Universidad Zaragoza

Trabajo Fin de Máster

Estudio implementación de algoritmos de compresión sin pérdidas para señales de audio utilizando técnicas de deep learning.

Study implementation of lossless compression algorithms for audio signals using deep learning techniques.

Autor

Rubén Grávalos Ruiz

Director y Ponente

Dr. Antonio Miguel Artiaga

ESCUELA DE INGENIERÍA Y ARQUITECTURA

2022

TRABAJOS DE FIN DE GRADO / FIN DE MÁSTER

Escuela de Ingeniería y Arquitectura
Universidad Zaragoza

DECLARACIÓN DE AUTORÍA Y ORIGINALIDAD

(Este documento debe remitirse a seceina@unizar.es dentro del plazo de depósito)

D./Dª. RUBÉN GRÁVALOS RUIZ ,

en aplicación de lo dispuesto en el art. 14 (Derechos de autor) del Acuerdo de 11 de septiembre de 2014, del Consejo de Gobierno, por el que se aprueba el Reglamento de los TFG y TFM de la Universidad de Zaragoza, Declaro que el presente Trabajo de Fin de Estudios de la titulación de Máster Universitario en Ingeniería de Telecomunicación (Título del Trabajo)

Estudio implementación de algoritmos de compresión sin pérdidas para señales de audio utilizando técnicas de deep learning

es de mi autoría y es original, no habiéndose utilizado fuente sin ser citada debidamente.

Zaragoza, 25 de noviembre de 2022

Fdo:

AGRADECIMIENTOS

En primer lugar, a la Universidad de Zaragoza por todos los conocimientos que he adquirido en ella, los cuales me han permitido formarme como ingeniero superior en el ámbito de las telecomunicaciones, y a su vez, crecer tanto en el sentido profesional como personal. Especial agradecimiento a Antonio Miguel Artiaga por brindarme la posibilidad de realizar este proyecto bajo su tutoría y de esta manera profundizar en el mundo del Deep Learning, siendo este de gran interés para mí. La predisposición y el apoyo por su parte no han podido ser mayor, además de la efectividad a la hora de resolver las posibles dudas que han podido surgirme durante el desarrollo del mismo.

A todos mis amigos y familia, por haberlos tenido siempre de la mano a lo largo de toda mi carrera formativa, y qué duda cabe, en la futura profesional.

Estudio implementación de algoritmos de compresión sin pérdidas para señales de audio utilizando técnicas de deep learning.

RESUMEN

Este trabajo fin de máster pone de manifiesto, una vez más, la efectividad de la utilización de técnicas basadas en Deep Learning, concretamente para el caso de compresión sin pérdidas de señales de audio enfocada a dar lugar a transmisión de los mismos con una diferente gama de ratios de compresión en función de una serie de parámetros clave propios de la red neuronal empleada, en este caso, las redes LSTM.

El proyecto se estructura en una serie de fases, comenzando por una propia de estudio del estado del arte de redes neuronales y sus beneficios en su aplicación al ejercicio de compresión sin pérdidas de archivos de diferentes tipos de información, siempre combinadas con distintas técnicas de codificación. Posteriormente, un proceso de entrenamiento y familiarización con ellas aplicadas a ejemplos sencillos, toma de decisión de las técnicas concretas a emplear, desarrollo del sistema final y realización de pruebas en función diferentes parámetros del mismo.

En esta memoria se comenzará presentando y describiendo las técnicas estudiadas de partida para este proyecto, a continuación se argumentará la decisión para el sistema final desarrollado, se profundizará en el funcionamiento de las diferentes partes que lo componen, y finalmente se expondrán una serie de pruebas realizadas con el mismo sobre un archivo de diferentes señales de audio.

Los resultados obtenidos permiten dar evidencia de la premisa de la que partimos, poniendo a las técnicas de Deep Learning como una excelente técnica de compresión sin pérdidas, en este caso de señales de audio.

Índice

1. Introducción **1**

1.1. Contexto 1
1.2. Motivación 2
1.3. Objetivos 2
1.4. Metodología 5
1.5. Gestión del proyecto 7
1.6. Estructura de la memoria 10

2. Tecnología **11**

2.1. Redes Neuronales Artificiales 11
2.2. Redes LSTM 12
2.3. Cross-Entropy Loss 17
2.4. Optimizador: Adam y SGD 17
2.5. Codificación Huffman 22

3. Implementación del sistema final **25**

3.1. Red 25
3.1.1. Proceso de entrenamiento 28
3.2. Codificación 29
3.2.1. Codificador 30
3.2.2. Decodificador 31
3.2.3. Tipo de señal 33
3.2.4. Parámetros clave 34

4. Resultados experimentales **37**

5. Conclusiones y líneas futuras **41**

5.1. Conclusiones 41
5.2. Líneas futuras 42

6. Bibliografía **45**

Lista de Figuras 47

Lista de Tablas 49

Anexos 50

A. Código 53

A.1. Red 53

A.2. Codificación 54

B. Tecnología empleada 57

B.1. Python 57

B.2. VS Code 60

B.3. MobaXterm 62

B.4. Linear NN 64

Capítulo 1

Introducción

1.1. Contexto

El Deep Learning está constituido por una serie de algoritmos que, de algún modo, pretenden entrenar a una computadora para que esta sea capaz de reproducir mecánicas propias del cerebro humano, como entre otras, reconocimiento del habla, identificación de imágenes o predecir eventos concretos. Este ejercicio se conoce como auto aprendizaje del sistema, y dada su gran capacidad, es empleado en diversos ámbitos como el del tratamiento de imágen, procesado del lenguaje y tecnologías del habla.

El fundamento de las redes neuronales son unidades matemáticas esencialmente sencillas, conocidas como neuronas, las cuales actúan interconectadas unas con otras, dando lugar a varias capas, que a su vez ponen a disposición diferentes niveles de abstracción, y de alguna manera, dan lugar a un aprendizaje más próximo al del ser humano que otros métodos de aprendizaje automático.

Entre las diferentes redes neuronales, cabe destacar las **Redes Neuronales Recurrentes** *(de ahora en adelante, RNNs)*, de esencial importancia en este proyecto. Las redes neuronales responden a una función de activación propia de cada una. La función de activación de las redes neuronales tradicionales *(no recurrentes)*, actúa en una sola dirección, desde la capa de entrada hacia la capa de salida, no recordando valores previos de salida de ellas. A diferencia de estas, las RNNs se caracterizan por incluir conexiones de retroalimentación entre neuronas dentro de las capas internas, lo cual permite procesar la información introducida en un momento dado en base a la información de la que el sistema dispone del procesado de información de los eventos anteriores, que jugará un papel similar al de una información de estado. Esto ocurre de la manera que se muestra en la Figura 1.1 [1].

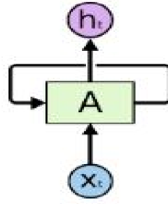


Figura 1.1: Red con realimentación.

Donde A es la red neuronal, x_t es la entrada y h_t la salida de dicha red. El bucle permite que la información se conserve en la red para momentos futuros.

De esta manera, si el sistema es capaz de predecir qué entrada será la próxima, la codificación de esa información se podrá llevar a cabo empleando una cantidad menor de bits. Y esto no es otra cosa que la premisa de la que parte el proyecto.

1.2. Motivación

Como es de saber, el proceso de digitalización abarca cada vez más campos y avanza más rápidamente en cada uno de ellos. El punto de **motivación** del presente proyecto está muy ligado a las comunicaciones telefónicas, por el hecho de operar con señales de voz, y dada la necesidad de transmisión de las mismas utilizando un menor ancho de banda, también por lograr una compresión eficiente de dichas señales. La **VoIP** *(Voz IP)* es el futuro de las comunicaciones telefónicas, y no es otra cosa que señales de voz que viajan sobre el protocolo de Internet, empleando pues el protocolo IP. Mediante VoIP, la señal de voz se transmite en diferentes paquetes, y se plantea la hipotética posibilidad de que el sistema de compresión sin pérdidas desarrollado en el presente proyecto podría operar hipotéticamente sobre dichos paquetes para reducir su tamaño, y de esta manera, el ancho de banda empleado para la transmisión de los mismos sin pérdidas, lo que se traduce en una reducción de coste económico.

El punto principal de interés relativo a lo anterior es que, el sistema que se desarrolla en el presente proyecto está preparado en esencia para no generar problema en paquetes futuros derivados de la pérdida de un paquete anterior. Esto sería posible ya que cada paquete se operaría como si fuera la primera muestra que entra a la red neuronal.

1.3. Objetivos

Partiendo de la introducción a las RNNs y aplicando su potencial a este proyecto, se pretende utilizar dicha recurrencia para predecir qué muestra será la próxima dentro de una señal de audio. Para cada muestra de entrada de la señal de audio, la salida de la red proporciona una serie de probabilidades sobre cual será la siguiente muestra

de entrada, de entre el conjunto de las posibles. Con esta probabilidad y utilizando la técnica de codificación Huffman, se asigna a cada muestra un conjunto de bits concretos, que a su vez corresponde a una palabra del diccionario del sistema Huffman.

Para utilizar de manera conjunta ambas técnicas RNNs y Huffman, como resultaría al trasladar el proyecto a un escenario real, surge la necesidad de desarrollar un sistema de codificación y otro de decodificación que las englobe a ambas, que correspondería a aquellos de los que dispondría la entidad emisora y receptora de dicho escenario real, respectivamente.

El archivo de audio con el que se trabaja en este proyecto tiene las siguientes características:

- Contiene 16.185 señales de audio extraidas de la base de datos TIMIT. [1]
- Todas las señales de audio de TIMIT se han recortado a una duración de 1 segundo y se han muestreado a 8K muestras/segundo.
- Cada muestra de la señal de audio tiene una resolución de 8 bits con *Ley Mu*, siendo la original de 16 bits.

El estudio de cuan bueno es el sistema en términos de compresión sin pérdidas se evaluará en función de la tasa de compresión obtenida, siendo esta calculada mediante la ecuación que se muestra en la Figura 1.2.

$$Tasa\ de\ compresión = \frac{8.000\ muestras\ x\ 8\ bits/muestra}{Nº\ bits\ de\ la\ señal\ codificada}$$

Figura 1.2: Fórmula para cálculo de tasa de compresión.

Así pues, para una misma señal de audio completa, se documentar las pruebas realizadas con diferentes parámetros clave del propio sistema, así como entrenando la red neuronal con diferente número de señales de audio e incluso utilizando unicamente fragmentos de diferente duración de las mismas.

Como parámetros propios de las redes neuronales con los que se experimenta, son los siguientes:

- Batch: número de señales de audio en cada paquete con el que se entrena iterativamente a la red.
- Epoch: número de iteraciones sobre el paquete completo de señales propias de entrenamiento con el que se entrena a la red.

[1]TIMIT es una base de datos referente que se ha utilizado durante décadas en la elaboración de diferentes papers [2] que tratan sobre tecnologías del habla.

– Número de capas de la red Long Short-Term Memory *(de ahora en adelante, LSTM).*

– Número de capas ocultas de la red LSTM.

A continuación se adjunta un gráfico a alto nivel que resume el sistema completo recién descrito, con los bloques propios del mismo, e ilustra los puntos de investigación claves del proyecto, como puede observarse en las Figuras 1.3 y 1.4.

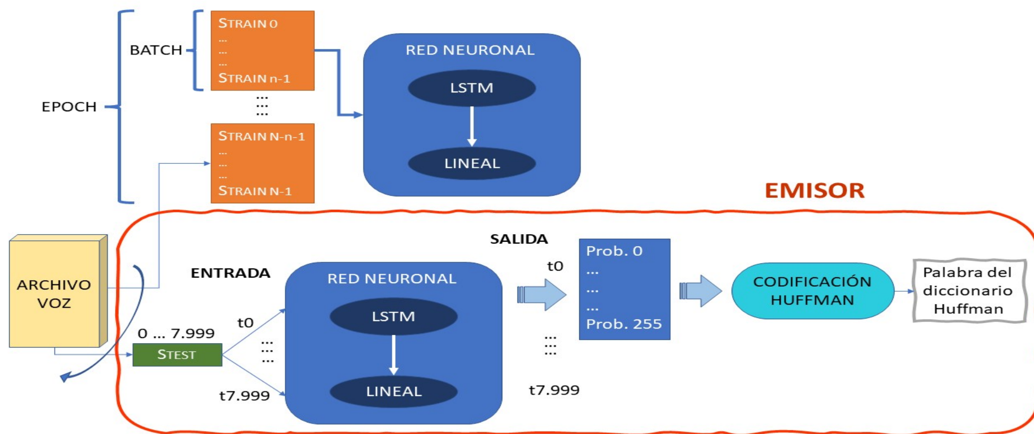


Figura 1.3: Bloque de entrenamiento de la red y bloque de emisión del sistema.

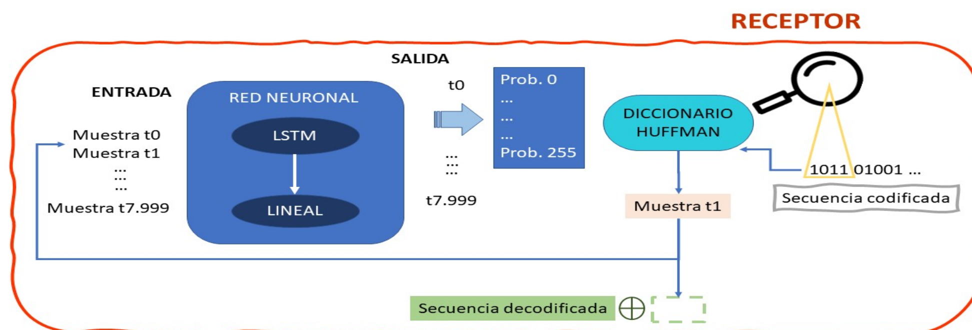


Figura 1.4: Bloque de recepción del sistema.

De la presentación sobre el gran rendimiento de la utilización de las RNNs para el caso de la compresión sin pérdidas en el apartado 1.1 y los objetivos recien expuestos, surge la necesidad de conseguir una **aplicación** programada en lenguaje Python que permita la realización de procesos de entrenamiento de la red del sistema, así como la automatización de las tareas propias a realizar en el ejercicio de emisión y transmisión, y estudiar concretamente los parámetros del sistema que mejor se ajustan al proyecto que nos ocupa, con las señales de audio concretas de las que se dispone.

1.4. Metodología

La metodología que se ha empleado para abordar uno a uno los objetivos propuestos previamente se estructura de la siguiente manera.

En primer lugar, un estudio teórico sobre el estado del arte de las diferentes técnicas de deep learning candidatas a ser utilizadas para el tema fundamental del proyecto, la compresión sin pérdidas.

Una de las principales técnicas que se han analizado es la de **Generación condicional de imágenes utilizanzo Redes Neuronales Convolucionales** *(de ahora en adelante, CNNs)* [3], en donde se explora el potencial del modelado condicional de imágenes adaptando y mejorando una variante convolucional de la arquitectura PixelRNN. En él, se trata la ventaja de devolver densidades de probabilidad explícitas, lo que hace que sea fácil de aplicar en dominios como la compresión y la planificación y exploración probabilística. La idea básica de la arquitectura es utilizar conexiones autorregresivas para modelar las imágenes píxel a píxel, descomponiendo la distribución conjunta de la imagen como un producto de condicionales. En el artículo original se proponen dos variantes: PixelRNN, donde las distribuciones de los píxeles se modelan con LSTM bidimensionales, y PixelCNN, donde se modelan con redes convolucionales. En el documento, se demuestra pues que un único modelo PixelCNN condicional puede utilizarse para generar imágenes de diversas clases como perros, cortadoras de césped y arrecifes de coral, simplemente condicionando una codificación de un solo disparo de la clase. Del mismo modo, se presenta la posibilidad de utilizar incrustaciones que capturan información de alto nivel de una imagen para generar una gran variedad de imágenes con características similares.

