro de un mismo sistema. ALSA provee audio y una MIDI para el ajuste de los parámetros de audio del sistema, destaca por las siguientes razones:

        - Soporta una gran cantidad de interfaces de audio desde tarjetas de audio tradicionales hasta tarjetas de audio profesionales.

        - Provee de controladores de sonido totalmente modularizados.

        - Provee una librería llamada alsa-libs que simplifica la programación de aplicaciones y otorga una funcionalidad de alto nivel.

    - PyAudio: Se trata del sistema que viene activado por defecto, es bastante flexible pues soporta el uso de ALSA y PulseAudio, dos sistemas de audio muy comunes en sistemas operativos GNU/Linux que permiten la reproducción y recogida de audio en la máquina en la que se instalan.

        Pyaudio es una herramienta que provee de bindings de Python a PortAudio. PortAudio es una API open source multi-plataforma que permite escribir programas en C y C++ para controlar el I/O de audio en un computador. Los bindings que provee Pyudio permiten utilizar funciones y librerías de C++ y C en Python como si fuesen propias del lenguaje, permitiendo así utilizar la API de PortAudio desde un entorno con Python. Pyaudio utiliza Python para reproducir o grabar audio en muchas plataformas diferentes.

    - HTTP Stream: Este sistema se utiliza cuando se desea introducir streams de audio remoto, permite introducir audio de 16 bits con una frecuencia de 16kHz, el audio es troceado en trozos sin intersección entre ellos y es transmitido vía HTTP 1.1 utilizando el protocolo POST. Cada segmento o “trozo” de audio recibido se recibe de manera independiente precedido de su tamaño en bytes, la transmisión acaba cuando se recibe un segmento con una longitud 0 [30].

Este sistema también ofrece la posibilidad en Rhasspy de recoger varios comandos o recoger solo uno del audio que recibe.

2. Sistema para reproducir audio: Rhasspy necesita de un sistema para sacar audio pues como veremos hay varias situaciones en las que el sistema debe reproducir algún sonido. De los sistemas que tiene integrados destacamos el sistema ALSA que ya hemos explicado anteriormente.

3. Sistema de Wake Word o Trigger Word: Este tipo de sistemas permite despertar el asistente mediante una palabra o palabras clave, algo similar a lo que hacen algunos productos que encontramos en el mercado, como por ejemplo Alexa [31]. Rhasspy provee 4 opciones, de las que comentaremos las tres que nos han parecido más destacables y que probaremos:

    - Porcupine: Sobre el papel, este sistema ofrece el mejor rendimiento en cuanto a tasa de detección de la wake word y al número de falsos positivos. Este sistema utiliza DNNs que han sido pre-entrenadas en entornos reales, destaca por funcionar en una gran variedad de plataformas y sistemas operativos, por su alta tasa de acierto y porque provee de bindings para múltiples lenguajes de programación como: Python, Java, JavaScript...

      Su integración en Rhasspy solo permite utilizar la palabra predefinida “porcupine” o una de las disponibles en el repositorio para despertar el asistente, esto nos ha resultado un inconveniente ya que, aunque Rhasspy da opción a utilizar una palabra personalizada, requiere de entrenar el modelo cada 30 días, pues pasado este tiempo la palabra se desactiva.

    - Snowboy: Este sistema, sobre el papel, rinde algo peor que Porcupine pero permite el uso de múltiples wake words, además otorga la posibilidad de utilizar una palabra personalizada, la desventaja de esto es que es necesario acceder a la página oficial de Snowboy para entrenar un modelo con la palabra personalizada deseada (una vez hecho no es necesario ningún acceso adicional a internet), además el modelo que se genera solo se entrena con las muestras de audio que se le proveen.

      Este sistema presume de tener unos consumos bajos en CPU y RAM en las distintas plataformas en las que funciona sin sacrificar en rendimiento como muestra la Tabla 1.

| Name | CPU | CPU Usage | RAM Usage |
|---|---|---|---|
| RPi 1 | single-core 700MHz ARMv6 | <10% | • Python: < 15MB<br>• C: < 2MB |
| RPi 2 | quad-core 900MHz ARMv7 | <5% | |
| RPi 3 | quad-core 1.2GHz ARMv8 | <5% | |
| RPi Zero | single-core 1GHz ARMv6 | <5% | |
| Macbooks | Intel Core i3/5/7 | <1% | |

*Tabla 1 Consumos de CPU y RAM Snowboy [http://docs.kitt.ai/snowboy/]*

Snowboy se basa en el uso de DNNs para la detección de las trigger words, más concretamente utiliza RNNs [32], por lo que en principio debería tener un buen rendimiento en la práctica.

- Pocketsphinx: Este es el sistema más flexible de los 3 pues permite definir cualquier palabra que se desee para despertar el asistente, las palabras deben tener de 3 a 4 sílabas. Sobre el papel tiene el peor rendimiento de los tres en términos de falsos positivos y negativos, no obstante Rhasspy deja definir un umbral con el que ser más flexible a la hora de detectar la palabra.

  Este sistema está construido sobre DNNs, aunque no hay mucha información acerca de su arquitectura.

4. Sistema escuchador de comandos: A la hora de recoger audio veíamos como Rhasspy cuenta con unas varias opciones que permiten la entrada de audio en el motor, sin embargo, también requiere de un sistema que grabe la voz detectando de forma automática cuando una persona empieza y termina de hablar. Para ello Rhasspy da varias opciones entre las que destacaremos dos:

   - WebRTCVAD: Este sistema se basa en el proyecto open source webRTC desarrollado por empresas como Google, Apple y Microsoft. WebRTC dota de comunicación en tiempo real a navegadores web y aplicaciones móviles mediante el envío en tiempo real de audio, video y datos genéricos entre pares, para ello provee una API que facilita la programación de aplicaciones y navegadores web que utilizan este sistema.

     La idea de WebRTC es conseguir aplicaciones para la comunicación en tiempo real que sean de calidad y que sean desarrolladas para navegadores web, plataformas móviles y dispositivos IoT, permitiendo la comunicación entre ellas utilizando protocolos comunes [33].

     La versión de WebRTC que integra Rhasspy es compatible con Python y utiliza un VAD (Voice Activity Detection) para clasificar el audio y detectar cuando hay y cuando no sonido. VAD es una técnica utilizada en Speech processing (Procesamiento del habla) que permite detectar la presencia o ausencia de habla humana en un audio, generalmente se utiliza en aplicaciones de Speech Recognition. Entre otras cosas evita una codificación y transmisión de audio cuando hay silencios.

   - Comandos: Rhasspy permite la opción de llamar mediante comandos a un programa externo instalado en la máquina que escuche el audio y devuelve un fichero WAV con el audio grabado.

5. Sistema Speech to Text: Este tipo de sistemas se encargan de transformar el audio recogido y grabado a texto o dicho de otra manera se encargan de transcribir el audio a texto. En el caso de Rhasspy además de transcribir el texto lo transforma a un formato JSON para su procesamiento posterior. Rhasspy trae consigo varios sistemas de los que estudiamos los únicos tres que soportan idioma español e inglés y que funcionan completamente offline:

- Pocketsphinx: Es el mismo sistema que hemos comentado para la wake word solo que en este caso se utiliza para realizar la transcripción del audio que es introducido en el sistema. Este sistema en su transcripción se restringe al modelo de lenguaje, modelo acústico y diccionario que se han utilizado durante el entrenamiento de Rhasspy, sin embargo existe la posibilidad de hacer una transcripción abierta utilizando un modelo de lenguaje más genérico para cada idioma.

- Comandos: Al igual que en el sistema para escuchar comandos, existe la posibilidad de llamar a un programa externo, que realice la transcripción del audio y la envíe de vuelta al motor.

- Servidor remoto HTTP: Como última opción está la posibilidad de utilizar un servidor remoto en el que realizar la transformación del fichero WAV a texto. Para ello los datos del audio son enviados vía HTTP con POST al servidor externo y se espera a que este de una contestación.

6. Sistema de reconocimiento de intents: Este tipo de sistemas se encargan de tomar el comando de voz transcrito e identificar el intent al que pertenece. Un intent podemos definirlo como un identificador para un conjunto de comandos que tienen por objeto realizar la misma acción, los veremos más adelante. Aquí las 6 mejores opciones que encontramos son:

   - Fsticuffs: Es menos flexible que otros reconocedores, pero es mucho rápido a la hora de entrenar y de reconocer intents. Según la documentación [34], resulta la mejor opción si Rhasspy se va a utilizar solo para reconocimiento vía audio de comandos y no mediante un chat de texto.

     Existe la opción de que fsticuffs sea menos estricto a la hora de hacer coincidir texto. Esto se hace que la función fuzzy, que, de forma predeterminada, está habilitada. Sin embargo, las palabras de los comandos con los que se entrena Rhasspy deben aparecer en el orden correcto, pero palabras adicionales detectadas durante la transcripción no causan un error en el reconocimiento del intent.

   - Fuzzywuzzy: Utiliza la distancia Levenshtein [35] (métrica de Strings, que permite calcular la diferencia entre dos secuencias de caracteres), entre el comando transcrito y cada uno de los comandos del conjunto de entrenamiento. Trabaja bien con cantidades pequeñas de frases o comandos.

   - Mycroft Adapt: Trabaja bien con una cantidad mediana de frases, puede reconocer comandos que no están en el fichero de entrenamiento. Se trata de un sistema que utiliza la librería open source Adapt. Adapt es un parser de intents, es decir es una librería que se encarga de convertir el lenguaje natural en un conjunto de estructuras de datos como JSON. Adapt es una librería multiplataforma, pues está escrita en Python.

   - Flair: Trabaja bien con grandes cantidades de comandos, puede reconocer palabras y frases que no están en el conjunto de entrenamiento. Una opción

interesante de Flair es que permite generar frases aleatorias (la cantidad se especifica en profile.json mediante el atributo num_samples), lo que produce modelos más precisos pero lleva un tiempo largo realizar el entrenamiento de estos. Flair es un framework de Pytorch, es multilenguaje y posee varios modelos pre-entreandos.

- Servidor remoto HTTP: Como última opción está la posibilidad de utilizar un servidor remoto en el que realizar la identificación del intent. Para ello el intent es enviado vía HTTP con POST al servidor externo y se espera a que este de una contestación.

- Comandos: Como hemos visto en otras ocasiones podemos ejecutar comandos para llamar a programas externos que en este caso nos permitirán reconocer los intents.

La Tabla 2 muestra un resumen de las distintas opciones disponibles.

| System | Ideal Sentence count | Training Speed | Recognition Speed | Flexibility |
|---|---|---|---|---|
| fsticuffs | 1M+ | very fast | very fast | ignores unknown words |
| fuzzywuzzy | 12-100 | fast | fast | fuzzy string matching |
| adapt | 100-1K | moderate | fast | ignores unknown words |
| rasaNLU | 1K-100K | very slow | moderate | handles unseen words |
| flair | 1K-100K | very slow | moderate | handles unseen words |

*Tabla 2 Opciones para intent recognition Rhasspy [https://rhasspy.readthedocs.io/en/latest/intent-recognition/]*

7. Sistema para el manejo de intents: Después de transcribir un comando e identificar el intent asociado se envía un evento en JSON a otro sistema encargado de realizar las acciones asociadas al intent. Aquí Rhasspy tiene 3 maneras de trabajar:

   - Home Assitant: Podemos manejar los eventos generados mediante sistema Home Assitant, si estamos corriendo Rhasspy en el sistema operativo Hass.io. Sin embargo no entraremos en detalles pues como explicaremos más adelante no utilizaremos Hass.io como sistema operativo.

   - Servidor HTTP: Podemos manejar los eventos mediante un servidor externo que recibe mediante HTTP vía POST el evento, el servidor realizará las acciones asociadas al evento recibido y devolverá una respuesta a Rhasspy.

   - Comandos: Como en el resto de ocasiones, podemos llamar a un programa externo mediante la ejecución de una serie de comandados que nos permita manejar cada uno de los eventos recibidos.

8. Sistema Text to Speech: Rhasspy incluye este sistema como algo opcional, básicamente la función que tiene es que una vez que un comando de voz ha sido transcrito y su intent ha sido manejado, es común devolver una respuesta auditiva al usuario (por esta razón

Rhasspy tiene la necesidad de un sistema que permitiera la salida de audio). Rhasspy nos brinda varios sistemas para hacer esto, pero solo dos de ellos funcionan para los idiomas inglés y español y de manera offline:

- eSpeak: eSpeak es un software sintetizador del habla que está disponible para varios idiomas, además es open source y multiplataforma. Entre otras funcionalidades permite: utilizar diferentes voces alterando las características de estas, traducir textos a un código de fonemas y almacenar el texto sintetizado en un fichero WAV. La voz sintetizada tiende a ser más robótica que en el resto de casos.

- PicoTTS: es una herramienta open source que permite sintetizar voz en múltiples idiomas. El sonido de la voz que produce suena más natural que en el caso de eSpeak.

Vistas las diferentes partes de Rhasspy, podemos resumir el proceso que lleva a cabo el motor cuando recibe audio del exterior de la siguiente manera:

1. Rhasspy recibe audio del exterior, mediante su sistema de wake word comprueba si el usuario ha dicho la palabra para despertar al asistente, en caso afirmativo pasaría al paso 2, si no simplemente quedaría a la espera de recibir más audio.

2. Rhasspy cambia de modo y comienza a recoger audio y grabarlo detectando cuando el usuario comienza a hablar y cuando termina.

3. Una vez grabado el audio, este es transcrito a texto y se deja en un formato JSON para su posterior procesamiento.

4. El audio transcrito es procesado en busca de coincidencias del comando de voz, si el comando coincide con alguno de los que Rhasspy ha aprendido obtiene el intent asociado y pasa al paso 5, en caso contrario devuelve un error.

5. Con el intent identificado genera un evento en formato JSON que es enviado y procesado por un sistema diferente a Rhasspy.

6. Rhasspy recibe la respuesta del programa que ha procesado el evento y reproduce una respuesta audible (de forma opcional).

Este proceso puede verse de manera más simplificada en la Figura 31.

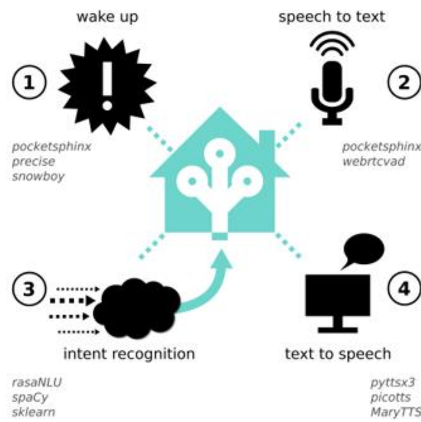



*Figura 31 Pasos Rhasspy [https://i1.wp.com/makezine.com/wp-content/uploads/2020/03/Rhasspy2.png?ssl=1]*

Debemos aclarar que todos los sistemas que integra Rhasspy son configurables y la información de configuración se almacena en un fichero llamado "profile.json", este fichero es independiente para cada idioma (profile) y tiene una estructura JSON.

Hemos comentado las diferentes partes que integra Rhasspy pasaremos a hablar de las formas que tenemos de interactuar con este. Aquí debemos de destacar las 3 más importantes:

1. Interfaz web: Rhasspy por defecto cuenta con una interfaz web que es accesible a través del navegador en la dirección 127.0.0.1 o localhost y en el puerto 12101. La interfaz permite configurar, probar y entrenar el motor de una forma sencilla e intuitiva. La interfaz se divide en varias pestañas como muestra la Figura 32, en cada una podemos configurar o probar diferentes funciones de Rhasspy. Los tres botones de la esquina superior derecha permiten:

    - Botón verde: Permite entrenar el motor para el idioma que se está utilizando (indicado en el cuadrado azul), el conjunto de intents y el diccionario de palabras personalizado.

    - Botón amarillo: Permite simular la wake word con lo que el asistente despierta y comienza a grabar audio.

    - Botón rojo: Permite reiniciar el servicio de Rhasspy.

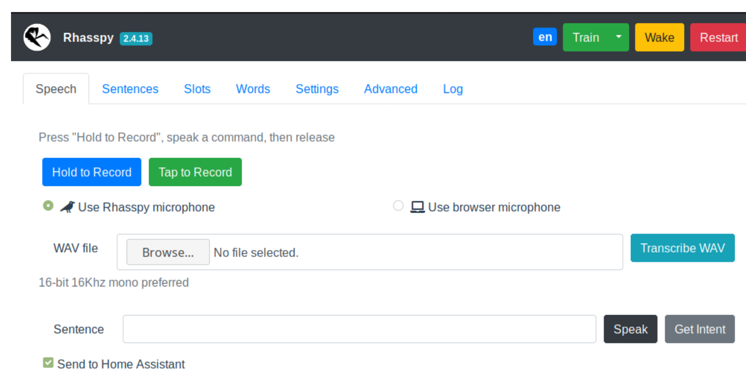



*Figura 32 Interfaz web Rhasspy [https://rhasspy.readthedocs.io/en/latest/img/web-speech.png]*

En la primera pestaña (Speech, Figura 32) podemos grabar audios manteniendo pulsado el botón “Hold to Record” o pulsando el botón “Tap to Record” para testear el funcionamiento de Rhasspy, subir de forma manual audios para obtener la transcripción de un audio almacenado en la máquina e introducir a mano un comando en el apartado “Sentences” para comprobar si el sistema es capaz de detectarlo.

En la segunda pestaña (Sentences, Figura 33) podemos configurar los comandos que queremos que el sistema pueda detectar, la estructura para declarar los comandos consiste en establecer un identificador o intent entre corchetes y debajo de este todos los comandos asociados. Todas las sentencias o comandos declarados juntos con sus respectivos intents son almacenados en un fichero llamado “sentences.ini”, Rhasspy da la posibilidad de cargar y utilizar otros ficheros de forma manual.

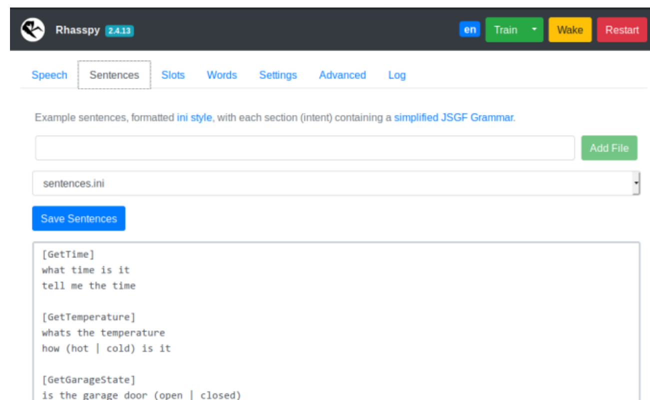



*Figura 33 Pestaña Sentences Rhasspy [https://rhasspy.readthedocs.io/en/latest/img/web-sentences.png]*

En la tercera pestaña (Slots, Figura 34) podemos declarar sets o conjuntos de valores que pueden ser utilizados en la declaración de comandos (Segunda pestaña), se utilizan a modo de atributo para uno o varios comandos, por ejemplo para un comando “pon las luces de color...”, podemos utilizar un set color en el que declarar varios colores posibles con los que utilizar ese comando. Estos sets tienen una estructura JSON, cada slot se identifica por un valor que actúa como clave (en la Figura 34 es el valor “person”) y tiene asociado un conjunto de valores.

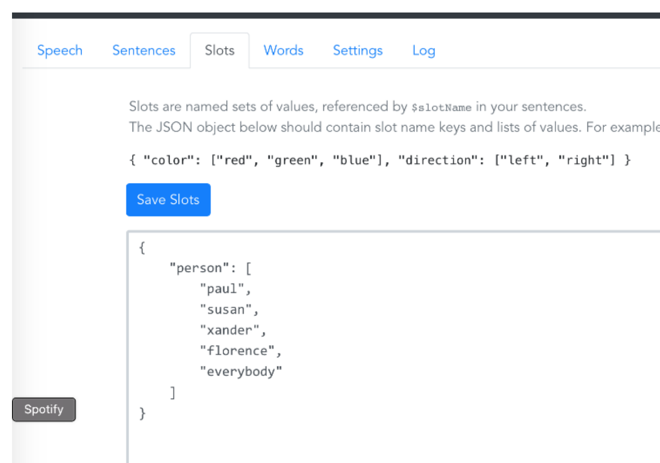



*Figura 34 Pestaña Slots Rhasspy [https://community.rhasspy.org/uploads/default/original/1X/336f9175ddf5127f032f2a24611bc5b4df698d7d.png]*

En la cuarta pestaña (Words, Figura 35) podemos añadir palabras que Rhasspy desconoce para el idioma seleccionado junto con sus respectivos fonemas, de esta forma conseguimos crear un diccionario de palabras personalizado. También tenemos la posibilidad de escuchar de forma individual los diferentes fonemas que utiliza el sistema (Figura 36) y de escuchar la pronunciación de las palabras que añadimos. Todas las palabras que se añaden en esta pestaña se almacenan en un fichero llamado "custom_words.txt".

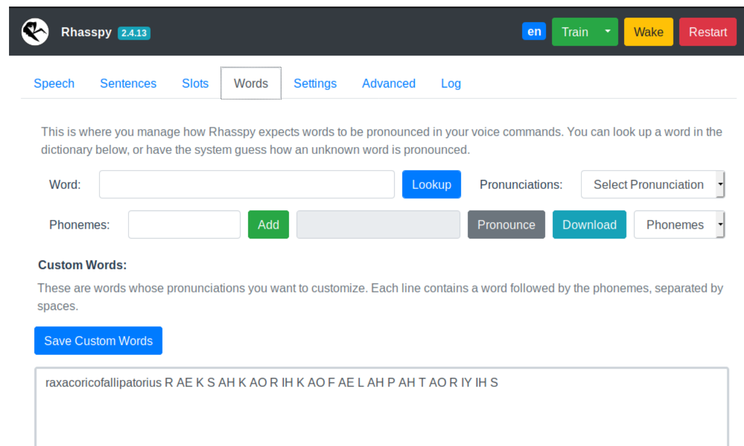



*Figura 35 Pestaña Words Rhasspy[https://rhasspy.readthedocs.io/en/latest/usage/]*

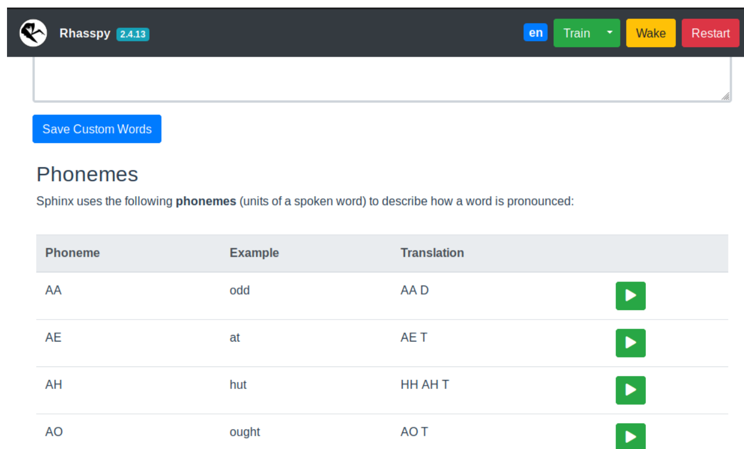



*Figura 36 Fonemas Rhasspy [https://rhasspy.readthedocs.io/en/latest/usage/]*

En la quinta pestaña (Settings, Figura 37) podemos modificar los ajustes de Rhasspy y de cada una de las herramientas que lo componen.

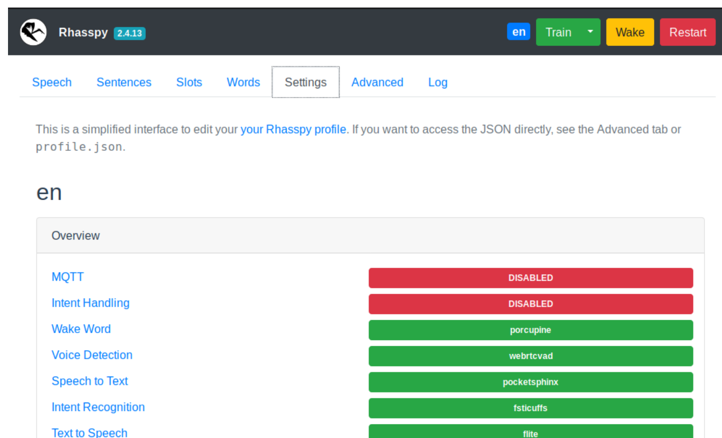



*Figura 37 Pestaña Settings Rhasspy [https://rhasspy.readthedocs.io/en/latest/usage/]*

En la sexta pestaña (Advanced, Figura 38) podemos modificar directamente el fichero "profile.json" desde la interfaz web.

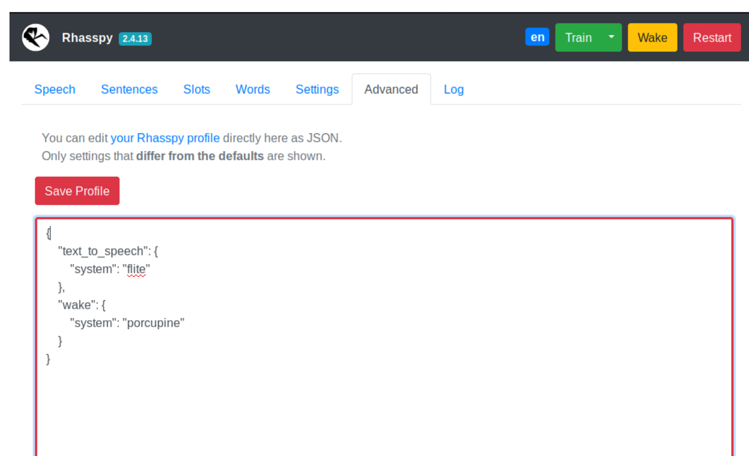



*Figura 38 Pestaña Advanced Rhasspy [https://rhasspy.readthedocs.io/en/latest/usage/]*

En la séptima (Log, Figura 39) y última pestaña podemos un registro o Log de los eventos que han ido ocurriendo en Rhasspy.

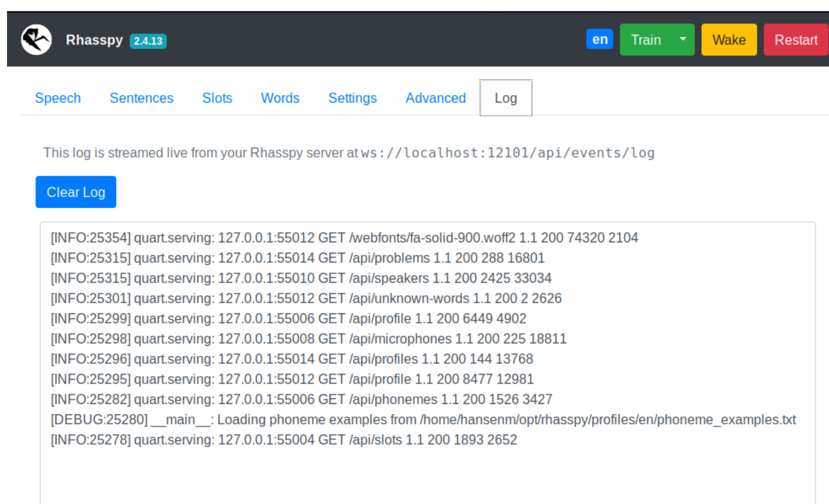



*Figura 39 Pestaña Log Rhasspy [https://rhasspy.readthedocs.io/en/latest/usage/]*

Cualquier modificación en alguna de las pestañas desde la segunda hasta la quinta (incluidas) requiere de un reentrenamiento del motor, de modo que pueda aprender los cambios que se han producido.

2. HTTP API: Rhasspy permite el uso de sus funcionalidades mediante peticiones HTTP GET y POST a determinados paths o directorios dentro del dominio en el que se está ejecutando el motor (Dirección IP : Puerto) llamados "endpoints"[36].

3. Comandos: Rhasspy permite utilizar parte de sus funcionalidades mediante línea de comandos y puede utilizarse tanto si estamos ejecutando Rhasspy en Docker como en un entorno virtual de Python.

## 3.3. Puesta en marcha

En este apartado vamos a ver cómo poner en marcha Rhasspy para empezar a utilizarlo y realizar pruebas para ir construyendo poco a poco el sistema que hemos planteado.

