Universidad del País Vasco
Euskal Herriko Unibertsitatea
ZIENTZIA ETA TEKNOLOGIA FAKULTATEA
FACULTAD DE CIENCIA Y TECNOLOGÍA

Gradu Amaierako Lana / Trabajo Fin de Grado
Gradu bikoitza Fisikan eta Ingeniaritza Elektronikoan
/ Doble Grado en Física y en Ingeniería Electrónica

Estudio de diferentes modelos de redes neuronales para el desarrollo de un clasificador de frases

Egilea/Autor/a:
Arkaitz Bidaurrazaga Barrueta
Zuzendaria/Director/a:
María Inés Torres
Zuzendarikidea/Codirector/a:
Raquel Justo Blanco

Leioa, 2019ko ekainaren 19a /Leioa, 19 de junio de 2019

Índice general

1. Introducción **1**

2. Estado del arte **2**
2.1. Sistemas de lenguaje natural . . . 2
2.2. Natural Language Understanding (NLU) . . . 3
2.3. Keras y Pytorch . . . 4

3. Redes neuronales **5**
3.1. RNN(Recurrent Neural Network) . . . 5
3.2. LSTM y GRU . . . 6
3.3. Redes Convolucionales . . . 8

4. Algoritmos Deep Learning **10**
4.1. *Back Propagation* . . . 10
4.2. Optimizadores . . . 11
4.2.1. AdaGrad . . . 11
4.2.2. RMSProp . . . 12
4.2.3. *Nesterov momentum* . . . 12
4.2.4. Adam . . . 13
4.3. Cómo minimizar el *overfitting* . . . 13
4.4. Búsqueda local . . . 14

5. Análisis del corpus **16**
5.1. Qué corpus, con qué objetivo . . . 16
5.2. Estructura del corpus . . . 18
5.3. Cómo evaluar los resultados (F1 score) . . . 20

6. Modelo de red **23**
6.1. Arquitectura de las redes . . . 23
6.2. Datos de entrenamiento . . . 25
6.3. *Train, validation* y *test* . . . 26

7. Resultados **27**
7.1. Comparación de las redes . . . 27
7.2. Optimización de la red . . . 31

8. Conclusiones **32**

Apéndice A **33**

Capítulo 1

Introducción

La Inteligencia Artificial (IA) está en auge, gracias al avance de los ordenadores, capaces de realizar un mayor número de cálculos de forma más rápida, así como también gracias a los nuevos métodos y modelos para desarrollar redes neuronales. Más aún, dentro de la IA existen también diversos subgrupos, ya que su versatilidad permite realizar diferentes tareas: análisis de imágenes, reconocimiento de voz, sistemas de diálogo...

En el caso del análisis de imágenes, fácilmente nos viene a la cabeza el reconocimiento facial que realizan los teléfonos móviles. Aunque poco a poco se van desarrollando nuevas IA-es capaces de localizar y reconocer diferentes objetos en una imagen. Como curiosidad, señalamos el software *Google Deep Dream* [1], el cual podemos entrenar para que busque cosas concretas en una imagen, es decir, podríamos entrenarlo para que buscara formas de ojos en una imagen, para que después situara ojos en los lugares de la imagen donde las haya 'visualizado'.

Los reconocedores de voz también están muy arraigados en los teléfonos móviles, entre otras aplicaciones. Los primeros sistemas, creados en 1952, sólo eran capaces de detectar la voz de una sola persona y solamente reconocían 10 palabras individualmente. A principios de los 70, la Agencia de Proyectos de Investigación Avanzada del Departamento de Defensa (DARPA), junto con la que participaron empresas como IBM, desarrolló un sistema que permitía reconocer hasta 1.000 palabras distintas. A partir de los años 80, se crearon sistemas que podían reconocer hasta 20.000 palabras, aunque solamente podían distinguir las palabras de forma individual. Hoy en día, los reconocedores de voz son capaces de identificar una gran variedad de palabras, de voces distintas y reconocerlas en conjuntos.

Por último, están los sistemas de diálogo, en los cuales nos centraremos en este trabajo. Más concretamente, nuestro trabajo se basará en desarrollar una red neuronal que analice semánticamente las frases de entrada (2.1), es decir, hará el proceso de *NLU* (Natural Language Understanding). Para ello, dispondremos del corpus proporcionado por el Proyecto Europeo *EMPATHIC* [2], el cual analizaremos en el capítulo 5. Este proyecto tiene como objetivo desarrollar un avatar para asistir a personas mayores, y hacer *coaching* en el ámbito de la salud física y psíquica de la persona, buscando siempre una conversación que exponga los problemas o malestares del usuario y poder motivarlo a mejorar su situación. Para mantener una conversación con el usuario se deben llevar a cabo estos procesos: transcripción del mensaje de voz, comprensión del texto (NLU), obtención de una representación del texto de salida[1] y por último creación y devolución la respuesta óptima.

[1] Al igual que nuestra red dará como salida una representación abstracta del texto de entrada, antes de construir la respuesta necesitaremos una representación de ésta.