Una de las grandes ventajas del modelado generativo **redes neuronales de recurrencia en píxel** [4] es que hay una cantidad prácticamente infinita de datos de datos de imágenes disponibles para aprender. Sin embargo, como las imágenes son muy dimensionales y están muy estructuradas, estimar la distribución de las imágenes naturales es un gran reto, y es el punto que se trata en este recurso.

Además, y enfocado directamente al tipo de red utilizado, aparecen las **redes LSTM**, las cuales ponen sobre la mesa el reto de dotar a un sistema de redes neurolanes de una dinámica lo más parecida posible a la del cerebro humano, y de esta manera, además de reconocer patrones, poder anticiparse a eventos posteriores, que no es otra cosa que predecir muestras futuras. En este punto, salta el problema al que se enfrentar las RNNs y que atajan las redes LSTM, la dependencia de largo término. De esta manera, nos pesenta a las redes LSTM como aquellas diseñadas explícitamente para evitar el problema de la dependencia a largo plazo, pudiendo retener información durante largos periodos de tiempo, como comportamiento por defecto de ellas.

Por último, se ha investigado sobre diferentes técnicas de codificación. La de más importancia y finalmente implementada en el proyecto es sobre la **codificación Huffman de longitud variable** [5], donde se analizan los códigos fuente de longitud variable, como los construidos por el conocido algoritmo de dos pasos de D. A. Huffman. Los costes de comunicación en los sistemas distribuidos empiezan a a dominar los costes de cálculo y almacenamiento internos. Los códigos de longitud variable de longitud variable suelen utilizar menos bits por letra de origen que los códigos de longitud fija, como ASCII y EBCDIC, lo cual puede suponer un enorme ahorro en los sistemas de comunicación basados en paquetes.

A continuación, la metodología pasa a un plano de más importancia a la práctica, que a su vez puede separarse en dos fases:

- Realización de cursos en línea **Intro to Machine Learning** [2] y **Intro to Deep Learning** [3].

- Trabajos de las asignaturas **Aprendizaje automático en datos multimedia** y **Deep Learning**, no cursadas anteriormente y que permiten familiarizarse con todo lo relativo a redes neuronales planteado anteriormente, utilizando técnicas y archivos de uso habitual en la Universidad de Zaragoza.

Posteriormente, se procede a desarrollar propiamente el código de lo que será el sistema final buscado, con las técnicas y particularidades objetivo planteadas anteriormente.

Por último, se lleva a cabo un paquete de pruebas completo en el que se estudia la efectividad del sistema ante diferentes escenarios posibles, en donde a su vez se llega a las conclusiones correspondientes a cada situación, y al proyecto en general.

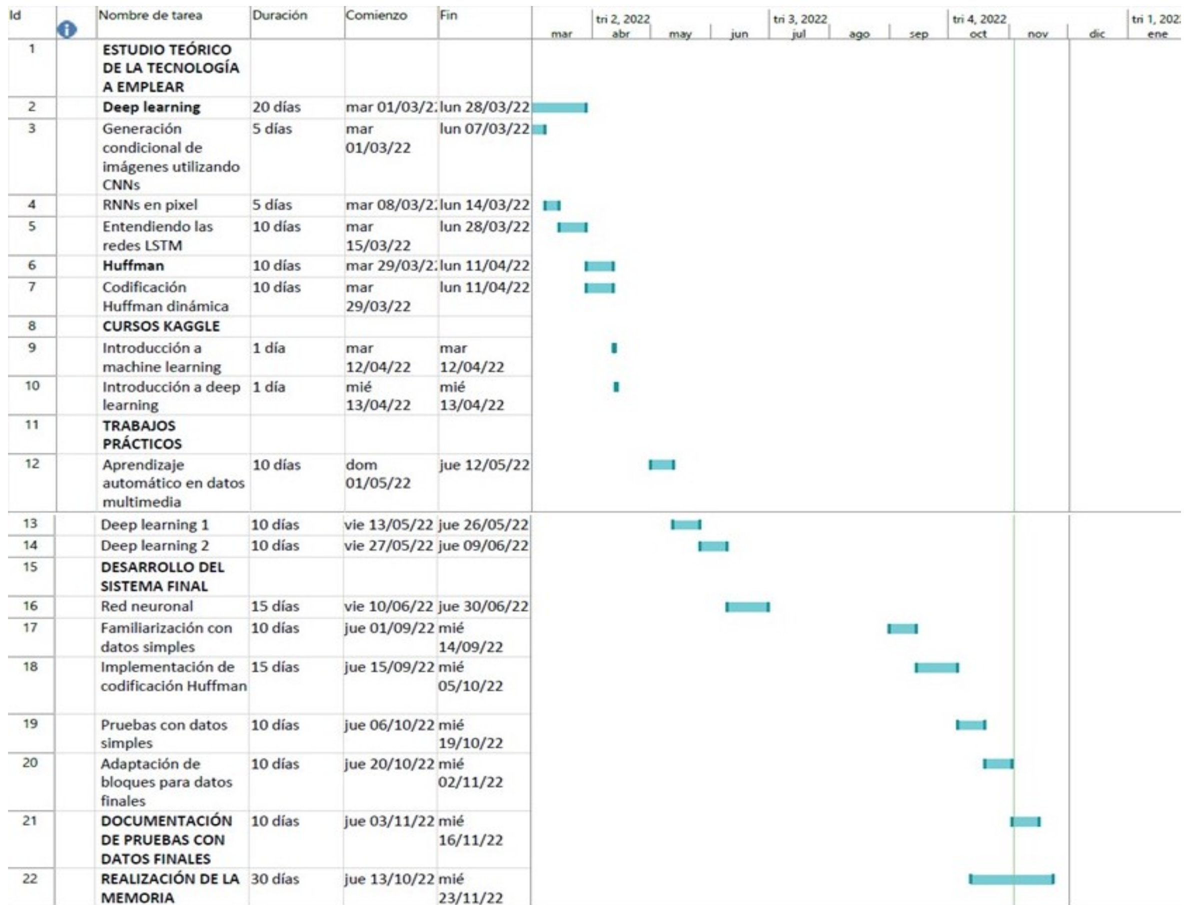
[2] Intro to Machine Learning es un curso de Kaggle de iniciación al mundo de Machine Learning.
[3] Intro to Deep Learning es un curso de Kaggle de iniciación al mundo de Deep Learning.

1.5. Gestión del proyecto

Una vez planteados los puntos a abordar a lo largo del proyecto, ya comentados y recogidos a alto nivel en la siguiente lista:

- Problemas a resolver.
- Metodología a poner en práctica.
- Hitos a cumplir.

Se ha planificado un mapa de ruta utilizando un diagrama de *Gantt*, en el que se propone un orden de ejecución de cada paquete de tareas con un tiempo estimado, con el fin de lograr los objetivos propuestos explotando al máximo el tiempo del que se dispone para la realización del proyecto. Dicho diagrama se puede observar en la Figura 1.5:



Id	Nombre de tarea	Duración	Comienzo	Fin
1	**ESTUDIO TEÓRICO DE LA TECNOLOGÍA A EMPLEAR**			
2	**Deep learning**	20 días	mar 01/03/2:	lun 28/03/22
3	Generación condicional de imágenes utilizando CNNs	5 días	mar 01/03/22	lun 07/03/22
4	RNNs en pixel	5 días	mar 08/03/2:	lun 14/03/22
5	Entendiendo las redes LSTM	10 días	mar 15/03/22	lun 28/03/22
6	**Huffman**	10 días	mar 29/03/2:	lun 11/04/22
7	Codificación Huffman dinámica	10 días	mar 29/03/22	lun 11/04/22
8	**CURSOS KAGGLE**			
9	Introducción a machine learning	1 día	mar 12/04/22	mar 12/04/22
10	Introducción a deep learning	1 día	mié 13/04/22	mié 13/04/22
11	**TRABAJOS PRÁCTICOS**			
12	Aprendizaje automático en datos multimedia	10 días	dom 01/05/22	jue 12/05/22
13	Deep learning 1	10 días	vie 13/05/22	jue 26/05/22
14	Deep learning 2	10 días	vie 27/05/22	jue 09/06/22
15	**DESARROLLO DEL SISTEMA FINAL**			
16	Red neuronal	15 días	vie 10/06/22	jue 30/06/22
17	Familiarización con datos simples	10 días	jue 01/09/22	mié 14/09/22
18	Implementación de codificación Huffman	15 días	jue 15/09/22	mié 05/10/22
19	Pruebas con datos simples	10 días	jue 06/10/22	mié 19/10/22
20	Adaptación de bloques para datos finales	10 días	jue 20/10/22	mié 02/11/22
21	**DOCUMENTACIÓN DE PRUEBAS CON DATOS FINALES**	10 días	jue 03/11/22	mié 16/11/22
22	**REALIZACIÓN DE LA MEMORIA**	30 días	jue 13/10/22	mié 23/11/22

Figura 1.5: Diagrama de Gantt.

Como se puede observar en el diagrama Gant, el proyecto se estructura en diferentes bloques principales que a su vez contienen subtareas, los cuales se han ido atajando de manera secuencial, en el otden que muestra el gráfico temporal de la parte derecha:

En primer lugar lugar, como se ha comentado previamente, se comienza estudiando el *Estado del arte* sobre la tecnología en torno a la que gira la parte técnica del proyecto.

Una vez completados dichos hitos de documentación previa a la ejecución del proyecto, se procede con la realización de cursos introductorios a Machine learning y Deep learning, a través de la plataforma Kaggle y obteniendo el correspondiente diploma tras su realización. Seguido de esto, se dedica un periodo de tiempo a la realización de trabajos prácticos donde se aplica lo aprendido anteriormente, trabajando ya con estructuras de datos similares a las del proyecto final y desarrollando el sistema en el lenguaje de programación Python, el mismo en el que se desarrollará el sistema final.

Una vez abordadas tareas de estudio y afianzamiento de la tecnología a emplear, se procede al desarrollo del sistema final, al completo. Este punto engloba tareas de familiarización con librerías a utilizar, desarrollo de fragmentos propios de cógido, así como la parte completa de codificación y decodificación, integración de las dos

anteriores y proceso de pruebas con los datos finales.

Finalmente, un proceso de generación de diferentes pruebas finales ha dado cuenta del correcto funcionamiento del sistema y ha permitido analizar las diferentes prestaciones del sistema para distintos tipos de configuración, documentando todas ellas e incluyendo las conclusiones pertinentes. Esta tarea se ha llevado a cabo en paralelo con la propia redacción de la memoria del proyecto.

1.6. Estructura de la memoria

La memoria del presente proyecto está formada por 5 capítulos. El primero de ellos introduce el **contexto** del trabajo: presentación de la tecnología que se va a emplear, y dentro de su gran versatilidad, énfasis en el potencial que se conoce en el campo en torno al que gira el proyecto, a la compresión sin pérdidas, en este caso, de señales de audio. Así pues, se localizan los diferentes objetivos a cumplir para poder desarrollar de manera propia un sistema que permita demostrar dicho potencial ya conocido de manera teórica, del estado del arte de la tecnología a utilizar.

El capítulo 2 gira en torno a la **tecnología** que se utiliza en el desarrollo del proyecto. Por una parte, la herramientas software en las que se ha desarrollado, MobaXterm y Visual Studio Code, así como el lenguaje de programación en el que se ha desarrollado el código del sistema. Además, del software, se dedica una sección a explicar conceptos que es necesario su entendimiento para comprender el funcionamiento de elementos como cada una de las redes empleadas, el tipo de error a través del cual se actualizan los parámetros de la red en cada iteración del proceso de entrenamiento, optimizador, scheduler y una sección particular para la codificación Huffman.

El capítulo 3 explica detalladamente la **implementación del sistema final**, donde se explica el funcionamiento bloque a bloque de todo el entramado que conforma el sistema final.

A continuación, el capítulo 4 recoge el conjunto de **pruebas experimentales** del proyecto, que han permitido evaluar las prestaciones del sistema desarrollado, y alcanzar los objetivos propuestos, demostrando que se cumplen las premisas de partida con respecto al Deep learning enfocado a la compresión sin pérdidas de señales de audio.

Por último, el capítulo 5 presenta una serie de **conclusiones** alcanzadas tras la elaboración del presente proyecto.

Capítulo 2

Tecnología

Este capítulo tiene como objetivo va a presentar la **tecnología** en torno a la que gira el presente proyecto, todo ello relacionado con el **deep learning** y diferentes conceptos propios de cualquier sistema que lo implemente. En él, se presentarán todos los puntos principales del presente sistema, pudiendo a su vez recogerlos en la parte relativa tanto al Deep learning como a la codificación Huffman.

2.1. Redes Neuronales Artificiales

La Red Neuronal Artificial (RNA) es probablemente la primera parada para cualquiera que se adentre en el campo del Deep Learning. Inspirada en la estructura de la red neuronal natural presente en nuestro cuerpo, la RNA imita una estructura y un mecanismo de aprendizaje similares.

La RNA no es más que un algoritmo para construir un modelo predictivo eficiente. Debido al algoritmo, y por tanto su implementación, se asemeja a una red neuronal típica, y el por ello que se denomina así. La funcionalidad de la RNA puede explicarse en los siguientes 5 sencillos pasos:

- Leer los datos de entrada
- Producir el modelo predictivo (una función matemática)
- Medir el error en el modelo predictivo
- Informar y aplicar las correcciones necesarias al modelo repetidamente hasta encontrar un modelo con el menor error
- Utilizar este modelo para predecir la incógnita

Para una explicación detallada del concepto de perceptrón, y cómo se agrupan varios de ellos para formar las capas de las redes neuronales, se recomienda visitar el Anexo B.4.

Teniendo en cuenta los cinco pasos anteriores, el proceso implica alimentar la entrada de una neurona en la siguiente capa para producir una salida utilizando una función de activación. Este proceso se denomina "Feed Forward". Después de producir la salida, se calcula el error (o la pérdida) y se envía una corrección a la red. Este proceso se denomina retropropagación".

Para la utilización de cualquier tipo de RNA, independientemente del software que se esté empleando, es necesario tener presentes los siguientes pasos a realizar:

- **Importar las librerías necesarias.** En el caso del presente proyecto serán las siguientes:
    - torch.nn [6]
- **Codificar la función de ajuste.** Sabemos que el algoritmo de descenso de gradiente requiere como entradas la tasa de aprendizaje *(learning rate)* y el número de iteraciones *(epoches)*.
- **Producir la salida y corregir el error.** Para el presente proyecto, como explicaremos más adelante en las secciones correspondientes, se han utilizado los siguientes métodos:
    - Pérdidas de Cross Entropía *(Cross-Entropy Lossess)* como función de cálculo de error.
    - Adam y Steepest Gradient Descent (de ahora en adelante, *SGD*) como optimizadores del modelo.
- **Ejecución del programa.** Para pasar las entradas y probar los resultados, es necesario elaborar unas mínimas líneas de código donde se incluyan los procesos de entrenamiento del modelo y de ejecución del mismo con datos de test para representar los resultados obtenidos y sacar las conclusiones pertinentes sobre la efectividad del mismo.

2.2. Redes LSTM

Como se menciona anteriormente en 1.4, las redes LSTM son un tipo concreto de RNN. Comenzamos por lo tanto presentando brevemente las RNN para exponer el problema al que se enfrentan en casos en los que es necesario que la red tenga memoria en un fragmento temporal amplio.

Para entender esta parte, se presenta la siguiente similitud. Los seres humanos no empiezan a pensar desde cero cada segundo. Por ejemplo, en la lectura de esta

memoria, se entiende cada palabra en base a la comprensión de las palabras anteriores. La información no es descartada, los pensamientos tienen persistencia.