A la hora de realizar la instalación de Rhasspy, observamos que la herramienta es bastante flexible, pues soporta diferentes tipos de arquitecturas de sistema en las puede ejecutarse como muestra la Figura 40. De forma básica la arquitectura de un sistema determina el conjunto de instrucciones que es capaz de ejecutar un procesador así como la cantidad de memoria virtual y física que es capaz de soportar, no entraremos en detalle pues este trabajo no tiene por objeto el estudio de las diferentes arquitecturas.

- Raspberry Pi 2-3 B/B+ ( `armhf` / `aarch64` )
- Desktop/laptop/server ( `amd64` )
- Raspberry Pi Zero ( `armv6l` )

*Figura 40 Arquitecturas compatibles con Rhasspy [https://rhasspy.readthedocs.io/en/latest/hardware/]*

En este punto hemos determinado que la arquitectura más óptima para nuestro caso de uso es aarch64, que es la que corresponde a la Raspberry 3B+ (Figura 40), entre otros motivos porque esto permitirá reducir el coste de hardware del futuro sistema que construyamos y porque permitirá introducir una cantidad menor de elementos y además de un tamaño más reducido en un quirófano real, haciendo que el espacio ocupado por nuestro sistema sea mínimo y facilitando las tareas de limpieza.

Una vez elegida la arquitectura pasamos a la forma de instalación, Rhasspy funciona en sistemas operativos basados en GNU/Linux, de este tipo de sistemas hay varios disponibles para Raspberry Pi entre los que destacan Raspbian y Hass.io:

- Raspbian: Es una distribución GNU/Linux basada en Debian y optimizada para el hardware y la arquitectura de la Raspberry. Destaca por la gran cantidad de soporte que tiene, su rendimiento y estabilidad.

- Hass.io: Hass.io es una evolución del sistema operativo Home Assistant, al que añade plugins que facilitan la tarea automatizando ciertas acciones o configuraciones que de otra manera se tendrían que hacer completamente a mano. Además, corre sobre

Docker, lo que aísla y unifica bastante las cosas. En sí es un panel de control para controlar dispositivos de todo tipo como: enchufes, cámaras, luces, persianas, sensores… (Figura 41).

Rhasspy puede instalarse mediante Docker o dentro de un entorno virtual de Python y en el caso de Hass.io mediante su sistema de Add-Ons, sin embargo descartamos esta última opción pues no utilizaremos Hass.io como sistema operativo, debido principalmente a que es un sistema operativo sencillo pensado para personas con poco conocimiento en sistemas operativos basados en GNU/Linux por lo que la capacidad de personalizar y manejar el código de Rhasspy se reduce en gran medida, además varios usuarios de Rhasspy han reportado incompatibilidades de hardware con este sistema operativo. Por tanto utilizaremos el sistema operativo Raspbian.

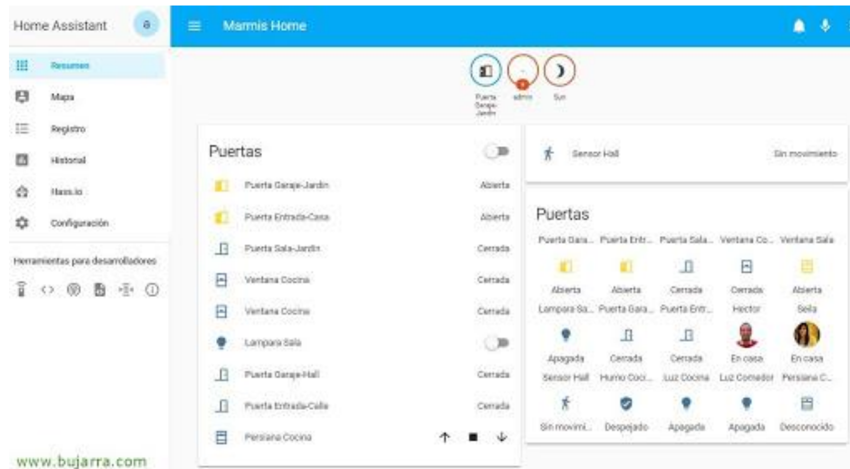



*Figura 41 Apariencia Hass.io [https://diy-smarthome.com/how-to-set-up-home-assistant-hass-io-as-a-home-automation-hub/]*

Para la instalación del sistema operativo es necesario grabar la imagen de este en una tarjeta de memoria microSD, para ello existen múltiples programas, nosotros vamos a utilizar "Balena Etcher". Una vez grabada la tarjeta de memoria con el sistema operativo Raspbian en su interior, la introducimos en la Raspberry Pi y comenzamos a realizar la instalación y configuración del sistema.

Una vez instalado el sistema operativo, siguiendo la recomendación de la documentación de Rhasspy [37] aumentamos el valor de la partición dedicada a SWAP o intercambio del sistema, esta partición permite almacenar datos de forma temporal en disco (La tarjeta de memoria en este caso) reduciendo así el gasto de memoria RAM. Esto lo hacemos debido a que la memoria RAM de una Raspberry PI 3B+ es bastante limitada con lo que tiende a llenarse con mucha facilidad. También llegados a este punto redimensionamos la partición "/root" del sistema para que ocupe todo el espacio restante de la tarjeta tras la instalación del sistema operativo, esto nos evitará problemas de almacenamiento posteriores. Este último paso es bastante común en sistemas pensados para Raspberry PI, pues las imágenes de estos sistemas están pensadas para funcionar en tarjetas de memoria pequeñas y tienen una tabla de particiones de memoria muy pequeña ya establecida.

El siguiente paso es proceder a la instalación de Rhasspy, aquí tenemos dos alternativas disponibles como hemos comentado más arriba, la primera es la instalación de la imagen de Docker de Rhasspy en un contenedor, mientras que la segunda consiste en instalar Rhasspy en un entorno virtual de Python. Decidimos que la mejor opción es la instalación del motor a través de un entorno virtual de Python, pues como veremos más adelante instalar Rhasspy en un

contenedor de Docker en muchas ocasiones provoca problemas para acceder al micrófono del sistema.

Después de instalar el motor, procedemos como paso opcional a configurar Rhasspy como un servicio del sistema operativo, esto entre otras cosas permite detener, reiniciar o iniciar automáticamente Rhasspy mediante la ejecución de comandos a través del terminal de Raspbian. Para ello hemos utilizado el fichero de configuración de ejemplo de la documentación de Rhasspy [37].

Como último paso ejecutamos Rhasspy a través de línea de comandos especificando el idioma a utilizar como perfil, para las pruebas utilizaremos el idioma español. Una vez iniciado Rhasspy accedemos a través del navegador a la dirección 127.0.0.1:12101, lo que nos mostrará la interfaz web del motor, la primera vez que entramos se nos solicita la descarga de una serie de elementos entre ellos: el modelo del lenguaje, el modelo acústico pre-entrenado y el diccionario general para el idioma (Figura 42). Cuando los elementos se han descargado entrenamos Rhasspy utilizando el botón correspondiente de la interfaz.

Una vez tengamos todo esto hecho, estaremos listos para utilizar Rhasspy y pasar al apartado de pruebas.

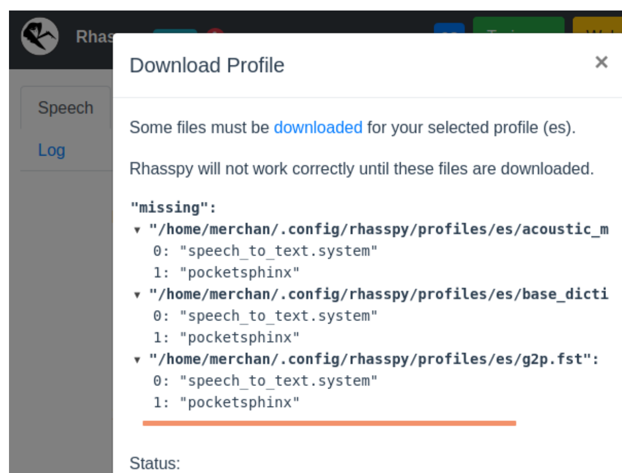



*Figura 42 Ficheros necesarios para el funcionamiento de Rhasspy*

## 3.4. Pruebas

### 3.4.1. Configuración previa de los sistemas antes de realizar las pruebas

Primero comenzamos configurando cada uno de los sistemas que utilizaremos en Rhasspy para la fase de pruebas:

1. Sistema para recoger el audio externo: ALSA. Hemos seleccionado el sistema por su alta compatibilidad que con ALSA de GNU/Linux (El que viene por defecto con Raspbian) y porque PyAudio (el sistema por defecto) ha dado muchos problemas durante las pruebas (tanto para Docker como para el entorno virtual de Python). Utilizaremos los ajustes por defecto del sistema y tan solo la modificaremos configuración relativa al dispositivo de entrada de audio, seleccionado el dispositivo adecuado.

2. Sistema para reproducir audio: ALSA. Al igual que antes porque es el que mejor compatibilidad tiene con ALSA de GNU/Linux. Utilizaremos los ajustes por defecto del sistema y tan solo modificaremos la configuración relativa al dispositivo de salida de audio, seleccionado el dispositivo adecuado.

3. Sistema escuchador de comandos: WebRTCVAD. Pues es un sistema que posee un buen rendimiento como lo demostrarán las pruebas. Por el momento se usaremos la configuración por defecto, aunque más adelante necesitaremos de unas pequeñas modificaciones.

4. Sistema Speech to Text: Pocketsphinx. Hemos elegido Pocketsphinx porque es el único que soporta los idiomas español e inglés y además trabaja de forma offline. Dejaremos la configuración por defecto para que utilice los ficheros por defecto del modelo acústico, del modelo del lenguaje, del diccionario base y del diccionario de palabras personalizadas.

5. Sistema de reconocimiento de intents: Fsticuffs. Hemos seleccionado este sistema pues a pesar de ser menos flexible que otros reconocedores es mucho rápido a la hora de entrenar y de reconocer intents que otros sistemas. Según la documentación resulta ser la mejor opción si Rhasspy se va a utilizar solo para reconocimiento vía audio de los comandos del conjunto de entrenamiento y no mediante un chat de texto. En este caso decidimos activar la opción fuzzy lo que le aportará una mayor flexibilidad a la hora de realizar el reconocimiento, el resto de la configuración será la que viene por defecto.

Como se puede observar omitimos la configuración del sistema de wake word, del sistema de text to Speech y del sistema de manejo de intents pues para las primeras pruebas no serán necesarios y conforme avanzamos en el desarrollo del sistema iremos integrando la utilización estos sistemas.

Antes de empezar a explicar las pruebas, consideramos importante mencionar que inicialmente realizamos una instalación de Rhasspy en un contenedor de Docker lo que inicialmente nos dio muchos problemas relacionados con el acceso al micrófono del sistema, hablaremos en profundidad de este problema más adelante.

### 3.4.2. Pruebas con comandos e intents por defecto y personalizados

Para las primeras pruebas como hemos dicho antes primero vamos a trabajar con el idioma español pues consideramos esencial que el sistema se comporte de manera óptima con este idioma antes de realizar las pruebas con inglés. Probamos de forma manual a despertar a Rhasspy y a decir varios comandos en alto, comprobamos que:

- Al menos 7 de cada 10 veces Rhasspy reconoce correctamente un comando de los que tiene almacenados por defecto en el fichero “sentences.ini” y por tanto los intents asociados.

- Las transcripciones de las palabras de cada frase las hace en la mayoría de casos a la perfección.

- Comprobamos que palabras no almacenadas en el diccionario básico del motor no son detectadas correctamente, como es el caso de: “estetoscopio”, “bisturí”, “esternocleidomastoideo”...

- Comprobamos que Rhasspy no hace distinción entre preguntas y afirmaciones por lo que comandos de voz como: “¿Qué dices?” y “que dices” tienen la misma transcripción y por tanto el mismo intent asociado.

Con estas pruebas realizadas procedemos a ampliar el conjunto de comandos con frases más largas y complejas como: “Enciende el estetoscopio esternocleidomastoideo”, “El ventrículo está obstruido”, “Aplicar esparadrapo rápidamente”... (Figura 43) Lo primero que vemos es que cuando modificamos el fichero “sentences.ini” y entrenamos Rhasspy para que aprenda los nuevos comandos, el motor nos detecta varias palabras como palabras desconocidas, algunas de ellas son: “estetoscopio”, “esternocleidomastoideo”, “ventrículo” y “obstruido” (Figura 44). Por lo tanto, las agregamos al diccionario de palabras personalizadas (pestaña words), Rhasspy añade junto a cada palabra el conjunto de fonemas que la forma de forma automática (Figura 45). Después de volver a entrenar el modelo con las nuevas palabras, realizamos pruebas de nuevo y comprobamos que:

- Alrededor de 7 u 8 de cada 10 veces Rhasspy reconoce correctamente los nuevos comandos y por tanto los intents asociados.

- Rhasspy reconoce las palabras que no reconocía antes (Figura 46).

- Rhasspy es capaz de reconocer números en los audios y en la transcripción se muestran de forma numérica, además los números decimales también los detecta tanto por ejemplo en la frase  “tres con catorce”, como en la frase “tres punto catorce”.

- Las grabaciones del audio del sistema WebRTCVAD tienen un límite de tiempo (30 segundos por defecto) por lo que si hablamos de continúo simulando ruido de fondo llega un momento que Rhasspy deja de recoger audio. En estas ocasiones el tiempo de procesamiento de Rhasspy es mayor y generalmente acaba devolviendo un error indicando que no ha podido reconocer el intent.



*Figura 43 Comandos personalizados añadidos*

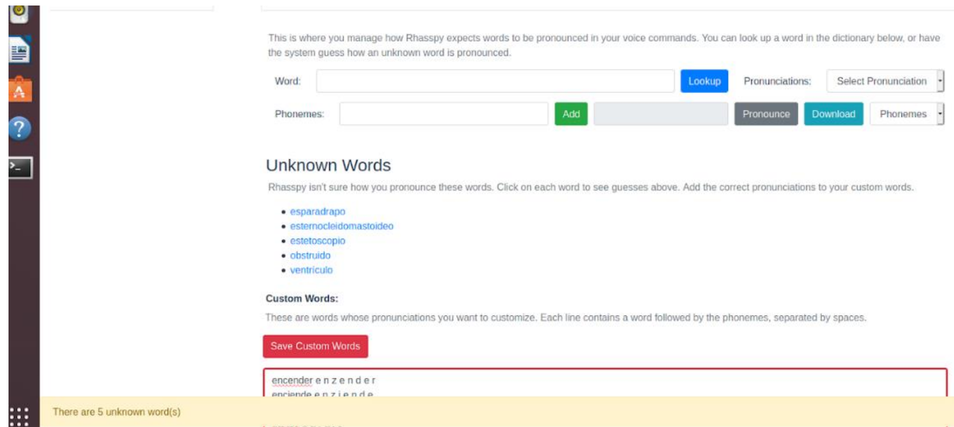



*Figura 44 Palabras desconocidas detectadas*

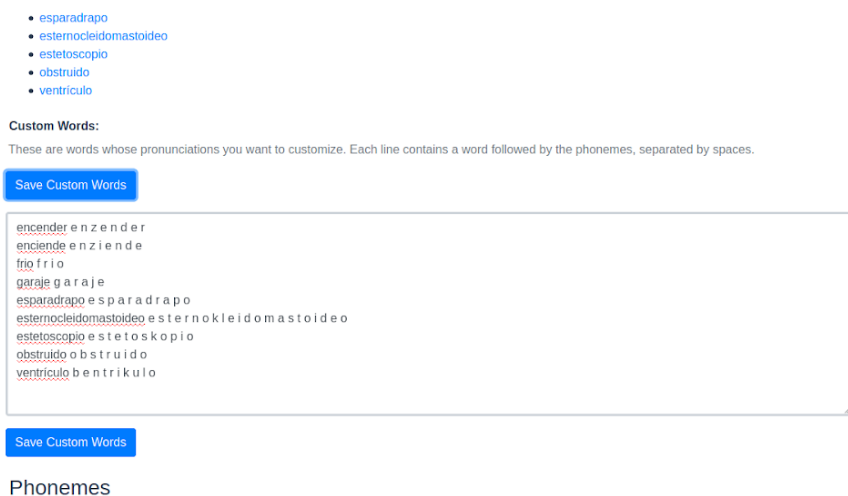



*Figura 45 Palabras desconocidas agregadas*

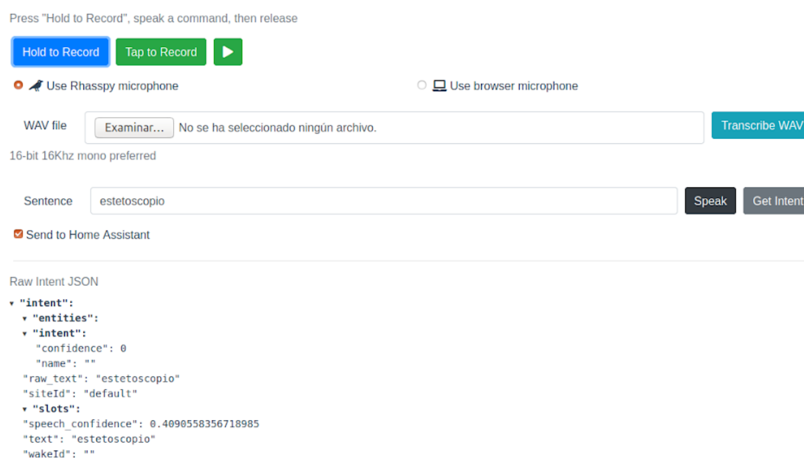



*Figura 46 Rhasspy detecta las palabras añadidas*

### 3.4.3. Configuración y pruebas con el sistema de wake word

Para las siguientes pruebas decidimos integrar el sistema de wake word de Rhasspy, en primer lugar utilizamos Porcupine (con su configuración por defecto) ya que como hemos explicado antes sobre el papel es el que mejor debería funcionar, sin embargo comprobamos que generalmente no detecta la palabra para despertar (aproximadamente 2 de cada 10 veces la detecta), probamos a subir la sensibilidad a 0.9 (en una escala de 0 a 1) con lo que el número de detecciones sube a 6 de cada 10 veces, no obstante una sensibilidad tan alta provoca que en muchas ocasiones falsos positivos.

Puesto que el rendimiento de Porcupine no ha sido el esperado vamos a probar a utilizar el sistema snowboy, inicialmente probamos el sistema con la configuración por defecto, por lo que la wake word es "snowboy", con esta configuración comprobamos que el comportamiento del sistema es similar a Porcupine. Decidimos utilizar palabras personalizadas para ver si el rendimiento del sistema mejora, para ello accedemos a la página web de snowboy [38], entrenamos un modelo con 3 muestras de nuestra voz para la palabra robot (Figura 47) y descargamos el fichero generado, hacemos lo mismo para la palabra Tedcas y añadimos ambos ficheros al directorio de Rhasspy.

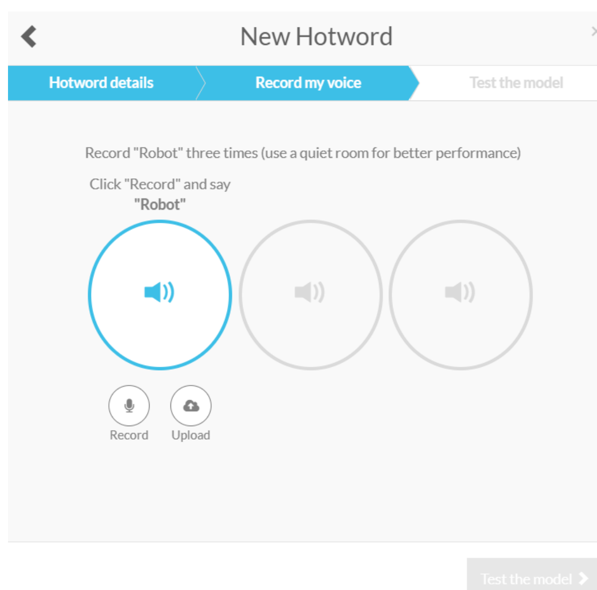



*Figura 47 Entrenamiento palabra robot Snowboy*

A través de la interfaz web de Rhasspy añadimos las dos palabras nuevas y procedemos a realizar pruebas, observamos que ambas palabras funcionan mejor que la palabra por defecto, pues al menos 7 de cada 10 veces son detectadas. Comprobamos además que el sistema es capaz de hacer distinción entre ambas palabras como muestran la Figura 48 y la Figura 49.

```
[DEBUG:633110] WebrtcvadCommandListener: loaded -> listening
[DEBUG:633109] SnowboyWakeListener: listening -> loaded
[DEBUG:633108] WebrtcvadCommandListener: Will timeout in 30 second(s)
[DEBUG:633106] DialogueManager: asleep -> awake
[DEBUG:633101] DialogueManager: Awake!
[DEBUG:633093] SnowboyWakeListener: Hotword(s) detected: ['snowboy/TedCas.pmdl']
```

*Figura 48 Log Rhasspy para wake word Tedcas*

```
[DEBUG:1252367] WebrtcvadCommandListener: loaded -> listening
[DEBUG:1252366] WebrtcvadCommandListener: Will timeout in 30 second(s)
[DEBUG:1252362] DialogueManager: asleep -> awake
[DEBUG:1252361] DialogueManager: Awake!
[DEBUG:1252343] SnowboyWakeListener: Hotword(s) detected: ['snowboy/Robot.pmdl']
```

*Figura 49 Log Rhasspy para wake word Robot*

Sin embargo en el caso de la palabra Tedcas es necesario ajustar la sensibilidad a 0.65 mientras que para la palabra Robot con la sensibilidad por defecto (0.5) es suficiente. Asumimos que esto se debe a que la palabra Robot ya tenía muestras previas de otras personas por lo que el modelo no solo ha entrenado con nuestras muestras de audio, lo que a priori debería hacerlo más robusto, mientras que la palabra Tedcas no existía y ha sido necesario crear el modelo solo con nuestras muestras (Figura 50 y Figura 51).

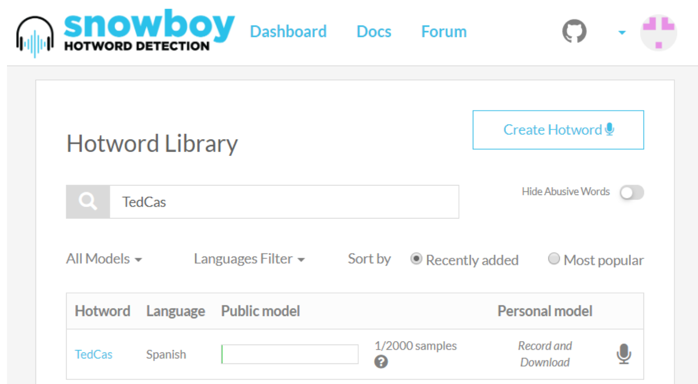



*Figura 50 Muestras disponibles para la palabra TedCas*

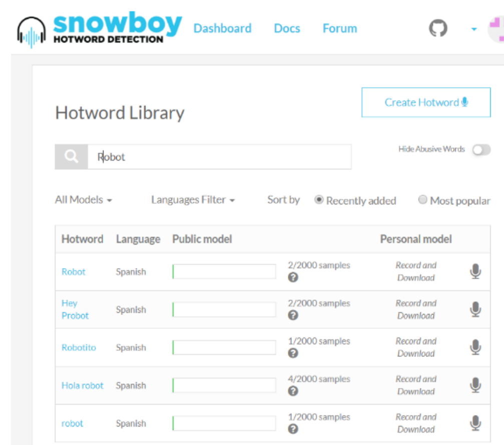



*Figura 51 Muestras disponibles para la palabra Robot*

### 3.4.4. Configuración del sistema de manejo de intents y desarrollo y pruebas de un servidor capaz de gestionar los intents de Rhasspy

Con el sistema de wake word integrado nuestro siguiente paso es añadir el sistema para manejar los intents, en este caso optamos por hacer a través de un servidor externo HTTP vía POST, de esta manera evitamos cargar más a la Raspberry. Para las primeras pruebas ponemos a ejecutar un servidor externo en la misma red local que Rhasspy con la herramienta “Netcat” y cambiamos la configuración de Rhasspy para que los intents reconocidos sean enviados a este servidor mediante HTTP POST. Los resultados obtenidos se muestran en la Figura 52.

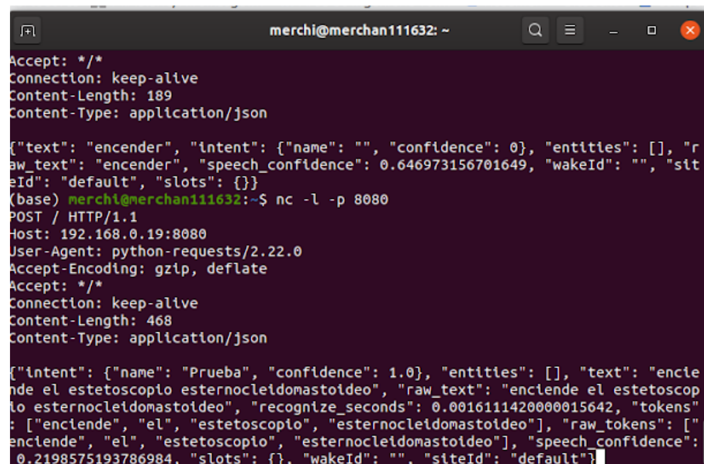



*Figura 52 Petición HTTP POST de Rhasspy en servidor Netcat*

Una vez comprobamos que Rhasspy es capaz de enviar peticiones a un servidor externo, comenzamos a construir un servidor HTTP que sea capaz de recibir las peticiones, procesarlas y enviar una respuesta, de manera que la comunicación se vuelva bidireccional. Para ello en primer lugar modificamos la configuración de Rhasspy para que se ejecute en una dirección IP distinta a localhost (127.0.0.1), de esta manera el servidor podrá comunicarse con él. Este servidor lo construiremos en Python pues nos resulta más fácil y rápido construir en este lenguaje. Este servidor (Figura 53):

1. Se ejecutará en una dirección IP y puerto determinado.

2. Recibirá peticiones POST y GET, procesará las primeras pues son las que esperamos recibir de Rhasspy y descartará las segundas.

3. Comprobará la estructura de la petición POST y si es correcta procesará la petición, en caso contrario la rechazará.

4. Comprobará que la estructura del evento JSON que envía Rhasspy es correcta y si no descartará la petición, pues asumimos que la estructura de los eventos de Rhasspy es fija por lo que si se recibe un evento que no cuenta con esta estructura, dicho evento no pertenece a Rhasspy.

5. Comprobará si el intent recibido en la petición es válido y no está vacío, es decir si es uno de los que el servidor tiene almacenados y que por tanto admite, si no es así mandará una respuesta a Rhasspy indicando que el intent no es correcto o está vacío.

6. Llevará a cabo las acciones correspondientes al intent y devolverá una respuesta a Rhasspy.

7. Llevará un log de todos los eventos que ocurren durante el servicio.

```
(base) merchi@merchan111632:~/Escritorio/TedCas$ python3 servidorHTTP.py
<_io.TextIOWrapper name='<stderr>' mode='w' encoding='UTF-8'> Inicializando el servidor...
<_io.TextIOWrapper name='<stderr>' mode='w' encoding='UTF-8'> Servidor ejecutandose...
HOLAAA
<_io.TextIOWrapper name='<stderr>' mode='w' encoding='UTF-8'> El intent recibido no cumple la estructura
192.168.2.91 - - [18/Feb/2020 12:08:48] "POST / HTTP/1.1" 400 -
HOLAAA
<_io.TextIOWrapper name='<stderr>' mode='w' encoding='UTF-8'> El intent recibido no cumple la estructura
192.168.2.91 - - [18/Feb/2020 12:08:53] "POST / HTTP/1.1" 400 -
```

*Figura 53 Servidor HTTP desarrollado*

### 3.4.5. Configuración y pruebas del sistema Text to Speech

Las respuestas del servidor tienen el mismo formato JSON que las peticiones de Rhasspy, solo que introducen un campo nuevo llamado “Speech”. Este campo podemos definirlo como la respuesta que el servidor da a Rhasspy. La función de este campo es que al recibir la respuesta si Rhasspy encuentra el campo “Speech” transforma su contenido de texto a sonido, es decir realiza un proceso de text to Speech, de esta manera conseguimos informar en todo momento al usuario de las acciones que se están llevando a cabo como respuesta a sus peticiones.

Para lograr este objetivo una vez desarrollado el servidor integramos el sistema text to speech de Rhasspy. Primero probamos a utilizar eSpeak, pero finalmente acabamos usando PicoTTS, pues como ya hemos comentado la voz de este segundo sistema suena mucho más natural que la del primero.