Capítulo 2

Estado del arte

En este capítulo veremos varios sistemas que hacen uso del lenguaje natural para llevar a cabo sus funciones (2.1), después discutiremos las diferentes herramientas que existen para realizar un primer análisis del corpus (2.2), y por último, compararemos dos paquetes que disponemos en *Python*, *Keras* y *Pytorch*, de los cuales uno será el que utilicemos para el desarrollo de la red neuronal (2.3).

2.1. Sistemas de lenguaje natural

Los sistemas de lenguaje natural se distinguen por su objetivo principalmente, en esta sección resumimos estos objetivos y damos algunos ejemplos:

Traductor: Se puede entrenar una IA para que ante el *input* que sería la frase a traducir, diera como *output* la frase traducida al idioma que nos interesa. La primera empresa en usar el análisis del lenguaje natural en un traductor fue IBM, con su traductor *Candide*. Más tarde, Google implementó esta idea en la red neuronal con la que funciona ahora su *Google Traductor*.

Diagnóstico: Existen sistemas capaces de diagnosticar rápidamente algunas enfermedades mediante radiografías, e incluso de estimar el riesgo de padecer cáncer de una persona. Como ejemplos tenemos *SOPHIA*, desarrollado por Francisco Dorr en su tesis, el cual se centra en el diagnóstico de cefaleas; por otro lado tenemos el *KD-NLP*, producto de una colaboración entre varias universidades, el cual se centra en el reconocimiento de la enfermedad Kawasaki.

Reseña de clientes: Ciertas empresas con un gran volumen de reseñas respecto a sus productos pueden usar esas reseñas para analizar qué productos son los más deseables, e incluso el sistema analítico podría dar consejos para mejorar cierto producto o proponer uno nuevo. Plataformas que son capaces de llevar a cabo estos análisis son por ejemplo: *SPSS*(de IBM), *NVivo* (de QSR International) y *Google Cloud*.

Resúmenes: En la medicina estos sistemas serían realmente útiles, pues los historiales médicos son a veces exageradamente largos, y un médico puede tardar mucho tiempo en leerlo y quedarse con las enfermedades importantes del paciente. En cambio, ciertos sistemas se están especializando en resumir textos, ya que si fuesen capaces de destacar solamente lo importante, podrían ahorrar mucho tiempo. Existen plataformas como *Summarizebot, Text Summarization API y MetaMind* (de Richard Socher).

Asistentes virtuales: El ejemplo más famoso es el asistente de *Apple*, *Siri*. Ésta se centra en entender las demandas del usuario, y devolverle un servicio para cubrirlas. Otros ejemplos son *Alexa* de Amazon y *Google Assistant*.

Análisis semántico: La semántica es el campo que analiza el significado de las palabras dentro de un contexto. Inteligencias artificiales capaces de llevar a cabo éste análisis son muy interesantes, ya que permiten un mejor entendimiento de los textos procesados y ayudan mucho a mejorar el rendimiento de otros sistemas de lenguaje natural, pudiendo llegar a un nivel de comprensión mayor. Para analizar semánticamente un texto, lo primero que se hace es procesar todo el enunciado, para tener un concepto general sobre el tema que trata el mismo, tras este paso, se procesan las palabras teniendo en cuenta las que están alrededor para darles su significado lógico dentro del texto. Expert System desarrolló *Cogito*, un sistema especializado en procesar el contenido de páginas web, redes sociales y big data.

Avatares: Se centran en mantener una conversación informal con el usuario. Aunque existen otra clase de avatares los cuales tienen como objetivo obtener cierta información del usuario, un ejemplo podría ser *ELIZA*, la cual fue desarrollada en el MIT, y su objetivo principal es el psicoanálisis del usuario más que tener una conversación informal. El objetivo de este trabajo es construir una red que haga la función de NLU, como hemos mencionado anteriormente (1). Éste sería el primer[1] paso a la hora de construir un avatar, clasificar las frases de entrada según el tema, intención y polaridad.

2.2. Natural Language Understanding (NLU)

Antes de usar nuestros datos para el entrenamiento de la red, haremos un primer procesamiento de los textos, para así poder facilitar más el etiquetado de las frases. Principalmente nos interesa detectar las entidades, como por ejemplo: lugares, nombres, números, etc. Ya que, dos palabras distintas pero de entidades idénticas, como por ejemplo Irlanda y Noruega, aportan la misma información al contexto de la frase porque las dos son localizaciones geográficas, por lo que sería más conveniente sustituirlas por una única palabra como *location*. Esto siempre es interesante, ya que reducir el vocabulario quitando palabras insignificantes conllevará a un menor ruido en el entrenamiento. En este apartado hacemos un pequeño resumen de diferentes paquetes de NLU que hemos tratado para comprobar su utilidad en este proceso.

NLTK (Natural Language ToolKit): NLTK es una herramienta desarrollada originalmente en el año 2001 por la Universidad de Pennsylvania, y está actualmente implementada en Python [3], lo cual facilita mucho su uso, ya que, como hemos mencionado anteriormente, será el lenguaje de programación que utilizaremos. Con él podremos etiquetar las palabras según la sintaxis [2], detectar entidades (como las mencionadas anteriormente, lo cual es lo que más nos interesa), etiquetar expresiones regulares como por ejemplo el gerundio (caminando), obtener el vocabulario del corpus y disponer de herramientas de análisis como por ejemplo las frecuencias con las que aparecen las palabras en él, etc. Al final de esta sección compararemos las herramientas que hemos utilizado para la detección de entidades, y argumentaremos cual hemos escogido para dicha tarea.