Las redes neuronales tradicionales están bastante limitadas en este sentido. Por ejemplo, para un caso de clasificación de qué tipo de evento está ocurriendo en cada momento de una película. No está claro cómo una red neuronal tradicional podría utilizar su razonamiento sobre los acontecimientos anteriores de la película para informar sobre los posteriores. Las redes neuronales recurrentes abordan este problema. Son redes con bucles que permiten que la información persista.

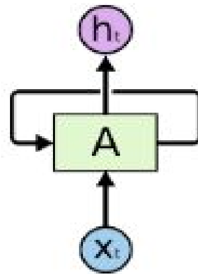


Figura 2.1: Estructura RNN.

En la Figura 2.1 se puede observar cómo un fragmento de red neuronal, A, observa una entrada xt y emite un valor ht. El bucle permite pasar la información de un paso de la red al siguiente.

Aunque el bucle de realimentación pueda hacer pensar lo contrario, las RNN no son tan diferentes de una red neuronal simple. Una RNN puede considerarse como múltiples copias de la misma red dispuestas en cascada, cada una de las cuales pasa un mensaje a su sucesora. En la Figura 2.2 se presenta la forma que tendría al ser dispuesta en cascada:

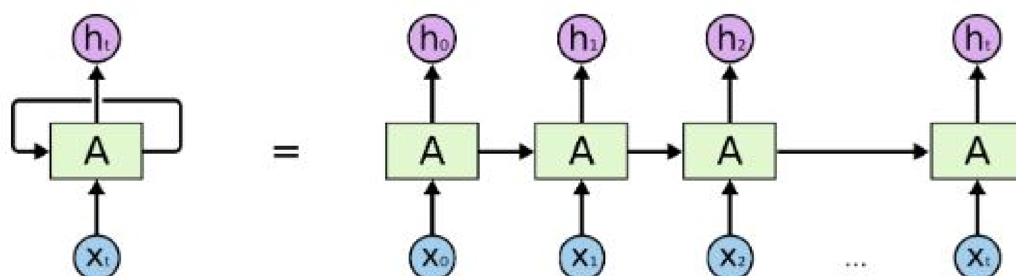


Figura 2.2: Estructura RNN en cascada.

Esta naturaleza encadenada revela que las redes neuronales recurrentes están íntimamente relacionadas con las secuencias y las listas, que son a su vez la arquitectura natural de las redes neuronales para usar esos datos. En los últimos años, se han conseguido increíbles éxitos aplicando las RNN a una gran variedad de problemas, entre ellos: reconocimiento del habla, modelado del lenguaje, traducción, subtitulación de imágenes...

Uno de los atractivos de las RNN es la idea de que puedan conectar la información anterior con la tarea actual, como por ejemplo, que el uso de fotogramas de vídeo anteriores pueda informar sobre la comprensión del fotograma actual. A veces, sólo necesitamos mirar la información reciente para realizar la tarea actual. En estos casos en los que la distancia entre la información relevante y el lugar en el que se necesita es pequeña, como el que se muestra en la Figura 2.3, las RNN resultan muy efectivas.

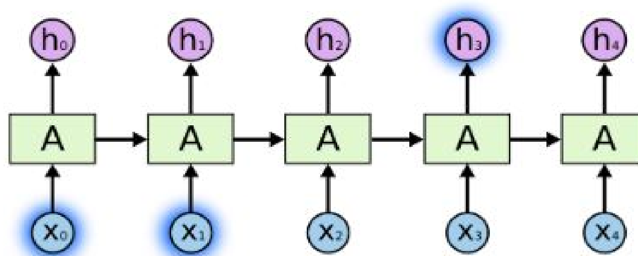


Figura 2.3: Estructura RNN en cascada con dependencia en periodo corto.

Pero hay casos en los que la brecha temporal entre la información relevante y el punto en el que se necesita es muy grande, como en el ejemplo de la Figura . Es aquí donde las RNN encuentran limitaciones a la hora de aprender a conectar la información, como se puede ver gráficamente en la Figura 2.4.

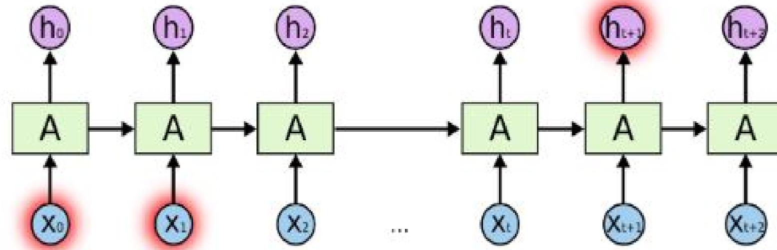


Figura 2.4: Estructura RNN en cascada con dependencia en periodo largo.

Las redes LSTM son un tipo especial de RNN, capaces de aprender dependencias en brechas temporales extensas. Así pues, están diseñadas explícitamente para evitar el problema de la dependencia a largo plazo. El módulo de repetición en una RNN estándar contiene una sola capa, como se observa en la Figura 2.5.

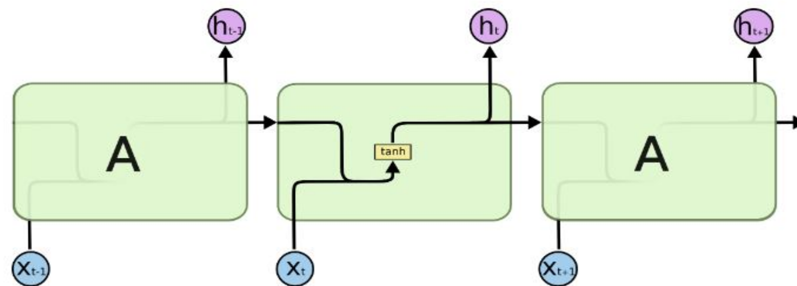


Figura 2.5: El módulo de repetición en una RNN estándar.

Las redes LSTM también tienen esta estructura en cadena, pero el módulo de repetición tiene una estructura diferente. En lugar de tener una sola capa de red neuronal, hay cuatro, de la manera que se muestra en la Figura 2.6

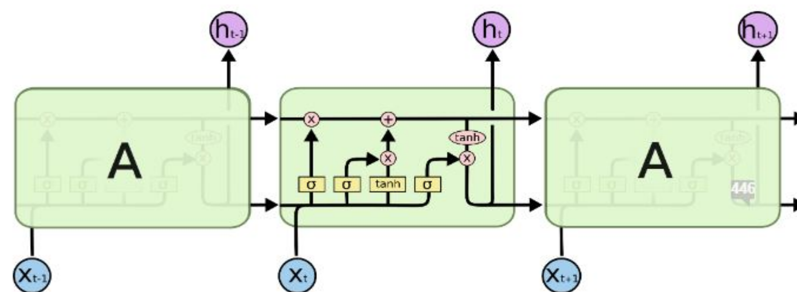


Figura 2.6: El módulo de repetición en una red LSTM.

A continuación, en la Figura 2.7, se adjunta la leyenda respectiva a la simbología empleada.

Figura 2.7: Leyenda de la simbología empleada en la Figura 2.6.

En la Figura 2.7, cada línea lleva un vector completo, desde la salida de un nodo hasta las entradas de otros. Los círculos rosas representan operaciones puntuales, como la suma de vectores, mientras que los recuadros amarillos son capas de la red neuronal aprendidas. Las líneas que se fusionan denotan la concatenación, mientras que una línea que se bifurca denota que su contenido se copia y las copias van a lugares diferentes.

La clave de las LSTM es el estado de las celdas, la línea horizontal que atraviesa la parte superior del diagrama. El estado de la célula es una especie de cinta transportadora. Corre en línea recta por toda la cadena, con sólo algunas interacciones lineales menores. De esta forma, es muy fácil que la información fluya a lo largo de ella sin cambios.

La red LSTM tiene la capacidad de eliminar o añadir información al estado de la célula, regulada por estructuras llamadas puertas, que son una forma de dejar pasar información de forma opcional. Las puertas están compuestas por una capa de red neuronal sigmoide y una operación de multiplicación puntual, que tiene lugar de la manera que se muestra en la Figura 2.8.

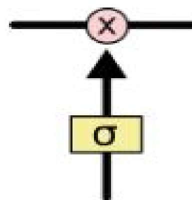


Figura 2.8: Capa sigmoide.

Para saber más a cerca del funcionamiento interno de las redes LSTM, se recomienda visitar la página citada en el capítlo 1.4.

2.3. Cross-Entropy Loss

La Cross-Entropy Loss *(Pérdidas de Entropía Cruzada)* es una de las funciones de pérdida más utilizadas para entrenar modelos de redes neuronales profundas, sobre todo en problemas de clasificación (multiclase). Cuando se aplica a datos categóricos, esta función de pérdida corresponde a una probabilidad logarítmica probabilística, lo que da lugar a propiedades de estimación favorables.

La explicación del método de Cross-Entropy Loss se apoya en la introducción de la distribución continua-categórica, donde se incluye un breve resumen notacional de la Cross-Entropy Loss categórica. Se recomienda consultar la página citada a continuación [7], la cual da cuenta de manera matemática de la estrecha conexión entre ambas.

En resumen, la Cross-Entropy Loss define un modelo probabilístico coherente para datos discretos sobre un conjunto de clases.

Se recuerda en este punto la función principal de nuestra red neuronal, la de adivinar cual será la muestra siguiente a la actual, que no es otra cosa que estimar cual es el valor con máxima probabilidad de ocurrencia. Así pues, al hilo de lo anterior y conectando con esta necesidad, se elije la Cross-Entropy Loss como método de error favorito para la elaboración del presente proyecto.

2.4. Optimizador: Adam y SGD

Las redes neuronales son algoritmos paramétricos. Los propios algoritmos se encargan de modelar el problema en función de los parámetros propios de la red, y no son otra cosa que una representación matemática propia de la red frente a dicho problema. Estos parámetros se estructuran en forma de matriz, para cada pareja de capas.

La función principal del optimizador es ajustar adecuadamente en cada iteración los valores de los parámetros, y de esta manera reducir la cantidad de error producido por la red. El término con el que se conoce comunmente a este proceso es *backpropagation*, y pese a ser complejo, no se considera oportuna una explicación detallada del mismo en la presente memoria.

El término más importante relativo al proceso de optimización es la **tasa de aprendizaje** *(learning rate)*. El valor del parámetro de tasa de aprendizaje determina a la velocidad que la red ajusta sus parámetros *"pesos"* para conseguir el mínimo error

en la salida de la misma. Así pues, una tasa de aprendizaje pequeña, hace que la red cambie el valor de los parámetros en dirección del vector de error en gran medida para cada iteración, mientras que para un valor pequeño de tasa de aprendizaje, la modificación en el valor de los pesos de la red será menor. Como se puede intuir, para cada escenario del entrenamiento es óptimo una tasa diferente, soliendo funcionar mejor tasas elevadas al comienzo y tasas reducidas conforme el error de la red tiende a estabilizarse en el mínimo.

A continuación, en la Figura 2.9, se muestran diferentes casos para tasas de aprendizaje ineficientes, y para el caso de tasa de aprendizaje óptimo:

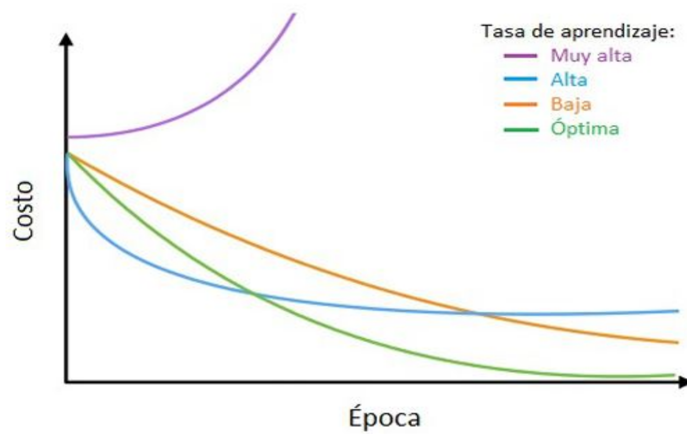


Figura 2.9: Error en función de diferentes tasas de aprendizaje.

Así pues, para el presente pryecto se considera la utilización del *Scheduler* de PyTorch, encargado de hacer decaer la tasa de aprendizaje de cada grupo de parámetros por el valor del parámetro gamma para cada número de epochs que se haya configurado. Los parámetros característicos de entrada de esta herramienta son los siguientes [8]:

– **optimizer *(Optimizador)*** - Optimizador envuelto, en nuestro caso, Adam y SGD.

– **step_size (int)** - Periodo de decaimiento de la tasa de aprendizaje.

– **gamma (float)** - Factor multiplicativo del decaimiento de la tasa de aprendizaje. Por defecto: 0,1.

– **last_epoch (int)** - El índice de la última época. Por defecto: -1.

– **verbose (bool)** - Si es True, imprime un mensaje en stdout para cada actualización. Por defecto: Falso.

Durante la realización del presente proyecto se han utilizado dos de los optimizadores más conocidos en la comunidad del Deep Learning, **Adam** y **SGD**:

– **Adam.** Adam es un algoritmo de optimización que puede utilizarse en lugar del procedimiento clásico de descenso de gradiente estocástico para actualizar los pesos de la red de forma iterativa basándose en los datos de entrenamiento. Al presentar el algoritmo, se enumeran las atractivas ventajas de utilizar Adam en problemas de optimización , a saber:

  - Fácil de implementar.
  - Es eficiente desde el punto de vista computacional.
  - Requiere poca memoria.
  - Es invariable a la reescalada diagonal de los gradientes.
  - Adecuado para problemas que son grandes en términos de datos y/o parámetros.
  - Apropiado para objetivos no estacionarios.
  - Apropiado para problemas con gradientes muy ruidosos o dispersos.
  - Los hiperparámetros tienen una interpretación intuitiva y suelen requerir poco ajuste.

Adam es diferente al descenso de gradiente estocástico clásico. El descenso de gradiente estocástico mantiene una única tasa de aprendizaje (denominada alfa) para todas las actualizaciones de pesos y la tasa de aprendizaje no cambia durante el entrenamiento. Se mantiene una tasa de aprendizaje para cada peso de la red (parámetro) y se adapta por separado a medida que se desarrolla el aprendizaje. Los autores describen a Adam como una combinación de las ventajas de otras dos extensiones del descenso de gradiente estocástico. En concreto:

- Algoritmo de Gradiente Adaptativo *(AdaGrad)* que mantiene una tasa de aprendizaje por parámetro que mejora el rendimiento en problemas con gradientes dispersos (por ejemplo, problemas de lenguaje natural y visión por ordenador).
- Propagación de la raíz cuadrada *(RMSProp)* que también mantiene tasas de aprendizaje por parámetro que se adaptan en función de la media de las magnitudes recientes de los gradientes para el peso (por ejemplo, la rapidez con la que está cambiando). Esto significa que el algoritmo funciona bien en problemas en línea y no estacionarios (por ejemplo, ruidosos).

Adam aprovecha las ventajas tanto de AdaGrad como de RMSProp.

En lugar de adaptar las tasas de aprendizaje de los parámetros basándose en la media como en RMSProp, Adam también hace uso de la media de los segundos momentos de los gradientes (la varianza no centrada).

En concreto, el algoritmo calcula una media móvil exponencial del gradiente y del gradiente al cuadrado, y los parámetros beta1 y beta2 controlan las tasas de decaimiento de estas medias móviles.

El valor inicial de las medias móviles y los valores de beta1 y beta2 cercanos a 1,0 (recomendado) provocan un sesgo de las estimaciones de momentos hacia cero. Este sesgo se supera calculando primero las estimaciones sesgadas antes de calcular después las estimaciones corregidas por el sesgo.