### 3.4.6. Configuración y pruebas de Rhasspy y el servidor HTTP para despertar a Rhasspy de forma encadenada

Lo siguiente que proponemos realizar es que el servidor sea capaz de despertar a Rhasspy de forma encadenada después de recibir un intent correcto, es decir que en vez de tener que despertar Rhasspy con cada petición, despertarlo una vez, realizar la petición y si esta es correcta (el intent que recibe el servidor no está vacío y corresponde con alguno de los almacenados) despertarlo automáticamente, así cada vez hasta que la petición que reciba el servidor no sea correcta. Para ello hacemos uso de la API HTTP de Rhasspy, de manera que cuando el servidor reciba una petición correcta, realice las acciones correspondientes y envíe una respuesta y además envíe una petición POST a la dirección https://direccion_Rhasspy:12101/api/listen-for-command, de esta manera conseguiremos despertar a Rhasspy automáticamente después de cada petición correcta.

Para realizar todo esto primero debemos activar la opción de cifrado SSL en Rhasspy, para ello cuando ejecutamos Rhasspy debemos pasarle un certificado y una clave de cifrado privada. Creamos la clave y el certificado nosotros mismos (certificado auto-firmado) como vemos en la Figura 54, si no, de otra manera, tendríamos que comprar el certificado a una AC (Autoridad de certificación). Por otro lado, modificamos nuestro servidor de manera que utilice el mismo cifrado que Rhasspy para que así se puedan comunicar y nuevamente generamos una clave y un certificado auto-firmados. Una vez hecho todo esto, modificamos la configuración de Rhasspy

cambiando la URL del servidor que maneja los intents para añadirle el protocolo SSL que va a utilizar a partir de ahora (pasamos de http:// a https://).

```
merchan@merchan-VirtualBox: ~
Archivo Editar Ver Buscar Terminal Ayuda
merchan@merchan-VirtualBox:~$ openssl req -x509 -newkey rsa:4096 -nodes -out cert.pem -keyout key.pem -days 365
Can't load /home/merchan/.rnd into RNG
139927960236480:error:2406F079:random number generator:RAND_load_file:Cannot open file:../crypto/rand/randfile.c:88:Filename=/home/merchan/.rnd
Generating a RSA private key
..........................................................................................................................
..........................................................................................................................
writing new private key to 'key.pem'
-----
You are about to be asked to enter information that will be incorporated
into your certificate request.
What you are about to enter is what is called a Distinguished Name or a DN.
There are quite a few fields but you can leave some blank
For some fields there will be a default value,
If you enter '.', the field will be left blank.
-----
Country Name (2 letter code) [AU]:ES
State or Province Name (full name) [Some-State]:Navarra
Locality Name (eg, city) []:Pamplona
Organization Name (eg, company) [Internet Widgits Pty Ltd]:TedCas
Organizational Unit Name (eg, section) []:TedCas Medical Systems S.L
Common Name (e.g. server FQDN or YOUR name) []:Julen
Email Address []:practicasTedcas2020@gmail.com
merchan@merchan-VirtualBox:~$
```

*Figura 54 Creación de un certificado auto-firmado*

Para las pruebas hemos necesitado desactivar en el servidor la comprobación del certificado y hemos tenido que modificar el código de Rhasspy con el mismo fin, ya que ambos detectan que los certificados del otro no están firmados por ninguna AC de confianza por lo que rechazan las conexiones.

### 3.4.7. Últimas configuraciones y pruebas de Rhasspy y el servidor HTTP

Por otro lado para la fase de pruebas hemos modificado el código de Rhasspy para permitir guardar de forma local el audio que recoge el sistema, de esta manera podemos escuchar el audio que graba Rhasspy y determinar en ciertos momentos porque falla al reconocer ciertos comandos y así ajustar la configuración del sistema.

Por último modificamos el servidor para hacerlo concurrente (para que maneje varias peticiones al mismo tiempo), esto es necesario pues cuando realicemos pruebas en quirófano trabajaremos con varias Raspberrys y un único servidor atendiendo peticiones.

## 3.5. Problemas y soluciones

1. En primer lugar como ya hemos comentado antes, al principio realizamos pruebas con Rhasspy instalado en un contenedor de Docker, esto provocaba problemas con los permisos para acceder al micrófono lo que impedía recoger audio y por tanto no podíamos despertar a Rhasspy y en muchas ocasiones tampoco podíamos grabar los comandos de voz.

2. Tuvimos problemas a la hora de utilizar el sistema PicoTTs pues este requiere de una serie de paquetes que no estaban instalados en la Raspberry por defecto.

3. Como ya hemos comentado para facilitarnos el trabajo configuramos Rhasspy como un servicio del sistema operativo, no obstante tuvimos problemas con el inicio automático

del mismo, lo solucionamos cambiando el fichero de servicio de Rhasspy a la ruta /etc/systemd/system.

4. En la comunicación entre el servidor y Rhasspy (sin cifrado) tuvimos problemas con el formato de las peticiones HTTP POST que enviaba el servidor que provocaba que Rhasspy rechazase todas las peticiones que recibía. Para solventar el problema en la petición se debe añadir primero una línea que indique la versión y tipo de servidor en la respuesta, en segundo lugar es necesario rellenar las cabeceras de respuesta para que Rhasspy sepa qué buscar en el contenido, por último se debe añadir al cuerpo de la respuesta el contenido del intent que se recibe de Rhasspy y después la información adicional que se desee.

5. En la comunicación entre el servidor y Rhasspy (con cifrado) tuvimos problemas con los certificados, pues como ya hemos comentado tanto Rhasspy como el servidor los identificaban como auto-firmados por lo rechazaban las conexiones, para solucionar los problemas nos vimos obligados a suprimir la comprobación de certificados en el servidor y a modificar una pequeña parte del código de Rhasspy con la misma intención.

6. Durante las pruebas en algunas ocasiones la Raspberry se ha quedado congelada por el proceso “python3 app.py” pues muchas veces ha llegado a ocupar casi el 90% de la memoria disponible (incluso después de reiniciar la Raspberry). Como solución nos vimos obligados a ampliar la partición “/root” del sistema para evitar congelaciones.

# 4. Wav2letter++

En esta sección de la memoria tratamos el funcionamiento de la herramienta Wav2letter++ y a realizamos pruebas con las que comprobar su rendimiento para nuestro caso de uso. Este apartado se estructura de la siguiente manera:

1. Comenzamos con una pequeña introducción a cerca de Wav2letter++, donde veremos qué es, por qué destaca y qué objetivos persigue.

2. Hablamos del funcionamiento de la herramienta y de cada una de las partes que la componen, además del proceso que sigue un fichero de audio desde que entra en el sistema hasta que devuelve una predicción de la transcripción.

3. Explicamos de una forma superficial los pasos a seguir para poner en marcha Wav2letter++ y la finalidad de cada una de las librerías de las que depende para su correcto funcionamiento.

4. Exponemos de forma extendida cada una de las pruebas que hemos realizado con el motor y los resultados que hemos obtenido, así como el planteamiento seguido para mejorar el rendimiento del sistema.

5. Comentamos tanto los problemas que hemos encontrado durante la fase de puesta en marcha como los que hemos tenido durante la fase de pruebas.

## 4.1. Introducción

Wav2letter++ es una herramienta open source desarrollada por el equipo de Facebook AI Research y tiene por objetivo facilitar la creación de modelos end-to-end para el reconocimiento del habla (sección 2). Wav2letter++ está escrito completamente en C++ lo que permite maximizar la eficiencia de la herramienta en cuanto a coste computacional y consumo de recursos [39], pues C++ permite un control exhaustivo de los recursos para sistemas de alto rendimiento, además otra ventaja es que muchas de las librerías nativas son fácilmente invocables desde otros lenguajes de programación (bindings). Además, utiliza librerías como ArrayFire y Flashlight. ArrayFire es una librería muy optimizada que permite la ejecución con soporte de GPU o de CPU. La estructura de librerías la podemos ver en la Figura 55.

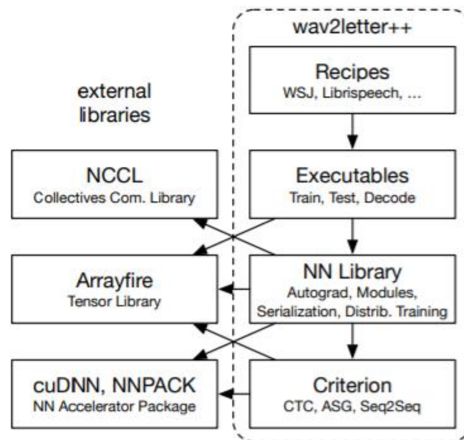



*Figura 55 Estructura de librerías Wav2letter++* [40]

Wav2letter++ se basa en la creación de modelos acústicos end-to-end basados en grafemas, a diferencia de los sistemas tradicionales que están basados en HMM (Hidden Markov Models) y GMM (Gaussian Mixture Models). Esto es debido a que poco a poco los sistemas end-to-end se están acercando cada vez más en cuanto a precisión a los sistemas tradicionales [40].

Wav2letter++ funciona en múltiples idiomas entre los que se encuentra el inglés y el español, además provee de una serie de bindings en Python para la utilización de las librerías propias de C++ que trae consigo el framework y que permiten, entre otras cosas, extraer características del audio, decodificar las muestras y utilizar funciones de coste más optimizadas para el proceso de descenso por gradiente.

Los desarrolladores de la herramienta presumen que es hasta dos veces más rápida que muchas de las alternativas de la competencia que existen en el mercado (Figura 56).

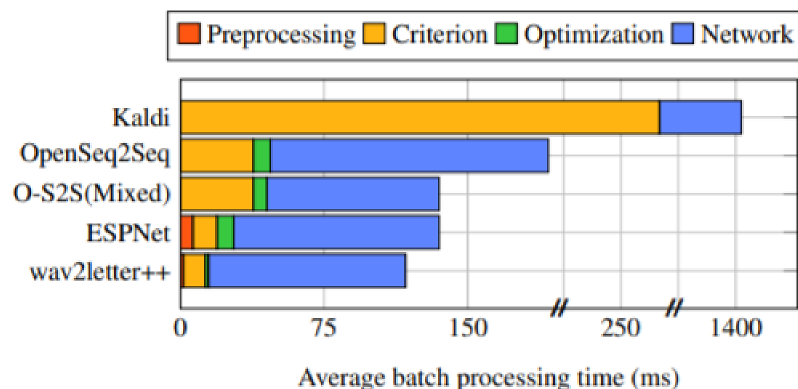



*Figura 56 Comparativa del promedio del tiempo de procesamiento de varias herramientas* [40]

El diseño de Wav2letter++ está centrado en lo siguiente (ver [41]):

1. Proveer las bases para entrenar de forma eficiente modelos en datasets grandes (que contengan miles de horas).

2. Ofrecer un modelo sencillo y extensible que permita incorporar nuevas arquitecturas de red, funciones de coste y otras operaciones típicas de los sistemas de reconocimiento del habla.

3. Agilizar el proceso de pasar de la investigación al desarrollo de modelos para Speech Recognition.

## 4.2. Funcionamiento

El sistema está compuesto por varias partes:

1. Preparación de datos y extracción de características: Wav2letter++ cuenta con un sistema para extraer las características de los datos de entrada (audios), lo que le permite preparar los datos antes de ser utilizados durante la fase de entrenamiento o decodificación. Además, Wav2letter puede tratar con diferentes formatos de audio (flac, WAV, mp3...) y con diferentes tipos de características de audio, entre las que destacan:

1.1. Audio en crudo o raw audio: Estas características se obtienen de un audio sin comprimir similar a WAV, este tipo de audio no incluye información de encabezado como la tasa de muestreo o el número de canales que utiliza, por lo que contienen mucha información (no toda es útil, como el ruido o los silencios).

1.2. MFCC (Mel Frecuency Cepstral Coefficients): MFCC tiene en cuenta la percepción humana de la sensibilidad a determinadas frecuencias del espectro de audio, por ello MFCC permite convertir la frecuencia convencional de los audios a escala Mel (una escala perceptual), esto hace que las características extraídas sean adecuadas para las tareas de reconocimiento del habla (ya que son adecuadas para comprender a los humanos y la frecuencia con la que los humanos hablan o pronuncian).
MFCC permite extraer la información relevante del audio y descartar información poco valiosa como el ruido de fondo. MFCC normalmente se calcula de la siguiente manera [42]:

1) Se divide el fichero de audio en un conjunto de pequeños frames o tramos.

2) Para cada frame se calcula el periodograma mediante la transformada de Fourier y se estima el espectro de energía. Un periodograma se utiliza para poder estimar la densidad espectral de una señal de audio, es decir, de una señal se estima una función matemática que nos informa de cómo está distribuida la potencia o la energía de dicha señal sobre las distintas frecuencias de las que está formada.

3) Se aplica el filtro de MFCC al espectro de energía obtenido en el paso anterior para filtrar aquella información que no es necesaria para el reconocimiento de voz. El problema de esto es que es muy difícil separar frecuencias que están espacialmente muy próximas (sobre todo en frecuencias altas), con MFCC se generan grupos de frecuencias y se suman las energías del espectro, para poder hacerse una idea de la cantidad de energía existente en cada región. Como resultado se obtiene un “banco de filtros”, que consiste en un array formado por más de un filtro, que separa

la señal de entrada en varias componentes, cada una de las cuales transporta la sub-banda de una sola frecuencia de la señal original.

4) Se toma el logaritmo de todas las energías obtenidas en el paso anterior (de cada filtro del banco de filtros).

5) El paso final es calcular la TDC (Transformada Discreta del Coseno) de los resultados del paso 4. Generalmente de los 26 coeficientes que se obtienen solo se suelen utilizar de 12 a 14. Esto se debe a que los coeficientes TDC más altos representan cambios rápidos en las energías del banco de filtros, lo que ha demostrado que reduce el rendimiento de los sistemas de Speech Recognition.

1.3. MFSC (Mel Frequency Spectral Coefficient): Su funcionamiento es similar a MFCC salvo que en el cuarto paso en vez de tomar logaritmos del banco de filtros se utiliza una función de transformación basada en una compresión no lineal (Figura 57).

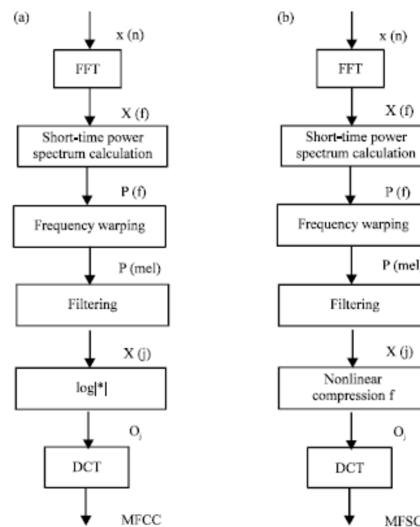



*Figura 57 Procesos de MFCC y MFSC [https://scialert.net/fulltext/?doi=jse.2015.350.361]*

Para hacer este proceso mucho más eficiente el motor lee, decodifica y extrae características de forma asíncrona y en paralelo.

2. Modelos: En Wav2letter++ todos los modelos están compuestos de una arquitectura de red y de una función de coste o criterion, esta herramienta soporta múltiples arquitecturas de red desde capas convolucionales de una dimensión o dos dimensiones a capas RNN, GRU, LSTM... también incluye varias funciones de activación que pueden ser utilizadas entre las que destacan: ReLu, Tanh (Tangente hiperbólitca) y la sigmoide. Todos estos elementos vienen integrados como módulos de la librería Flashlight [43]. Además, Wav2letter++ soporta diferentes tipos de funciones de coste entre las que destacan:

2.1. CTC (Connectionist Temporal Classification): El CTC es un tipo de función de coste que busca resolver el problema de alinear los caracteres transcritos del audio a los frames que componen el mismo. Este tipo de funciones ayudan a Wav2letter++ a entender cómo son las transiciones de letras a lo largo del tiempo (en su representación de

ondas), lo que ayuda a mapear la representación de un sonido con su letra correspondiente [44].

CTC trabaja con probabilidades, a la hora de clasificar, buscaremos maximizar la probabilidad a posteriori de una clase determinada dada una entrada, estos valores de probabilidad dependen de como CTC alinee los valores de la entrada con los valores de salida. Para maximizar la probabilidad de que la salida seleccionada sea la correcta y esperada es necesario computar de forma eficiente las probabilidades condicionales, por lo que la función coste de CTC utiliza el descenso por gradiente.

CTC es de alineado libre, no requiere de alineación entre la entrada y la salida, sino que obtiene la probabilidad de la salida a partir de la entrada sumando las probabilidades de todas las posibles diferentes alineaciones, seleccionando la de mayor probabilidad.

Este algoritmo lo que hace, básicamente, es que si tenemos un audio de, por ejemplo 10 segundos a una frecuencia de 100Hz acabamos teniendo aproximadamente 1.000 entradas, por lo que podíamos pensar que tendríamos aproximadamente 1.000 salidas esperadas diferentes o sea 1.000 caracteres, sin embargo lo que CTC hace es obtener 1.000 salidas repitiendo caracteres a lo largo del tiempo, para poder alinear de esta manera la entrada con la salida y una vez se obtiene la salida completa comprime los caracteres obtenidos en la transcripción. Un ejemplo de esto que acabamos de explicar es la Figura 58.

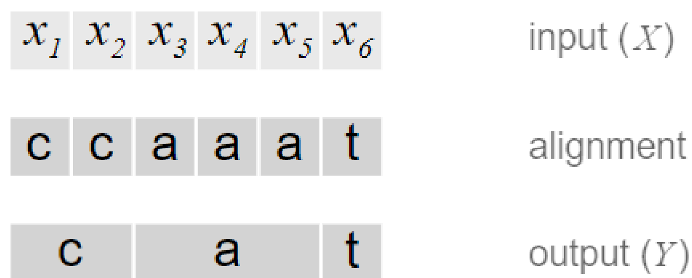



*Figura 58 Alineación entre la entrada y la salida con CTC* [44]

Sin embargo, esto que acabamos de ver tiene dos problemas, el primero es que no tiene sentido forzar a que por cada entrada haya siempre una salida ya que existen momentos en los audios en los que hay silencio, el segundo es que puede haber palabras que tengan un carácter repetido como: “perro”, “olla”, “sierra”... y al comprimir los caracteres lo que haría CTC es eliminar uno de estos caracteres con lo que obtendríamos transcripciones como: “pero”, ”ola”, ”siera”...

Para solventar este problema este método introduce un token o carácter especial llamado “blank token” que normalmente se denota con el símbolo ε, este carácter no tiene ningún significado, se introduce en momentos en los que se detecta silencio y entre repeticiones de caracteres diferentes, por lo que a la hora de comprimirlos no se eliminan elementos como veíamos antes. Observamos esta solución para el ejemplo de la Figura 58 en la Figura 59.

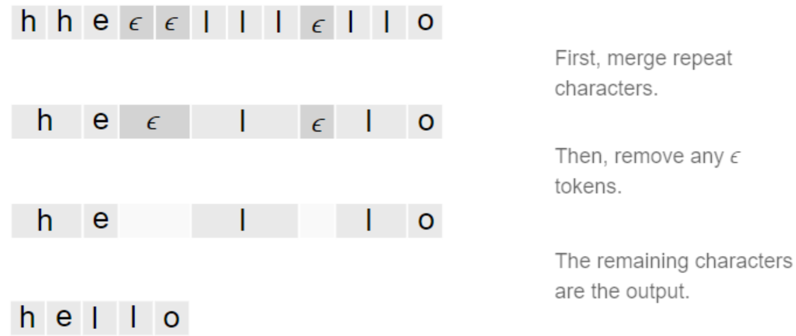



*Figura 59 Uso de "Blank token" en CTC* [44]

Para reconocimiento del habla, CTC se combina con modelos del lenguaje para ofrecer una mejora en la precisión de salida de la red neuronal, por lo que la salida a la hora de clasificar suele tener la forma de la Figura 60.

$$Y^* = \underset{Y}{\operatorname{argmax}} \quad p(Y \mid X) \quad \cdot \quad p(Y)^\alpha \quad \cdot \quad L(Y)^\beta$$

The CTC conditional probability. The language model probability. The "word" insertion bonus.

*Figura 60 Salida en CTC junto con un modelo del lenguaje* [44]

En referencia a la Figura 60, P(Y|X) es la probabilidad condicional con CTC, P(Y) es la probabilidad del modelo de lenguaje y L(Y) representa la longitud del modelo de lenguaje en términos del número de tokens que componen el modelo. Los parámetros α y β se seleccionan mediante validación cruzada y dan más o menos peso en función de su valor.

Este algoritmo aplica una búsqueda beam search que explicaremos más adelante, donde se explora un grafo expandiendo el nodo que sea más prometedor sumando para ello las probabilidades de cada secuencia de letras, de esta forma CTC contempla la posibilidad de que una salida tenga varias formas de alinearse.

2.2. ASG (AutoSegCritetrion): ASG funciona de una manera similar a CTC salvo por dos cuestiones, en primer lugar no hace uso de un carácter especial para indicar repetición en las letras sino que lo indica mediante un “2” [45] lo que ayuda a reducir el grafo de posibles secuencias de una palabra haciendo que las búsquedas sean más sencillas (Figura 61 y Figura 62), en segundo lugar CTC requiere de una normalización en los datos de entrada, mientras que ASG no.

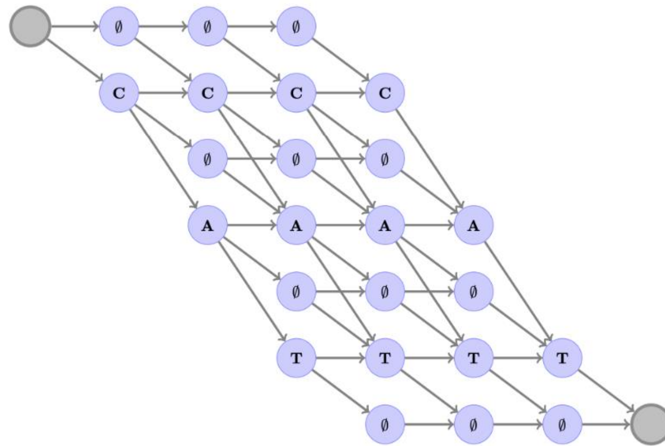



*Figura 61 Grafo CTC para la palabra "CAT"* [45]

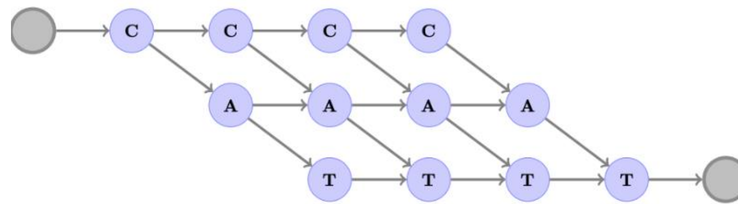



*Figura 62 Grafo ASG para la palabra "CAT" [41]*

3. Entrenamiento escalable: Wav2letter++ soporta tres tipos de entrenamiento:

    3.1. "train": consiste en entrenar un modelo desde cero.

    3.2. "continue": consiste en seguir entrenando un modelo a partir de un punto guardado (checkpoint) para ajustar la configuración de hiper-parámetros.

    3.3. "fork": consiste en crear y entrenar un nuevo modelo a partir de uno que este guardado, suele utilizarse para adaptar un modelo a un nuevo dataset.

    Además, soporta diferentes algoritmos de optimización estándar como el SGD y otros basados en descenso por gradiente como Adam.

4. Decodificación: El proceso de decodificación en Wav2letter++ está basado en la búsqueda Beam Search, este tipo de búsqueda es una variante optimizada del algoritmo best-first que consume menor cantidad de memoria, beam search construye el grafo o árbol de búsqueda ordenando las soluciones parciales de menor a mayor coste de acuerdo a un coste de heurística. En cada nivel del grafo de búsqueda sólo se almacenan un número predeterminado de estados llamado "beam width" (cuanto mayor es su valor más memoria es requerida y menor número de estados son descartados o podados en cada nivel del árbol) y el resto son podados del árbol de acuerdo con un valor de umbral, de forma que todo camino cuyo coste de heurística sea mayor que el umbral es descartado. podemos representar el algoritmo mediante la Figura 63. Durante la fase de decodificación puede utilizarse un modelo del lenguaje lo que permite optimizar la búsqueda.

Crea una agenda de un elemento (el nodo raíz)
*hasta* que la agenda esté vacía o se alcance la meta
  *si* el primer elemento es la meta
  *entonces* acaba
  *si no* elimina el primer elemento,
    añade sus sucesores a la agenda,
    ordena todos los elementos de la agenda y
    selecciona los $k$ mejores (elimina los demás)

*Figura 63 Algoritmo de Beam Search [https://es.slideshare.net/whaleejaa/beam-search-57693084]*

Debemos de explicar también que Wav2letter++ cuenta con una serie de ficheros que vamos a utilizar durante la fase de pruebas:

- Fichero de arquitectura: En este fichero se define la arquitectura de la red neuronal, cada línea del fichero representa una capa de la red o una función, que viene indicada por un identificador y una serie de parámetros (establecidos en la librería Flashlight [43]).

- Fichero de configuración: Contiene una serie de flags o etiquetas que determinan la configuración del sistema para el entrenamiento y la decodificación (generalmente se utiliza un diferente en cada fase).

- Fichero de léxico: Contiene el léxico (las palabras) permitidas en el lenguaje, junto con el conjunto de grafemas que las forman.

- Fichero de tokens o diccionario: Contiene las unidades básicas del lenguaje permitidas.

- Fichero de entrenamiento/test/validación: Contienen varias líneas donde cada línea contiene un único ejemplo del dataset y está formada por un identificador, la ruta a un ejemplo del conjunto de datos, la duración de este (en milisegundos) y su transcripción.

- Fichero del modelo del lenguaje: Este fichero contiene el modelo de lenguaje.

Cada una de las partes y los pasos que da Wav2letter++ se pueden representar mediante la Figura 64.

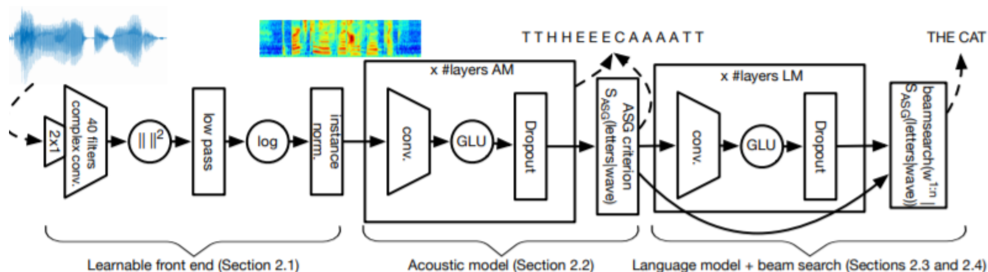



Figure 1: *Overview of the fully convolutional architecture.*

*Figura 64 Estructura de Wav2letter++* [46]

## 4.3. Puesta en marcha

Para la puesta en marcha del sistema es necesario que instalemos y configuremos una serie de dependencias antes de poder instalar y utilizar Wav2letter++:

1. Puesto que Wav2letter++ está completamente escrito en C++, es necesario que comprobemos si contamos con un compilador para C++, con soporte para c++11 y en caso de no tenerlo instalarlo. Instalaremos g++.

2. Es necesario instalar la librería Flashlight de la que ya hemos hablado, para ello es necesario que previamente instalemos y configuremos unas dependencias:

   2.1. ArrayFire: Como ya hemos explicado antes se trata de una librería que simplifica y optimiza el rendimiento en computación mediante la ejecución de lotes de operaciones en paralelo, la librería soporta dispositivos con arquitecturas x86, ARM, CUDA, y OpenCL. Instalamos ArrayFire y añadimos una variable al sistema que guarde la ruta a la librería.

   2.2. Libfreeimage3: Se trata de las dependencias necesarias para dar soporte gráfico a ArrayFire.

   2.3. Google Test: Instalamos este framework de Google para el testo en c++, para ello compilamos las librerías que trae el paquete gtest y las copiamos en el directorio de librerías de Linux (/usr/lib/).

   2.4. OpenMPI: Es un conjunto de herramientas para el paso de mensajes utilizando el estándar MPI, MPI es un estándar que define la sintaxis y la semántica de las funciones contenidas en una biblioteca de paso de mensajes. El paso de mensajes se utiliza para aportar sincronización entre procesos y permitir la exclusión mutua, de manera similar a como se hace con los semáforos, monitores…

   2.5 Nvidia Cuda: Se trata del software de Nvidia que permite la utilización de la gráfica Nvidia del sistema para tareas de computación en paralelo ejecutadas sobre GPU.