Spacy: Spacy es otra herramienta para NLP (Natural Language Processing), de código libre implementado en Python, que comenzó a desarrollarse en el año 2015. Dispone de más herramientas que NLTK y presenta ciertas mejoras:

- Detección de entidades y POSTagger(Mejorado respecto a NLTK)
- Modelos de redes neuronales(no implementado en NLTK)
- Word Vectors entrenados(no implementado en NLTK)
- Visualizador de relaciones sintácticas(no implementado en NLTK)

Watson NLU: Watson es una API (Application Programming Interface) desarrollada por IBM, la cual ofrece un gran abanico de herramientas para analizar textos. Entre ellas encontramos: sentimiento del texto (negativo o positivo), emoción (parámetros de disfrute,odio,enfado,tristeza...), detección de palabras clave, entidades, análisis sintáctico, etc. La herramienta para analizar el sentimiento de un texto (o una frase) es interesante, ya que aporta una información crucial a la hora de identificar la intención de éste. Sin embargo, al disponer de un corpus en el que las frases están etiquetadas por su polaridad (sentimiento), no veremos necesario utilizar esta API, puesto que el resto de funciones (entidades, sintaxis...) estarán bien cubiertas por el paquete que acabaremos utilizando, el cual expondremos al final de esta sección.

[1] Es el primer paso suponiendo que el usuario está comunicándose directamente mediante texto, si no es así, primero se tendría que transcribir la información oral a escrita.

[2] También conocido como POSTagger, Part Of Speech en inglés

Standford CoreNLP: Standford comenzó a desarrollar CoreNLP a partir de 2010 [4], es una de las mejores herramientas para el análisis del lenguaje natural, por razones que veremos más adelante. Está implementada en Python y mayormente en Java, por lo que hemos utilizado su implementación en Java para obtener mejores resultados.

Las funciones que puede efectuar esta herramienta son parecidas a las que efectuaban los anteriores, sin embargo, destaca por los buenos resultados que obtiene, sobre todo en el reconocimiento de entidades lo cual es de nuestro mayor interés [5]. Como podemos ver en la tabla 2.1, CoreNLP detecta mucho mejor las entidades que NLTK o Spacy.

Frase	**NLTK**	**Spacy**	**CoreNLP**
Por lo menos 3 o 4 días a la semana	GPE lo menos 3 o 4 días a la semana	Por lo menos 3 o 4 días a la semana	Por lo menos NUMBER o NUMBER días a la semana .
Encantado , Natalia	GPE , GPE	Encantado , LOC	Encantado , PERSON .
Hoy en día la lectura , paseos y ... cuestiones	Hoy en día la lectura , paseos y ... cuestiones	Hoy en día la lectura , paseos y ... cuestiones	DATE en día la lectura , paseos y ... cuestiones .

Cuadro 2.1: Tabla de comparación de detección de entidades con las herramientas NLTK, Spacy y CoreNLP. Etiquetas: GPE (entidad geo-política), LOC (localización) y en el caso de CoreNLP no son abreviaturas si no las entidades en inglés simplemente.

En esta tabla podemos ver tres ejemplos claros del etiquetado que lleva a cabo cada herramienta. Es evidente que CoreNLP realiza un etiquetado prácticamente excelente, cabe decir que no es perfecto ya que hay algunos pocos casos en los que no detecta bien las entidades, no obstante, NLTK y Spacy demuestran un mal funcionamiento, esto se puede deber a que hayamos utilizado unos malos modelos de detectores de entidades. Asimismo, es obvio que CoreNLP dispone de una amplia variedad de entidades, mientras que Spacy no es capaz de reconocer números y fechas, y NLTK asocia un lugar geo-político prácticamente a toda palabra que comience en mayúscula.

Por todas estas razones, CoreNLP será la herramienta que utilizaremos para la detección de entidades.

2.3. Keras y Pytorch

En esta sección comentaremos los aspectos importantes del software necesario para construir el modelo de nuestra red neuronal. La mayor diferencia entre *Keras* y *Pytorch* es el nivel de programación. *Keras* (respaldado por Google desde 2015) es una API de alto nivel, utiliza algoritmos de otros paquetes para agilizar los cálculos, siendo TensorFlow o Theano[3] los principales soportes. Al ser de alto nivel nos permite construir modelos con una gran variedad de redes predefinidas, teniendo éstos un funcionamiento establecido, sin embargo, esto limita la capacidad de crear redes muy complejas. Por otro lado, *Pytorch* (respaldado por Facebook desde 2016) es una API de bajo nivel, es decir, podemos crear nuevas redes neuronales con funcionalidades y dinámicas distintas, ésto otorga una gran versatilidad, y por esta razón estos últimos años su uso ha aumentado entre los académicos.