Los parámetros de configuración característicos del método de optimización Adam son los siguientes:

- **alfa.** También se denomina tasa de aprendizaje o tamaño del paso. Es la proporción en que se actualizan los pesos (por ejemplo, 0,001). Los valores más grandes (por ejemplo, 0,3) dan lugar a un aprendizaje inicial más rápido antes de que se actualice la tasa. Los valores más pequeños (por ejemplo, 1,0E-5) ralentizan el aprendizaje durante el entrenamiento.

- **beta1.** La tasa de decaimiento exponencial para las estimaciones del momento de primer orden (por ejemplo, 0,9).
- **beta2.** La tasa de decaimiento exponencial para las estimaciones del momento de segundo orden (por ejemplo, 0,999). Este valor debe establecerse cerca de 1,0 en problemas con un gradiente disperso (por ejemplo, problemas de PNL y de visión por ordenador).
- **epsilon.** Es un número muy pequeño para evitar cualquier división por cero en la implementación (por ejemplo, 10E-8).

– **SGD.** SGD es uno de los métodos de optimización más utilizado en el mundo del Deep Learning, cuya principal característa por la que destaca de entre sus competidores es la gran reducción del tiempo de computación que se experimenta con su utilización. El hecho de que consiga simplificar el proceso y ahorrar tiempo se debe a que solo emplea una muestra aleatoria de entre las posibles del conjunto de las de entrenamiento, y en base a ella se actualizan los pesos de la red. Es una particularización del algoritmo Gradient Descent *(de ahora en adelante, GD)*, el cual considera el total de muestras de cada iteración para contribuir a la actualización de pesos, lo que supone, en casos para los que se trabaja con cantidades de información muy grandes, un coste computacional mucho mayor. Así pues, mientras con GD el error converge de manera más directa, utilizando SGD se experimentan los conocidos como *'serpenteos'* en la función que calcula el error, pero como ventaja, suele alcanzar el error mínimo más rápido. Por esto, depende del conjunto de datos con el que se trabaje, será más efectivo la utilización de uno u otro. A continuación, en la Figura 2.10, puede observarse las curvas de error para cada uno de los dos métodos:

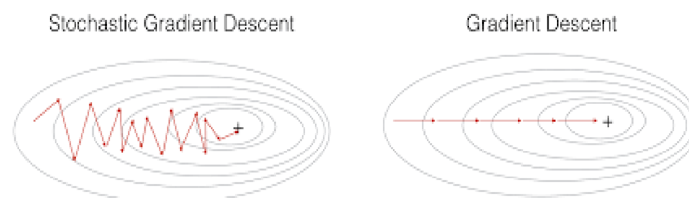


Figura 2.10: Convergencia para los métodos de optimización SGD y GD.

Dado el set de datos con el que se ha trabajado en el presente proyecto, se ha considerado utilizar el algoritmo de optimización SGD junto a Adam, para los que veremos más adelante, cuan efectivos han sido ambos para diferentes casos de prueba.

2.5. Codificación Huffman

La codificación Huffman es un algoritmo de compresión de datos sin pérdidas. La idea es asignar códigos de longitud variable a los caracteres de entrada; las longitudes de los códigos asignados se basan en las frecuencias de los caracteres correspondientes. El carácter más frecuente recibe el código más pequeño y el menos frecuente el más grande. Los códigos de longitud variable asignados a los caracteres de entrada son códigos prefijados, lo que significa que los códigos (secuencias de bits) se asignan de tal manera que el código asignado a un carácter no es el prefijo del código asignado a cualquier otro carácter. Así es como la codificación Huffman se asegura de que no haya ambigüedad al decodificar el flujo de bits generado.

Existen dos partes principales en el proceso de codificación Huffman:

- Construir un Huffman Tree *(Árbol de Huffman)* a partir de los caracteres iniciales.
- Recorrer el Huffman Tree y asignar códigos a dichos caracteres.

Para comprender los principios de la codificación Huffman, a continuación se plantea un **sencillo problema** [9]. Supongamos que se dispone de un texto que está compuesto únicamente de 5 caracteres, *'A'*, *'B'*, *'C'*, *'D'* y *'E'*, y ocurren con frecuencia 15, 7, 6, 6 y 5, respectivamente. A continuación, la Figura 2.11 muestra los caracteres junto a sus frecuencias, lo que llamaremos nodos.

Para construir el Huffman Tree y obtener la codificación de cada caracter, el primer paso del algoritmo es crear una cola de prioridad con los nodos, donde los de mayor prioridad son los que tienen menor frecuencia, como se muestra en la Figura 2.11.

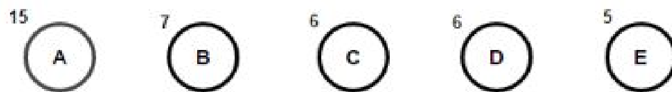


Figura 2.11: Nodos ordenados según prioridad (mayor a la derecha).

Mientras exista más de un nodo en la cola de prioridad, el algoritmo fusiona los dos nodos de prioridad más alta y crea un nuevo nodo interno, con dichos dos nodos como nodos hijos. La frecuencia del nuevo nodo es la suma de las frecuencias de los dos nodos hijos. A continuación, en la Figura 2.12, se muestra cómo quedaría la cola de prioridad tras la primera fusión.

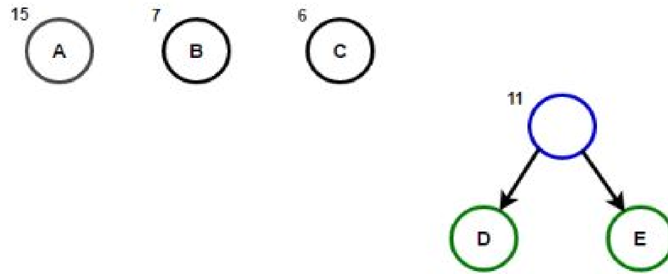


Figura 2.12: Cola de prioridad tras fusión de nodos 'D' y 'E'.

El proceso se repite hasta que solamente quede un solo nodo en la cola de prioridad, como se sigue en las Figuras 2.13 y 2.14.

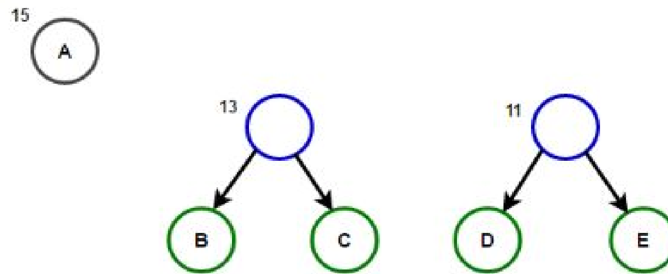


Figura 2.13: Cola de prioridad tras tercera fusión de nodos.

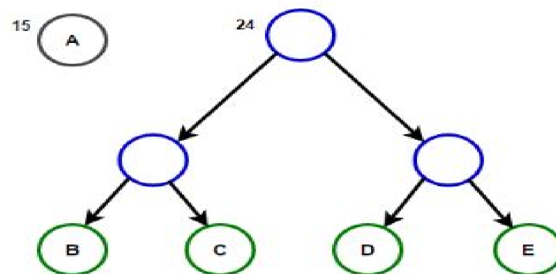


Figura 2.14: Cola de prioridad tras última fusión de nodos.

Este será el nodo raiz del Huffman Tree, como se muestra en la Figura 2.15.

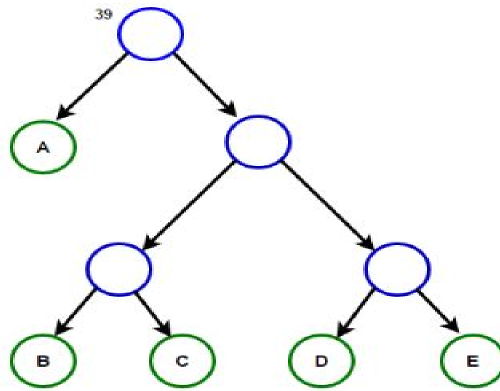


Figura 2.15: Huffman Tree completado.

Una vez construido el Huffman Tree, se asigna un 1 y un 0 a cada camino que sale de un nodo a otro. Finalmente, cada caracter tendrá la codificación resultante de concatenar los unos y ceros que sigue la ruta que va desde el nodo raiz hasta cualquier caracter, y el proceso de codificación Huffman habrá sido completado, como se muestra en la Figura 2.16.

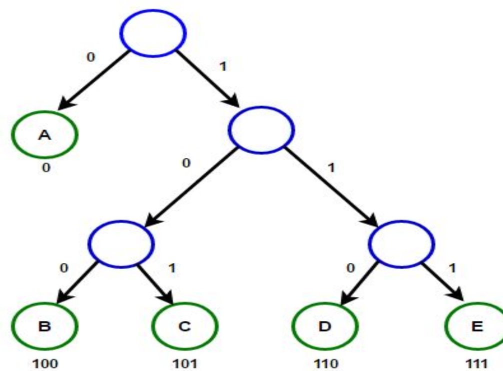


Figura 2.16: Codificación de cada caracter.

Capítulo 3

Implementación del sistema final

Este capítulo se enfoca puramente en el análisis detallado del sistema final implementado en el presente proyecto. En él se se hará un estudio detallado de la red neuronal utilizada, y se detallará todo el proceso que tendría lugar en la parte del emisor, al igual que para la parte del receptor, con los procesos de entrenamiento de la red, codificación y decodificación necesarios. Además, se expondrá el tipo de señal con el que se ha trabajado, además de los parámetros clave del sistema que han sido clave para la realización de pruebas simulando diferentes escenarios, y gracias a los cuales se ha llegado a elaborar las conclusiones pertinentes que concluyen el presente proyecto.

3.1. Red

La red empleada para el proyecto, como se ha mencionado anteriormente, se compone, primeramente, por una red LSTM, y posteriormente, por una capa Lineal, como se muestra en la Figura 3.1. La estructura propia de la red como todas las funciones definidas dentro de ella se incluyen en la Figura A.1.

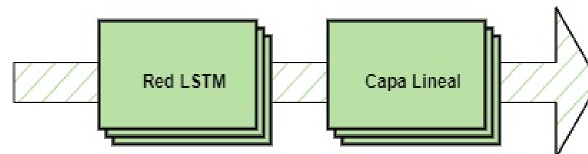


Figura 3.1: Estructura de la red neuronal empleada.

Comenzaremos por analizar en primer lugar la parte de la **red LSTM**. Para su implementación, se ha empleado la librería nn.LSTM de PyTorch, donde se puede encontrar toda la documentación relativa a la misma [10].

Gracias a esta librería, se aplica una RNN multicapa de memoria a corto-largo plazo (LSTM) a una secuencia de entrada. Para cada elemento de la secuencia de entrada, cada capa calcula la función que se ve en la imágen de la Figura 3.2:

$$
\begin{aligned}
i_t &= \sigma(W_{ii}x_t + b_{ii} + W_{hi}h_{t-1} + b_{hi}) \\
f_t &= \sigma(W_{if}x_t + b_{if} + W_{hf}h_{t-1} + b_{hf}) \\
g_t &= \tanh(W_{ig}x_t + b_{ig} + W_{hg}h_{t-1} + b_{hg}) \\
o_t &= \sigma(W_{io}x_t + b_{io} + W_{ho}h_{t-1} + b_{ho}) \\
c_t &= f_t \odot c_{t-1} + i_t \odot g_t \\
h_t &= o_t \odot \tanh(c_t)
\end{aligned}
$$

Figura 3.2: Función calculada para cada capa de la red LSTM, para cada elemento de la secuencia de entrada.

donde h_t es el estado oculto en el momento t, c_t es el estado de la célula en el momento t, x_t es la entrada en el momento t, h_{t-1} es el estado oculto de la capa en el tiempo t-1 o el estado oculto inicial en el tiempo 0, e i_t, f_t, g_t y o_t son las puertas de entrada, olvido, célula y salida, respectivamente. σ es la función sigmoidea, y $\odot$ es el producto de Hadamard, punto a punto.

En una red LSTM multicapa, la entrada $x_t^{(l)}$ de la capa l-ésima ($l >= 2$) es el estado oculto $h_t^{(l-1)}$ de la capa anterior multiplicado por el término olvidado $\delta_t^{(l-1)}$, donde cada $\delta_t^{(l-1)}$ es una variable aleatoria Bernoulli que es de valor 0 con probabilidad *dropout (parámetro de entrada)*.

Así pues, los parámetros de entrada de la función de la red LSTM de interés en el proyecto son los siguientes:

– *input_size*. El número de elementos esperadas en la entrada x, en nuestro caso:

  - **Entrenamiento de la red.** Para el entrenamiento de la red se introducen, salvo para determinadas pruebas, todas las muestras de cada señal de audio del conjunto de audios de entrenamiento, menos la primera de ellas.
  - **Proceso de codificación/decodificación.** En el proceso de codificación/decodificación, unicamente se introduce una muestra a la entrada de la red, ya que la única finalidad es obtener como salida el vector de probabilidades de la muesta siguiente.

– *hidden_size*. El número de elementos en el estado oculto h.

– *num_layers*. Número de capas recurrentes. Por ejemplo, establecer $num_layers = 2$ significaría apilar dos LSTMs juntos para formar un LSTM apilado, con el segundo LSTM tomando las salidas del primer LSTM y calculando los resultados finales.

– *bias*. Si es *False*, la capa no utiliza los pesos de sesgo b_ih y b_hh. Por defecto, el valor es $True$

- *batch_first*. Si es *True*, los tensores de entrada y salida se proporcionan como (*batch*, *seq*, *feature*) en lugar de (*seq*, *batch*, *feature*). Es importante tener en cuenta que esto no se aplica a los estados ocultos o de celda. Para más detalles, consulte las secciones de *Entradas/Salidas* de la página de la librería. Por defecto, el parámetro se inicializa a *False*.

- *dropout*. Si es distinto de cero, introduce una capa de eliminación de información en las salidas de cada capa LSTM excepto la última, con una probabilidad de eliminación igual a la de *dropout*. Por defecto, el parámetro se inicializa a 0.

- *bidirectional*. Si es *True*, se convierte en un LSTM bidireccional. Por defecto, el parámetro se inicializa a *False*.

Después de la red LSTM, el sistema está compuesto por una **capa Lineal**. Para su implementación, se ha empleado la librería nn.LSTM de PyTorch, donde se puede encontrar toda la documentación relativa a la misma [11].

El sistema aplica la siguiente transformación lineal a los datos entrantes:

$$y = x * A^T + b$$

A continuación se listan los parámetros de entrada de la función de la capa Lineal de interés:

- *in_features (int)*. Tamaño de la muestra de entrada.

- *out_features (int)*. Tamaño de la muestra de salida.

- *bias (bool)*. Si se establece en *False*, la capa no aprenderá un sesgo aditivo. Por defecto, el parámetro se inicializa a *True*.

Además, al inicializar la red, se declara el elemento que utilizaremos para calcular el error cometido por la red, que como se ha mencionado anteriormente, es de tipo entropía cruzada. Para su utilización, se ha empleado la librería de PyTorch que la implementa [12].

Este criterio calcula la pérdida de entropía cruzada entre los elementos de entrada y el objetivo. Es útil cuando se entrena un problema de clasificación con C clases, como el caso del presente proyecto, donde cada clase, como se ha mencionado anteriormente, es uno de los 256 valores que puede tomar la señal de audio. Si se proporciona, el argumento opcional *weight* debe ser un tensor 1D que asigne peso a cada una de las clases. Esto es particularmente útil cuando se tiene un conjunto de entrenamiento desequilibrado.