   2.6. Nvidia CuDDN (Nvidia Cuda Deep Neural Networks): Es un conjunto de librerías de Nvidia que utiliza la GPU para el procesamiento acelerado para redes neuronales profundas.

   2.7. Nvidia NCCL (Nvidia Collective Communication Library): Es un conjunto de librerías de Nvidia que permite la comunicación colectiva multi GPU.

   2.8. OpenBlas: Se trata de un conjunto de librerías que contienen un conjunto de rutinas muy optimizadas para tareas de algebra lineal básico.

   Con todo esto instalado y configurado, procedemos a la instalación de la librería Flashlight.

3. Instalamos libsndfile: Una librería que permite leer y escribir ficheros de audio.

4. Procedemos a instalar la librería Intel MKL: Se trata de una librería con rutinas matemáticas optimizadas para ciencia, ingeniería y finanzas, es muy similar a OpenBlas.

Para ello, compilamos la librería y añadimos una variable al sistema que guarde la ruta a la misma.

5. Realizamos la instalación de FFTW: FFTW es una biblioteca de subrutinas en C para calcular la transformada discreta de Fourier (DFT) en una o más dimensiones, de tamaño de entrada arbitrario y de datos reales y complejos.

6. Instalamos KenLM: Se trata de una librería eficiente que implementa dos estructuras de datos para la construcción de modelos de lenguaje eficientes, reduciendo tanto el tiempo como los costes de memoria. La estructura de datos Probing utiliza tablas hash de sondeo lineal y está diseñada para ser rápida. La estructura de datos Trie es una terna con empaquetamiento a nivel de bit, registros ordenados, búsqueda de interpolación y cuantificación opcional dirigida a un menor consumo de memoria. Trie usa simultáneamente menos memoria y menos CPU. Para ello previamente instalamos:

   6.1. Libboost: Es un conjunto de librerías que permite manipular rutas, directorio y ficheros de una forma más fácil y óptima.

   6.2. Libeigen3: Es una librería para el cálculo de algebra lineal de una forma óptima y eficiente.

   Una vez instaladas las dos librerías procedemos a instalar la librería de KenLM y exportamos en una variable la ruta al directorio de la librería.

7. Instalamos Gflags y Libgflags2v5: El paquete Gflags contiene una librería C++ que implementa el procesamiento de flags vía línea de comandos. Incluye soporte para tipos estándar como String y la capacidad de definir flags dentro del archivo fuente en el que se utilizan, esto lo utilizaremos para los ficheros de configuración (para entrenamiento y decodificación) que comentábamos más arriba.

8. Instalamos Glogs: es una librería para C++ desarrollada por Google con propósitos de logging.

Después de instalar y configurar todo lo que acabamos de mencionar, procedemos a instalar Wav2letter++. Para ello, descargamos el código de Wav2letter++ del repositorio [39] e instalamos el motor. Wav2letter++ se ejecuta en Docker por lo que instalamos la herramienta Docker en nuestro sistema y, además, como utilizaremos soporte de GPU, también será necesario Nvidia-Docker.

Desde el repositorio de la herramienta nos dan opción a utilizar una imagen Docker que ha sido construida por los desarrolladores o también tenemos la opción de crear la nuestra. En nuestro caso utilizaremos una preconstruida, para ello lanzamos a ejecutar el contenedor de Docker con la imagen que deseamos utilizar (en este caso la última versión con soporte de gráficos), esto hará que comience a descargarse y configurarse la imagen seleccionada. A la hora de lanzar el contenedor añadimos la opción en Docker para que este se inicie cada vez que arranca el sistema operativo.

Una vez finalizado el proceso accedemos al contenedor generado y probamos realizar los tests de las librerías Flashlight y KenLm y los test de Wav2letter++, si hemos realizado bien

el proceso, todos los tests deben ejecutarse si ningún error y por tanto ya tendremos lista la herramienta para empezar a utilizarla.

## 4.4. Pruebas

En esta sección vamos a comentar las pruebas que hemos realizado con la herramienta Wav2letter++ para generar modelos acústicos que puedan resolver nuestro problema de la mejor forma posible.

### 4.4.1. Pruebas con dataset en inglés

En esta fase hemos comenzado utilizando un dataset en idioma inglés de la página web Librispeech, para ello vamos a seguir un tutorial de la página oficial de Wav2letter++ [47], de esta manera podremos comprobar si el motor funciona de forma correcta y aprender a utilizarlo. Empezamos descargando los ficheros de audio de la página junto con sus correspondientes transcripciones, que ya vienen divididos en los conjuntos de entrenamiento, test y validación.

Ejecutamos un script en Python que trae consigo Wav2letter++. Este script permite preparar los datos y dejarlos en un formato adecuado para que el motor pueda reconocerlos, además permite generar el fichero de léxico, el diccionario y las listas de entrenamiento, test y validación. Para este caso el diccionario contiene las letras del alfabeto inglés y el apóstrofe, por lo que los únicos caracteres permitidos serán los contenidos en este fichero. El formato del fichero del diccionario consiste en un conjunto de líneas donde cada línea contiene un único carácter, el cual representa la unidad básica del lenguaje que estamos considerando (grafemas, fonemas, silabas...). En la Figura 65 mostramos el contenido del diccionario para un ejemplo en el que la unidad básica son los grafemas. Para nuestro primer caso la unidad básica son los grafemas por lo que el aspecto de nuestro diccionario es similar al de Figura 65.

```
# tokens.txt
|
'
a
b
c
...
... (and so on)
z
```

*Figura 65 Estructura de un diccionario de grafemas en Wav2letter++ [https://github.com/facebookresearch/wav2letter/wiki/Data-Preparation]*

En cuanto al fichero de léxico, la estructura de este está dividida en un conjunto de líneas donde cada línea contiene una palabra permitida del lenguaje junto con la secuencia de tokens que la conforma separada por una tabulación, los tokens al igual que en el caso del diccionario dependerán de la unidad básica considerada para el lenguaje. Cada token está separado por un

espacio en blanco y la secuencia de tokens finaliza con "|" que indica la separación entre dos palabras. La Figura 66 muestra un ejemplo basado en grafemas de lo que acabamos de explicar.

```
# lexicon.txt
a a |
able a b l e |
about a b o u t |
above a b o v e |
...
hello-kitty h e l l o | k i t t y |
...
... (and so on)
```

*Figura 66 Estructura de un fichero de léxico de grafemas en Wav2letter++ [https://github.com/facebookresearch/wav2letter/wiki/Data-Preparation]*

Con los datos pre-procesados descargamos un modelo de lenguaje de Librispeech que está preconstruido para el lenguaje considerado. El siguiente paso que vamos a dar será seleccionar el tipo de características que vamos a utilizar, en este caso usaremos las MFCC que hemos visto más arriba, para ello en el fichero de configuración para el entrenamiento definimos la etiqueta o flag "--mfcc=true", de igual manera para la función de coste utilizaremos CTC y añadiremos el flag "--ctc=true" en el fichero de configuración del entrenamiento. Otros flags destacables que vamos a utilizar para el entrenamiento son:

- "--lr =0.1": El Learning rate o la tasa de aprendizaje.
- "--batchsize=4": El tamaño de batch o lote a utilizar en el descenso por gradiente.
- "--iter=25": Es el número de iteraciones o epoch que realizará Wav2letter++ durante el proceso de entrenamiento. Un epoch representa haber recorrido todo el conjunto de datos 1 vez y haber hecho el entrenamiento con todos ellos, cada iteración o epoch está compuesta de un número de steps o pasos que se obtienen dividiendo el tamaño del conjunto de entrenamiento entre el tamaño de batch.

Por último, antes de proceder a realizar el primer entrenamiento, definimos la arquitectura de la red neuronal que vamos a utilizar para entrenar y generar el modelo acústico, en este caso vamos a utilizar una arquitectura completamente convolucional, de 8 capas de convolución de 2 dimensiones y como función de activación entre capas, la función ReLu. Por último, en las dos capas finales usaremos capas lineales para obtener la transcripción a nivel de caracteres para las características que se han recibido en la entrada. Estas capas actúan como fully-connected. Podemos ver un resumen de la arquitectura en la Figura 67.

```
V -1 1 NFEAT 0
C2 NFEAT 256 8 1 2 1 -1 -1
R
C2 256 256 8 1 1 1 -1 -1
R
C2 256 256 8 1 1 1 -1 -1
R
C2 256 256 8 1 1 1 -1 -1
R
C2 256 256 8 1 1 1 -1 -1
R
C2 256 256 8 1 1 1 -1 -1
R
C2 256 256 8 1 1 1 -1 -1
R
C2 256 256 8 1 1 1 -1 -1
R
RO 2 0 3 1
L 256 512
R
L 512 NLABEL
```

*Figura 67 Arquitectura de capas convolucionales 2D Wav2letter++*

La capa con el identificador V representa una capa de View que permite transformar las dimensiones de las características de entrada a las correspondientes para ser utilizadas por las capas convolucionales. En cuanto a C2, es una capa de convolución de dos dimensiones y R representa la función de activación ReLu. Por otro lado, R0 es una capa de Reorder, cuya función es bastante similar a la de View. Por último, el identificador L representa una capa lineal.

Con todo esto hecho realizamos el primer entrenamiento y obtenemos los resultados que vemos en la Tabla 3.

| TER-Train(%) | WER-Train(%) | Loss-Train | TER-Val(%) | WER-Val(%) | Loss-Val |
|---|---|---|---|---|---|
| 25.3698 | 41.2125 | 4.04589 | 34.3564 | 52.6358 | 5.64759 |

*Tabla 3 Resultados del primer entrenamiento de Librispeech*

Con el entrenamiento realizado pasamos a realizar el proceso de decodificación, en este caso utilizamos una configuración diferente a la que hemos utilizado en el proceso de entrenamiento, algunos de los parámetros destacables son:

- "--beamsize=500": Representa el número de nodos o estados a mantener en cada paso de beam search.
- "—beamthreshold=25": Representa el valor de umbral para el coste de heurística de cada nodo.

Decidimos hacer una comparativa de resultados entre no utilizar un modelo del lenguaje (lo que Wav2letter++ llama Greedy Search), utilizando un modelo de lenguaje 3-grama, usando un modelo de lenguaje 4-grama. Estos resultados los mostramos en la Tabla 4.

| | Sin modelo de lenguaje | Utilizando un modelo 3-grama | Utilizando un modelo 4-grama |
|---|---|---|---|
| WER (%) | 85.2642 | 35.9632 | 73.3072 |
| LER (%) | 40.1474 | 19.4185 | 26.7172 |

*Tabla 4 Resultados para la decodificación utilizando diferentes modelos del lenguaje*

Lo que nos sorprende a la vista de los resultados es que el modelo de lenguaje 4-grama da peores resultados que el 3-grama, cuando cabría pensar que esto fuese al revés, esto puede ser debido a que para un mismo corpus o base de palabras únicas en el caso del 3-grama el número de valores generados es mucho menor que para un 4-grama, esto provoca que en un 4-grama las probabilidades que se obtienen son mucho más bajas, debido a que las frecuencias de conteo se diluyen. Por esto, a pesar de que en teoría un modelo n-grama mayor debería contener una mayor cantidad de información acerca del contexto de las palabras, si el dataset tiene una cantidad escasa de información o datos es muy difícil que generalice ante nuevos datos por lo que acaba sobre-aprendiendo muy fácilmente.

Otra cosa que hemos observado es que los tiempos de decodificación son mayores cuanto mayor es el valor n del n-grama, algo esperable debido a que un 4-grama contiene un mayor número de combinaciones que un 3-grama, lo que hace que las búsquedas sean cada vez más lentas. También probamos a transformar el modelo de formato ".arpa" a binario, en esta transformación el modelo del lenguaje se transforma en una estructura de datos de KenLM llamada "probing hash table". Los n-gramas son hasheados a enteros de 64 bits, donde dichos enteros actúan como claves de la tabla y como valores actúan las probabilidades, con lo que a priori debería mostrar un incremento en la velocidad de decodificación. Los resultados se muestran en la Tabla 5.

| | Sin modelo | 3-grama (arpa) | 3-grama (binario) | 4-grama (arpa) | 4-grama (binario) |
|---|---|---|---|---|---|
| Tiempo (s) | 115.356 | 217.888 | 194.2697 | 296.023 | 271.741 |

*Tabla 5 Comparativa del tiempo de ejecución con distintos modelos del lenguaje*

Vemos cómo utilizar un modelo de lenguaje en este formato reduce los tiempos de ejecución, pese a incrementar el consumo de memoria. Hemos comprobado que el sistema funciona correctamente, aunque los resultados obtenidos al menos para el idioma inglés no son demasiado buenos, pues las tasas de error son demasiado altas en todos los casos.

### 4.4.2. Pruebas con dataset en español

#### *4.4.2.1. Creación de un modelo del lenguaje*

Pasamos a realizar pruebas con el idioma español, para ello, en primer lugar, creamos un modelo del lenguaje, en este caso utilizamos un corpus del idioma español con aproximadamente 1.5 billones de palabras. El corpus está formado por textos y artículos de la página Wikipedia.

Transformamos la información de los textos extrayendo las palabras que los forman y pasándolas a minúsculas para ello utilizamos un script en Python generado por nosotros. Por último, utilizamos la librería KenLM para generar el modelo del lenguaje con los datos que hemos mencionado. El proceso tarda aproximadamente 6 horas en finalizar y a la salida obtenemos el modelo del lenguaje en formato ".arpa", junto con la salida que muestra la Figura 68.

```
=== 1/5 Counting and sorting n-grams ===
File stdin isn't normal.  Using slower read() instead of mmap().  No
progress bar.
df -h
Unigram tokens 1420671095 types 3244633
=== 2/5 Calculating and sorting adjusted counts ===
Chain sizes: 1:38935596 2:4578600960 3:8584877056
Statistics:
1 3244633 D1=0.672608 D2=1.03849 D3+=1.35909
2 70231511 D1=0.753828 D2=1.08483 D3+=1.3645
3 282239777 D1=0.761612 D2=1.14383 D3+=1.37715
Memory estimate for binary LM:
type    MB
probing 6532 assuming -p 1.5
probing 6947 assuming -r models -p 1.5
trie    2811 without quantization
trie    1644 assuming -q 8 -b 8 quantization
trie    2644 assuming -a 22 array pointer compression
trie    1477 assuming -a 22 -q 8 -b 8 array pointer compression and
quantization
=== 3/5 Calculating and sorting initial probabilities ===
Chain sizes: 1:38935596 2:1123704176 3:5644795540
----5---10---15---20---25---30---35---40---45---50---55---60---65---70---7
5---80---85---90---95--100
####################################################################
################################################
=== 4/5 Calculating and writing order-interpolated probabilities ===
Chain sizes: 1:38935596 2:1123704176 3:5644795540
----5---10---15---20---25---30---35---40---45---50---55---60---65---70---7
5---80---85---90---95--100
####################################################################
################################################
=== 5/5 Writing ARPA model ===
----5---10---15---20---25---30---35---40---45---50---55---60---65---70---7
5---80---85---90---95--100
****************************************************************************************************
****************************
Name:lmplz    VmPeak:13287440 kB    VmRSS:5600 kB
RSSMax:9117556 kB    user:901.094    sys:74.0663    CPU:975.161
real:21931.4
```

*Figura 68 Salida de la librería KenLM para la generación del modelo de lenguaje en español*

Lo siguiente que realizamos es transformar el fichero ".arpa" a formato binario, pues, como hemos visto esto nos permite mejorar los tiempos de decodificación a costa de tener un mayor coste de memoria durante la decodificación.

#### *4.4.2.2. Pre-procesamiento y pruebas utilizando el dataset de Kaggle y la arquitectura de capas convolucionales 2D para el entrenamiento del modelo*

Para la fase de pruebas utilizamos un dataset de la página Kaggle [48], el cual consiste en un conjunto de audios en español grabados por una misma persona que lee frases del Quijote. Antes de entrenar el modelo, generamos un notebook en Python para realizar un pre-procesamiento de los datos (tanto de los audios como de sus transcripciones) para ello el notebook:

1. Pondrá el fichero de transcripciones en formato correcto para que el entrenamiento pueda:

1.1. Eliminar símbolos y caracteres no contemplados en el diccionario del lenguaje como: ., :, ¡,!...

1.2. Realizar un tratamiento especial con las palabras compuestas con guion. Se eliminarán los guiones que puedan aparecer al principio o al final de una palabra, sin embargo, se mantienen aquellos que aparecen en medio de una palabra compuesta como: hispano-argentino o Sierra-Morena.

1.3. Eliminar datos incompletos como audios sin su correspondiente transcripción.

1.4. Eliminar transcripciones duplicadas en el fichero.

2. Realizará el particionamiento de la lista de transcripciones en 3 conjuntos de datos: Entrenamiento, Validación y Test:

2.1. Para el particionamiento, como se cuenta con pocas muestras (aproximadamente 11.000), se aplicará un particionamiento Hold-out siguiendo la regla típica de 60/20/20.

3. Almacenará las listas generadas en 3 ficheros distintos dentro de una ruta:

3.1. Cada fichero tendrá el formato apropiado para que Wav2letter++ pueda realizar el entrenamiento y la decodificación de forma correcta.

3.2. Las rutas de almacenamiento de los ficheros serán generadas durante la fase de pre-procesamiento.

4. Generará los ficheros de léxico y el diccionario que se utilizarán para el entrenamiento y la decodificación:

4.1. El diccionario contendrá todas las letras del alfabeto español incluyendo la ñ, además incluirá todas las vocales con tilde, además del símbolo "Ç" y el guion.

4.2. El fichero de léxico contendrá todas las palabras tratadas observadas en el fichero de transcripciones (sin repetición) durante la fase de pre-procesamiento, junto con sus respectivas secuencias de tokens.

4.3. En el fichero de léxico las palabras compuestas separadas por guion en su parte de grafemas se dividirán en dos palabras independientes utilizando para ello el separador "|".

4.4. En el fichero de léxico no habrá posibilidad de palabras vacías.

Con el pre-procesamiento realizado pasamos a realizar el primer entrenamiento. Para ello utilizaremos en primer lugar, la misma configuración que en el caso del idioma inglés. En cuanto al resto de los hiper-parámetros, los ajustaremos más adelante. En lo que respecta a la arquitectura de la red neuronal, probaremos primero con la arquitectura que tenemos en la

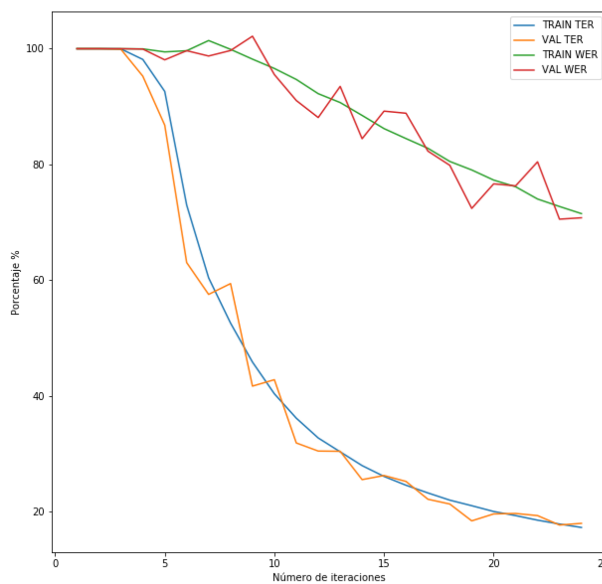
Figura 67 para comprobar su funcionamiento para el idioma español, aunque también buscaremos probar otras basadas en RNN, LSTM o GRU.

Realizamos un primer entrenamiento y obtenemos los resultados que muestra la Tabla 6.

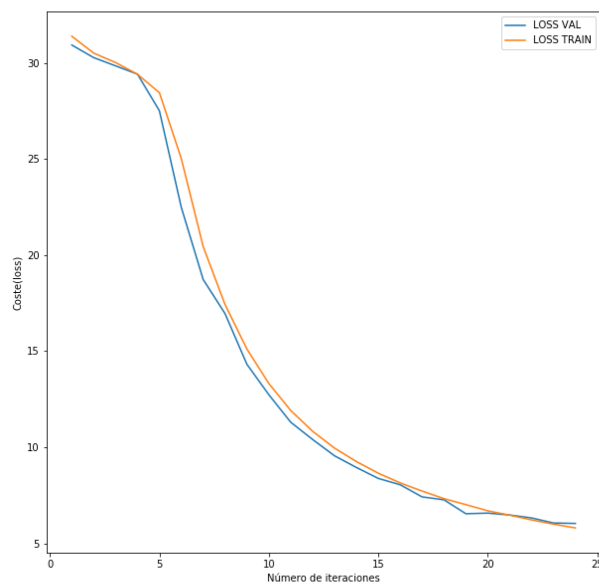
| TER-Train (%) | WER-Train (%) | Loss-Train | TER-Val (%) | WER-Val (%) | Loss-Val |
|---|---|---|---|---|---|
| 17.28 | 71.51 | 5.8401 | 17.99 | 70.79 | 6.03618 |

*Tabla 6 Resultados configuración por defecto*

Creamos un notebook en Python con el que poder realizar un análisis de los resultados obtenidos en la fase de entrenamiento. Mostramos de forma gráfica los resultados en la Gráfica 1 y la Gráfica 2.




*Gráfica 1 Comparativa TER y WER en entrenamiento y validación para configuración por defecto*




*Gráfica 2 Comparativa coste (loss) entre entrenamiento y validación para configuración por defecto*

A la vista de los resultados, podemos deducir que el modelo tiene un problema de bias ya que los valores de WER son muy altos para entrenamiento, más de lo que cabría esperar. Para reducir el problema de bias podemos utilizar un arquitectura diferente o más grande, entrenar durante mucho más tiempo o utilizar otra configuración de hiper-parámetros.

Aún con los resultados obtenidos probamos a realizar una decodificación utilizando el modelo acústico generado, así como el modelo del lenguaje que hemos creado antes. Los resultados que mostramos en la Tabla 7 no son nada buenos la tasa de WER resulta demasiado alta.

| WER (%) | TER (%) | Tiempo/muestra (s) |
|---|---|---|
| 19.2064 | 8.40756 | 0.498 |

*Tabla 7 Resultados decodificación con configuración por defecto*

Probamos a realizar un refinamiento de los hiper-parámetros utilizados en la fase de entrenamiento para ver si conseguimos mejorar los resultados. En primer lugar, comenzamos modificando los siguientes flags de la configuración:

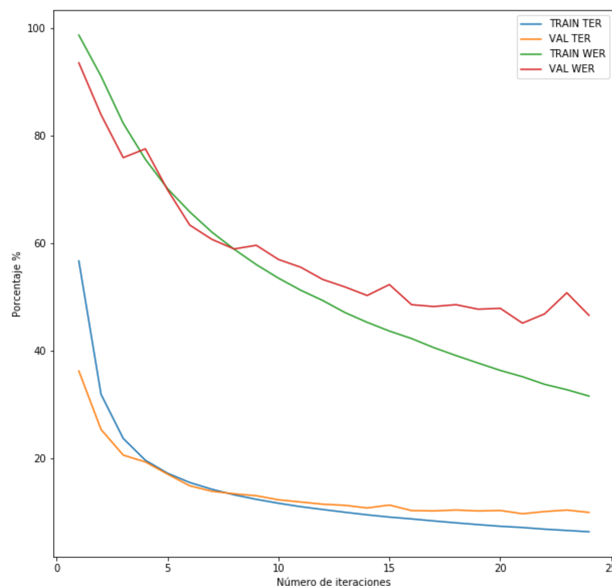
1. Para el tamaño de batch, utilizaremos el más grande que permita nuestro sistema, en este caso para este dataset es 24.

2. Incrementamos el número de hilos responsables de la ejecución del entrenamiento hasta 8 (el máximo del sistema) para acelerar el rendimiento.

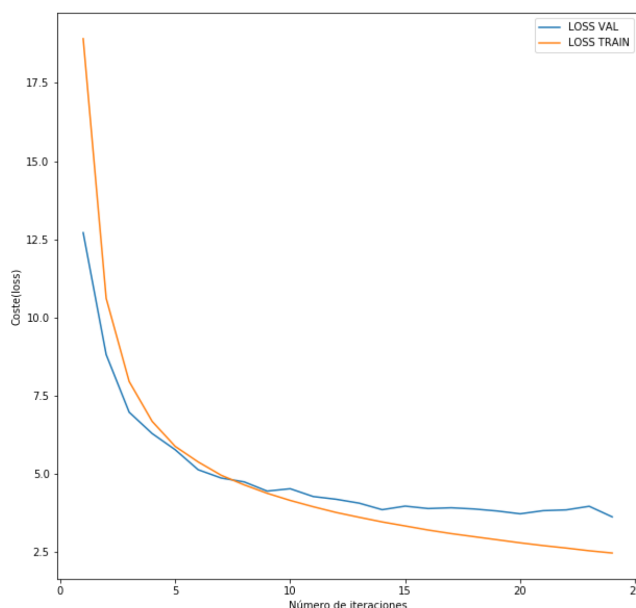
Los resultados que se obtienen son los que muestra la Tabla 8, la Gráfica 3 y la Gráfica 4.

| TER-Train (%) | WER-Train (%) | Loss-Train | TER-Val (%) | WER-Val (%) | Loss-Val |
|---|---|---|---|---|---|
| 6.27 | 31.53 | 2.46223 | 9.87 | 46.56 | 3.61879 |

*Tabla 8 Resultados configuración por defecto y batch 24*




*Gráfica 3 Comparativa TER y WER para entrenamiento y validación con batch 24*




*Gráfica 4 Comparativa coste (loss) entre entrenamiento y validación para batch 24*

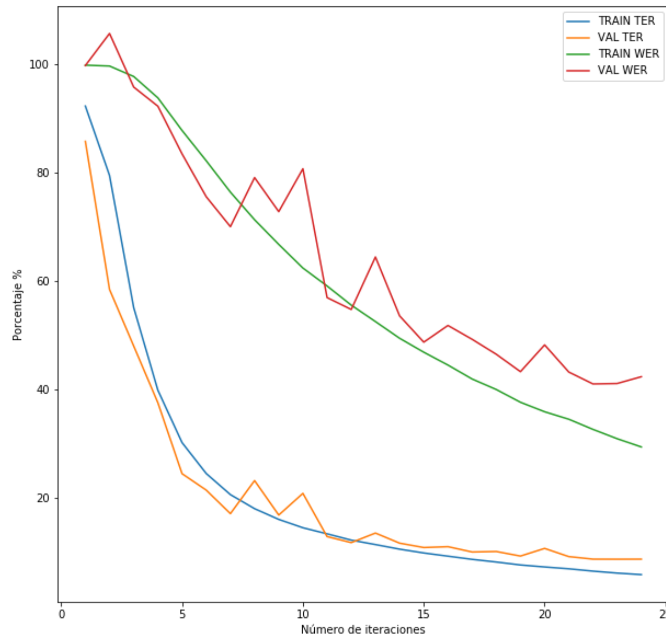
Los resultados obtenidos muestran como el problema de bias está comenzando a desaparecer y por otro lado aparece un problema de varianza pues como vemos comienza a haber una diferencia notable entre el WER y el coste (loss) en validación y entrenamiento.

Trataremos de ajustar el valor de la tasa de aprendizaje para mejorar los resultados que hemos obtenido acelerando la velocidad de convergencia, ya que según pasan las iteraciones vemos como el porcentaje de error y el valor de coste (loss) mejora, pero lo hacen de una forma muy lenta. Para ello probamos con varios valores del flag "--lr" como muestra la Tabla 9.