Nuestra mayor cuestión ahora es saber cual de las dos utilizaremos, para ello primero tendremos que decidir la arquitectura del modelo que emplearemos para nuestra red, y si es una estructura viable en *Keras* no tendremos por qué complicarnos demasiado utilizando *Pytorch*.

[3]TensorFlow fue desarrollado por Google, mientras que Theano fue desarrollado por la Universidad de Montreal.

Capítulo 3

Redes neuronales

En este capítulo explicaremos las arquitecturas de redes neuronales más importantes a la hora de analizar secuencias de palabras. Comenzaremos con las redes recurrentes (3.1) siendo éstas las más simples, después las LSTM y GRU (3.2), y por último las redes convolucionales (3.3). Al tratarse las frases de una secuencia de palabras, definiremos el tiempo t como el índice de cada palabra en la secuencia, y servirá para referirnos al momento que nos encontramos en la inferencia, como se puede observar en la Figura 3.1.

3.1. RNN(Recurrent Neural Network)

Se trata de una adaptación de las redes *feed-forward*[1], se especializa en el análisis de secuencias. Al tratar con secuencias, la inferencia se realiza de elemento en elemento, por esta razón se define el tiempo para las *RNN*-s, siendo el tiempo el paso en el que nos encontramos en la inferencia. Estas redes tienen un estado interior, el cual se usa como *input* adicional en cada momento de la inferencia, ésto permite que la red sea "consciente" del contexto, o dicho de otra forma, el estado interno sirve como una memoria que da información sobre los elementos anteriores de la secuencia. Las *RNN*-s a lo largo del tiempo se podrían ver como varias redes *feed-forward* conectadas entre sí, como podemos apreciar en la imagen de abajo:

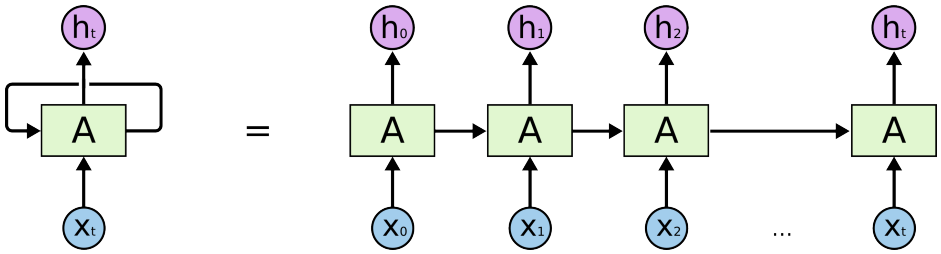


Figura 3.1: A la izquierda vemos la estructura de la red donde la entrada y salida cambia con el tiempo (esquema enrollado), y a la derecha la estructura como una secuencia de redes densas conectadas entre sí (esquema desenrollado). Ref: [6].

Las *RNN*-s presentan un problema al entrenarlas, puesto que el método de descenso de gradiente (4.1) es inestable en estos sistemas. Esto se debe a que el gradiente disminuye exponencialmente a lo largo del tiempo hasta llegar a anularse (a veces también diverge), causando la parálisis del entrenamiento. Una solución que se propuso para evitar el *vanishing gradient problem* fue implementar pequeñas unidades de memoria que fueran capaces de guardar información a largo plazo, ya que una *RNN* estándar solo es capaz de almacenar información a corto plazo. De esta idea nacieron las *LSTM* (Long Short Term Memory) y *GRU* (Gated Recurrent Unit), capaces de procesar largas secuencias con éxito, a diferencia de las *RNN*.

[1] Las redes feed-forward son las redes densas, que constan de un cierto número de capas, estando los nodos en las capas y todos los nodos están conectados con todos nodos anteriores y posteriores.

3.2. LSTM y GRU

La arquitectura del LSTM (Long Short Term Memory) se centra en una memoria que puede mantener su estado a lo largo del tiempo, a la vez que en una serie de funciones de activación[2] no lineales, las cuales regulan el tráfico de la información.

En este apartado nos centraremos en la arquitectura denominada *vanilla* [7]. En la imagen de abajo vemos un esquema que nos facilitará el entendimiento de la estructura:

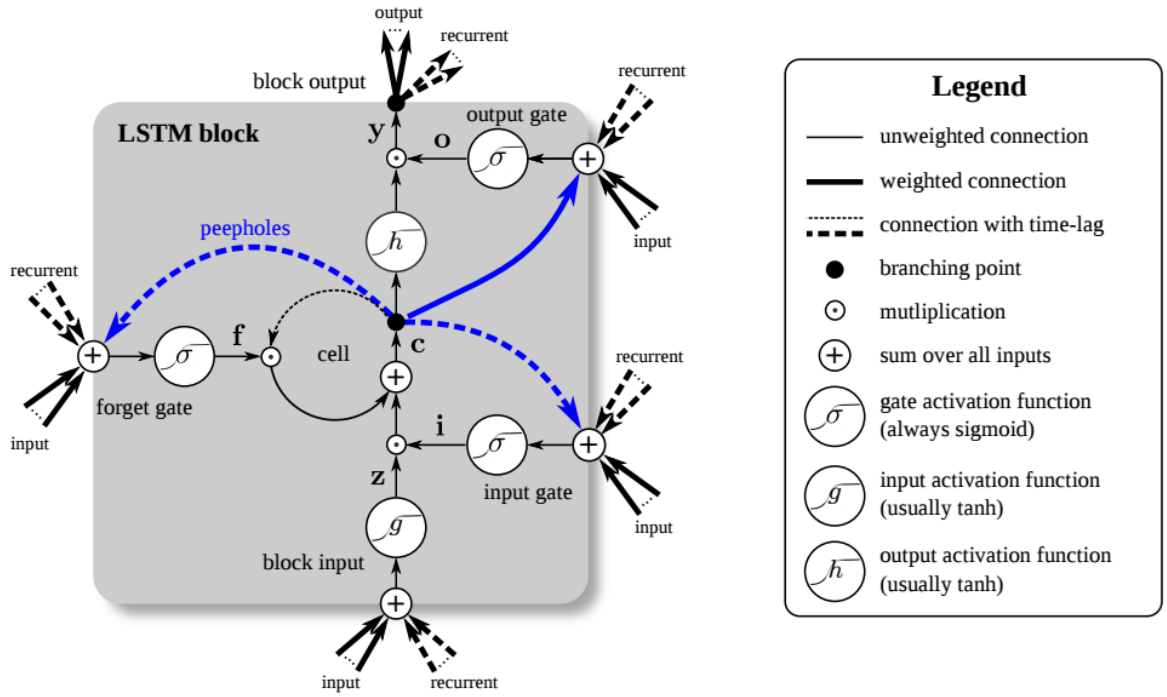


Figura 3.2: Esquema del bloque *LSTM vanilla*. Está formada por varias entradas, diferentes funciones de activación, y conexiones como la *Forget gate*(f), esta implementación permite a la red resetear su propio estado interno (memoria) y *Peephole connections*, gracias a estas conexiones la *LSTM* obtiene un mayor control sobre el tiempo. Ref: [8].

Uno de los componentes más importantes es la unidad de estado $c_i^{(t)}$, la cual depende de sí misma linealmente y está regulada por la puerta de olvido $f_i^{(t)}$(*forget gate*), variando el peso desde 0 a 1 mediante la función sigmoide:

$$f_i^{(t)} = \sigma\left(b_i^f + \sum_j U_{i,j}^f x_j^{(t)} + \sum_j W_{i,j}^f y_j^{(t-1)}\right) \tag{3.1}$$

Donde $\mathbf{x}^{(t)}$ es la entrada en el momento t, $\mathbf{y}^{(t-1)}$ la salida de la red en el momento anterior, y $\mathbf{b}^f, \mathbf{U}^f, \mathbf{W}^f$ siendo respectivamente los umbrales, pesos de entrada y pesos recurrentes para la *forget gate*.

[2]Las funciones de activación determinan la salida para cada valor de la entrada, suelen utilizarse funciones acotadas y que cumplen cierta relación con sus derivadas. Las más utilizadas son la sigmoide($\sigma(x)$) y la tangente hiperbólica($tanh(x)$).

Por otro lado, tenemos la puerta de entrada exterior(en el gráfico se denomina como **i** pero para no confundirnos con la notación nos referiremos a ella como **g**) que regula como afecta la entrada misma en el unidad de estado (3.3). Se calcula exactamente igual que la puerta de olvido, solo que con sus propios pesos:

$$g_i^{(t)} = \sigma\Big(b_i^g + \sum_j U_{i,j}^g x_j^{(t)} + \sum_j W_{i,j}^g y_j^{(t-1)}\Big) \tag{3.2}$$

Una vez calculados las puertas de entrada exterior y de olvido, la unidad de estado viene dada por esta formula:

$$c_i^{(t)} = f_i^{(t)} c_i^{(t-1)} + g_i^{(t)} \sigma\Big(b_i + \sum_j U_{i,j} x_j^{(t)} + \sum_j W_{i,j} y_j^{(t-1)}\Big) \tag{3.3}$$

Por último, la salida estará dada por la tangente hiperbólica de la unidad de salida[3], y estará regulada por la puerta de salida:

$$\begin{aligned} y_i^{(t)} &= tanh(c_i^{(t)}) q_i^{(t)} \\ q_i^{(t)} &= \sigma\Big(b_i^o + \sum_j U_{i,j}^o x_j^{(t)} + \sum_j W_{i,j}^o y_j^{(t-1)}\Big) \end{aligned} \tag{3.4}$$

Otra alternativa para las celdas LSTM son las denominadas GRU (Gated Recurrent Unit). Podría entenderse como una simplificación de lo anterior, ya que la mayor diferencia entre ellas es que mientras en la LSTM tenemos una puerta que controla el olvido y otra la actualización de la unidad de estado, en el caso de la GRU una sola unidad controla ambos procesos a la vez, en nuestro caso la denominaremos $z_i^{(t)}$. Dicho estado ($h_i^{(t)}$) viene dado por la siguiente ecuación:

$$h_i^{(t)} = z_i^{(t-1)} h_i^{(t-1)} + (1 - z_i^{(t-1)}) \sigma\Big(b_i + \sum_j U_{i,j} x_j^{(t-1)} + \sum_j W_{i,j} r_j^{(t-1)} h_j^{(t-1)}\Big) \tag{3.5}$$