Por último, destacar la función desarrollada *predict*, la cual se encarga de pasar los datos de entrada por la red, y devuelve la los datos de salida de la misma, tras ser pasados previamente por la función *softmax* [1], además del estado de la red LSTM. De esta manera, los datos de salida de la red están preparados para ser utilizados por el método Huffman, el cual espera un vector de probabilidades de donde cada elemento pertenezca al conjunto [0,1] y la suma de todos ellos sea igual a 1. Así pues, el elemento de salida *estado* será el utilizado en los procesos de codificación/decodificación para que la red LSTM sepa en cada momento el estado en el que queda tras introducir cada entrada.

3.1.1. Proceso de entrenamiento

Como se ha indicado anteriormente, el objetivo final con el que se entrena a la red es el de que sea capaz de predecir el siguiente valor de la señal de entrada. Para complementar gráficamente la manera mediante la que se consigue dicho resultado, se muestra el conjunto de entradas y salidas a la red LSTM en la Figura 3.3.

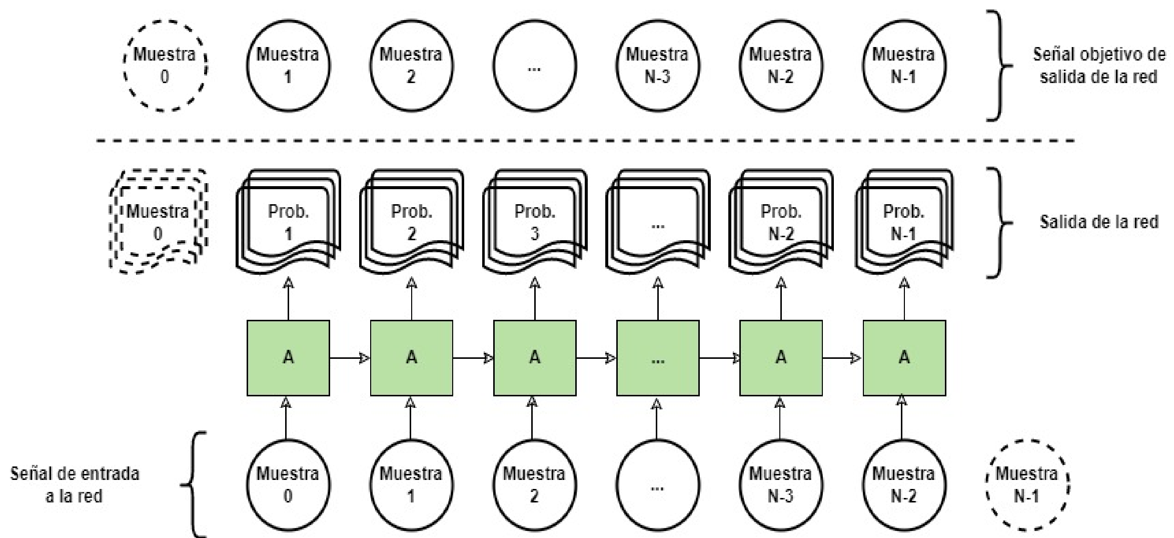


Figura 3.3: Conjunto de entradas y salidas a la red LSTM.

Como se aprecia en el diagrama recién presentado, la **entrada** para una iteración concreta corresponde con una señal de audio a la que le falta la última muestra. De manera intuitiva, la señal de **salida objetivo** habrá de ser la misma señal de entrada desplazada, de manera que contenga todas las muestras de la señal salvo la primera. Así, para cada muestra de entrada, la red LSTM da como salida una matriz de 256 elementos, donde cada uno corresponde a uno de los posibles valores de la señal de audio, y el contenido de cada elemento es la probabilidad de que la muestra siguiente

[1] La función *softmax* es la encargada de transformar los elementos de un vector a otros de valores entre [0,1], de manera que la suma de todos los elementos de dicho vector sea igual a 1.

sea la asociada a dicho elemento. De esta manera, la salida total obtenida en cada iteración es una matriz de $N_{muestras_entrada}x256$.

La red estará mejor entrenada para predecir la siguiente muestra, por lo tanto, cuando el valor del elemento que corresponde a la muestra siguiente sea máximo (máxima probabilidad) y el de los elementos restantes sea el menor posible (menor probabilidad).

3.2. Codificación

La idea principal de la sinergia entre la red neuronal del sistema y el proceso de codificación, como se viene comentando, es conseguir que la red sea capaz de predecir la muestra siguiente del archivo de audio con la mayor exactitud posible. De esta manera, el codificador asignará el menor número de bits a los valores comprendidos en un conjunto lo más pequeño y concentrado posible, dejando mayor número de bits para los demás valores. Esto no es otra cosa que estimar la función de probabilidad de cada muestra. El mejor de los casos sería una función de tipo pico, ya que se asignaría un bit al valor del pico y mayor número de bits a los valores laterales a medida que decrementa dicha función. Del mismo modo, el peor de los casos sería una función de tipo uniforme, ya que todos los valores tendrían la misma probabilidad de ocurrencia y la asignación de bits sería totalmente aleatoria.

Así pues, podemos entender los dos extremos con los siguientes ejemplos, supuesto que se conocen valores previos de cada uno:

- Predecible con certeza: señal sinusoidal.

- No predecible con certeza: lanzamiento de un dado.

Para ilustrar la explicación anterior, se muestra en la Figura 3.4 la representación del caso de una señal sinusoidal, y cómo se decidiría la asignación de número de bits.

Para simplificar y automatizar al máximo posible cada proceso singular del ejercicio de **codificación** y **decodificación**, se ha creado una clase llamada ***CodDecod*** que incorpora las funciones para ámbos ejercicios.

Como se puede ver en la Figura A.2, en la función de inicialización del método se encuentran los siguientes parámetros:

- x_true. Toma el valor de la muestras reales de la señal de audio.

- x_t0. Toma el valor de la primera muestra que se introduce a la red.

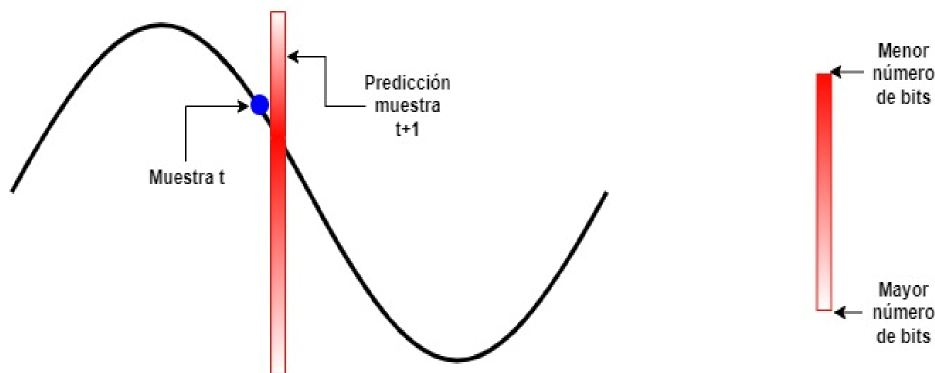


Figura 3.4: Ejemplo de predicción y asignación de número de bits.

- *symbols*. Es un vector que contiene el abanico completo de valores posibles de las muestras de la señal de audio.

- *model*. Es el modelo de la red neuronal.

3.2.1. Codificador

Para el proeso de codificación, se ha creado una función dentro de la clase *CodDecod* llamada ***Code***, la cual se muestra en la Figura A.3.

En ella puede verse que está formada principalmente por un bucle encargado de iterar tantas veces como elementos existen en el elemento que contiene la señal de audio a codificar. Así pues, para cada iteración, la función sigue los siguientes pasos:

- Se llama a la función *predict* introduciendo el valor de estado s (inicializado a *None*) devuelto por la red y la muestra de audio x_t de la iteración anterior del bucle. De esta manera, la función devuelve un nuevo estado y una matriz de probabilidades de ocurrencia para la muestra siguiente.

- Se actualiza el valor de la muestra real siguiente, que será introducida a la red en la iteración siguiente, de la misma forma descrita en el punto anterior.

- Se crea el diccionario *dictionary* con la matriz de símbolos *symbols* y las probabilidades de ocurrencia de cada uno, en formato adecuado.

- Se crea el arbol de Huffman mediante el cual se codificará la muestra en cuestión.

- Se llama a la función $huffman_encode$ y se guarda la muestra dodificada u en la variable que alberga las muestras anteriores ya codificadas, $message_encoded$.

Una vez finalizadas las iteraciones, se encuentra la señal al completo codificada en la matriz *message_encoded*, y esta es devuelta por la función.

A continuación, se adjunta en la Figura 3.5 un diagrama de bloques que permite seguir el proceso de codificación recien descrito, a alto nivel y de una forma más conceptual.

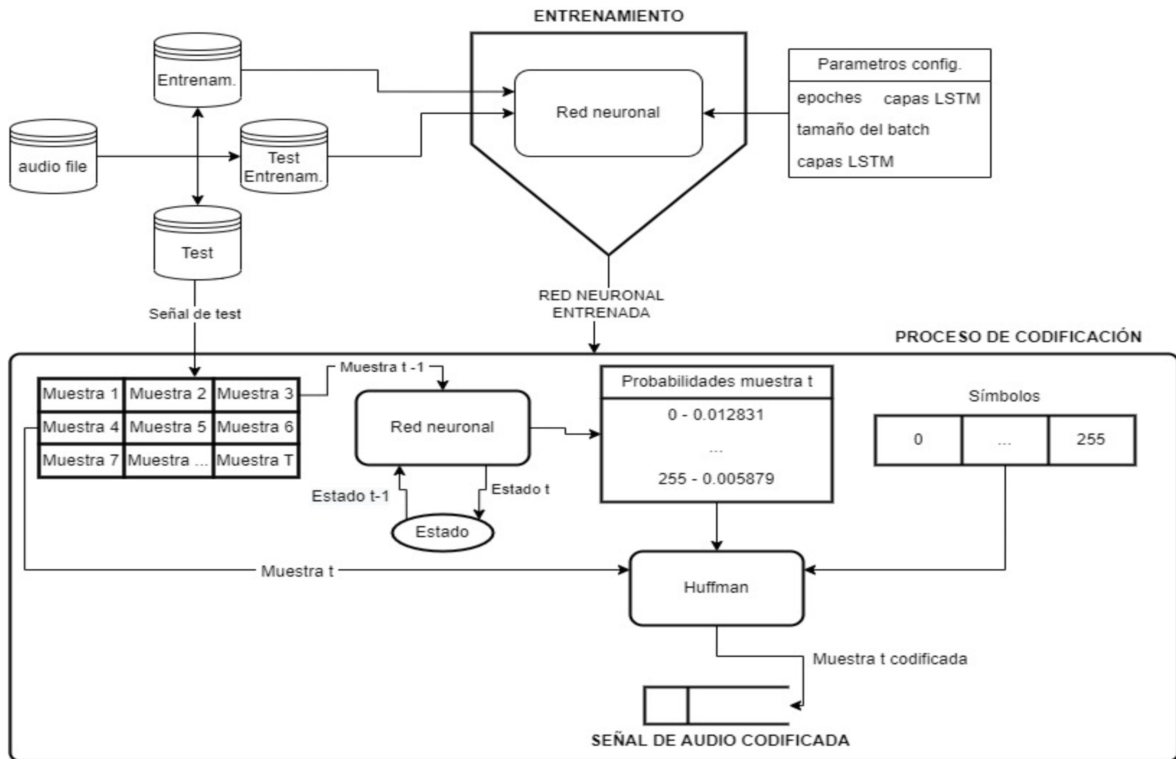


Figura 3.5: Diagrama de bloques del proceso de codificación.

3.2.2. Decodificador

De la misma manera que para el proceso de codificación, se ha creado una función dentro de la clase *CodDecod* llamada ***Decode*** para el proceso de decodificación, la cual se muestra en la Figura A.4.

De manera muy similar al proceso que tiene lugar en la función de codificación, en la decodificación se itera repetidamente sobre un bucle tantas iteraciones como muestras de la señal. Para el caso concreto del proyecto, se ha optado por simplicidad por delimitar el número de elementos a recibir como si el receptor lo conociese con anterioridad. En el caso práctico, existirían diferentes formas de detectar el final del mensaje, como podría ser un caracter determinado, o un conjunto de ellos. Así pues, para cada iteración, se siguen los siguientes pasos:

- Se llama a la función *predict* introduciendo el valor de estado s (inicializado a *None*) devuelto por la red y la muestra de audio recibida y decodificada x_t de la

iteración anterior del bucle. De esta manera, la función devuelve un nuevo estado y una matriz de probabilidades de ocurrencia para la muestra que se espera recibir.

– Se crea el diccionario *dictionary* con la matriz de símbolos *symbols* y las probabilidades de ocurrencia de cada uno, en formato adecuado.

– Se crea el arbol de Huffman mediante el cual se decodificará la muestra en cuestión.

– Para cada bit recibido, se concatena en la variable $word_encoded$ y se comprueba si existe en el diccionario creado para la iteración en cuestión una palabra codificada coincidente con la misma.

  - En caso afirmativo, se llama a la función $huffman_decode$ introduciendo la palabra codificada recibida y el diccionario, y la misma, devuelve el valor de la muestra real recibida, se concatena en la variable *message_decoded*.
  - En caso negativo, se concatena el siguiente vez recibido y se repite el proceso de intento de mapeo de dicha palabra recibida con los símbolos del diccionario.

Una vez finalizadas las iteraciones, se ha recibido y decodificado la señal al completo y se encuentra almacenada en la variable *message_decoded*, y esta es devuelta por la función.

A continuación, se adjunta en la Figura 3.6 un diagrama de bloques que permite seguir el proceso de decodificación recien descrito, a alto nivel y de una forma más conceptual.

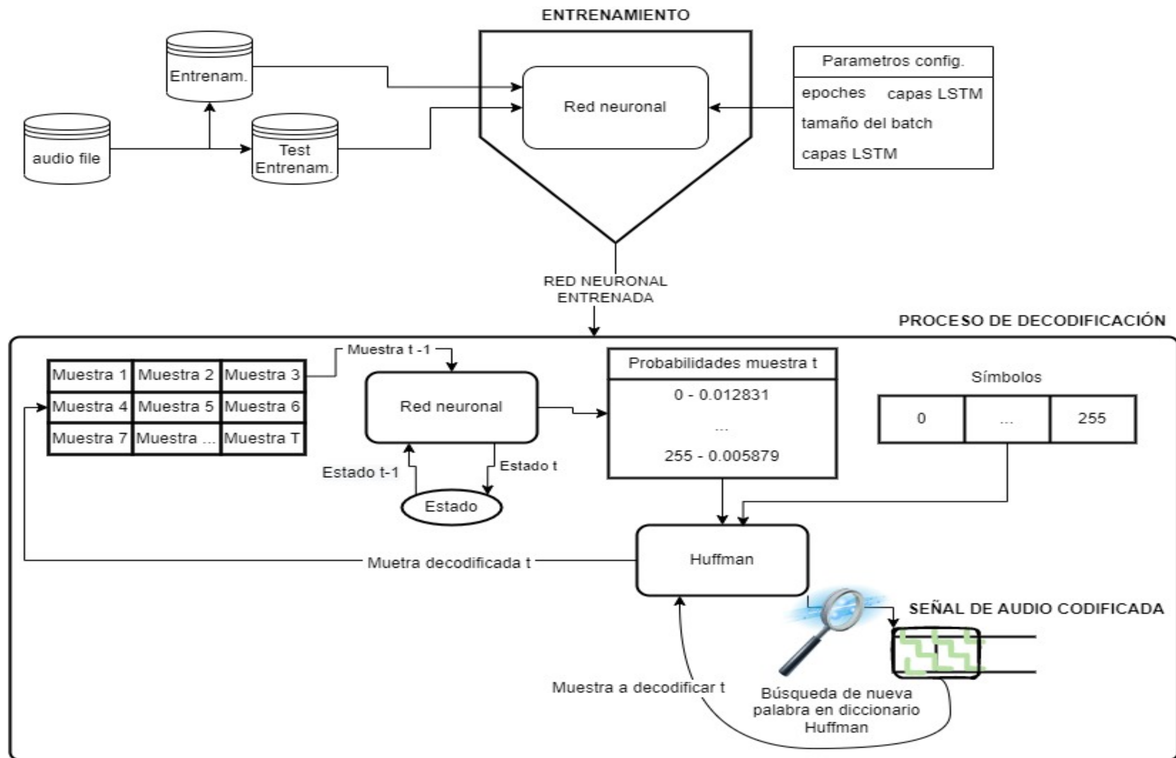


Figura 3.6: Diagrama de bloques del proceso de codificación.