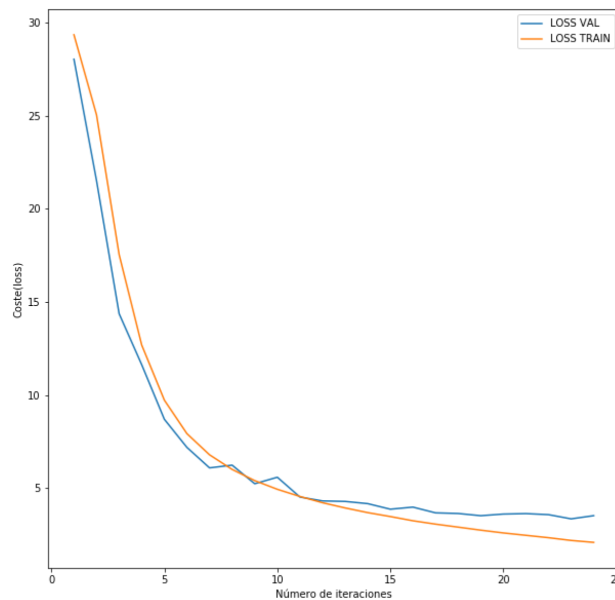
| Tasa de aprendizaje | TER-Train (%) | WER-Train (%) | Loss-Train | TER-Val (%) | WER-Val (%) | Loss-Val |
|---|---|---|---|---|---|---|
| 0.2 | 6.01 | 30.02 | 2.34178 | 9.15 | 44.82 | 3.56419 |
| 0.3 | 5.57 | 29.63 | 2.15426 | 8.74 | 43.02 | 3.53610 |
| 0.42 | 5.79 | 29.33 | 2.08131 | 8.64 | 42.28 | 3.51697 |

*Tabla 9 Resultados para diferentes valores de tasa de aprendizaje*

Observamos que la tasa de aprendizaje que mejores resultados da es 0.42, sin embargo, seguimos viendo un claro problema de bias y varianza (Gráfica 5 y Gráfica 6). No obstante, probamos a realizar una decodificación para ver si los resultados han mejorado respecto al caso inicial.




*Gráfica 5 Comparativa TER y WER para entrenamiento y validación con batch 24 y tasa de aprendizaje 0.42*




*Gráfica 6 Comparativa coste (loss) entre entrenamiento y validación para batch 24 y tasa de aprendizaje 0.42*

Como podemos ver en la Tabla 10 los resultados a priori parecen ser bastante buenos, por lo que decidimos grabar un pequeño dataset de 10 audios con posibles comandos que se utilizarán en un entorno real (en quirófano), para ver los resultados del modelo acústico.

| WER (%) | TER (%) | Tiempo/muestra (s) |
|---|---|---|
| 9.44916 | 3.92132 | 0.366 |

*Tabla 10 Decodificación para batch 24 y tasa de aprendizaje 0.42*

En el proceso de decodificación nos detecta que ciertas palabras no están contempladas en el léxico del lenguaje (Figura 69), por lo que las añadimos al fichero de léxico y volvemos a realizar la prueba.

```
Skipping unknown entry: 'comprobación'
Skipping unknown entry: 'paciente'
Skipping unknown entry: 'quirúrgico'
Skipping unknown entry: 'comfirmación'
Skipping unknown entry: 'paciente'
Skipping unknown entry: 'quirúrgico'
Skipping unknown entry: 'comprobación'
Skipping unknown entry: 'anestesia'
Skipping unknown entry: 'comprobación'
Skipping unknown entry: 'quirúrgico'
Skipping unknown entry: 'comprobación'Skipping unknown entry: 'comprobación'
Skipping unknown entry: 'sangrado'

Skipping unknown entry: 'pretransfusionales'
Skipping unknown entry: 'previas'
Skipping unknown entry: 'comfirmación'
Skipping unknown entry: 'paciente'
Skipping unknown entry: 'identidad'
Skipping unknown entry: 'paciente'
Skipping unknown entry: 'quirófano'
Skipping unknown entry: 'cumplimentado'
Falling back to using letters as targets for the unknown word: contaje
Falling back to using letters as targets for the unknown word: instrumental
Skipping unknown entry: 'contaje'
Skipping unknown entry: 'instrumental'
```

*Figura 69 Palabras no detectadas en el fichero de léxico del lenguaje*

Los resultados que obtenemos son realmente malos como observamos en la Tabla 11. Esto puede ser debido a tres razones: la primera es que, como hemos visto en la Figura 69, inicialmente en el proceso de decodificación se desconocen varias palabras que aparecen en las transcripciones, lo que implica que durante el proceso de entrenamiento estas palabras nos han aparecido en ningún momento, esto puede provocar que al escucharlas por primera vez en la decodificación el modelo falle en exceso. El segundo motivo, que consideramos de gran peso, es que, como comentábamos, tenemos un problema de bias y varianza, es decir, nuestro modelo no se ajustar bien a los datos de entrenamiento, pero es que además, no generaliza bien para los datos nuevos. La tercera razón es que el dataset con el que estamos realizando el entrenamiento es muy poco diverso, pues la voz en todos los audios corresponde a la misma persona, esto provoca que cuando el modelo recibe un audio de otra persona con un acento diferente falle más de lo que debería.

| WER (%) | TER (%) | Tiempo/muestra (s) |
|---|---|---|
| 122.642 | 78.1473 | 0.448 |

*Tabla 11 Resultados decodificación con comandos personalizados*

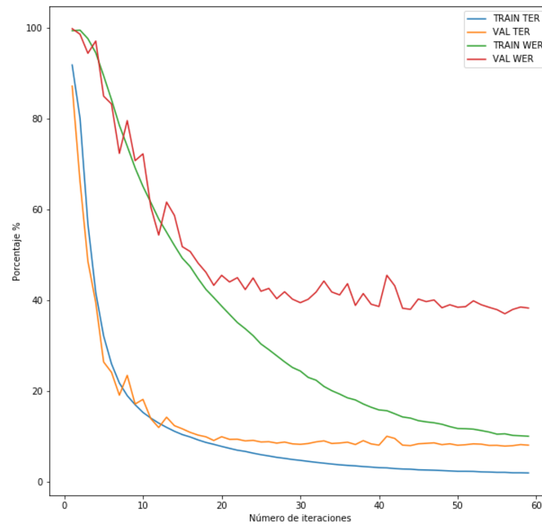
Vamos a tratar de resolver los problemas de bias y varianza antes de probar otras alternativas como la ampliación del dataset. En primer lugar, probaremos a aumentar el número de iteraciones del entrenamiento, pues como hemos visto tanto el coste como el error tienden a disminuir conforme pasan las iteraciones, pero por los resultados que obtenemos vemos como 25 iteraciones resultan insuficientes. En la Tabla 12 mostramos los resultados para 40, 50 y 60 iteraciones.

| Iteraciones | TER-Train (%) | WER-Train (%) | Loss-Train | TER-Val (%) | WER-Val (%) | Loss-Val |
|---|---|---|---|---|---|---|
| 40 | 3.28 | 17.17 | 1.18632 | 9.10 | 41.49 | 4.02198 |
| 50 | 2.43 | 12.19 | 0.80792 | 8.40 | 39.02 | 4.64225 |
| 60 | 1.98 | 10.06 | 0.63557 | 8.10 | 38.30 | 4.60894 |

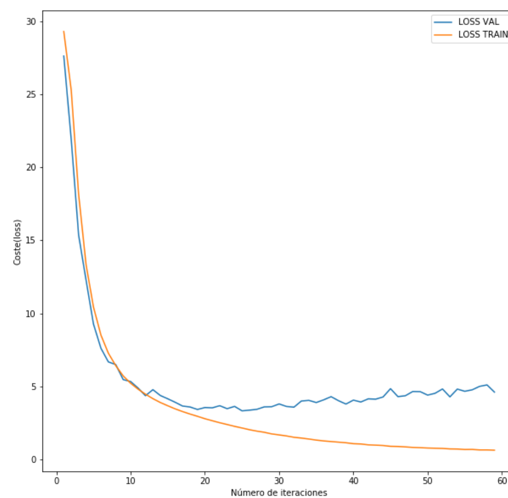
*Tabla 12 Comparación de resultados con diferentes cantidades de iteraciones*

Comprobamos por un lado que el problema de bias tiende a desaparecer conforme pasan las iteraciones, algo que ya esperábamos pues como vemos tanto el valor de coste como el valor

de los errores desciende de forma notable con el paso de las iteraciones. Sin embargo, algo que nos llama mucho la atención es que vemos una reducción en el WER de validación conforme pasan las iteraciones, pero por otro lado vamos que el valor de la función de coste no disminuye si no que más bien aumenta de 3.51697 que teníamos para 25 iteraciones (Tabla 9) a 4.60894, esto puede indicar que nuestra tasa de aprendizaje es muy alta y que a partir de un cierto número de iteraciones debemos reducirla o bien que nuestro modelo está sobre-aprendiendo, algo que es bastante probable viendo la gran diferencia entre el coste y el error de entrenamiento y validación. Decidimos analizar gráficamente (Gráfica 7 y Gráfica 8) los resultados para intentar visualizar los problemas que estamos teniendo.




*Gráfica 7 Comparativa TER y WER para entrenamiento y validación con batch 24 y tasa de aprendizaje 0.42 y 60 iteraciones*




*Gráfica 8 Comparativa coste (loss) entre entrenamiento y validación para batch 24 y tasa de aprendizaje 0.42 y 60 iteraciones*

Observamos en la Gráfica 7 como a partir del epoch 20 el TER para validación comienza a incrementarse mientras que el WER está continuamente incrementando y disminuyendo, de forma similar en la Gráfica 8 a partir de la epoch 20 el coste para validación comienza a incrementar, mientras que el de entrenamiento sigue disminuyendo. Probaremos a realizar una decodificación para ver si los resultados mejoran con respecto a la Tabla 10 en la que realizábamos 25 iteraciones. Los resultados de la decodificación los mostramos en la Tabla 13,

como esperábamos los resultados han empeorado, por lo que nuestro modelo parece estar sobre-aprendiendo ya que, aunque el error durante la fase de entrenamiento disminuye sustancialmente, el modelo generaliza mal ante nuevos ejemplos dando un peor rendimiento en test.

| WER (%) | TER (%) | Tiempo/muestra (s) |
|---|---|---|
| 11.2701 | 4.37019 | 0.297 |

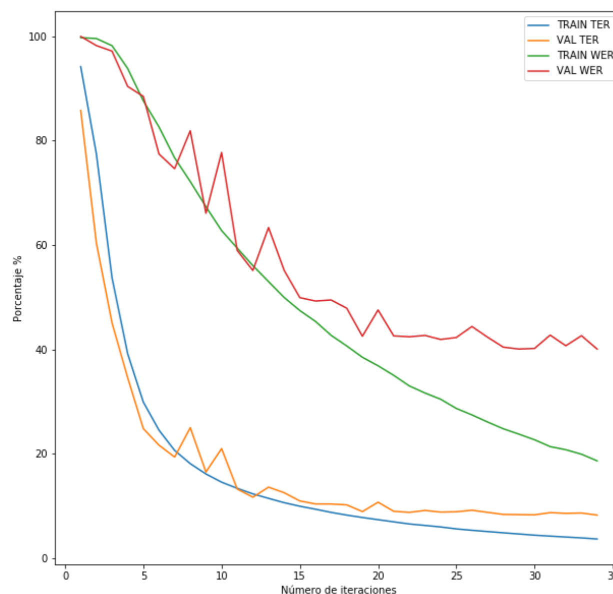
*Tabla 13 Resultados decodificación para 60 iteraciones*

Nos planteamos la posibilidad de que este problema también esté debido a que la tasa de aprendizaje sea demasiado alta y que a partir de la fase 20 sea necesario disminuir su valor, por lo que probamos a entrenar el modelo durante 20 epoch con una tasa de aprendizaje de 0.42 y a partir de ahí reducir la tasa y entrenar el modelo 10 iteraciones más para comprobar si se soluciona el problema. Para ello probamos con los valores de tasa de aprendizaje de la Tabla 14.

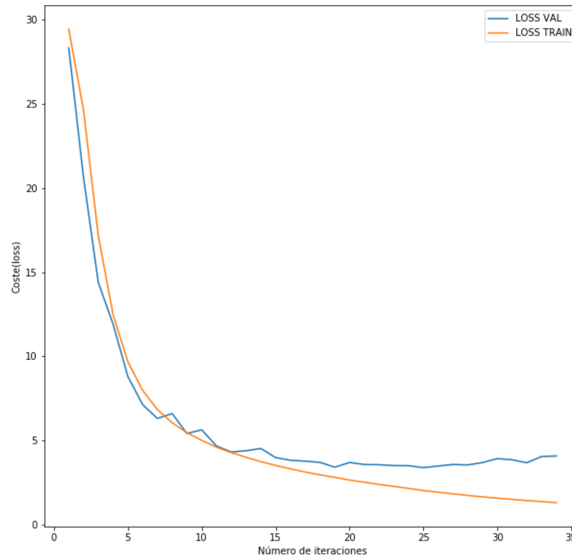
| Tasa de aprendizaje | TER-Train (%) | WER-Train (%) | Loss-Train | TER-Val (%) | WER-Val (%) | Loss-Val |
|---|---|---|---|---|---|---|
| 0.2 | 3.75 | 19.02 | 1.30988 | 8.37 | 41.48 | 4.06611 |
| 0.1 | 3.72 | 18.79 | 1.30427 | 8.43 | 39.86 | 4.08428 |
| 0.01 | 3.70 | 18.67 | 1.29943 | 8.28 | 40.08 | 4.07716 |

*Tabla 14 Resultados después de cambiar la tasa de aprendizaje a partir del epoch 20 y de entrenar 15 epoch más*

Vemos como sigue ocurriendo lo mismo aunque no de una manera tan pronunciada, los valores de error siguen disminuyendo mientras que el valor de la función de coste para validación sigue aumentando, mostramos además los resultados en la Gráfica 9 y la Gráfica 10. Obviamos representar los resultados para el resto de valores pues su representación es prácticamente la misma.




*Gráfica 9 Comparativa TER y WER para entrenamiento y validación cambiando la tasa de aprendizaje a 0.01 a partir de epoch 20*




*Gráfica 10 Comparativa coste (loss) entre entrenamiento y validación cambiando la tasa de aprendizaje a 0.01 a partir de epoch 20*

Comprobamos además el resultado de la decodificación para cada uno de estos modelos generados, para ver si mejoran los resultados de la Tabla 13. Como vemos en la Tabla 15 los resultados tienden a ser mejores que en la Tabla 13 pero siguen siendo peores que los de la Tabla 10 (de los que partíamos), esto puede ser debido a que estos modelos solo se han ejecutado 35 epoch y no 60 como en el caso de la Tabla 13, por lo que el modelo ha sobre-aprendido pero durante menos tiempo.

| Tasa de aprendizaje | WER (%) | TER (%) | Tiempo/muestra (s) |
|---|---|---|---|
| 0.2 | 10.59641 | 4.15478 | 0.326 |
| 0.1 | 10.00247 | 3.98714 | 0.338 |
| 0.01 | 10.21789 | 4.01587 | 0.335 |

*Tabla 15 Comparativa resultados decodificación modificando las tasas de aprendizaje a partir de epoch 20*

#### *4.4.2.3. Entrenamiento utilizando medidas de regularización para el dataset de Kaggle y la arquitectura de capas convolucionales de 2 dimensiones*

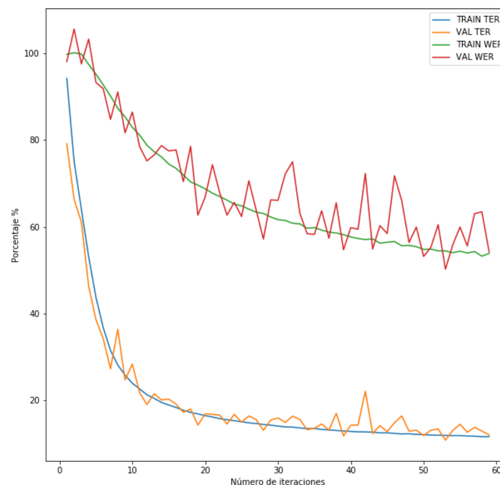
Como hemos visto el cambio en el valor de la de tasa de aprendizaje durante el entrenamiento no nos proporciona prácticamente ninguna mejora, por lo que pasaremos a utilizar técnicas de regularización. En primer lugar, vamos a probar a entrenar el modelo utilizando regularización l2, para ello utilizaremos el flag "--weightdecay" de Wav2letter++. Probamos diferentes valores de regularización para una tasa de aprendizaje de 0.42, un número de iteraciones de 35 y un tamaño de batch de 24. Los resultados se muestran en la Tabla 16.

| Valores de regularización | TER-Train (%) | WER-Train (%) | Loss-Train | TER-Val (%) | WER-Val (%) | Loss-Val |
|---|---|---|---|---|---|---|
| 0.1 | 99.09 | 99.96 | 64.09473 | 99.10 | 99.96 | 63.35076 |
| 0.01 | 100.0 | 100.0 | 31.55224 | 100.0 | 100.0 | 30.47775 |
| 0.001 | 13.51 | 59.64 | 4.71930 | 13.25 | 58.40 | 4.84289 |

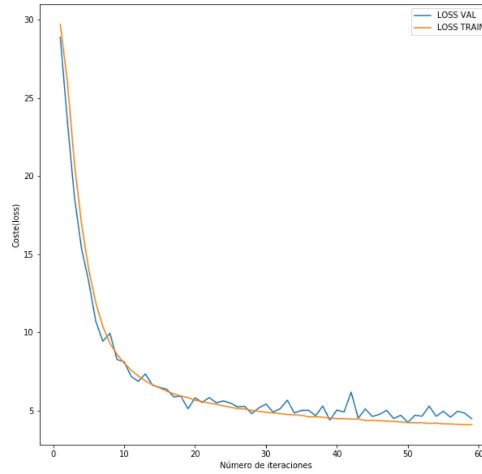
*Tabla 16 Comparativa de rendimiento para diferentes valores de regularización (l2)*

Observamos como los valores de regularización 0.1 y 0.01 resultan ser demasiado altos para nuestro problema, pues como vemos, aunque la diferencia en coste y error entre entrenamiento y validación es prácticamente nula (lo que indica que el problema de sobre-aprendizaje ha desaparecido), comprobamos que los valores son demasiado altos por lo que el modelo ahora está generalizando en exceso y ni siquiera se está ajustando a los datos de entrenamiento. Por otro lado, el valor 0.001 parece que empieza a dar mejores que los otros dos, pues aparentemente el problema de sobre-aprendizaje ha desaparecido y los valores de coste y error son mucho más bajos que en los otros dos casos.

Probamos a entrenar el modelo durante un mayor número de iteraciones con una regularización de 0.001 para intentar que el modelo converja y obtener mejores resultados durante la decodificación que en la Tabla 10. El proceso de aprendizaje se muestra en la Gráfica 11 y en la Gráfica 12, comprobamos como los valores de coste y error siguen disminuyendo a partir del epoch 35 pero fluctúan mucho y provocan los picos que vemos en las gráficas.




*Gráfica 11 Comparativa TER y WER para entrenamiento y validación con regularización 0.001 para 60 iteraciones*




*Gráfica 12 Comparativa coste (loss) entre entrenamiento y validación para regularización de 0.001 y 60 iteraciones*

En vista de estos resultados probamos a realizar una decodificación (Tabla 17) y comprobamos como empeoran respecto a la Tabla 13.

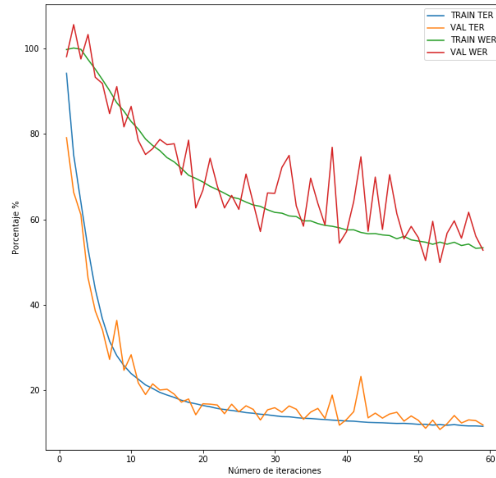
| WER (%) | TER (%) | Tiempo/muestra (s) |
|---|---|---|
| 15.986 | 7.10322 | 0.682 |

*Tabla 17 Resultados decodificación para regularización 0.001 y 60 iteraciones*

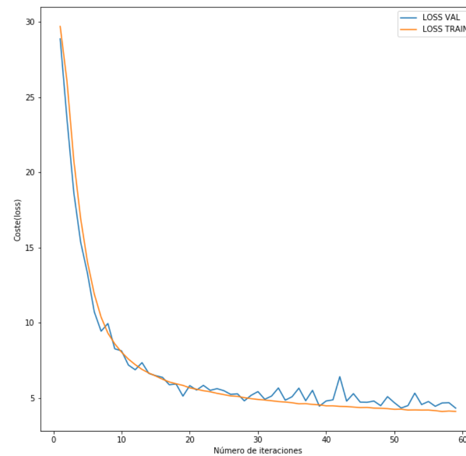
Nos planteamos utilizar a partir del epoch 35 una tasa de aprendizaje más pequeña que 0.42 como hicimos en la Tabla 14, para ver si mejoramos el rendimiento del modelo. Tras realizar las pruebas obtenemos los resultados de la Tabla 18. Además en la Gráfica 13 y en la Gráfica 14, mostramos el proceso de aprendizaje para un cambio de tasa de 0.42 a 0.01 (las demás gráficas son prácticamente idénticas por lo que decidimos no mostrarlas). En estas gráficas vemos que los resultados son bastante similares a los de la Gráfica 11 y la Gráfica 12, sin embargo gracias a la regularización vemos como ya no tenemos el problema de sobre-aprendizaje que se veía en la Gráfica 9 y en la Gráfica 10.

| Tasa de aprendizaje | TER-Train (%) | WER-Train (%) | Loss-Train | TER-Val (%) | WER-Val (%) | Loss-Val |
|---|---|---|---|---|---|---|
| 0.2 | 12.17 | 56.02 | 4.21890 | 13.02 | 55.21 | 4.44012 |
| 0.1 | 11.71 | 54.07 | 4.09532 | 11.77 | 53.29 | 4.32164 |
| 0.01 | 11.61 | 53.56 | 4.09765 | 11.92 | 52.79 | 4.31151 |

*Tabla 18 Resultados para regularización de 0.001 reduciendo la tasa de aprendizaje a partir del epoch 35*




*Gráfica 13 Comparativa TER y WER para entrenamiento y validación con regularización 0.001 para 60 iteraciones cambiando la tasa de aprendizaje a 0.01 en epoch 35*




*Gráfica 14 Comparativa coste (loss) entre entrenamiento y validación para regularización de 0.001 y 60 iteraciones cambiando la tasa de aprendizaje a 0.01 en epoch 35*

Comprobamos además los resultados al decodificar (Tabla 19) para cada uno de los casos de la Tabla 18. Vemos como los resultados siguen siendo peores que en el caso inicial (Tabla 10), pero es que son incluso peores que antes de aplicar la regularización l2 “weightdecay” (Tabla 15), lo que puede ser debido a que esta regularización no funciona correctamente para nuestro problema o a que el valor de regularización es bastante elevado.

| Tasa de aprendizaje | WER (%) | TER (%) | Tiempo/muestra (s) |
|---|---|---|---|
| 0.2 | 18.0401 | 8.14301 | 0.547 |
| 0.1 | 16.2537 | 7.30382 | 0.702 |
| 0.01 | 15.5166 | 7.12104 | 0.681 |

*Tabla 19 Resultados decodificación para regularización 0.001, 60 iteraciones y disminuyendo la tasa de aprendizaje a partir del epoch 35*

Nos planteamos probar otro tipo de regularización, en este caso dropout para ver si conseguimos mejorar los resultados respecto a utilizar regularización l2, para ello debemos modificar la arquitectura de nuestra red convolucional. En la primera aproximación vamos a

probar a aplicar dropout con diferentes probabilidades antes de cada una de las capas fully-connected (las dos capas lineales del final) siguiendo los consejos de algunas páginas web y artículos científicos [49], [50]. La arquitectura una vez modificada tiene el aspecto de la Figura 70.

```
(0): View (-1 1 40 0)
(1): Conv2D (40->256, 8x1, 2,1, SAME,SAME, 1, 1) (with bias)
(2): ReLU
(3): Conv2D (256->256, 8x1, 1,1, SAME,SAME, 1, 1) (with bias)
(4): ReLU
(5): Conv2D (256->256, 8x1, 1,1, SAME,SAME, 1, 1) (with bias)
(6): ReLU
(7): Conv2D (256->256, 8x1, 1,1, SAME,SAME, 1, 1) (with bias)
(8): ReLU
(9): Conv2D (256->256, 8x1, 1,1, SAME,SAME, 1, 1) (with bias)
(10): ReLU
(11): Conv2D (256->256, 8x1, 1,1, SAME,SAME, 1, 1) (with bias)
(12): ReLU
(13): Conv2D (256->256, 8x1, 1,1, SAME,SAME, 1, 1) (with bias)
(14): ReLU
(15): Conv2D (256->256, 8x1, 1,1, SAME,SAME, 1, 1) (with bias)
(16): ReLU
(17): Reorder (2,0,3,1)
(18): Dropout (0.500000)
(19): Linear (256->512) (with bias)
(20): ReLU
(21): Dropout (0.500000)
(22): Linear (512->37) (with bias)
```

*Figura 70 Arquitectura de capas convolucionales de 2 dimensiones con dropout en las capas fully-connected*

Los resultados para diferentes probabilidades de dropout, en 35 epoch, con tasa de aprendizaje de 0.42 y un batch de tamaño 24 se muestran en la Tabla 20. Comprobamos que el valor de dropout que mejores resultados de da es 0.2 no solo durante las fases de entrenamiento y validación, sino también durante test o decodificación (Tabla 21). Vemos como utilizando dropout por fin podemos mejorar los resultados que obtuvimos en la Tabla 10 para la fase de decodificación.

| Probabilidad de dropout | TER-Train (%) | WER-Train (%) | Loss-Train | TER-Val (%) | WER-Val (%) | Loss-Val |
|---|---|---|---|---|---|---|
| 0.5 | 7.24 | 34.56 | 2.80214 | 9.05 | 41.84 | 3.60140 |
| 0.2 | 3.58 | 17.71 | 1.27682 | 8.16 | 38.08 | 4.25618 |
| 0.1 | 6.81 | 33.63 | 2.44292 | 9.08 | 42.26 | 3.65462 |

*Tabla 20 Resultados del entrenamiento y validación aplicando dropout con diferentes probabilidades antes de las capas fully-connected*

| Probabilidad de dropout | WER (%) | TER (%) | Tiempo/muestra (s) |
|---|---|---|---|
| 0.5 | 8.60031 | 3.69336 | 0.391 |
| 0.2 | 8.39494 | 3.47452 | 0.278 |
| 0.1 | 9.15 | 3.93624 | 0.419 |

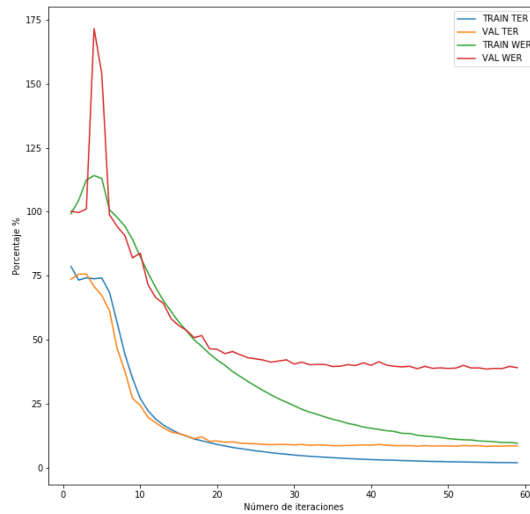
*Tabla 21 Resultados decodificación para diferentes valores de dropout en las capas fully-connected*

Puesto que hemos conseguido mejorar los resultados iniciales, volvemos a intentar decodificar el conjunto de datos que contiene posibles comandos que utilizaremos en un entorno real, por desgracia vemos que los resultados siguen sin mejorar (Tabla 22).

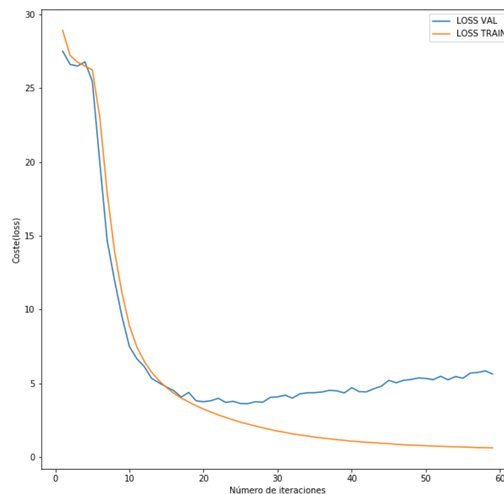
| WER (%) | TER (%) | Tiempo/muestra (s) |
|---|---|---|
| 160.377 | 74.3468 | 0.639 |

*Tabla 22 Rendimiento del modelo utilizando dropout de 0.2 en las capas fully-connected para el dataset de comandos reales*

Vamos a probar a entrenar el modelo durante un número mayor de iteraciones para comprobar si los resultados mejoran con un dropout de 0.2, entrenaremos el modelo durante 60 epoch. Mostramos el proceso de entrenamiento en la Gráfica 15 y en la Gráfica 16, donde vemos como nuevamente vuelve a aparecer un claro problema de sobre-aprendizaje a partir del epoch 20.