Donde **z** representa la puerta *update* y **r** la de *reset*. Ambas vienen dadas por las siguientes ecuaciones:

$$\begin{aligned} z_i^{(t)} &= \sigma\Big(b_i^z + \sum_j U_{i,j}^z x_j^{(t)} + \sum_j W_{i,j}^z h_j^{(t)}\Big) \\ r_i^{(t)} &= \sigma\Big(b_i^r + \sum_j U_{i,j}^r x_j^{(t)} + \sum_j W_{i,j}^r h_j^{(t)}\Big) \end{aligned} \tag{3.6}$$

En el esquema presentado abajo podemos visualizar la labor de cada puerta. Gracias a la función sigmoide actúan como interruptores, observando por ejemplo los casos extremos en el que z_i^t valiese 1, el termino de la derecha en la ecuación 3.5 se anula, es decir, no se actualizaría el estado interno de la GRU dejándolo igual que estaba en el momento anterior, ese sería el caso que observamos en el esquema 3.3 en el que el interruptor z está orientado hacia la izquierda. Si z_i^t fuese igual a 0, estaríamos en el caso contrario y el interruptor z estaría orientado hacia la derecha. Con la puerta *reset* sucede algo parecido, pero éste simplemente controlará si el estado interno actual afectará al siguiente estado.

[3]Se suele utilizar la tangente hiperbólica como función de salida, pero *Keras* nos permite escoger que función utilizaremos para la salida.

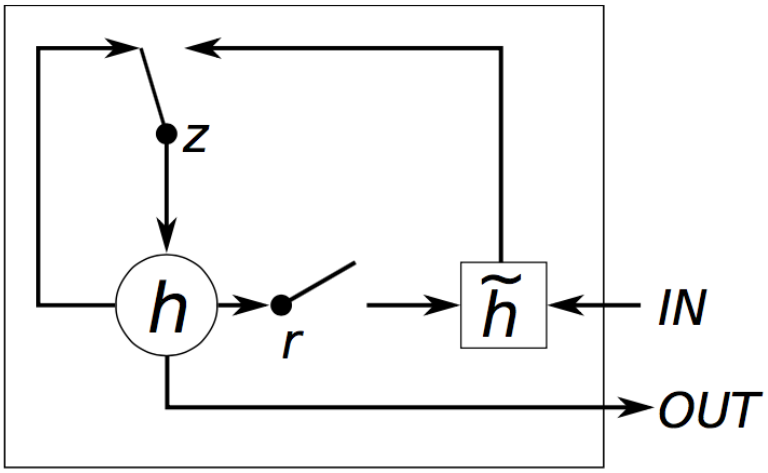


Figura 3.3: Esquema simplificado de una GRU. Ref: [9]

Las redes recurrentes son claves a la hora de procesar secuencias, sin embargo, existen otro tipo de redes que pueden ser también útiles a la hora de hacer NLU, y esas son las redes convolucionales, de las cuales hablaremos en la siguiente sección. Para más información detallada sobre las celdas LSTM y GRU véase [10].

3.3. Redes Convolucionales

La definición estrictamente matemática de la convolución vendría dada por la siguiente ecuación[4]:

$$(f * g)(x) \equiv \int f(t)g(x-t)dt \tag{3.7}$$

Sin embargo, las redes neuronales convolucionales aplican una operación que podría entenderse como una generalización discreta de la ecuación 3.7. En el caso de dos dimensiones, si tenemos dos matrices I y K[5], se define la convolución de esta forma:

$$S(i,j) = (I * K)(i,j) = \sum_m \sum_n I(i+m, j+n)K(m,n) \tag{3.8}$$

Esta operación se puede comprender más fácilmente con la imagen que presentamos a continuación:

[4] A esta operación se le denomina la convolución de $f(x)$ sobre $g(x)$.
[5] La matriz K suele llamarse filtro o *kernel*.