3.2.3. Tipo de señal

Como se ha comentado en todo momento en la presente memoria del proyecto, el tipo de información con el que se ha desarrollado el mismo es un archivo de 16.185 señales de audio muestreadas a 8KHz, con una precisión de muestra de 8 bits.

Habiendose alterado en diferentes ocasiones la proporción de señales dedicadas a entrenamiento, evaluación durante entrenamiento y testeo, la proporción que más se ha empleado ha sido la siguiente:

- Entrenamiento de la red: 15.985 señales de audio.
- Evaluación durante entrenamiento: 100 señales de audio.
- Testeo: 100 señales de audio.

Cada señal pues, corresponde a un fragmento de voz de 1 segundo, lo que se corresponde, según la tasa de muestreo, a una matriz de 8.000 elementos.

Las señales de audio tienen la apariencia que se muestra en la Figura 3.7, donde puede apreciarse que efectivamente están formadas por 8.000 muestras de valor comprendido entre 0 y 255.

Cabe destacar que la representación muestra la señal de audio original y la señal de audio recuperada tras el proceso de codificación-decodificación. Del hecho de desarrollar

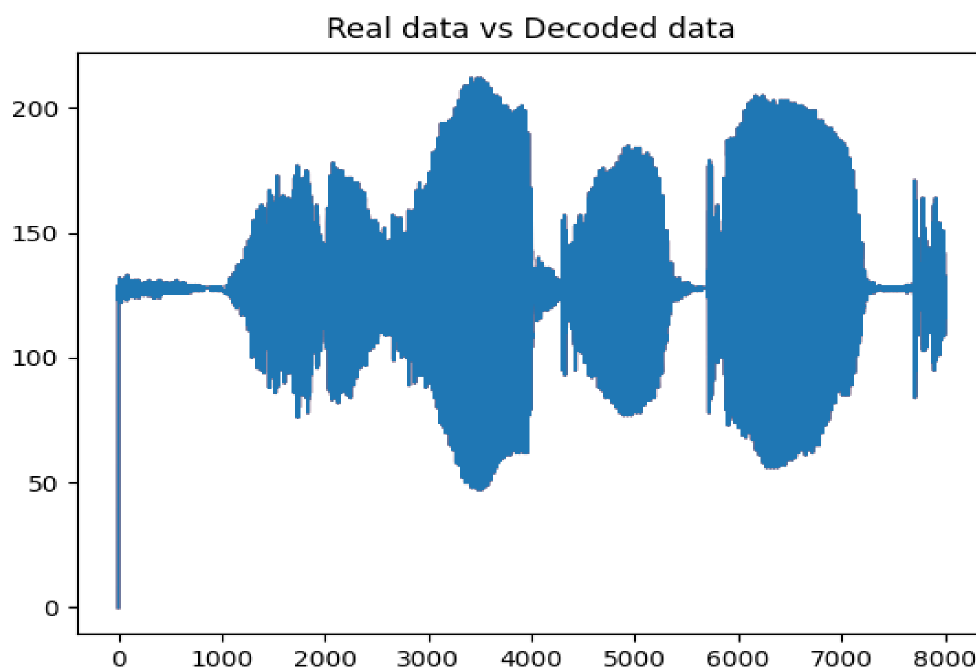


Figura 3.7: Señal de audio sin procesar y señal de audio decodificada.

un sistema de compresión sin pérdidas, se aprecia solamente una señal ya que se superpone una de ellas sobre la otra. Además, mencionar que la primera de las muestras de todas las señales se pone a 0 para que la red sea capaz de identificar el comienzo de la señal.

3.2.4. Parámetros clave

De entre todos los parámetros que constituyen el sistema final, una serie de ellos intervienen realmente en funciones que determinan el resultado final en términos de tasa de compresión, y en ellos es en los que el presente proyecto pone el foco, y altera en diferentes ocasiones para sacar conclusiones de las diferencias observadas tras la ejecución.

Dichos parámetros clave del proyecto son los que se listan a continuación:

- T. Tiempo de la señal. Es el número de muestras que tiene cada señal. En ocasiones, se considera reducir este parámetro para estudiar el resultado de compresión para una red que ha sido entrenada con fragmentos menores de señal.

- N. Número de señales totales. Además de experimentar con la proporción dedicada a cada parte del proceso (recordamos: entrenamiento, evaluación durante entrenamiento y testeo), también se experimenta a reducir el número

total de señales del conjunto del que se extraen las dedicadas a cada una de dichas partes.

- *nb_lstm_layers*. Número de capas de red LSTM.
- *hidden_dim*. Número de capas (dimensiones) ocultas, internas a cada capa LSTM.

Con la variación de cada uno de los parámetros clave del presente proyecto, listados anteriormente, se obtendrán diferentes resultados de compresión, como veremos más adelante.

Capítulo 4

Resultados experimentales

Una vez presentados tanto requerimientos como tecnología que fundamenta el presente proyecto, además de parámetros clave que pondrán a prueba la efectividad del sistema desarrollado en el mismo, en esta sección capítulo se pretende presentar con detalle las diferentes tasas de compresión obtenidas para los distintos escenarios considerados en función de los diferentes parámetros clave del sistema 3.2.4, elaborando las conclusiones oportunas para cada uno de los casos.

Con esto, también se pretende comparar el sistema desarrollado en el presente proyecto con sus homólogos comúnmente utilizados en la industria actual, dando cuenta de lo lejos que se ha llegado en este campo.

Así pues, para cada escenario determinado, se presenta en formato de tabla los parámetros clave de configuración del sistema y se incluyen ciertos gráficos que pueden ayudar a interprestar los resultados obtenidos. Los parámetros que ocupan la cabecera de la tabla se corresponden, de izquierda a derecha, con el número de *epoches* del que consta el entrenamiento de la red, el número de señales dedicadas al entrenamiento de la red, el número de muestras que contiene cada señal de las del conjunto de entrenamiento, el número de capas LSTM, el número de capas ocultas, el tamaño de la señal original (bits/muestra), el tamaño de la señal comprimida (bits/muestra) y la tasa de compresión resultante.

Con ánimo de comparar el sistema desarrollado con alguno de sus similares, de características semejantes y utilización general, se presentan los resultados obtenidos por el sistema presente modificando diferentes parámetros junto con los resultados de compresión de los sistemas **Shorten** y **ZIP**, obtenidos mediante dos archivos ejecutables *.py* que comprimen 100 de las señales contenidas en el conjunto utilizado en el presente proyecto. Puesto que no utilizan técnicas de Deep Learning para el proceso de codificación, los campos dentro de la tabla relativos a ello se completan con el carácter " $-$ ".

Antes de nada, se presenta una breve presentación de cada uno de ellos:

- **Shorten.** Esta técnica reduce el tamaño de aquellos archivos con forma de onda (como el audio) utilizando la codificación Huffman de los residuos de predicción y una cuantificación adicional de carácter opcional. En el modo sin pérdidas, la cantidad de compresión obtenida depende de la naturaleza de la forma de onda. Las compuestas por bajas frecuencias y bajas amplitudes dan la mejor compresión, que puede ser de 2:1 o superior, y esta es la modalidad que corresponde al carácter del presente proyecto [13].

- **ZIP.** ZIP es un formato de archivo que admite la compresión de datos sin pérdidas.

En el primer experimento realizado, se emplea un total de 15.985 señales de audio dedicadas al entrenamiento de la red, lo que se considera una cantidad suficientemente elevada. El entrenamiento se lleva a cabo empleando todas las muestras de las señales, y el valor del resto de parámetros clave, junto a estos, pueden observarse en la Tabla 4.1.

Técnica	Epoches	Señales entren.	Señales test	Capas LSTM	Capas ocultas	Bits / muestra original	Bits / muestra comp.	**Tasa comp.**
Propia	10	15.985	100	2	256	8	3,76	**2,13**
Shorten	-		100	-		8	5,13	1,56
ZIP	-		100	-		8	6,00	1,33

Tabla 4.1: Primera prueba con parámetros clave en valores razonables estimados.

En el primer caso, obtenemos una tasa de compresión de 2.13, esto es, la señal de audio codificada y preparada para ser enviada ocupa **menos de la mitad** de lo que lo hacía la **original**.

A continuación, se **aumenta el número de *epoches* a 50**, es decir, se entrena el modelo 5 veces más con el mismo conjunto de datos de entrenamiento, el cual recordamos que continúa siendo muy amplio, con 15.985 señales. A cotinuación, en la Tabla 4.2 podemos ver el resultado obtenido en este caso.

Técnica	Epoches	Señales entren.	Señales test	Capas LSTM	Capas ocultas	Bits / muestra original	Bits / muestra comp.	**Tasa comp.**
Propia	50	15.985	100	2	256	8	3,77	**2,12**
Shorten	-		100	-		8	5,13	1,56
ZIP	-		100	-		8	6,00	1,33

Tabla 4.2: Aumento de las *epoches* a 50.

En este caso se comprueba que la **tasa de compresión** es a aspectos prácticos la **misma para el caso anterior**, con 10 *epoches*, de lo que se entiende que con 15.985 señales, la red converge al error mínimo que puede alcanzar dado el escenario antes de las 10 *epoches*. No es necesaria tanta computación. Por esto, se decide **reducir el número de señales dedicadas a entrenamiento a la mitad (7.992 señales)**, conservando el resto de parámetros del caso anterior, 10 *epoches*. El resultado obtenido es el que se muestra en la Tabla 4.3.

Técnica	Epoches	Señales entren.	Señales test	Capas LSTM	Capas ocultas	Bits / muestra original	Bits / muestra comp.	**Tasa comp.**
Propia	10	7.992	100	2	256	8	3,82	**2,1**
Shorten	-		100	-		8	5,13	1,56
ZIP	-		100	-		8	6,00	1,33

Tabla 4.3: Reducción de señales de entrenamiento a 7.992.

Tras a penas obtener diferencia en el escenario anterior, ahora se opta por **reducir drásticamente el número de señales dedicadas al entrenamiento a 100**, lo que supone muy poco abanico de señales diferentes, lo que se espera que ocasiones un aprendizaje mucho más limitado de la red. A cotinuación, en la Tabla 4.4 podemos ver el resultado obtenido en este caso.

Como es de esperar, en este caso la tasa de compresión es **más baja** en comparación de los escenarios anteriores.

Técnica	Epoches	Señales entren.	Señales test	Capas LSTM	Capas ocultas	Bits / muestra original	Bits / muestra comp.	**Tasa comp.**
Propia	100	100	100	2	256	100	4,17	**1,92**
Shorten	-		100	-		8	5,13	1,56
ZIP	-		100	-		8	6,00	1,33

Tabla 4.4: Reducción de señales de entrenamiento a 100.

Probamos a emplear nuevamente las 7.992 señales, pero en este caso reducimos el **tamaño del *batch* de 32 a 16**. Esto provocará que los pesos de la red se actualicen el doble de veces de lo que lo han hecho en el escenario de la Tabla 4.3. El resultado obtenido tras la ejecución para este caso es el que se muestra en la Tabla 4.5.

Técnica	Epoches	Señales entren.	Señales test	Capas LSTM	Capas ocultas	Bits / muestra original	Bits / muestra comp.	**Tasa comp.**
Propia	10	7.992	100	2	256	8	4,01	**1,99**
Shorten	-		100	-		8	5,13	1,56
ZIP	-		100	-		8	6,00	1,33

Tabla 4.5: Reducción de tamaño de *batch* a 16 señales.

En este caso, la diferencia es de 0,1 bit por muestra, lo que supone una diferencia prácticamente nula. Esto da cuenta de que para un número de señales de entrenamiento tan elevado y suficientes *epoches*, función de error de la red ya ha convergido en el entrenamiento al error mínimo alcanzable para este caso. Por lo tanto, para un mismo resultado, la lectura que se hace es que es **más conveniente un tamaño de batch de 32 frente a 16**, en términos de coste de computación.

Se concluye el presente capítulo con resultados satisfactorios frente a diferentes arquitecturas y configuraciones del sistema, obteniendo en todo momento mejores tasas de compresión que para los métodos de conocimiento común y utilización general Shorten y ZIP.

Capítulo 5

Conclusiones y líneas futuras

El presente capítulo pretende proyectar la conclusión alcanzada tras las consideración conjunta de los resultados experimentales presentados en el capítulo anterior 4, interpretando el resultado global de la efectividad en cuestión de compresión sin pérdidas del sistema desarollado en el presente proyecto.

5.1. Conclusiones

Este proyecto ha consistido en desarrollar un sistema propio de compresión sin pérdidas para archivos de audio empleando redes neuronales profundas y codificación Huffman. Durante la realización del mismo se ha comprendido la esencia de las redes recurrentes LSTM, de importancia clave en el trabajo, mediante ejemplos realizados con señales sencillas y posteriormente aplicado a las señales reales de audio con las que se ha puesto a prueba el sistema. También se ha puesto en práctica los conocimientos adquiridos en la titulación sobre el método de codificación Huffman.

De esta manera, se ha cubierto enteramente el objetivo del proyecto, y se ha podido comprobar mediante las pruebas experimentales como el sistema comprime los archivos de audio con diferentes tasas, para alteraciones de parámetros clave del sistema y cantidad de señales de test.

Así pues, se ha podido comparar la efectividad del propio sistema frente a métodos de compresión sin pérdidas de uso común en la actualidad. Frente a estos, se ha podido comprobar sobre todos los casos de prueba recogidos en el Capítulo 4 que el propio sistema desarrollado obtiene mejores tasas de compresión, a falta de determinar experimentalmente la comparativa de los costes de computación de cada uno de los métodos frente a las mismas señales.

Los resultados experimentales ponen de manifiesto la importancia de las redes neuronales profundas, concretamente para el tipo de señal empleada, ya que a diferencia de Shorten y ZIP, son capaces de aprovechar la renundancia propia de una señal de

audio para finalmente conseguir mayor compresión.

La valoración general del presente proyecto es positiva y gratificante. Los conocimientos que han sido adquiridos durante la realización del mismo son cuantiosos y ampliamente aplicables a diferentes áreas en la nueva etapa profesional. Además, el proyecto sirve como introducción real y completa a un campo de actualidad y cada vez más presente en todos los ámbitos, donde se han entendido las bases, se ha avanzado satisfactoriamente frente a su complejidad y abstracción, y todo esto da pie a profundizar mucho más allá en cualquiera de los caminos hacia los que se ramifica el aprendizaje automático, y los diferentes datos sobre los que puede operar en aplicaciones reales.

5.2. Líneas futuras

Como continuación del presente proyecto, se considera de importancia presentar una serie de lineas futuras que se estima que de ser llevada adelante, se podría profundizar más en cuanto a resultados experimentales, comprender mejor la influencia de diferentes puntos críticos del sistema tras la alteración de los mismos, y quizá alcanzar mejores resultados en los que es el objetivo real del mismo, alcanzar mayores tasas de compresión, teniendo en cuenta siempre el coste computacional y logrando un equilibrio óptimo entre ambos.