*Gráfica 15 Comparativa TER y WER para entrenamiento y validación con dropout de 0.2 en las capas fully-connected en 60 epoch*




*Gráfica 16 Comparativa coste (loss) entre entrenamiento y validación con dropout de 0.2 en las capas fully-connected en 60 epoch*

Para intentar solucionar el problema de la Gráfica 15 y de la Gráfica 16, vamos a probar a realizar una segunda aproximación utilizando dropout, esta vez utilizaremos dropout en las capas convolucionales después de aplicar la función de activación además de utilizarlo en las fully-connected, para ello nuevamente debemos modificar la arquitectura, el aspecto de la nueva arquitectura es el de la Figura 71.

```
V -1 1 NFEAT 0
C2 NFEAT 256 8 1 2 1 -1 -1
R
DO 0.2
C2 256 256 8 1 1 1 -1 -1
R
DO 0.2
C2 256 256 8 1 1 1 -1 -1
R
DO 0.2
C2 256 256 8 1 1 1 -1 -1
R
DO 0.2
C2 256 256 8 1 1 1 -1 -1
R
DO 0.2
C2 256 256 8 1 1 1 -1 -1
R
DO 0.2
C2 256 256 8 1 1 1 -1 -1
R
DO 0.2
C2 256 256 8 1 1 1 -1 -1
R
DO 0.2
RO 2 0 3 1
DO 0.2
L 256 512
R
DO 0.2
L 512 NLABEL
```

*Figura 71 Arquitectura de capas convolucionales de 2 dimensiones aplicando dropout en las capas convolucionales y las fully-connected*

Realizamos múltiples pruebas con diferentes valores de dropout y documentamos los resultados tanto para entrenamiento y validación (Tabla 23) como para decodificación (Tabla 24). Debemos mencionar que durante las pruebas para valores de probabilidad mayores a 0.2 el entrenamiento fallaba debido al problema de vanishing gradients, debido a esto los valores de coste tomaban el valor NaN o “Not a Number”.

| Probabilidad de dropout | TER-Train (%) | WER-Train (%) | Loss-Train | TER-Val (%) | WER-Val (%) | Loss-Val |
|---|---|---|---|---|---|---|
| 0.2 | 5.77 | 27.44 | 2.29658 | 5.49 | 25.45 | 2.41017 |
| 0.1 | 3.83 | 18.33 | 1.52126 | 5.64 | 26.05 | 2.88618 |
| 0.05 | 3.24 | 15.10 | 1.26872 | 6.14 | 28.46 | 3.24212 |

*Tabla 23  Resultados del entrenamiento y validación aplicando dropout con diferentes probabilidades antes de las capas fully-connected y entre capas convolucionales para 35 epoch*

| Probabilidad de dropout | WER (%) | TER (%) | Tiempo/muestra (s) |
|---|---|---|---|
| 0.2 | 7.03039 | 3.01611 | 0.408 |
| 0.1 | 7.25014 | 3.07148 | 0.414 |
| 0.05 | 7.34985 | 3.13506 | 0.292 |

*Tabla 24 Resultados decodificación para diferentes valores de dropout en las capas fully-connected y convolucionales*

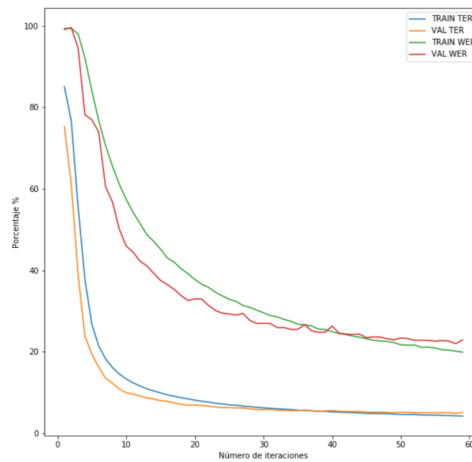
Comprobamos como los resultados han mejorado durante la fase de entrenamiento, validación y test al añadir el dropout en las capas convolucionales, por lo que procedemos a decodificar el dataset de comandos personalizados para ver si los resultados también han mejorado. Para ello utilizamos dropout de 0.2 pues es el que mejor ha funcionado y los resultados son los de la Tabla

25, los resultados han mejorado con respecto a la Tabla 22 y la Tabla 11, pero seguimos viendo que son resultamos demasiado malos.

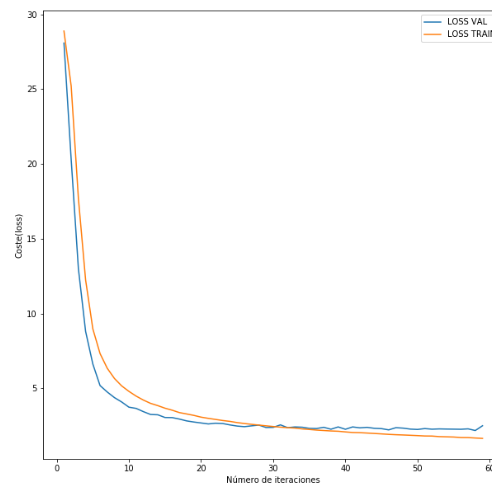
| WER (%) | TER (%) | Tiempo/muestra (s) |
|---|---|---|
| 98.1132 | 88.5986 | 0.477 |

*Tabla 25 Decodificación del dataset de comandos personalizados utilizando dropout de 0.2 en las capas convolucionales y las fully-connected*

Antes de aplicar otra solución probamos aumentar el número de iteraciones en el entrenamiento utilizando dropout de 0.2 en capas convolucionales. El proceso de entrenamiento se muestra en la Gráfica 17 y en la Gráfica 18, observamos que el problema de sobre-aprendizaje que teníamos en la Gráfica 15 y en la Gráfica 16 ha desaparecido por completo por lo tanto, el valor de coste y los valores de error de validación se aproximan más a los valores del proceso de entrenamiento. Esta mejora también se ve reflejada en el proceso de decodificación con el dataset de Kaggle (Tabla 26) y con el de comandos personalizados (Tabla 27). No obstante el rendimiento con comandos personalizados sigue siendo demasiado bajo.




*Gráfica 17 Comparativa TER y WER para entrenamiento y validación con dropout de 0.2 en las capas fully-connected y en las de convolución para 60 epoch*




*Gráfica 18 Comparativa coste (loss) entre entrenamiento y validación con dropout de 0.2 en las capas fully-connected y en las convolucionales para 60 epoch*

| WER (%) | TER (%) | Tiempo/muestra (s) |
|---|---|---|
| 7.00105 | 2.99875 | 0.397 |

*Tabla 26 Resultados decodificación para dropout de 0.2 en las capas fully-connected y convolucionales para 60 epoch*

| WER (%) | TER (%) | Tiempo/muestra (s) |
|---|---|---|
| 95.97410 | 84.57438 | 0.412 |

*Tabla 27 Decodificación del dataset de comandos personalizados utilizando dropout de 0.2 en las capas convolucionales y las fully-connected para 60 epoch*

Llegados a este punto y viendo los resultados que hemos ido obteniendo, nos planteamos dos alternativas para mejorar el rendimiento del clasificador la primera será ampliar el conjunto de datos para aumentar la diversidad de estos a la hora de realizar el entrenamiento y mejorar el rendimiento del modelo acústico, además como hemos ido viendo tenemos un problema de varianza y añadir más ejemplos a nuestro conjunto de entrenamiento podría ayudar a solventar este problema. La segunda alternativa consiste en cambiar la arquitectura de la red que estamos utilizando, para ello exploraremos diferentes alternativas siguiendo la literatura científica.

#### 4.4.2.4. Pre-procesamiento y pruebas utilizando el dataset de Kaggle y el de Common Voice junto con la arquitectura de capas convolucionales 2D para el entrenamiento del modelo

Comenzamos ampliando el conjunto de datos, para ello utilizamos el dataset de Common Voice [51], este conjunto de datos está formado por voces de personas anónimas de todo el mundo, cada audio contiene una frase aleatoria. El tamaño de este conjunto de datos es mucho mayor que el de Kaggle, cuenta con aproximadamente 148370 elementos, lo que supone un total de unas 167 horas de audio.

Al igual que en el caso del dataset de Kaggle realizamos un pre-procesamiento de los datos para evitar que el rendimiento del modelo que vamos a generar se vea reducido. Para ello creamos un notebook en Python similar al caso de Kaggle pero con algunas diferencias, pues en este caso transformamos los ficheros de audio a formato WAV, eliminamos ficheros de audio vacíos, el particionamiento que hacemos sigue la regla 80/10/10... Posteriormente juntamos el fichero de léxico y las listas de entrenamiento, validación y test generadas por este notebook junto con las que ya tenemos de Kaggle para ampliar el conjunto de datos y procedemos a realizar pruebas.

Lo primero que observamos en una diferencia considerable en los tiempos de ejecución en la fase de entrenamiento para 25 epoch (Tabla 28), algo que resulta comprensible pues hemos pasado de trabajar con 6609 ejemplos a 73884. Esto además ha supuesto un coste mayor de memoria para la computadora, por lo que nos hemos visto obligados a reducir el tamaño de batch de 24 a 10, que es el máximo que hemos podido utilizar sin salirnos de memoria GPU en ningún momento.

| | Dataset Kaggle | Dataset conjunto |
|---|---|---|
| Tiempo de ejecución (s) | 2375 | 52500 |

*Tabla 28 Comparativa de tiempos de ejecución entre dataset de Kaggle y dataset conjunto*

Probamos con varias tasas de aprendizaje y comprobamos que para los valores: 0.42 y 0.3 después de unas 10 epoch (aproximadamente 5 horas de ejecución) ocurre el problema de los vanishing gradients (Figura 72), tratamos de reducir la tasa de aprendizaje en estos casos a partir

de la epoch 10 para evitar este problema, sin embargo nos encontramos con la misma situación. Decidimos probar a entrenar el modelo con tasas de aprendizaje más bajas desde la epoch 1 para ver si solventamos el problema, mostramos los resultados en la Tabla 29.

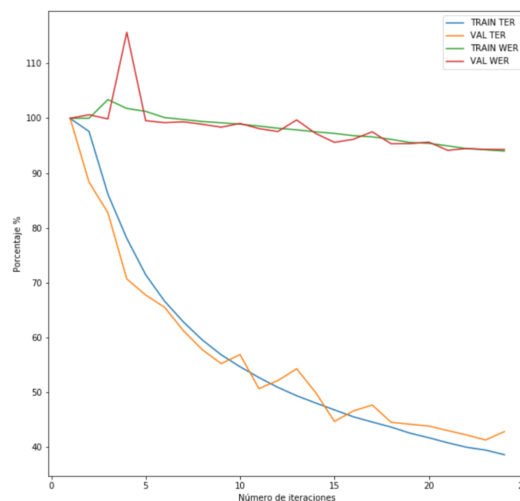
```
I0519 19:19:02.030728  3820 Train.cpp:250] [Network Params: 3904549]
I0519 19:19:02.030740  3820 Train.cpp:251] [Criterion] ConnectionistTemporalClassificationCriterion
I0519 19:19:02.030757  3820 Train.cpp:259] [Network Optimizer] SGD
I0519 19:19:02.030766  3820 Train.cpp:260] [Criterion Optimizer] SGD
I0519 19:19:02.626427  3820 W2lListFilesDataset.cpp:141] 73884 files found.
I0519 19:19:02.627506  3820 Utils.cpp:102] Filtered 0/73884 samples
I0519 19:19:02.632290  3820 W2lListFilesDataset.cpp:62] Total batches (i.e. iters): 7389
I0519 19:19:02.849833  3820 W2lListFilesDataset.cpp:141] 24628 files found.
I0519 19:19:02.850409  3820 Utils.cpp:102] Filtered 0/24628 samples
I0519 19:19:02.851986  3820 W2lListFilesDataset.cpp:62] Total batches (i.e. iters): 2463
I0519 19:19:02.853003  3820 Train.cpp:557] Shuffling trainset
I0519 19:19:02.853911  3820 Train.cpp:564] Epoch 12 started!
F0519 19:50:01.761185  3820 Train.cpp:608] Loss has NaN values. Samples - train_common28920,train_common49738,train_common2274,train_common51352,train_common51227,train_common35798,train_common35489,train_common
53817,train_common7717,train_common33208
*** Check failure stack trace: ***
    @     0x7f4e650365cd  google::LogMessage::Fail()
    @     0x7f4e65038433  google::LogMessage::SendToLog()
    @     0x7f4e6503615b  google::LogMessage::Flush()
    @     0x7f4e65038e1e  google::LogMessageFatal::~LogMessageFatal()
    @           0x4947da  _ZZ4mainENKUlSt10shared_ptrIN2fl6ModuleEES_IN3w2l17SequenceCriterionEES_INS3_10W2lDatasetEES_INS0_19FirstOrderOptimizerEES9_ddblE3_clES2_S5_S7_S9_S9_ddbl.constprop.12679
    @           0x41c337  main
    @     0x7f4e643e0830  __libc_start_main
    @           0x48e539  _start
    @              (nil)  (unknown)
Aborted (core dumped)
```

*Figura 72 Problema vanishing gradients durante el entrenamiento del modelo con el dataset conjunto*

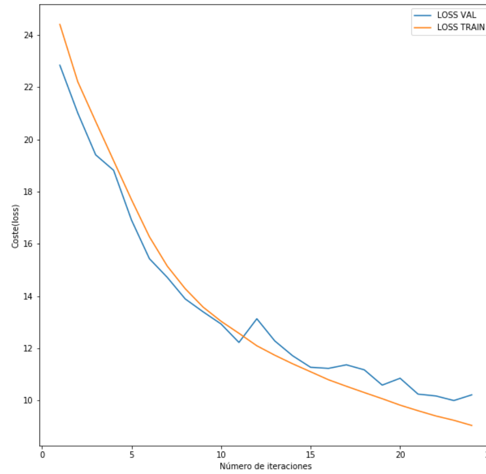
| Tasa de aprendizaje | TER-Train (%) | WER-Train (%) | Loss-Train | TER-Val (%) | WER-Val (%) | Loss-Val |
|---|---|---|---|---|---|---|
| 0.3 | 39.75 | 94.12 | 9.20707 | 42.97 | 96.60 | 10.24563 |
| 0.2 | 38.53 | 94.02 | 9.04289 | 42.76 | 94.31 | 10.21507 |
| 0.1 | 39.02 | 94.05 | 9.15478 | 42.86 | 95.74 | 10.23107 |

*Tabla 29 Resultados del entrenamiento con el dataset conjunto para diferentes valores de tasa de aprendizaje y 25 epoch*

Observamos como seguimos teniendo un problema de bias pero apenas hay problema de varianza, para intentar solucionarlo aumentamos el número de iteraciones. En la Gráfica 19 y en la Gráfica 20 se puede observar el proceso de aprendizaje para una tasa de aprendizaje de 0.2. Los resultados en el proceso de entrenamiento no son nada buenos, pero probamos a realizar una decodificación con cada uno de ellos para ver los resultados (Tabla 30).




*Gráfica 19 Comparativa TER y WER para entrenamiento y validación para el dataset conjunto con tasa 0.2 y 25 epoch*




*Gráfica 20 Comparativa coste (loss) entre entrenamiento y validación con tasa de aprendizaje 0.2 para el dataset conjunto en 25 epoch*

| Tasa de aprendizaje | WER (%) | TER (%) | Tiempo/muestra (s) |
|---|---|---|---|
| 0.3 | 67.7003 | 39.2890 | 0.635 |
| 0.2 | 76.8049 | 52.0713 | 0.689 |
| 0.1 | 78.2476 | 54.02704 | 0.622 |

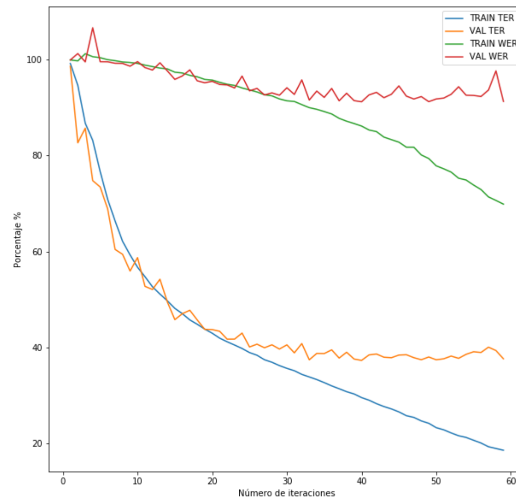
*Tabla 30 Resultados decodificación para el dataset conjunto para diferentes tasas de aprendizaje*

Pese a los resultados pésimos durante la fase de entrenamiento y validación, en la fase de decodificación vemos una cierta mejora, aun así los resultados siguen siendo mucho peores que para el dataset de Kaggle (Tabla 26). Comprobamos que la decodificación sigue siendo muy deficiente para el dataset de comandos personalizados (Tabla 31).

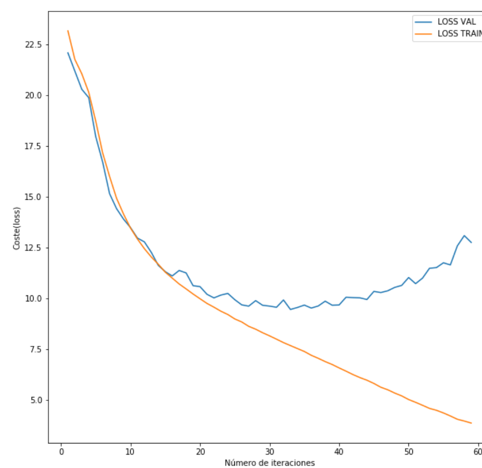
| WER (%) | TER (%) | Tiempo/muestra (s) |
|---|---|---|
| 100.0 | 85.7482 | 0.493 |

*Tabla 31 Resultados para la decodificación del dataset de comandos personalizados utilizando el modelo acústico generado con el dataset conjunto*

Como hemos hecho en otras ocasiones trataremos de reducir el problema de bias aumentando el número de epoch en la fase de entrenamiento y después nos ocuparemos del problema de varianza. Aumentamos el número de epoch a 60 para la tasa de aprendizaje 0.3. El proceso de la fase de entrenamiento se muestra en la Gráfica 21 y la Gráfica 22, donde claramente a partir del epoch 25 vemos una clara diferencia entre los valores de coste y error entre validación y entrenamiento, lo que indica un claro sobre-aprendizaje.




*Gráfica 21 Comparativa TER y WER para entrenamiento y validación para el dataset conjunto con tasa 0.3 y 60 epoch*



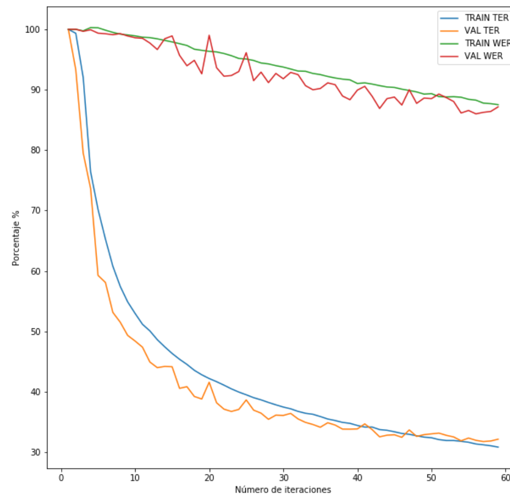

*Gráfica 22 Comparativa coste (loss) entre entrenamiento y validación con tasa de aprendizaje 0.3 para el dataset conjunto en 60 epoch*

Con estos resultados nuevamente realizamos una decodificación con la partición de test del dataset conjunto y con el dataset de comandos personalizados para analizar si los resultados han sufrido alguna mejoría (Tabla 32).

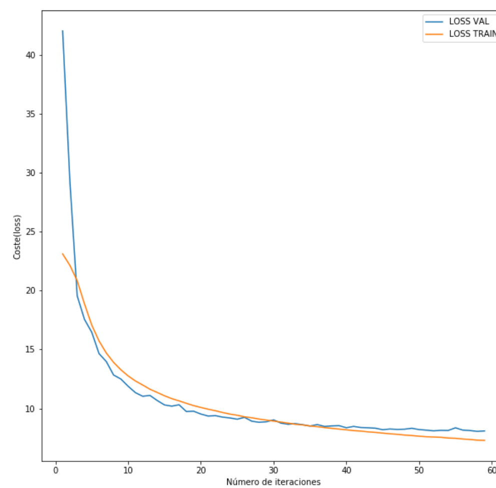
| Dataset | WER (%) | TER (%) | Tiempo/muestra (s) |
|---|---|---|---|
| Conjunto | 46.0204 | 24.2238 | 0.626 |
| Comandos | 94.3396 | 81.9477 | 0.421 |

*Tabla 32 Resultados decodificación para el dataset conjunto y el de comandos utilizando el modelo acústico entrenado con el dataset conjunto durante 60 iteraciones usando una tasa de aprendizaje de 0.3*

Los resultados mejoran de forma notable para el dataset conjunto y de forma mínima para el dataset de comandos personalizados. Probamos a introducir dropout de 0.2 de la misma manera que en la Figura 71, para intentar corregir el problema de varianza y como vemos en la Gráfica 23 y la Gráfica 24 el problema se soluciona de forma correcta.




*Gráfica 23 Comparativa TER y WER para entrenamiento y validación para el dataset conjunto con tasa 0.3 para 60 epoch utilizando dropout de 0.2 en capas convolucionales y fully-connected*




*Gráfica 24 Comparativa coste (loss) entre entrenamiento y validación con tasa de aprendizaje 0.3 para el dataset conjunto en 60 epoch y aplicando dropout de 0.2 en capas convolucionales y fully-connected*

Pasamos a comprobar los resultados para la decodificación con el dataset conjunto y con el de comandos personalizados, los documentamos en la Tabla 33.

| Dataset | WER (%) | TER (%) | Tiempo/muestra (s) |
|---|---|---|---|
| Conjunto | 24.5368 | 10.174 | 1.54 |
| Comandos | 93.2264 | 85.782 | 0.592 |

*Tabla 33 Resultados decodificación para el dataset conjunto y el de comandos utilizando el modelo acústico entrenado con el dataset conjunto durante 60 iteraciones usando una tasa de aprendizaje de 0.3 y utilizando dropout de 0.2 en capas convolucionales y fully-connected*

Hemos comprobado que ampliando el dataset los resultados son mucho peores a la hora de decodificar la partición de test del dataset conjunto, sin embargo los resultados con el conjunto de comandos personalizados han mejorado mínimamente (aunque siguen siendo muy malos) respecto a los que habíamos obtenido hasta el momento, esto puede ser debido a varias razones:

1. La calidad de algunos de los datos (audios) del conjunto Common Voice no es muy buena, por lo que hay grabaciones que son difíciles de entender incluso para el ser humano.

2. Existe una gran variedad de voces con diferentes acentos, pero quizás por cada tipo de acento no hay demasiadas voces, lo que hace que el modelo no aprenda de forma correcta.

3. La arquitectura de la red puede ser demasiado sencilla para la cantidad de datos que estamos tratando por lo que su rendimiento en test no es demasiado bueno. Por lo que podría solucionarse con un cambio de arquitectura o ampliando la cantidad de capas y neuronas existente.

4. Utilizar capas convolucionales de 2 dimensiones para nuestro caso de uso no es una buena opción pues no mantienen demasiado bien el contexto y en audios de una duración alta tienden a fallar más por este motivo.

#### *4.4.2.5. Pruebas utilizando el dataset de Kaggle y diferentes arquitecturas de red para el entrenamiento del modelo*

Decidimos probar la otra alternativa para mejorar el rendimiento del modelo acústico, es decir, cambiar de arquitectura. Buscamos arquitecturas en papers científicos y encontramos una arquitectura basada en capas convolucionales de 1 dimensión [52] que a priori parece tener muy buenos resultados. La arquitectura consta de 12 capas convolucionales de 1 dimensión, las dos primeras capas con stride y las dos últimas con una anchura o width de los filtros de convolución de 1, lo que equivale a dos capas fully-connected. La Figura 73 muestra la arquitectura que hemos explicado.

Para realizar las pruebas creamos un fichero de arquitectura en Wav2letter++ y construimos la arquitectura de la Figura 73 con ayuda de la documentación de la librería Flashlight [43]. El aspecto de la arquitectura tras la construcción del fichero es el que muestra en la Figura 74, para esta arquitectura al igual que la inicial (Figura 67) es necesario una primera capa View que ponga las dimensiones de la entrada (las características) en el orden correcto y por otro lado una última capa de Reorder que reordene la salida de la red.

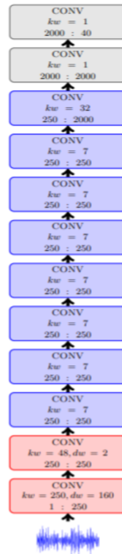


*Figura 73 Arquitectura con capas convolucionales 1D* [52]

```
V -1 1 NFEAT 0
AC NFEAT 250 250 160 -1 0.5 1
R
AC 250 250 48 2 -1 0.5 1
R
AC 250 250 7 1 -1 0.5 1
R
AC 250 250 7 1 -1 0.5 1
R
AC 250 250 7 1 -1 0.5 1
R
AC 250 250 7 1 -1 0.5 1
R
AC 250 250 7 1 -1 0.5 1
R
AC 250 250 7 1 -1 0.5 1
R
AC 250 250 7 1 -1 0.5 1
R
AC 250 2000 32 1 -1 0.5 1
R
AC 2000 2000 1 1 -1 0.5 1
R
AC 2000 NLABEL 1 1 -1 0.5 1
RO 2 0 3 1
```

*Figura 74 Arquitectura 1D en Wav2letter++*

Para las pruebas, en primer lugar decidimos probar la arquitectura con el dataset de Kaggle pues al ser un dataset pequeño nos ayudará rápidamente a comprobar el rendimiento de la nueva arquitectura, para saber si funciona bien o no para nuestro problema. Lo primero que de lo que nos damos cuenta, es que el número de parámetros de la red es mucho mayor que en el caso de la arquitectura de capas convolucionales de 2 dimensiones, sin embargo el tiempo de ejecución del proceso de entrenamiento resulta ligeramente menor (Tabla 34), esto puede ser debido a que las operaciones que realiza internamente la red son más sencillas y menos costosas que en el caso de la arquitectura de capas de dos dimensiones.

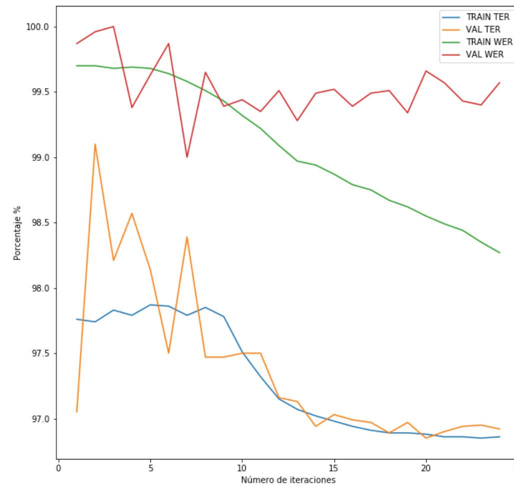
| Arquitectura | Tiempo (s) | Número de parámetros |
|---|---|---|
| Capas convolucionales 2D | 2375 | 3904036 |
| Capas convolucionales 1D | 1125 | 28642787 |

*Tabla 34 Comparativa de tiempos de ejecución y número de parámetros para las dos arquitecturas convolucionales*

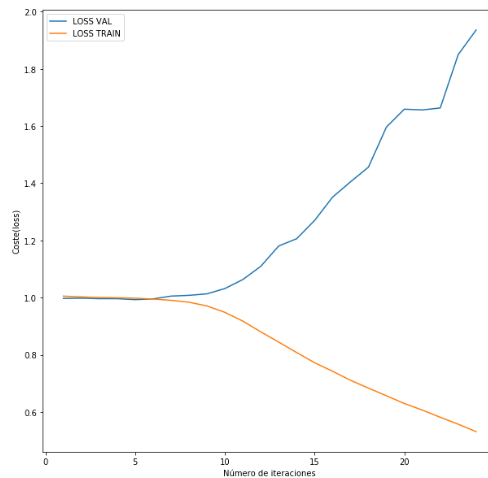
Para ello, utilizamos la configuración por defecto (la misma que en el caso de LibriSpeech) y obtenemos los resultados de la Tabla 35, la Gráfica 25 y la Gráfica 26.

| TER-Train (%) | WER-Train (%) | Loss-Train | TER-Val (%) | WER-Val (%) | Loss-Val |
|---|---|---|---|---|---|
| 96.86 | 98.27 | 0.53145 | 96.92 | 99.57 | 1.93600 |

*Tabla 35 Resultados configuración por defecto arquitectura de capas convolucionales de 1 dimensión*




*Gráfica 25 Comparativa TER y WER para entrenamiento y validación para arquitectura de capas convolucionales de 1 dimensión (parámetros por defecto)*




*Gráfica 26 Comparativa coste (loss) entre entrenamiento y validación para arquitectura de capas convolucionales de 1 dimensión (parámetros por defecto)*

Vemos como los resultados son realmente malos para el caso del error, mientras que los valores de coste son realmente bajos, sin embargo, comprobamos como existe un claro problema de sobre-aprendizaje que se visualiza claramente en el WER y en el coste, pero también de bias pues el error en entrenamiento es demasiado alto. Esto posiblemente es debido a que esta arquitectura no rinde bien para nuestro problema. Probamos a realizar una primera decodificación para ver qué resultados se obtienen y como esperábamos son muy malos (Tabla 36), aunque observamos que la velocidad de decodificación es mucho mayor que en la anterior arquitectura, posiblemente esto se debe a lo que comentábamos a cerca de la Tabla 34.