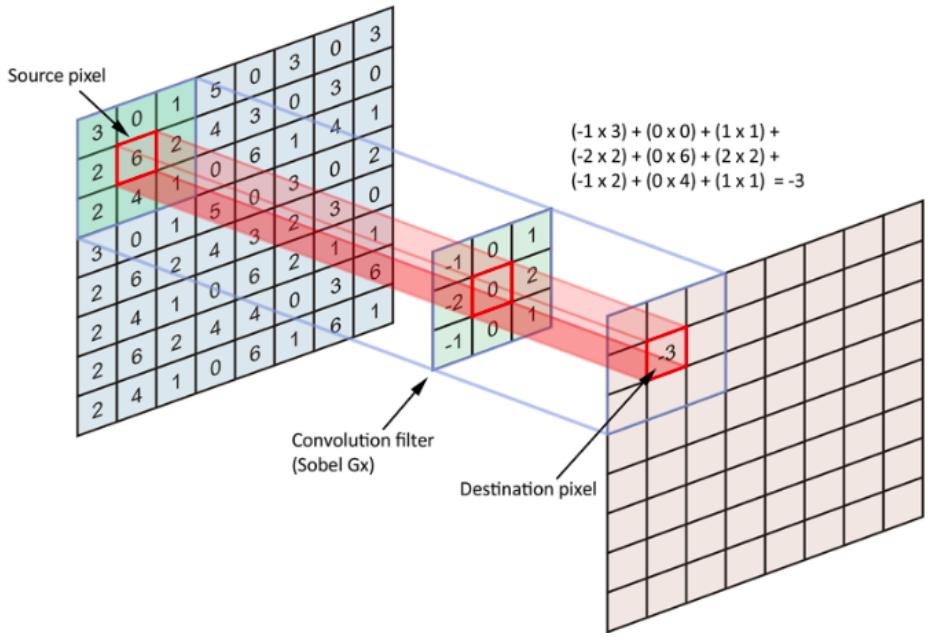


Figura 3.4: Visualización de la convolución. El pixel de salida es calculado multiplicando cada pixel de la imagen de entrada con los elementos del filtro, y después sumándolos. El proceso de convolución sería hacer este mismo proceso con todos los píxeles de la imagen de entrada. Ref: [11].

En el caso del procesamiento de imágenes, los pesos del filtro son de una gran importancia a la hora de saber que efecto crearán en la imagen, ya que se suele utilizar la convolución para difuminar, reducir la calidad de la imagen,etc. Por lo tanto, con una red convolucional lo que se desea es conseguir los pesos óptimos para detectar las características deseadas de una imagen o de una frase. Una arquitectura general para un modelo basado en una red convolucional con el objetivo de clasificar frases es mostrada a continuación.

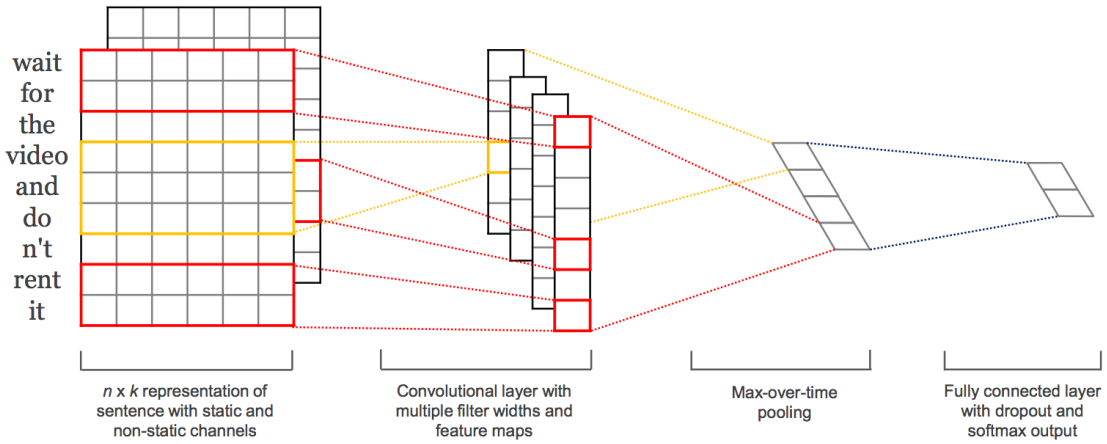


Figura 3.5: Estructura general de un clasificador usando una red convoucional. La tercera capa aplica la operación *pooling*, la cual selecciona una de las salidas a lo largo de toda la frase, haciendo así desaparecer la dependencia en la longitud de la frase a analizar. Por ejemplo, el *Maxpooling* escoge solamente el valor más grande que haya tenido cada salida. Ref: [12].

Capítulo 4

Algoritmos Deep Learning

El *Deep Learning* ha progresado muchísimo estos últimos años, y muchos son los algoritmos y técnicas que se han desarrollado para optimizar la inteligencia artificial. En este capítulo, hablaremos del algoritmo más importante en el ámbito del aprendizaje supervisado, el *Back Propagation*, después expondremos algunos métodos para evitar el *overfitting* , y finalmente explicaremos el algoritmo de búsqueda local que utilizaremos para optimizar el resultado de nuestras redes.

4.1. *Back Propagation*

El *Back Propagation* es la base para optimizar la redes neuronales, habiendo definido una función de coste del sistema (o función de error) el algoritmo trata de minimizarlo propagando el error hacia atrás en la red. Para ello hay que tener en cuenta que la función de coste dependerá de los datos de entrada($\mathbf{x}^i$), salida (y^i) y de los parámetros de la red ($\boldsymbol{\theta}$)[1]: $L = L(\mathbf{x}^i, y^i, \boldsymbol{\theta})$.