Todos los esfuerzos van focalizados pues a un mismo punto, el de experimentar con diferentes arquitecturas y configuraciones. A continuación, se lista una serie de posibles líneas futuras del proyecto:

- Comprobar la efectividad y dependencia que tiene con diferentes segmentos de la señal de audio, en lugar de operar sobre la señal completa. Esto daría cuenta de la aplicabilidad que tiene el sistema en el mundo de VoIP, que ha sido punto clave de la motivación de este proyecto, como se presenta en la Sección 1.2.

- Completar el conjunto de pruebas alterando la anchura y la profundidad de la red, esto es, el número de capas LSTM y el número de capas internas a las mismas (capas ocultas), respectivamente. Las pruebas experimentales se han realizado con la configuración que permite el mayor número de ambas, limitados en este aspecto por la capacidad computacional del equipo empleado.

- Experimentar con el tiempo de codificación del sistema para diferentes escenarios, tratar de optimizar el equilibrio entre tasa de compresión y tiempo de codificación, y compararlo con sus homólogos.

Por último, fuera de las señales de audio, el sistema es de naturaleza completamente adaptable, pudiendo ser modificado para comprimir diferentes archivos. Sería de grán interés comprobar su eficacia frente a archivos de texto, imágen y vídeo.

Capítulo 6

Bibliografía

[1] Christopher Olah. http://colah.github.io/posts/2015-08-understanding-lstms/. 2015.

[2] J. S. Garofolo. https://ui.adsabs.harvard.edu/abs/1993stin...9327403g/abstract. 1993.

[3] Aäron van den Oord. Conditional image generation with pixelcnn decoders. 2016.

[4] Koray Kavukcuoglu Aäron van den Oord, Nal Kalchbrenner. Pixel recurrent neural networks. 2016.

[5] Jeffrey Scott Vitter. Design and analysis of dynamic huffman codes. 1987.

[6] PyTorch Foundation. https://pytorch.org/docs/stable/nn.html. 2022.

[7] Elliott Gordon-Rodriguez. Uses and abuses of the cross-entropy loss: Case studies in modern deep learning. 2020.

[8] PyTorch Foundation. https://pytorch.org/docs/stable/generated/torch.optim.lr_scheduler.stepl 2022.

[9] Techie Delight. https://www.techiedelight.com/es/huffman-coding/. 2022.

[10] PyTorch Foundation. https://pytorch.org/docs/stable/generated/torch.nn.lstm.html. 2022.

[11] PyTorch Foundation. https://pytorch.org/docs/stable/generated/torch.nn.linear.html. 2022.

[12] PyTorch Foundation. https://pytorch.org/docs/stable/generated/torch.nn.crossentropyloss.htm 2022.

[13] Tony Robinson and SoftSound Ltd. https://linux.die.net/man/1/shorten. 1999.

[14] Amazon Web Services. https://aws.amazon.com/es/what-is/python/. 2022.

[15] Visual Studio Code. https://code.visualstudio.com/docs. 2022.

[16] Stack Overflow. https://insights.stackoverflow.com/survey/2021-most-popular-technologies-new
2021.

[17] Git. https://git-scm.com/. 2022.

[18] Kubernetes. https://kubernetes.io/es/docs/concepts/overview/what-is-kubernetes/.
2022.

[19] Microsoft. https://azure.microsoft.com/es-es/. 2022.

[20] OpenWebinars. https://openwebinars.net/blog/que-es-visual-studio-code-y-que-ventajas-ofrece/
2022.

[21] MobaXterm. https://mobaxterm.mobatek.net/. 2022.

[22] Mobatek. https://www.mobatek.net/. 2022.

[23] Raja Suman C. https://analyticsindiamag.com/ann-with-linear-regression/. 2019.

[24] GeeksforGeeks. Huffman coding — greedy algo-3
https://www.geeksforgeeks.org/huffman-coding-greedy-algo-3/. 2022.

[25] Dropbox. https://experience.dropbox.com/resources/what-is-a-zip-file. 2022.

Lista de Figuras

1.1. Red con realimentación. 2
1.2. Fórmula para cálculo de tasa de compresión. 3
1.3. Bloque de entrenamiento de la red y bloque de emisión del sistema. . . 4
1.4. Bloque de recepción del sistema. 4
1.5. Diagrama de Gantt. 9

2.1. Estructura RNN. 13
2.2. Estructura RNN en cascada. 14
2.3. Estructura RNN en cascada con dependencia en periodo corto. 14
2.4. Estructura RNN en cascada con dependencia en periodo largo. 15
2.5. El módulo de repetición en una RNN estándar. 15
2.6. El módulo de repetición en una red LSTM. 15
2.7. Leyenda de la simbología empleada en la Figura 2.6. 16
2.8. Capa sigmoide. 16
2.9. Error en función de diferentes tasas de aprendizaje. 18
2.10. Convergencia para los métodos de optimización SGD y GD. 21
2.11. Nodos ordenados según prioridad (mayor a la derecha). 22
2.12. Cola de prioridad tras fusión de nodos ’D’ y ’E’. 23
2.13. Cola de prioridad tras tercera fusión de nodos. 23
2.14. Cola de prioridad tras última fusión de nodos. 23
2.15. Huffman Tree completado. 24
2.16. Codificación de cada caracter. 24

3.1. Estructura de la red neuronal empleada. 25
3.2. Función calculada para cada capa de la red LSTM, para cada elemento de la secuencia de entrada. 26
3.3. Conjunto de entradas y salidas a la red LSTM. 28
3.4. Ejemplo de predicción y asignación de número de bits. 30
3.5. Diagrama de bloques del proceso de codificación. 31
3.6. Diagrama de bloques del proceso de codificación. 33

3.7. Señal de audio sin procesar y señal de audio decodificada. 34

A.1. Clase Net de la red neuronal utilizada. 53
A.2. Inicialización de la clase CodDecod. 54
A.3. Función de codificación de la clase CodDecod. 54
A.4. Función de decodificación de la clase CodDecod. 55

B.1. Editores de código fuente favoritos de 2021. 61
B.2. Estructura típica de la función de transferencia del perceptrón [23]. . . 65
B.3. Estructura típica del perceptrón. 65
B.4. Estructura típica de una red neuronal con múltiples perceptrones. . . . 65

Lista de Tablas

4.1. Primera prueba con parámetros clave en valores razonables estimados. . 38
4.2. Aumento de las *epoches* a 50. 39
4.3. Reducción de señales de entrenamiento a 7.992. 39
4.4. Reducción de señales de entrenamiento a 100. 40
4.5. Reducción de tamaño de *batch* a 16 señales. 40

Anexos

Anexos A

Código

Este Anexo sirve para incluir partes clave del código desarrollado durante este proyecto a las que se hace referencia en diferentes partes de la presente memoria.

A.1. Red

La clase Net creada para el proyecto, la cual estructura la red neuronal utilizada y define diferentes funciones para su explotación es la siguiente:

```
class Net(nn.Module):
    def __init__(self, input_dim, hidden_dim, output_dim, nb_lstm_layers, dropout=0.5):
        super(Net, self).__init__()
        self.lstm = nn.LSTM(input_dim, hidden_dim, nb_lstm_layers, batch_first=True, bidirectional=False, dropout=dropout)
        self.fc = nn.Linear( hidden_dim, output_dim)
        self.J = nn.CrossEntropyLoss()

    def forward(self, x):
        x = x.float()
        x, state = self.lstm(x)
        x = self.fc(x)
        return x

    def loss(self, out, y):
        n, t, d = y.shape
        return self.J(out.reshape(n*t,-1), y.reshape(n*t,))

    def predict(self, x, state=None):
        x = x.float()
        x, state = self.lstm(x, state)
        x = self.fc(x)
        return x.softmax(-1), state

model = Net(input_dim, hidden_dim, output_dim, nb_lstm_layers)
```

Figura A.1: Clase Net de la red neuronal utilizada.

A.2. Codificación

El código de inicialización de la clase creada para realizar las tareas de codificación y decodificación es el siguiente:

```
class CodDecod(object):
    def __init__(self, x_true, x_t0, symbols, model):
            self.x_true = x_true.cuda()
            self.x_t0 = x_t0.cuda()
            self.symbols = symbols
            self.model = model
```

Figura A.2: Inicialización de la clase CodDecod.

Por otro lado, la función de codificación de la clase CodDecod desarrollada es la siguiente:

```
def Code(self):
  s = None
  message_encoded = []
  x_t = self.x_t0
  for t in range(len(self.x_true[0])):
      p, s = self.model.predict(x_t, s)
      x_t = (torch.FloatTensor([[[self.x_true[0,t]]]])).cuda()
      w_ = p.reshape((len(p[0,0]), -1))
      w_ = w_.tolist()
      dictionary = [ (self.symbols[j], w_[j]) for j in range(len(w_)) ]
      h = make_huffman_tree(dictionary)

      d = flatten_to_dict(h)
      w = [int(x_true[0, t])]
      u = huffman_encode(w, d)

      message_encoded.append(u)

  message_encoded = [item for sublist in message_encoded for item in sublist]
  return message_encoded
```

Figura A.3: Función de codificación de la clase CodDecod.

La función de decodificación de la clase CodDecod desarrollada es la siguiente:

```
def Decode(self, message_encoded):
  x_t = self.x_t0
  s = None
  message_decoded = []
  word_encoded = []
  i = 0
  for t in range(len(self.x_true[0])):
    p, s = self.model.predict(x_t, s)
    w_ = p.reshape((len(p[0,0]), -1))
    w_ = w_.tolist()
    dictionary = [ (self.symbols[j], w_[j]) for j in range(len(w_)) ]
    h = make_huffman_tree(dictionary)
    d = flatten_to_dict(h)

    word_encoded = []
    word_encoded.extend(message_encoded[i:i+1])
    while ( word_encoded not in d.values() ) and ( i < len(message_encoded)-1 ):
      i = i+1
      word_encoded.extend(message_encoded[i:i+1])
    i = i+1
    word_decoded = huffman_decode(word_encoded, h)
    message_decoded.append(word_decoded)
    x_t = (torch.FloatTensor([[word_decoded]])).cuda()
  return message_decoded
```

Figura A.4: Función de decodificación de la clase CodDecod.

Anexos B

Tecnología empleada

En este Anexo tiene como objetivo presentar la **tecnología** software empleada como herramienta de ayuda para la realización del presente proyecto. En él se recogerán características generales y algunos datos y gráficos de interés.

Además, en él se pueden encontrar algunos de los aspectos básicos de las Redes Neuronales Lineales, que simplifican la comprensión del trabajo para aquellos que no tengan uso de utilización del Deep Learning.

B.1. Python

Python [14] es un lenguaje de programación de uso común en aplicaciones web, desarrollo de software, ciencia de datos y machine learning (ML), entre otros. El atractivo en la utilización de Python viene de que es eficiente y fácil de aprender, a parte de permitir ser ejecutado en gran multitud de plataformas, su descarga es gratuita, la integración es satisfactoria con gran número de sistemas y permite una velocidad del desarrollo significativametne alta.

De entre un gran número de ellos, se listan a continuación los principales beneficios que se obtienen del uso de Python:

- Los programas de Python son intuitivos, gracias a la sintaxis básica que caracteriza al lenguaje.

- Python aporta productividad en el trabajo a desarrollar, ya que en comparación con sus competidores, es muy barato en términos de líneas de cógigo.

- Pueden encontrarse gran cantidad de bibliotecas con código reutilizable para la mayoría de los casos.

- Es sencillo combinar Python con otros lenguajes de programación.

– Es sencillo y rápido encontrar soporte para la solución de cualquier problema dada la gran utilización de Python a nivel mundial, existiendo contacto en foros y demás portales de internet con fácil acceso.

– Se dispone de gran cantidad de recursos útiles disponibles en internet para aprender el lenguaje.

– Puede ser enviado mediante gran número de sistemas operativos de computadora, entre ellos, Windows, macOS, Linux y Unix.

Los principales casos de uso de Python son los que se listan a continuación:

– **Aplicaciones de Inteligencia Artificial y Machine Learning.** En el mundo de las Tecnologías de la Información, la Inteligencia Artificial y el Machine learning se han convertido en dos tendencias de inmenso crecimiento en los últimos años. Exigen a su vez cierta complejidad y potencia en sentido de computación, y Python brinda los recursos de los que necesitan, dotando a las computadoras de capacidad para memorizar datos y efectuar predicciones precisas. Hay un sinfín de aplicaciones en este area, y las numerosas librerías de las que se dispone con Python permiten entrenar de manera sencilla los modelos de Machine Learning y crear clasificadores de datos con precisión.

– **Ciencia de datos.** De la mano del crecimiento del marketing digital de los últimos años han venido los avances en ciencia de datos, que no es otra cosa que extraer datos importantes de las empresas y conocimientos valiosos en estrategias de marketing y comerciales. Las computadoras procesan una cantidad de datos y posteriormente aprenden patrones derivados de los mismos, que les dotan de capacidad para realizar predicciones precisas. De entre diferentes utilidades que tiene Python en este area, destacan los listados a continuación:

  - Renombrar archivos de manera automática y de una sola vez
  - Conversiones automáticas de tipo de archivo
  - Eliminar duplicidades en un mismo fichero
  - Efectuar procesos matemáticos
  - Transmitir mensajes vía correo electrónico
  - Realizar multitud de descargas
  - Analizar conjuntos de registros
  - Localizar y corregir partes erroneas en ficheros

    - Localizar datos concretos relativos a un determinado tema y extraer su contenido de manera interpretable
    - Sacar reportes estadísticos en diferentes tipos de gráficos en función de los datos analizados

– **Desarrollo de software.** La utilidad más suculenta que aporta Python en el desarrollo de software es la automatización de tareas complejas desde el lado del servidor, esto es, desde el backend, desde donde los sitios web recogen y transforman la información para mostrarla posteriormente al usuario. Unas de las funcionos principales que quedan cubiertas mediante este lenguaje de programación son la interacción con las bases de datos de las que el servidor recoge la información, la comunicación con otros sitios web y la protección de la información cuando esta es transmitida a través de la red.

  Uno de los grandes atractivos para los desarrolladores de software, como se ha mencionado anteriormente, es la cantidad de bibliotecas de facil acceso via web de las que se dispone, y de donde pueden extraer código escrito para modificarlo y adaptarlo a las tareas particulares en las que se esté trabajando. Gracias a esto, conjuntos de funciones complejas de backend quedan simplificadas y desarrolladas en menor tiempo.

  Algunas de las tareas que más suelen realizar los desarrolladores mediante Python son las que se listan a continuación:

    - Localización de fragmentos de código erróneos
    - Automatizar el desarrollo del código software
    - Administración de procesos de software
    - Desarrollo de prototipos de software
    - Utilización de bibliotecas de interfaz gráfica de usuario *(GUI)* para el desarrollo de aplicaciones de escritorio
    - Combinación de juegos sencillos para el desarrollo de videojuegos de mayor complejidad

– **Automatización de pruebas de software.** Las pruebas de sofware es un proceso indispensable tras cada desarrollo del mismo, ya que la intención principal de estas pruebas es verificar que el resultado obtenido del desarrollo se corresponde con el resultado que se espera. De esa manera, queda garantizado que el desarrollo se encuentra sin errores que interfieran en la función que pretende cubrir.

- Python dispone de marcos de prueba de unidad, como de entre ellos, PyUnit, Robot y Unittest
- Es de uso común Python como lenguaje de desarrollo para elaboración de ficheros de prueba que pongan en cuarentena, diferentes partes del sistema sistema final.