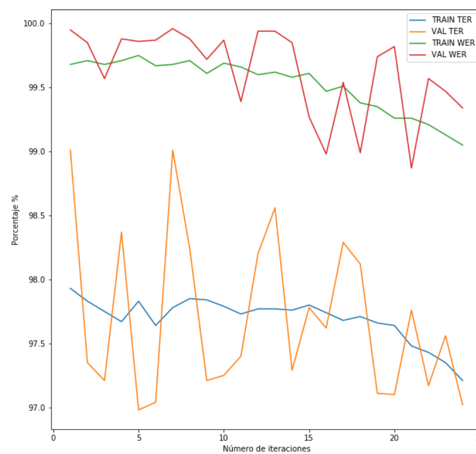
| WER (%) | TER (%) | Tiempo/muestra (s) |
|---|---|---|
| 99.895 | 99.0231 | 0.00416 |

*Tabla 36 Resultados decodificación para arquitectura de capas convolucionales de 1 dimensión*

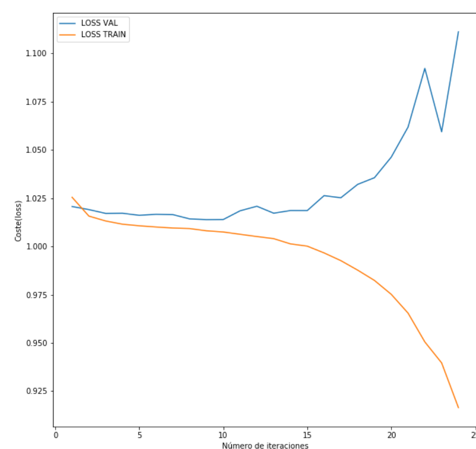
Probamos a aumentar el tamaño de batch al máximo que permite nuestro sistema para esta arquitectura de red, en este caso 64 y probamos diferentes valores de tasas de aprendizaje para ver si los resultados mejoran (Tabla 37), el resto de los hiper-parámetros no se modifican. Comprobamos como los resultados prácticamente no mejoran y seguimos teniendo un problema de bias y varianza. Un ejemplo del proceso de aprendizaje se muestra en la Gráfica 27 y en la Gráfica 28 donde vemos claramente los problemas de bias y varianza, además pese a una pequeña mejora en WER en validación vemos como los valores fluctúan mucho lo que da lugar a unos picos muy pronunciados.

| Tasa de aprendizaje | TER-Train (%) | WER-Train (%) | Loss-Train | TER-Val (%) | WER-Val (%) | Loss-Val |
|---|---|---|---|---|---|---|
| 0.2 | 97.98 | 99.66 | 0.99978 | 98.25 | 99.95 | 1.01956 |
| 0.3 | 97.50 | 99.30 | 0.96722 | 97.56 | 99.52 | 1.04288 |
| 0.42 | 97.21 | 99.05 | 0.91644 | 97.02 | 99.34 | 1.11124 |

*Tabla 37 Resultados para diferentes tasas de aprendizaje con la arquitectura de capas convolucionales de 1 dimensión*




*Gráfica 27 Comparativa TER y WER para entrenamiento y validación para arquitectura de capas convolucionales de 1 dimensión tasa de aprendizaje 0.42 y batch 64*




*Gráfica 28 Comparativa coste (loss) entre entrenamiento y validación para arquitectura de capas convolucionales de 1 dimensión tasa de aprendizaje 0.42 y batch 64*

Mostramos los resultados de la decodificación utilizando los modelos generados con las diferentes tasas de aprendizaje en la Tabla 38 para poder contrastar los valores obtenidos. Como vemos en el resultado es prácticamente igual de malo para todas las tasas de aprendizaje. Cuando probamos a decodificar el dataset de comandos personalizados nos ocurre algo similar (Tabla 39).

| Tasa de aprendizaje | WER (%) | TER (%) | Tiempo/muestra (s) |
|---|---|---|---|
| 0.42 | 99.9772 | 99.1653 | 0.00485 |
| 0.3 | 99.9886 | 99.2925 | 0.00458 |
| 0.2 | 99.9977 | 99.3271 | 0.00454 |

*Tabla 38 Resultados para la decodificación del dataset Kaggle utilizando diferentes tasas de aprendizaje con el dataset de Kaggle y la arquitectura de capas 1D*

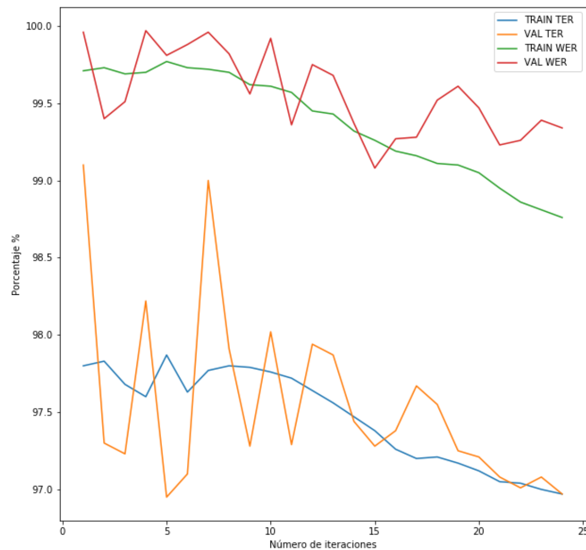
| Tasa de aprendizaje | WER (%) | TER (%) | Tiempo/muestra (s) |
|---|---|---|---|
| 0.42 | 100.0 | 96.6746 | 0.00385 |
| 0.3 | 100.0 | 97.3872 | 0.00210 |
| 0.2 | 100.0 | 97.8622 | 0.00393 |

*Tabla 39 Resultados para la decodificación del dataset de comandos personalizados utilizando diferentes tasas de aprendizaje con el dataset de Kaggle y la arquitectura de capas 1D*

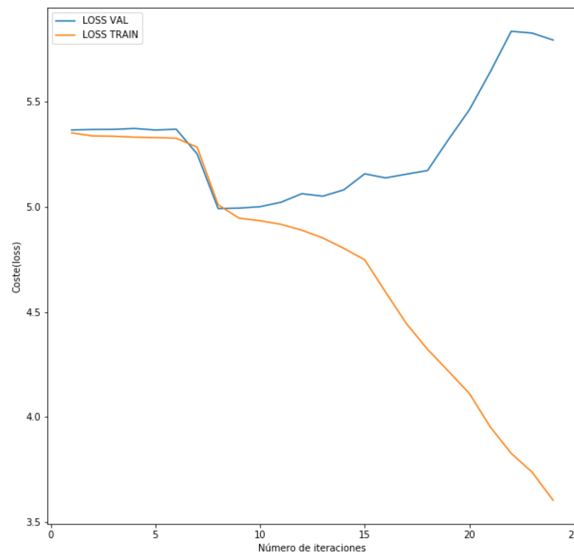
Para intentar solucionar el problema de bias visto que el incremento en batch y tasa de aprendizaje no lo solucionan cambiaremos la función a optimizar de CTC a ASG. Por desgracia comprobamos que los resultados son prácticamente los mismos (Tabla 40, Gráfica 29, Gráfica 30), seguimos viendo un problema claro tanto de varianza alta como de bias alto, por lo que el modelo no se está ajustando correctamente a los datos de entrenamiento.

| TER-Train (%) | WER-Train (%) | Loss-Train | TER-Val (%) | WER-Val (%) | Loss-Val |
|---|---|---|---|---|---|
| 96.88 | 98.27 | 3.17673 | 96.91 | 99.47 | 6.82694 |

*Tabla 40 Resultados del aprendizaje para la función de coste ASG con la arquitectura de capas convolucionales de 1 dimensión y tasa de aprendizaje 0.42*




*Gráfica 29 Comparativa TER y WER para entrenamiento y validación para arquitectura de capas convolucionales de 1 dimensión tasa de aprendizaje 0.42, batch 64 y función loss ASG*




*Gráfica 30 Comparativa coste (loss) entre entrenamiento y validación para arquitectura de capas convolucionales de 1 dimensión tasa de aprendizaje 0.42, batch 64 y función loss ASG*

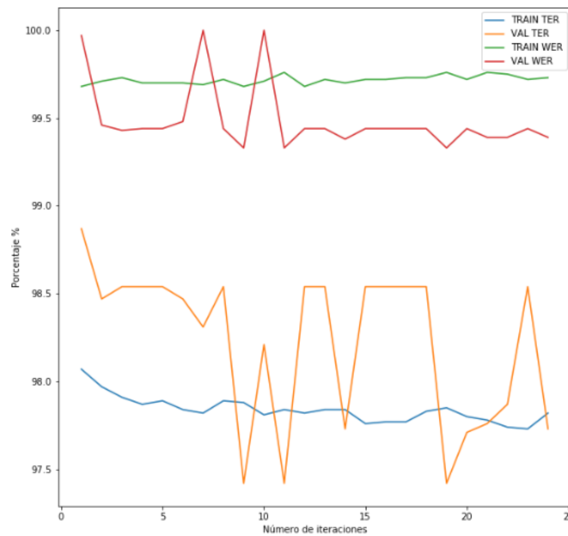
Como última solución probamos a introducir dropout en la red (Figura 75) para probar si el rendimiento de la red mejora. Al igual que en anteriores situaciones (Tabla 20 y Tabla 23) probamos con diferentes valores de probabilidad para una tasa de aprendizaje de 0.42, un batch de 64 y 25 epoch. Los resultados de los procesos de entrenamiento y validación se están en la Tabla 41, como podemos observar los resultados siguen siendo muy malos, aunque nos damos cuenta con la y la que el problema de sobre-aprendizaje comienza a corregirse aunque aún está presente.

```
(0): View (-1 1 40 0)
(1): AsymmetricConv1D (Conv2D (40->250, 250x1, 160,1, SAME,0, 1, 1) (with bias))
(2): ReLU
(3): Dropout (0.200000)
(4): AsymmetricConv1D (Conv2D (250->250, 48x1, 2,1, SAME,0, 1, 1) (with bias))
(5): ReLU
(6): Dropout (0.200000)
(7): AsymmetricConv1D (Conv2D (250->250, 7x1, 1,1, SAME,0, 1, 1) (with bias))
(8): ReLU
(9): Dropout (0.200000)
(10): AsymmetricConv1D (Conv2D (250->250, 7x1, 1,1, SAME,0, 1, 1) (with bias))
(11): ReLU
(12): Dropout (0.200000)
(13): AsymmetricConv1D (Conv2D (250->250, 7x1, 1,1, SAME,0, 1, 1) (with bias))
(14): ReLU
(15): Dropout (0.200000)
(16): AsymmetricConv1D (Conv2D (250->250, 7x1, 1,1, SAME,0, 1, 1) (with bias))
(17): ReLU
(18): Dropout (0.200000)
(19): AsymmetricConv1D (Conv2D (250->250, 7x1, 1,1, SAME,0, 1, 1) (with bias))
(20): ReLU
(21): Dropout (0.200000)
(22): AsymmetricConv1D (Conv2D (250->250, 7x1, 1,1, SAME,0, 1, 1) (with bias))
(23): ReLU
(24): Dropout (0.200000)
(25): AsymmetricConv1D (Conv2D (250->250, 7x1, 1,1, SAME,0, 1, 1) (with bias))
(26): ReLU
(27): Dropout (0.200000)
(28): AsymmetricConv1D (Conv2D (250->2000, 32x1, 1,1, SAME,0, 1, 1) (with bias))
(29): ReLU
(30): Dropout (0.200000)
(31): AsymmetricConv1D (Conv2D (2000->2000, 1x1, 1,1, SAME,0, 1, 1) (with bias))
(32): ReLU
(33): Dropout (0.200000)
(34): AsymmetricConv1D (Conv2D (2000->37, 1x1, 1,1, SAME,0, 1, 1) (with bias))
(35): Reorder (2,0,3,1)
```

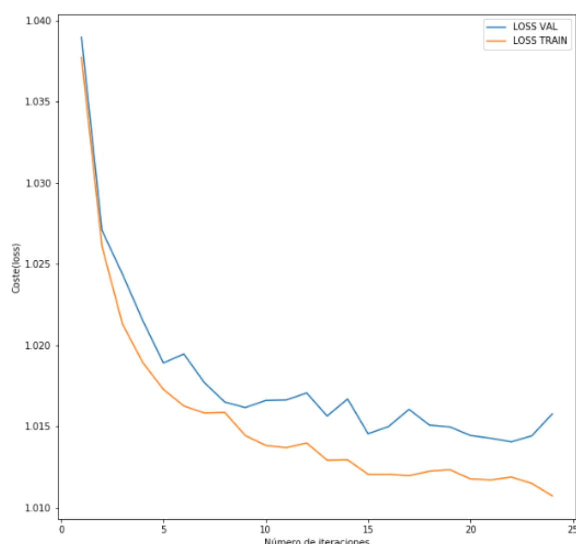
*Figura 75 Arquitectura de capas convolucionales de 1 dimensión con dropout*

| Probabilidad de dropout | TER-Train (%) | WER-Train (%) | Loss-Train | TER-Val (%) | WER-Val (%) | Loss-Val |
|---|---|---|---|---|---|---|
| 0.5 | 97.82 | 99.73 | 1.01072 | 97.73 | 99.39 | 1.01577 |
| 0.3 | 97.85 | 99.77 | 1.00590 | 99.01 | 99.96 | 1.02482 |
| 0.2 | 97.87 | 99.72 | 1.00222 | 98.34 | 99.99 | 1.02297 |

*Tabla 41 Resultados para diferentes valores de probabilidad de dropout para la arquitectura de capas convolucionales de 1 dimensión*




*Gráfica 31 Comparativa TER y WER para entrenamiento y validación para arquitectura de capas convolucionales de 1 dimensión, tasa de aprendizaje 0.42, batch 64 y dropout de 0.5*




*Gráfica 32 Comparativa coste (loss) entre entrenamiento y validación para arquitectura de capas convolucionales de 1 dimensión, tasa de aprendizaje 0.42, batch 64 y dropout de 0.5*

Probamos a mostrar los resultados de la decodificación para los datasets de Kaggle (Tabla 42) y el de comandos personalizados (Tabla 43) para los diferentes valores de dropout y así comprobar si el rendimiento del modelo en test ha mejorado. Lamentablemente como podemos ver los resultados empeoran respecto a los obtenidos en la Tabla 38 y en la Tabla 39 para la tasa de aprendizaje de 0.42, tanto en WER y en TER como en la velocidad de decodificación.

| Probabilidad de dropout | WER (%) | TER (%) | Tiempo/muestra (s) |
|---|---|---|---|
| 0.5 | 100.0 | 100.0 | 0.00507 |
| 0.3 | 100.0 | 100.0 | 0.00492 |
| 0.2 | 100.0 | 100.0 | 0.00489 |

*Tabla 42 Resultados para la decodificación del dataset Kaggle utilizando diferentes valores de dropout con tasa de aprendizaje 0.42 y la arquitectura de capas 1D*

| Probabilidad de dropout | WER (%) | TER (%) | Tiempo/muestra (s) |
|---|---|---|---|
| 0.5 | 100.0 | 100.0 | 0.00487 |
| 0.3 | 100.0 | 100.0 | 0.00438 |
| 0.2 | 100.0 | 100.0 | 0.00636 |

*Tabla 43 Resultados para la decodificación del dataset de comandos personalizados utilizando diferentes valores de dropout con tasa de aprendizaje 0.42 y la arquitectura de capas 1D*

Finalmente, viendo los resultados tan malos que se obtienen decidimos descartar la arquitectura y pasar directamente a utilizar modelos secuenciales (RNN, LSTM, GRU…). Para ello, al igual que en el caso de la arquitectura de capas convolucionales de 1 dimensión (Figura 73) buscamos arquitecturas en artículos científicos sobre Speech Recognition y encontramos dos muy interesantes [53], [54]. Sin embargo, tras varios intentos fallidos de implementar las arquitecturas en Wav2letter++ y después de buscar información al respecto (hablaremos más al

respecto en el siguiente apartado), finalmente descartamos la idea de utilizarlas con lo que terminamos nuestras pruebas con la herramienta Wav2letter++.

## 4.5. Problemas y soluciones

Durante la instalación de Wav2letter++ y sus dependencias tuvimos los siguientes problemas:

1. Los drivers para utilizar la tarjeta gráfica del sistema no estaban correctamente instalados lo que provocaba errores en la instalación de las librerías de Nvidia, para solucionarlo, reinstalamos los drivers correspondientes.

2. A la hora de instalar Nvidia Cuda, tuvimos que hacerlo con el instalador ".deb" ya que el ".sh" fallaba constantemente en la instalación.

3. Wav2letter++ era incapaz de encontrar las librerías de KenLM durante el proceso de instalación, para corregir esto fue necesario:

    3.1. Copiar las librerías de KenLM a la ruta /usr/local/lib/ del sistema.

    3.2. Añadir una variable al sistema con la ruta al fichero con las cabeceras de KenLM.

    3.3. Modificar el fichero "CMakeCache.txt" para la instalación de Wav2letter++ y añadir manualmente la ruta a las librerías y cabecera de KenLM.

4. Wav2letter++ no conseguía encontrar el fichero sndfile de libsndfile, para solventar este problema tuvimos instalar la versión dev de la librería, esta versión contiene los ficheros de cabeceras, por lo que Wav2letter++ fue capaz de detectarlo.

5. Al instalar Wav2letter++ no detectaba el directorio de ArrayFire por lo que tuvimos que añadir el flag "-DArrayFire_DIR" durante la instalación para especificar la ruta a la instalación de la librería.

Después de la instalación y configuración de Wav2letter++ y todas sus dependencias tuvimos los siguientes problemas:

1. La herramienta no pasaba todos los tests de KenLM y Wav2letter++ (Figura 76). Después de investigar descubrimos que debíamos actualizar la imagen de Nvidia Docker tras su instalación en un contenedor, pues la imagen por defecto contiene elementos que están desactualizados, lo que provoca un mal funcionamiento del sistema. Para ello actualizamos la librería ArrayFire, junto con Flashlight y Wav2letter++.

```
root@e59caaa85d67:~/flashlight/build# make test
Running tests...
Test project /root/flashlight/build
      Start  1: AutogradTest
 1/10 Test  #1: AutogradTest .....................   Passed   40.44 sec
      Start  2: OptimTest
 2/10 Test  #2: OptimTest ........................   Passed    1.57 sec
      Start  3: ModuleTest
 3/10 Test  #3: ModuleTest .......................   Passed    4.09 sec
      Start  4: SerializationTest
 4/10 Test  #4: SerializationTest ................   Passed   11.05 sec
      Start  5: UtilsTest
 5/10 Test  #5: UtilsTest ........................   Passed    1.30 sec
      Start  6: DatasetTest
 6/10 Test  #6: DatasetTest ......................   Passed    3.33 sec
      Start  7: MeterTest
 7/10 Test  #7: MeterTest ........................   Passed    0.88 sec
      Start  8: AllReduceTest
 8/10 Test  #8: AllReduceTest ....................***Failed    1.67 sec
      Start  9: ContribModuleTest
 9/10 Test  #9: ContribModuleTest ................   Passed    2.83 sec
      Start 10: ContribSerializationTest
10/10 Test #10: ContribSerializationTest .........   Passed    2.07 sec

90% tests passed, 1 tests failed out of 10

Total Test time (real) =  69.24 sec

The following tests FAILED:
          8 - AllReduceTest (Failed)
Errors while running CTest
Makefile:71: recipe for target 'test' failed
make: *** [test] Error 8
```

*Figura 76 Resultados de los tests de Flashlight*

2. Durante las pruebas comprobamos que el contenedor por defecto no detectaba correctamente la codificación UTF-8, esto provocaba que las transcripciones que contenían símbolos no contemplados en ASCII se detectasen de forma incorrecta.

3. Encontramos problemas con los permisos de Docker para modificar ciertos ficheros e instalar ciertos paquetes dentro del contenedor de la herramienta, la solución a este problema consiste en crear un “Docker group” y añadir al usuario que pretende utilizar el contenedor a dicho grupo.

4. Tuvimos problemas con el almacenamiento de Docker durante las pruebas, la partición en la que se almacenaban las imágenes se llenaba constantemente por lo que tuvimos que modificar la ruta de almacenamiento y lectura de imágenes de Docker, esto funcionó de manera temporal, pero al cabo de un tiempo nos vimos obligados a reinstalar el sistema operativo que estábamos utilizando (Ubuntu) ampliando la partición asignada.

5. En la fase de entrenamiento utilizando el dataset conjunto (Kaggle + Common Voice), en algunas ocasiones se nos presentó el problema de vanishing gradients, debido a esto los valores de los gradientes eran tan pequeños que acababa habiendo divisiones entre 0 por lo que en los cálculos aparecía el símbolo NaN o “Not a Number”, provocando errores y deteniendo el entrenamiento.

6. Durante una de las pruebas el ordenador que estábamos utilizando para testear Wav2letter++ se apagó y al reiniciarlo no funcionaba correctamente, lo que nos obligó nuevamente a reinstalar el sistema operativo y Wav2letter++ junto con sus dependencias.

7. Debido a una actualización del motor el flag “--iter” pasó de ser el número de epoch al número de steps por epoch, para solucionarlo tuvimos que calcular en cada prueba la cantidad de steps o pasos por epoch (número de muestras entre el tamaño de batch) y multiplicarlo por el número de epoch deseado.

8. Relacionado con el punto anterior, una actualización provocó el valor de la tasa de aprendizaje (flag “--lr”) que establecíamos no fuese fijo, la solución que encontramos fue desactivar un nuevo flag que había sido añadido llamado “--warmup”, el cual establece un número de steps para ir “calentando” la tasa de aprendizaje, es decir para utilizar un valor variable e incremental de tasa de aprendizaje en cada step, hasta llegar al valor fijado por el usuario.

9. Como ya hemos comentado durante la fase de pruebas, lo errores que devuelve la herramienta en la fase de entrenamiento y decodificación son muy poco descriptivos, por lo que tuvimos que preguntar en varias ocasiones a los desarrolladores de la herramienta que es lo que estábamos haciendo mal, esto hizo que nos llevase mucho más tiempo construir y evaluar la segunda arquitectura convolucional.

10. Relacionado con el punto anterior nos fue imposible probar arquitecturas basadas en capas RNN, LSTM o GRU ya que la herramienta devolvía errores poco descriptivos y los desarrolladores de Wav2letter++ no pudieron ayudarnos mucho con este problema, por lo que nunca pudimos poner en marcha ninguna arquitectura basada en modelos secuenciales.

# 5. Comparativa experimental

En este apartado resumimos y comparamos lo que hemos realizado en secciones anteriores: los resultados de las pruebas, la utilidad de las herramientas, la facilidad de puesta en marcha y uso y el consumo de recursos del sistema. También vamos a hablar acerca de la posible integración del sistema para su utilización en un entorno real.

## 5.1. Comparativa de herramientas y resultados

Empezamos hablando de las herramientas que hemos utilizado en las secciones 3 y 4 del documento, es decir vamos a comentar la experiencia que hemos tenido con Rhasspy y Wav2letter++ y comparar los resultados que hemos obtenido con ambas herramientas.

Comenzando por Rhasspy hemos podido ver que la herramienta resulta mucho más sencilla e intuitiva que Wav2letter++, no solo a nivel de uso, sino también durante el proceso de instalación y puesta en marcha, pues requiere de un menor número de dependencias (librerías, programas, paquetes...) que el motor de Facebook, por lo que su puesta en marcha es mucho menos costosa y genera una menor cantidad de problemas. Por otro lado, esta reducción del número de dependencias nos ha facilitado la tarea de modificar el código para añadir funcionalidades o alterar el funcionamiento del motor (evitar la comprobación de los certificados en HTTPS, almacenar el audio recogido...).

Relacionado con lo último que hemos comentado, otra ventaja que hemos encontrado al utilizar Rhasspy frente a Wav2letter++ es que el primero está escrito completamente en Python un lenguaje sencillo, que conocemos y que nos ha permitido el desarrollo rápido y sencillo de funcionalidades para el motor de reconocimiento de voz así como la construcción del servidor encargado de recibir y manejar las peticiones del usuario.

Algo de lo que presume Rhasspy y que nos ha resultado muy útil e importante para nuestro caso de uso es que puede funcionar en diferentes arquitecturas hardware, destacando la arquitectura de la Raspberry Pi 3B+ (aarch64), esto como hemos mencionado en más de una ocasión en el documento permite reducir los costes hardware de la posible implementación final del sistema y a su vez reduce la cantidad de elementos a introducir en un quirófano, así como el espacio ocupado por el sistema. Por otro lado, facilita la limpieza y esterilización del equipo necesario para hacer funcionar el sistema. Wav2letter++ sin embargo, requiere de un hardware más potente para el entrenamiento y la decodificación y por el momento los modelos acústicos que se generan con el sistema no tienen soporte para la arquitectura de la Raspberry pi.

Relacionado con el entrenamiento hemos podido comprobar como en Rhasspy es un proceso muy sencillo (pues solo es pulsar un botón en su interfaz gráfica) y rápido (al menos para el conjunto de datos que hemos probado) pues el motor no está generando un modelo acústico al entrenar, sino que aprende las palabras (diccionario o léxico) y comandos nuevos que el usuario introduce en el sistema, esto hace que no sea necesario un dataset de audios con los que entrenar el sistema, eliminando por tanto la necesidad de la búsqueda y pre-procesamiento de los mismos. Esto en gran parte es lo que hace que Rhasspy requiera de un hardware tan sencillo para funcionar y que sus entrenamientos sean tan rápidos. Por otro lado, hemos visto que en Wav2letter++ el proceso es completamente distinto pues requerimos de una gran cantidad de

datos para las fases de entrenamiento y decodificación, además pese a que la herramienta está muy bien optimizada, es necesario un hardware con una gran potencia, de hecho en muchas ocasiones durante la fase de pruebas nos hemos visto limitados en el proceso de mejora del rendimiento del modelo acústico debido a que el hardware con el que contamos no daba más de sí.

Un tema que resulta diferencial entre Rhasspy y Wav2letter++ es que el primero incluye una serie de funcionalidades o herramientas como el sistema de trigger word o el sistema de text to speech, lo que nos ha facilitado la creación del sistema de asistente voz sin tener que recurrir a herramientas de terceros para añadir estas funcionalidades a nuestro proyecto. Por otro lado, Wav2letter++ solo permite el entrenamiento de modelos acústicos y la decodificación de audio mediante el uso de estos y de modelos del lenguaje.

También hemos podido comprobar como el sistema de speech to text (el reconocimiento del habla) ha sido más preciso en cuanto a las transcripciones de audio en Rhasspy que en los modelos que hemos generado con Wav2letter++, al menos con muestras de nuestra voz. No obstante, consideramos que estos resultados pueden ser mejorados, como vamos a comentar más adelante.