Además, Python ofrece diferentes herramientas de ejecucución automática de scripts de prueba, conocidas como herramientas de integración e implementación contínuas *(CI/CD)*, gracias a las cuales los desarrolladores y los encargados de validar el software ahorran tiempo de ejecución manual sucesiva de dichos scripts individuales. Los desarrolladores pueden utilizar varias herramientas para ejecutar scripts de prueba de manera automática. De esta manera, para cada modificación de código por parte de los desarrolladores, las pruebas son ejecutadas automáticamente y se muestran los resultados obtenidos.

Como conclusión, Python es uno de los lenguajes de programación más utilizados actualmente, implementado en cientos de empresas e instituciones educativas como herramienta para la programación. Más allá de su sencillez, de entendimiento, Python tiene un inmenso campo de uso, siendo uno de los lenguajes de programación que en un futuro próximo podría quitas de los primeros lugares a otros lenguajes que llevan más de 30 años en el mercado.

B.2. VS Code

Visual Studio Code (de ahora en adelante, VS Code) [15] es un editor de código fuente desarrollado por Microsoft. VS Code se caracteriza por ser ligero pero a su vez potente, de ejecución sencilla en el mismo escritorio y disponible para Windows, macOS y Linux. Además, cuenta con soporte incorporado para los lenguajes de programación JavaScript, TypeScript y Node.js, y cuenta con una amplia gama de extensiones para otros lenguajes y distintos tiempos de ejecución (entre ellos, C++, C#, Java, Python, PHP, Go, .NET).

VS Code es uno de los editores de código fuente de mayor popularidad y aceptación en la comunidad de desarrollo de la actualidad. A continuación, se puede observar en la Figura B.1 una comparativa de los editores de código favoritos por los desarrolladores, producto de una encuesta realizada por Stack Overflow en mayo del año 2021 a más de 80.000 desarrolladores [16]. En la Figura B.1, salta a la vista que VS Code es el favorito de ellos, con un porcentaje del 71,06 % [20].

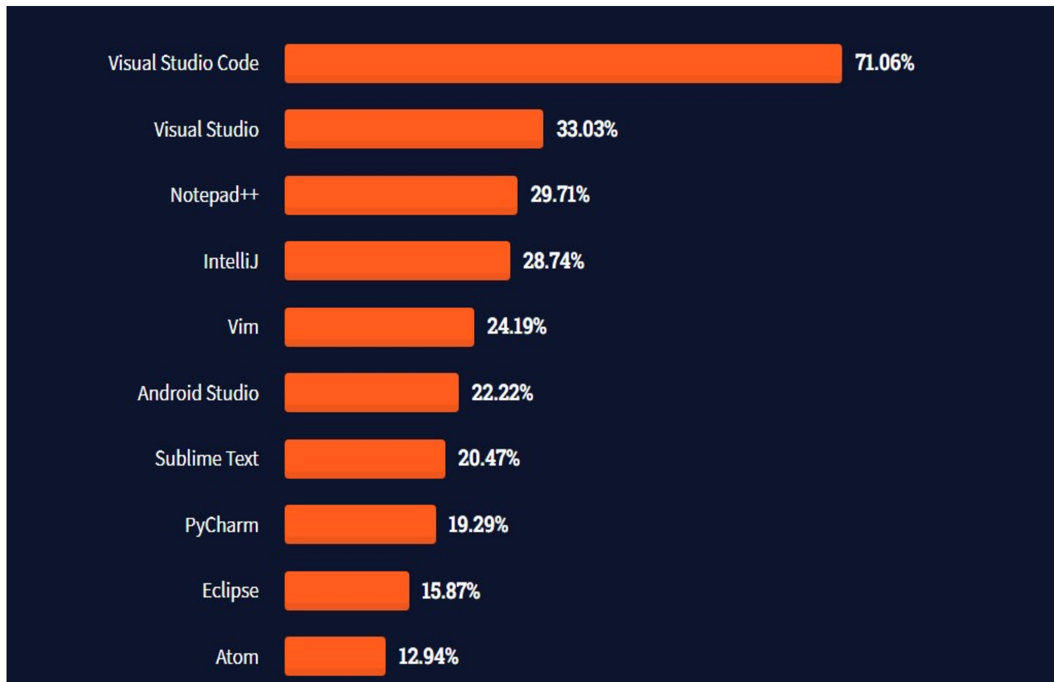


Figura B.1: Editores de código fuente favoritos de 2021.

A continuación se listan las principales características y funcionalidades de VS Code que le hacen tan popular adentro de la comunidad del desarrollo software:

– **Es multiplataforma.** Como se ha mencionado anteriormente, VS Code está disponible para Windows, GNU/Linux y macOS.

– **IntelliSense.** Esta característica permite sugerencias, autocompletado y señalización de fragmentos de código, lo cual agiliza la tarea de desarrollo. Existe una gran cantidad de extensiones que el usuario puede descargar e instalar para de este modo personalidar su IntelliSense, en función del escenario en el que se esté programando.

– **Depuración.** Esta función permite al individuo detectar el error y su localización en la parte exacta del código.

– **Uso del control de versiones.** Gracias a la completa compatibilidad que tiene VS Code con Git [17], el usuario puede facilmente operar sobre ficheros controlando de manera sencilla las modificaciones de cada fichero y la versión a la que pertenece, entre otras funcionalidades, desde la misma plataforma de VS Code.

– **Extensiones.** Es uno de los puntos de más interes de VS Code, ya que las extensiones le agregan potencia y dan oportunidad de personalizar y enriquecer

las funcionalidades de manera vertical e independiente una de otra. El resultado de lo anterior es una mejor experiencia de usuario sin afectar al rendimiento de VS Code, ya que son ejecutados en procesos independientes.

VS Studio Code cuenta con una terminal que es encendida de manera sencilla desde el propio directorio de trabajo, y cuenta con la posibilidad de emplear cualquier Shell del que se disponga en el equipo. Durante el desarrollo, se requiere de la ejecución de diferentes comandos necesarios, y es ahí donde se encuentra el atractivo de la terminal de VS Code. Una vez más, en un intento de agilizar las tareas del propio desarrollo.

De entre la gran cantidad de funcionalidades que tiene VS Code, a continuación se mencionan algunas de las más populares dentro de las profesiones de Tecnologías de la Información:

- Permite el desarrollo completo de aplicaciones
- Mediante SSH es posible conectarse de manera remota a máquinas virtuales, contenedores y Windows Subsystem for Linux (WSL)
- Acceder a conjuntos de archivos
- Desde el terminal de VS Code, gestión total sobre directorios locales y remotos
- Gestión de aplicaciones desde contenedores
- Gestión de *clusters* en Kubernetes [18]
- Integración completa con Microsoft Azure [19]
- Personalización total respecto a instalación de herramientas, de acuerdo a las necesidades del usuario, en cambio de otros IDEs completos de serie

B.3. MobaXterm

MobaXterm es una herramienta que facilita la realización de tareas informáticas de manera remota, de gran utilización a nivel global y concretamente en los siguientes colectivos:

- Programadores.
- Maestros de web.
- Gestores del área de Tecnologías de la Información.

Así pues, MobaXterm es la herramienta que se ha empleado durante el desarrollo del proyecto para tareas de gestión de ficheros con contenido de utilidad para la parte práctica, como archivos de audio y scripts de Python. Además, se ha utilizado para ejecutar dichos archivos de Python en un cluster propocionado por la universidad de Zaragoza, ya que agiliza los procesos de ejecución por disponer de mayor recursos que la propia computadora desde donde se trabaja, y los procesos de entrenamiento de las redes neuronales se caracterizan, en general, por ser muy caros en terminos de computación, y por lo tanto, abarcan periodos de ejecución que pueden durar días [21].

A continuación se listan algunas de las herramientas de red remotas esenciales que MobaXterm pone a disposición del usuario:

- SSH
- X11
- RDP
- VNC
- FTP
- MOSH

Además de dichas herramientas de red, un buen número de comandos de Unix son aportados al escritorio de Windows mediante un archivo exe, que puede utilizarse desde el primer momento de la instalación de la herramienta. Alguno de los comandos de Unix más importantes se listan a continuación:

- bash
- ls
- cat
- sed
- grep
- awk
- rsync

Las ventajas que ofrece tener a disposición una herramienta de red como MobaXterm son numerosas. Una de ellas, de especial interes, es la de la aparición automática de un navegador gráfico SFTP que permite modificar archivos remotos en el momento en que se conecta el usuario a un servidor remoto a través de SSH. Además, las aplicaciones remotas del usuario aparecerán de igual manera en el escritorio de Windows.

Una de los puntos fuertes de la herramienta es que ofrece a los usuarios un número de funciones principales de obtención gratuita, además de otro programa de pago que da acceso a un conjunto completo de las funcionalidades de la herramienta, enfocado para el uso en el ámbito profesional, como por ejemplo, por organizaciones y empresas.

Según comentan los desarrolladores de MobaXterm, el objetivo pricipal de la herramienta ha sido en todo momento brintar al usuario una interfaz intuitiva que le de acceso de manera sencilla a servidores remotos mediante distintas redes o sistemas. La herramienta MobaXterm está actualmente en continuo desarrollo y actualización, por parte de Mobatek [22].

B.4. Linear NN

Un principiante en la ciencia de los datos, después de pasar por los conceptos de regresión, clasificación, ingeniería de características, etc. y entrar en el campo del Deep Learning , sería muy beneficioso que pudiese relacionar la funcionalidad de los algoritmos en el Deep Learning con los conceptos anteriores.

Antes de adentrar en la RNA, es esencial entender el concepto de perceptrón, que es un bloque de construcción básico de la RNA. El perceptrón es el nombre que se da inicialmente a un clasificador binario. Sin embargo, podemos ver el perceptrón como una función que toma ciertas entradas y produce una ecuación lineal que no es más que una línea recta. Esto puede utilizarse para separar ciertos datos fácilmente separables, como se muestra en la Figura B.2. Sin embargo, en los escenarios del mundo real, las clases no son tan fácilmente separables.

Así pues, la estructura típica del perceptrón puede observarse en la Figura B.3:

La estructura de una red neuronal con múltiples perceptrones en su interior sería la que se muestra en la Figura B.4:

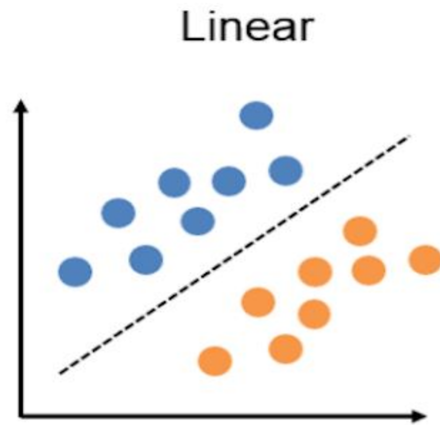


Figura B.2: Estructura típica de la función de transferencia del perceptrón [23].

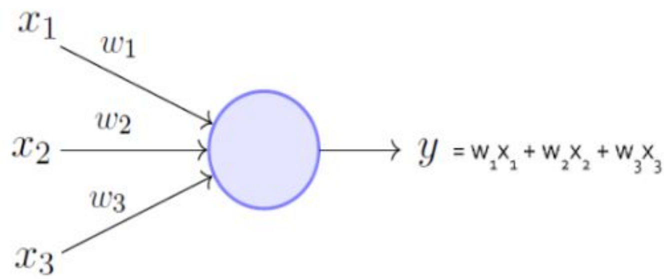


Figura B.3: Estructura típica del perceptrón.

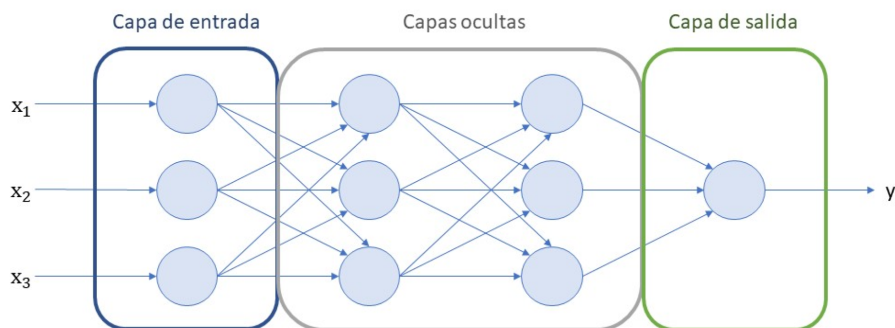


Figura B.4: Estructura típica de una red neuronal con múltiples perceptrones.

Con la agrupación de varios perceptrones, podemos construir diferentes capas de conjuntos de ellos. Esto significa generar múltiples ecuaciones lineales en múltiples

puntos. Estos perceptrones también pueden llamarse neuronas o nodos, que en realidad son los bloques de construcción básicos en la red neuronal natural dentro de nuestro cuerpo. En la figura anterior, el primer conjunto vertical de 3 neuronas es la capa de entrada. Los dos siguientes conjuntos verticales de neuronas forman parte de la capa intermedia, que suele denominarse capa oculta, y la última neurona individual es la capa de salida. La red neuronal de la figura anterior es una red de 3 capas. Esto se debe a que la capa de entrada no suele contarse como parte de las capas de la red. Cada neurona de la capa de entrada representa un atributo (columna) de los datos de entrada (es decir, x1, x2, x3, etc.). Lo que ocurre en la red anterior es que los datos de entrada se introducen en un conjunto de neuronas, y cada una produce una salida. De nuevo, cada una de estas salidas alimenta a otras neuronas que, a su vez, producen otra salida, que alimenta de nuevo a la capa de salida. El error calculado en esta capa de salida se envía de nuevo a la red para refinar aún más las salidas de cada neurona, que alimentan de nuevo a la neurona en la capa de salida para producir una salida más refinada que antes. Como se explica en el proceso de 5 pasos anterior, este proceso se repite hasta que se obtiene una salida con un error mínimo.

El proceso de producir salidas, calcular errores y volver a alimentarlas para producir una mejor salida es generalmente un proceso confuso, especialmente para quienes no tienen uso de trabajar con redes neuronales. Por cuestión de simplicidad conceptual, se expone el proceso que tendría lugar con una sola neurona y una capa. Una vez que se entiende este concepto básico, ampliarlo a una red neuronal más grande no es difícil.

Es de aceptación general el hecho de que la regresión lineal simple es lo más sencillo en el aprendizaje automático o, al menos, lo primero que se aprende. Por lo tanto, vamos a tratar de entender este concepto de aprendizaje profundo también con una regresión lineal simple, mediante la resolución de un problema de regresión utilizando ANN.

De lo anterior se de concluye con que cada una de las neuronas de la RNA, excepto la capa de entrada, produce una salida. La salida será dependiente de aquella función que se utilice. Esta función se denomina "función de activación". Como la RNA se utiliza principalmente para fines de clasificación, generalmente se utiliza la función sigmoidea u otros algoritmos de clasificación similares. Como el caso que nos ocupa es un problema de regresión lineal, la función de activación no es más que una 'Ecuación Lineal Simple' de la forma que se muestra en la Ecuación B.1:

$$y = w0 + w1 * x1 + w2 * x2 + w3 * x3 + \dots wn * xn \tag{B.1}$$

donde x1, x2, x3.. xn son los atributos independientes en los datos de entrada, w1, w3... wn son los pesos (coeficientes) de los atributos correspondientes, y w0 es el sesgo.

Como nuestra salida debe ser una sola línea , debemos configurar nuestra RNA con una sola neurona. Como la salida de la neurona es la propia línea, esta neurona se colocará en la capa de salida. Las capas ocultas son necesarias cuando intentamos clasificar objetos con el uso de múltiples líneas (o curvas). Por lo tanto, no es necesaria ninguna capa oculta.

Por lo tanto, la RNA para resolver un problema de regresión lineal, consiste en una capa de entrada con todos los atributos de entrada y una capa de salida con sólo una neurona.