Sin embargo, Rhasspy cuenta con una serie de desventajas:

1. Pese a que por cada uno de los sistemas que integra ofrece diferentes posibilidades con diferentes configuraciones, como por ejemplo en el caso del sistema de wake word donde podemos elegir entre otros entre Porcupine o Snowboy, resulta ser una herramienta muy poco personalizable pues si queremos añadir o modificar ciertas funcionalidades de la herramienta en muchos casos nos vemos obligados a modificar el código del motor, lo que puede ser realmente costoso dependiendo de lo que queramos realizar.

2. Aunque el sistema snowboy para la wake word generalmente ha funcionado muy bien, hemos visto que en varias ocasiones ha generado falsos positivos en la detección.

3. Pese a que el reconocimiento del habla en Rhasspy ha funcionado bien en nuestro entorno de prueba, no hemos podido probarlo en un entorno real y en caso de un funcionamiento deficiente resulta complicado modificar o re-entrenar el modelo acústico que trae por defecto, mientras que en el caso de Wav2letter++ bastaría con añadir más muestras, modificar la arquitectura de la red y/o los hiper-parámetros del entrenamiento.

4. Cada vez que añadimos uno o varios comandos y/o una o varias palabras, es necesario reentrenar el motor para que Rhasspy las aprenda, sin embargo en el caso de Wav2letter++ basta con añadir las palabras nuevas al fichero de léxico del sistema para que sean detectadas correctamente.

5. Nos hemos dado cuenta de que Rhasspy es una herramienta relativamente nueva y presenta fallos puntuales que lastran la experiencia del usuario, pues en algunas ocasiones el motor no ha realizado de forma correcta los entrenamientos, no ha conseguido detectar las palabras desconocidas pese a estar en el diccionario de palabras personalizadas, ha lanzado fallos de ejecución... Además ligado a este problema encontramos que existe poca documentación a cerca del funcionamiento de Rhasspy.

6. Para el manejo de los comandos es necesario de un software externo pues Rhasspy no da la posibilidad de manejar los intents por sí mismo.

Con respecto a Wav2letter++ como comentábamos el rendimiento en cuanto a el reconocimiento del habla no ha sido tan bueno como en el caso de Rhasspy ni como esperábamos en un principio. En las pruebas hemos demostrado que para el dataset de Kaggle los resultados han sido muy buenos tanto en entrenamiento y en validación (Tabla 23) como en test (Tabla 26), sin embargo cuando hemos cambiado a nuestro dataset de comandos personalizados con nuestra voz hemos visto como los resultados obtenidos han sido realmente malos, no llegando a reducir en ningún caso el WER más allá de 90% (Tabla 22, Tabla 25, Tabla 27).

La ampliación del conjunto de datos que hemos realizado utilizando el dataset de Common Voice no parece haber ayudado a mejorar los resultados, más bien estos han empeorado no solo en entrenamiento y validación (Tabla 29), sino también en decodificación (Tabla 32, Tabla 33). Esto puede ser debido a varias razones como comentábamos en la sección 4, entre las que destacamos: que la calidad de muchos de los audios no sea demasiado buena y resulten difíciles de entender, que existe una gran variedad de voces con diferentes acentos pero para cada tipo de acento no hay demasiadas muestras, que la arquitectura de capas convolucionales de 2 dimensiones que hemos utilizado es demasiado sencilla para nuestro problema. Hemos llegado a estas conclusiones pues, como hemos podido ver al ampliar el dataset, el modelo presentaba un claro problema de bias pese a aumentar el número de epoch hasta 60 (Gráfica 23 y Gráfica 24), lo que hace evidente que el modelo tiene un problema de under-fitting pues ni siquiera se ajusta a los datos de entrenamiento, con lo que el error en WER y TER es demasiado alto.

Una de las posibles soluciones a este problema, además de aumentar el número de epoch o conjunto de entrenamiento, es aumentar el tamaño de la red neuronal utilizada o directamente cambiar la arquitectura de esta, algo que hemos intentado realizar utilizando la arquitectura de capas convolucionales de 1 dimensión (Figura 73). Sin embargo, como ya hemos podido comprobar en las múltiples pruebas que hemos realizado, los resultados tienden a empeorar en todas las fases del proceso (entrenamiento, validación y test) tanto para el dataset de Kaggle (Tabla 36, Tabla 37, Tabla 38 y Tabla 41) como para el de comandos personalizados (Tabla 39, Tabla 43).

Los resultados tan malos obtenidos con la arquitectura de capas convolucionales de 1 dimensión pueden ser debidos a que la arquitectura es demasiado simple para nuestro problema y por tanto no resulta adecuada para realizar un buen reconocimiento del habla, sin embargo en su momento decidimos realizar pruebas con una arquitectura de red que utilizase este tipo de capas pues habíamos comprobado que los resultados que se habían obtenido en otras investigaciones eran bastante prometedores, al menos para el idioma inglés [52], [55].

Como ya hemos comentado también tratamos de utilizar modelos secuenciales para resolver nuestro problema e intentar mejorar el rendimiento que habíamos obtenido. Sin embargo, no pudimos llegar a realizar ninguna prueba con una arquitectura basada en capas RNN, LSTM o GRU, ya que la Wav2letter++ devolvía unos errores muy poco descriptivos que no dejaban ver cuál era el problema por el que este tipo de arquitecturas no funcionaban de forma correcta. Tras muchas pruebas e investigación acerca de los errores que devolvía la herramienta, decidimos terminar las pruebas con Wav2letter++ y descartar el uso de la herramienta.

## 5.2. Integración final

Debido a la situación de confinamiento, no hemos podido probar nuestro sistema en un entorno real, por lo que no hemos llegado a integrar el sistema de forma completa. En este apartado explicaremos de una forma breve lo pasos que seguiríamos para lograr la creación de un sistema completamente funcional capaz de manejar equipos de consulta médicos.

Si decidiéramos utilizar Rhasspy para la construcción sistema tendríamos que:

1. Completar el servidor añadiendo las funciones necesarias para poder controlar los equipos de consulta médicos dependiendo del tipo de intent recibido.

2. Utilizar certificados de una AC (Autoridad de Certificación) para poder habilitar las comprobaciones de seguridad tanto en el servidor como en Rhasspy.

3. Añadir las medidas de seguridad necesarias al servidor para la puesta en producción.

4. Añadir los comandos, intents y las palabras personalizadas necesarias para su uso en un quirófano real.

5. Ajustar los parámetros de la configuración de los distintos sistemas que utiliza Rhasspy para ajustarlo a un entorno real.

6. Recoger más muestras de audio para el sistema de wake word de Snowboy y mejorar así su precisión.

Si decidiéramos utilizar Wav2letter++ tendríamos que:

1. Probar más arquitecturas de red, configuraciones de hiper-parámetros y datasets hasta dar con una combinación con la que obtener un modelo acústico con un buen rendimiento para nuestro caso de uso.

2. Hacer uso del framework "Inference" de wav2letter++ para poder introducir un stream de audio vía micrófono al modelo acústico y de esta manera poder realizar una decodificación en tiempo real.

3. Construir o utilizar algún sistema o programa externo que haga las funciones de un sistema de trigger word, de manera que al identificar un o varias palabras preconfiguradas sea capaz de activar la entrada de audio a través de un micrófono e introducirlo en el modelo acústico generado por Wav2letter++ (similar al funcionamiento de Snowboy).

4. Configurar Wav2letter++ para poder enviar la transcripción que realizar el modelo a un servidor o a un programa capaz de manejar el comando recibido.

5. Integrar un sistema capaz de hacer las funciones de text to speech, es decir recibir la respuesta del servidor o del programa que maneja los comandos de vox en forma de texto y transformarla a audio (similar al funcionamiento de PicoTTS).

Como vemos en el caso de Wav2letter++, buscamos que el sistema final tenga un comportamiento similar al que tiene Rhasspy (Figura 31), pues consideramos que este debería ser el funcionamiento ideal de nuestro sistema.

Por último, tendríamos que diseñar el sistema a nivel de hardware, decidiendo qué elementos serían necesarios para el correcto funcionamiento del asistente de voz en un entorno real.

# 6. Conclusiones y líneas futuras

En esta última parte de la memoria vamos a incluir las conclusiones de todo el trabajo que hemos realizado, además vamos a comentar las posibles líneas futuras que podría seguir el proyecto, es decir, las cosas que quedan por hacer, las mejoras que se podrían realizar o las alternativas que podríamos considerar para mejorar los resultados que hemos obtenido durante las fases de pruebas.

## 6.1. Conclusiones

A la vista de todo lo analizado en este trabajo fin de grado, podemos extraer las siguientes conclusiones:

1. Aunque Speech Recognition no es una rama nueva de la IA (puesto que ya encontramos los primeros estudios en 1970 [56]), hemos comprobado que en los últimos años ha habido un auge en la aplicación de las técnicas de esta rama para el desarrollo de productos no solo orientados al ámbito profesional como Dragon Speech Recognition de la compañía Nuance [57], sino al ámbito doméstico como Alexa [31].

2. Existen herramientas bien desarrolladas que nos facilitan la creación de no solo modelos end-to-end como Wav2letter++ sino también de asistentes de voz completos como Rhasspy, evitando de esta manera tener que crear un sistema desde cero.

3. Con las pruebas que hemos hecho en las secciones 3 y 4 con cada una de las herramientas y tras realizar la comparativa en la sección 5, hemos dejado claro que el sistema que mejor nos ha funcionado ha sido Rhasspy, pues además de ofrecer un buen rendimiento a la hora de reconocer el habla (al menos en el entorno de pruebas), también nos ha facilitado la creación del asistente de voz gracias a la integración de sistemas como SnowBoy para la detección de la wake word o PicoTTS para transformar el texto en voz.

4. Wav2letter++ es una herramienta con mucho potencial y consideramos que con una configuración adecuada de la arquitectura de la red neuronal y de los hiper-parametros para el entrenamiento podemos llegar a conseguir un modelo acústico de calidad con un rendimiento muy bueno para el reconocimiento del habla del idioma español, además estamos convencidos que una arquitectura basada en capas LSTM o GRU podría haber dado muy buenos resultados en nuestro caso de uso.

5. Nos hemos dado cuenta de que el hecho de trabajar con voz implica no solo un consumo más elevado de recursos (Almacenamiento, CPU, GPU...), sino también unos tiempos de ejecución más elevados especialmente para la fase de entrenamiento. Por tanto, un análisis de resultados y un planteamiento previo antes de cada entrenamiento ha cobrado un mayor peso a la hora de realizar las pruebas, evitando de esta manera realizar pruebas innecesarias ahorrándonos una gran cantidad de tiempo y obteniendo mejores resultados.

6. Hemos comprobado que el pre-procesamiento en un dataset de audio implica una mayor dificultad que en otros conjuntos de datos, pues debemos manejar el códec, la frecuencia de muestreo, el número de canales y otras características típicas de los ficheros de voz, además de tratar los archivos con las transcripciones para eliminar símbolos no deseados, datos inconsistentes o incompletos, datos duplicados...

7. La aplicación de técnicas de Speech Recognition en el campo de la medicina tiene un gran futuro, pues va a permitir el desarrollo de productos que van a facilitar y mejorar el trabajo de los profesionales de este sector.

## 6.2. Líneas futuras

Comenzamos comentado las cosas que han quedado por realizar:

1. Quedaría realizar la integración final del sistema, algo que ya habíamos comentado en la sección 5 de este documento.

2. Relacionado con el punto anterior, como ya habíamos mencionado antes, nos han faltado realizar pruebas en entornos reales para probar el funcionamiento del sistema y poder obtener un feedback con el que mejor y/o ampliar las funcionalidades de este.

3. En el caso de Wav2letter++ como ya hemos comentado no hemos podido probar arquitecturas basadas en capas RNN, LSTM o GRU y es algo que quedaría pendiente pues consideramos que una arquitectura basada en este tipo de capas podría funcionar muy bien para nuestro problema.

4. Nos ha quedado pendiente hacer pruebas y comparar resultados con otras alternativas que vamos a explicar más adelante como DeepSpeech.

Vemos necesario explicar y comentar algunas alternativas a las herramientas que hemos utilizado (Rhasspy, Wav2letter++), pues consideramos que podrían tener un buen funcionamiento y convendría realizar pruebas con ellas en un futuro:

1. DeepSpeech: Ya habíamos hablado previamente de esta herramienta en la sección 2, se trata de una herramienta open source que funciona completamente offline y está escrita en Python, por lo que a priori no debería ser tan eficiente como Wav2letter++, ya que como ya hemos explicado Wav2letter++ está escrito en C++ y una de las cosas de las que presume es de un uso eficiente de recursos (CPU, GPU, memoria...), así como

de una velocidad de hasta dos veces superior a herramientas de la competencia (Figura 77).

*Figura 77 Comparativa del tiempo de entrenamiento entre Wav2letter++, DeepSpeech y Kaldi [https://cdn-images-1.medium.com/freeze/max/1000/1*VoYBADzyCkhwqGBi5XkWDw.jpeg?q=20]*

La arquitectura de DeepSpeech es similar a la descrita en el paper del sistema de Speech Recognition de Baidu [29] (Figura 78), con la diferencia que utiliza capas RNN unidireccionales en vez de bidireccionales y además las unidades de estas capas son LSTM en vez de GRU [59], [60].

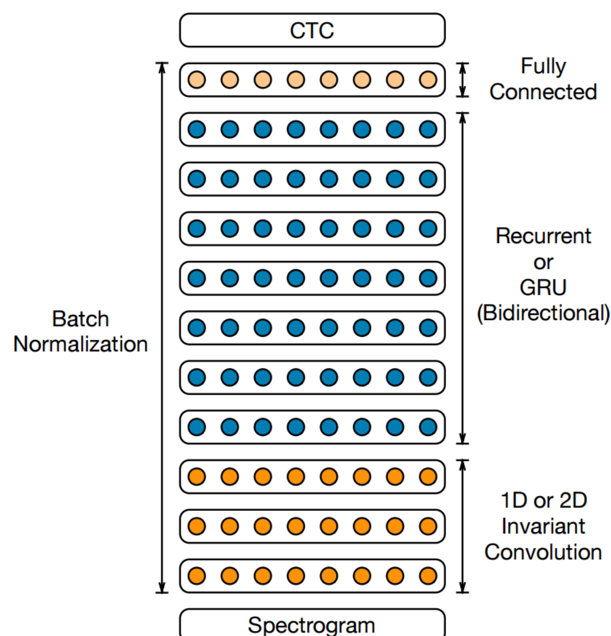



*Figura 78 Arquitectura del sistema de Speech Recognition de Baidu* [29]

Consideramos que puede ser una buena alternativa porque utiliza redes RNN, es decir, modelos secuenciales, lo que en principio debería buenos resultados para nuestro caso de uso, ya que, como hemos mencionado en esta memoria, este tipo de redes neuronales tienden a funcionar muy bien en problemas de reconocimiento del habla.

2. Utilizar un framework para Deep Learning: Un framework se puede definir como un entorno de trabajo que busca facilitar la tarea de programación a la hora de escribir código o de desarrollar cualquier programa o aplicación. Para ello, automatiza ciertos procesos y provee el esqueleto o implementación de ciertas funciones y algoritmos.

   Para nuestro problema, hemos valorado la opción de construir nuestro sistema desde cero, construyendo la arquitectura de la red neuronal utilizando uno de los muchos frameworks de Deep Learning que hay disponibles en Python como: Keras, Pytorch, TensorFlow... Existe mucha documentación sobre estos y además construir el sistema desde cero nos daría un control absoluto sobre el mismo, facilitándonos las tareas de modificación del código.

   A la hora de elegir un framework habría que tener en cuenta lo siguiente:

   - La facilidad a la hora de programar la red neuronal para trabajar sobre ella y para una posible puesta en producción.

   - La velocidad de ejecución.

   - Que el framework sea open source durante un largo periodo de tiempo.

# 7. Bibliografía


[1] ]A. M. Turing, "Computing Machinery and Intelligence," *Mind*, vol. LIX, no. 236, pp. 433–460, Oct 1950.

[2] A.M. Turing, "On computable numbers with an application to the Entscheindungsproblem," *Artif. Intell. Law*, vol. 25, no. 2, pp. 181–203, 2017, Nov 1936.

[3] R. Browse, J. Glasgow, "Programming Artificial Languages for," *Comp. & Maths. with Appls.*, vol. 11, no. 5, pp. 431-448, Sept 1985.

[4] R. High, "The Era of Cognitive Systems: An Inside Look at IBM Watson and How it Works," *Int. Bus. Mach. Corp.*, vol. 1, no. 1, pp. 1–14, Dec 2012.

[5] K. Warwick and H. Shah, "Can machines think? A report on Turing test experiments at the Royal Society," *J. Exp. Theor. Artif. Intell.*, vol. 28, no. 6, pp. 989–1007, 2016.

[6] R. Sierra, "Informe Empleos Emergentes", Linkedin, Mountain View, CA, USA, en. 2020.

[7] G. Berger, "Emerging Jobs Report 2020," Linkedin, Mountain View, CA, USA, Jan, 2020.

[8] G. Linden, M. Conover, and J. Robertson, "The Netflix prize, computer science outreach, and Japanese mobile phones," *Commun. ACM*, vol. 52, no. 10, pp. 8–9, Oct 2009.

[9] B. S. Abu-nasser, "Medical Expert Systems Survey," *International Journal of Engineering and Information Systems*, vol. 1, no. 7, pp. 218–224, Oct 2017.

[10] S. P. Somashekhar *et al.*, "Watson for Oncology and breast cancer treatment recommendations: Agreement with an expert multidisciplinary tumor board," *Ann. Oncol.*, vol. 29, no. 2, pp. 418–423, Jan 2018,

[11] B. G. B. and E. A. Feigenbaum, "Dendral and Meta-Dendral: Their applications dimension," *Standford University Computer Science Department*, no. STAN-CS-78-649, pp. 26, Feb 1978.

[12] E. H. Shortliffe, R. Davis, S. G. Axline, B. G. Buchanan, C. C. Green, and S. N. Cohen, "Computer-based consultations in clinical therapeutics: Explanation and rule acquisition capabilities of the MYCIN system," *Comput. Biomed. Res.*, vol. 8, no. 4, pp. 303–320, June 1975

[13] T. Panch, H. Mattie, and L. A. Celi, "The 'inconvenient truth' about AI in healthcare," *npj Digit. Med.*, vol. 2, no. 1, pp. 4–6, Aug 2019.

[14] M. G. Idoate, "Tema 1 introducción a machine learning."

[15] M. Kaytan and I. B. Aydilek, "A review on machine learning tools," *International Journal of Advanced Research in Computer Engineering & Technology (IJARCET)*, vol. 6, no. 10, pp. 1–4, Oct 2017.

[16] A. K. Jain, M. N. Murty, and P. J. Flynn, "Data clustering: A review," *ACM Comput. Surv.*, vol. 31, no. 3, pp. 264–323, Sept 1999.

[17] X. Zhu, "Semi-Supervised Learning Literature Survey," University of Wisconsin, Madison, WI, USA, Sept, 2005.

[18] D. D. Lewis, “Naive (Bayes) at Forty: The Independece Assumption in Information Retrieval,” *Nédellec C., Rouveirol C. (eds) Machine Learning: ECML-98*, vol. 1398, pp. 4-15, June 1998.

[19] K. P. Murphy, “Naive Bayes classifiers,” *Lect. Notes Comput. Sci. (including Subser. Lect. Notes Artif. Intell. Lect. Notes Bioinformatics)*, vol. 4701 LNAI, pp. 371–381, Oct 2006.

[20] V. Metsis, I. Androutsopoulos, and G. Paliouras, “Spam filtering with Naive Bayes - Which Naive Bayes?,” in *3rd Conf. Email Anti-Spam - Proceedings, CEAS 2006*, pp. 1-9.

[21] “Multinomial distribution", *Wikipedia*, 2004. [Online]. Available: https://en.wikipedia.org/wiki/Multinomial_distribution. [Accessed: 08-may-2020].

[22] E. Anguiano Hernández, “Naive Bayes Multinomial para Clasificación de Texto Usando un Esquema de Pesado por Clases”, pp. 1–8, abr. 2009.

[23] A. M. Kibriya, E. Frank, B. Pfahringer, and G. Holmes, “Multinomial naive bayes for text categorization revisited,” *Lect. Notes Artif. Intell. (Subseries Lect. Notes Comput. Sci.)*, vol. 3339, pp. 488–499, Jan 2004.

[24] “Understanding Neural Networks", *Towards Data Science*, 2019. [Online]. Available: https://towardsdatascience.com/understanding-neural-networks-19020b758230. [Accessed: 03-may-2020].

[25] O. Abdel-hamid, A. Mohamed, H. Jiang, L. Deng, G. Penn, and D. Yu, “Convolutional Neural Networks for Speech Recognition,” *IEE/ACM Transactions on audio, speech, and language processing*, vol. 22, no. 10, Oct 2014.

[26] S. Hochreiter, “The vanishing gradient problem during learning recurrent neural nets and problem solutions,” *Int. J. Uncertainty, Fuzziness Knowlege-Based Syst.*, vol. 6, no. 2, pp. 107–116, Apr 1998.

[27] “Illustrated Guide to LSTM’s and GRU’s: A step by step explanation.”, *Towards Data Science*, 2018. [Online]. Available: https://towardsdatascience.com/illustrated-guide-to-lstms-and-gru-s-a-step-by-step-explanation-44e9eb85bf21. [Accessed: 04-may-2020].

[28] “What is an Acoustic Model?", *Voxforge*, 2010. [Online]. Available: http://www.voxforge.org/home/docs/faq/faq/what-is-an-acoustic-model. [Accessed: 05-may-2020].

[29] D. Amodei *et al.*, “Deep Speech 2 : End-to-End Speech Recognition in English and Mandarin,” *Baidu Research – Silicon Valley AI Lab*, pp. 1–28, Dec 2005.

[30] “Chunked transfer encoding", *Wikipedia*, 2006. [Online]. Available: https://en.wikipedia.org/wiki/Chunked_transfer_encoding. [Accessed: 06-may-2020].

[31] “Keyword Research, Competitive Analysis, & Website Ranking.”, *Alexa*, 2010. [Online]. Available: https://www.alexa.com/. [Accessed: 16-may-2020].

[32] A. Amberkar, P. Awasarmol, G. Deshmukh, and P. Dave, “Speech Recognition using Recurrent Neural Networks,” in *Proc. 2018 Int. Conf. Curr. Trends Towar. Converging Technol. ICCTCT 2018*, 2018, no. June, pp. 1–4.

[33] “Toda la información relevante y preguntas frecuentes.”, *WebRTC*, 2018. [En línea]. Disponible en: https://www.3cx.es/webrtc/. [Accedido: 07-may-2020].

[34] "Intent Recognition." *Rhasspy*, 2019. [Online]. Available: https://rhasspy.readthedocs.io/en/latest/intent-recognition/. [Accessed: 07-May-2020].

[35] "Distancia de Levenshtein", *Wikipedia*, 2007. [En línea]. Disponible en: https://es.wikipedia.org/wiki/Distancia_de_Levenshtein. [Accedido: 07-may-2020].

[36] "Referenc

e", *Rhasspy*, 2019. [Online]. Available: https://rhasspy.readthedocs.io/en/latest/reference/#http-api. [Accessed: 09-may-2020].

[37] "Installation", *Rhasspy*, 2019. [Online]. Available: https://rhasspy.readthedocs.io/en/latest/installation/. [Accessed: 09-may-2020].

[38] "Snowboy Hotword Detection.", *Snowboy*, 2016. [Online]. Available: https://snowboy.kitt.ai/. [Accessed: 10-may-2020].

[39] "facebookresearch/wav2letter: Facebook AI Research's Automatic Speech Recognition Toolkit.", *GitHub*, 2018. [Online]. Available: https://github.com/facebookresearch/wav2letter. [Accessed: 11-may-2020].

[40] V. Pratap *et al.*, "Wav2Letter++: A Fast Open-source Speech Recognition System," in *ICASSP, IEEE Int. Conf. Acoust. Speech Signal Process. - Proc.*, 2019, vol. 2019-May, pp. 6460–6464.

[41] "Introducing Wav2letter++", *Towards Data Science*, 2018. [Online]. Available: https://towardsdatascience.com/introducing-wav2latter-9e94ae13246. [Accessed: 11-may-2020].

[42] "Mel Frequency Cepstral Coefficient (MFCC) tutorial", *Practical Cryptography*, 2013. [Online]. Available: http://practicalcryptography.com/miscellaneous/machine-learning/guide-mel-frequency-cepstral-coefficients-mfccs/. [Accessed: 11-may-2020].

[43] "Modules — flashlight documentation.", *Flashlight*, 2018. [Online]. Available: https://fl.readthedocs.io/en/latest/modules.html. [Accessed: 12-may-2020].

[44] "Sequence Modeling with CTC.", *Distill*, 2017. [Online]. Available: https://distill.pub/2017/ctc/. [Accessed: 12-may-2020].

[45] "Better, Faster Speech Recognition with Wav2Letter's Auto Segmentation Criterion.", *Towards Data Science*, 2019. [Online]. Available: https://towardsdatascience.com/better-faster-speech-recognition-with-wav2letters-auto-segmentation-criterion-765efd55449. [Accessed: 12-may-2020].

[46] N. Zeghidour, Q. Xu, V. Liptchinsky, N. Usunier, G. Synnaeve, and R. Collobert, "Fully Convolutional Speech Recognition," *Facebook A.I. Research, Paris, France; New York & Menlo Park, USA, CoML, ENS/CNRS/EHESS/INRIA/PSL Research University, Paris, France*, pp. 25–29, Apr 2018.

[47] "wav2letter/tutorials/1-librispeech_clean at master · facebookresearch/wav2letter.", *GitHub*, 2018. [Online]. Available: https://github.com/facebookresearch/wav2letter/tree/master/tutorials/1-librispeech_clean. [Accessed: 13-may-2020].

[48] "Spanish Single Speaker Speech Dataset", *Kaggle*, 2018. [Online]. Available: https://www.kaggle.com/bryanpark/spanish-single-speaker-speech-dataset. [Accessed:

14-may-2020].

[49] "Where should I place dropout layers in a neural network? - Cross Validated.", *Stack Exchange*, 2016. [Online]. Available: https://stats.stackexchange.com/questions/240305/where-should-i-place-dropout-layers-in-a-neural-network. [Accessed: 17-may-2020].

[50] G. E. Hinton, N. Srivastava, A. Krizhevsky, I. Sutskever, and R. R. Salakhutdinov, "Improving neural networks by preventing co-adaptation of feature detectors," pp. 1–18, 2012, [Online]. Available: http://arxiv.org/abs/1207.0580.

[51] "Common Voice.", *Mozilla*, 2017. [En línea]. Disponible en: https://voice.mozilla.org/es/datasets. [Accedido: 15-may-2020].

[52] R. Collobert, C. Puhrsch and G. Synnaeve, "Wav2letter++: An End-to-End ConvNet-Based Speech Recognition System," in *Iclr*, 2017, no. 2014, pp. 1–16.

[53] A. Graves, N. Jaitly, and A. Mohamed, "Hybrid Speech Recognition with Deep Bidirectional LSTM," in *IEEE Workshop on Automatic Speech Recognition and Understanding,* ASRU 2013, pp. 273–278.

[54] S. Watanabe, T. Hori, and J. R. Hershey, "Language independent end-to-end architecture for joint language identification and speech recognition," in *IEEE Autom. Speech Recognit. Underst. Work. ASRU 2017 - Proc.*, 2017, vol. 2018-Janua, pp. 265–271.

[55] S. Kriman *et al.*, "Quartznet: Deep Automatic Speech Recognition with 1D Time-Channel Separable Convolutions," in *ICASSP 2020 - 2020 IEEE International Conference on Acoustics, Speech and Signal Processing (ICASSP), Barcelona, Spain*, 2020, pp. 6124-6128.

[56] "Speech recognition", *Wikipedia*, 2002. [Online]. Available: https://en.wikipedia.org/wiki/Speech_recognition. [Accessed: 23-may-2020].

[57] "Dragon Speech Recognition - Get More Done by Voice", *Nuance*, 2016. [Online]. Available: https://www.nuance.com/dragon.html. [Accessed: 23-may-2020].

[58] "Current DeepSpeech architecture - Deep Speech", *Mozilla Discourse*, 2019. [Online]. Available: https://discourse.mozilla.org/t/current-deepspeech-architecture/41471/7. [Accessed: 23-may-2020].

[59] A. Hannun *et al.*, "Deep Speech: Scaling up end-to-end speech recognition," *Baidu Research – Silicon Valley AI Lab*, pp. 1–12, Dec 2014.