El algoritmo comienza con unos valores iniciales para los parámetros de la red[2], calcula el gradiente de la función de coste respecto a los parámetros de la red para cada dato de entrenamiento y se calcula la media[3]:

$$\boldsymbol{g}_t = \frac{1}{m} \sum_{i=1}^{m} \nabla_{\boldsymbol{\theta}_t} L(\mathbf{x}^i, y^i, \boldsymbol{\theta}_t) \tag{4.1}$$

Sin embargo, el coste computacional de este procedimiento es del orden $\mathcal{O}(m)$. Ésto es un grave problema ya que el tamaño de los datos que se suelen utilizar para entrenar suele rondar los millones, por lo que no es viable calcular el gradiente de esta forma en cada iteración. Para evitar este problema se desarrolló el *Stochastic Gradient Descent* (SGD) [13], el cual simplemente implica utilizar *minibatches* (pequeñas muestras) en cada iteración, siendo estos de un tamaño reducido (m'). Una vez calculado el gradiente se actualizarán los parámetros de la red restándoles dicho gradiente por el ratio de aprendizaje[4] (ϵ), se debe entender en la siguiente ecuación que cada parámetro se actualiza con su correspondiente componente del gradiente:

$$\begin{aligned} \boldsymbol{g}_t &= \frac{1}{m'} \sum_{i=1}^{m'} \nabla_{\boldsymbol{\theta}_t} L(\mathbf{x}^i, y^i, \boldsymbol{\theta}_t) \\ \boldsymbol{\theta}_{t+1} &= \boldsymbol{\theta}_t - \epsilon \mathbf{g}_t \end{aligned} \tag{4.2}$$

[1] El símbolo $\boldsymbol{\theta}$ engloba todos los pesos ($\boldsymbol{W}$,$\boldsymbol{U}$) y umbrales ($\boldsymbol{b}$) de la red.

[2] La inicialización de los parámetros se puede personalizar con los *initializers* que dispone Keras, pudiendo iniciarlos todos a un valor constante o aleatoriamente siguiendo una distribución Gaussiana.

[3] La cantidad total de datos de entrenamiento lo denominaremos m, y t indicará la iteración.

[4] El ratio de aprendizaje suele ser $\epsilon \in [0, 1)$. No tiene por que ser constante durante el entrenamiento, se suele definir un parámetro de declive α que determina como irá reduciéndose linealmente ϵ_t en cada iteración.

Este proceso se podría visualizar con un ejemplo de una función en una dimensión, como podemos ver en la Figura 4.1. En el ejemplo al no encontrarse el punto inicial en el mínimo de la función de error, existe una pendiente no nula en ese punto, en este caso positiva, por lo que el mínimo debe encontrarse en el sentido contrario de la pendiente, de ahí que se reste el gradiente a los parámetros, por que de esta forma nos aseguramos que los valores de los parámetros se moverán en el sentido contrario a la que crece la función.

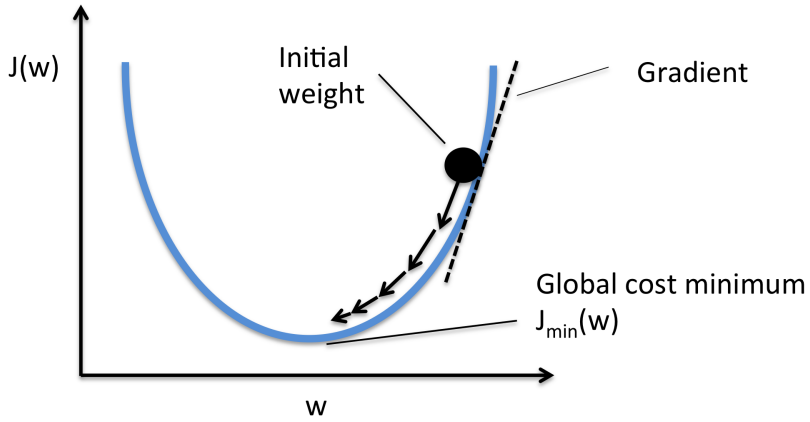


Figura 4.1: Representación del algoritmo de *Back Propagation* en una dimensión. Donde a los parámetros de la red se les denomina w y a la función de error $J(w)$. Ref: [14].

Aun así, la función de error es función de todos los parámetros de la red, éstos suelen rondar entre miles y millones de parámetros, por lo que la minimización es mucho más complicado cuando tenemos una función con tantas variables. Es más, pueden existir mínimos locales, o incluso puntos de inflexión, los cuales evitarán que encontremos el mínimo absoluto siguiendo el procedimiento 4.2. En la siguiente sección explicaremos un par de soluciones a este problema.

4.2. Optimizadores

Dentro del algoritmo de *Back Propagation* existen distintas formas de actualizar los parámetros ($\boldsymbol{\theta}$) de la red, entre otros están el Adam (*Adaptative moment estimation*)y *Nesterov momentum*, los cuales compararemos en este trabajo. A éstos se les denomina optimizadores, y en esta sección presentaremos cuatro, primero daremos una descripción general de los algoritmos y al final de cada subsección definiremos cómo se ajustan los parámetros ($\boldsymbol{\theta}$). En todas las subsecciones se debe entender que si no se dice lo contrario $\mathbf{g}_t$ se calcula de la forma 4.2 presentada anteriormente y que las operaciones entre matrices se hacen por elementos[5].

4.2.1. AdaGrad

AdaGrad adapta los ratios de aprendizaje de cada parámetro de la red ajustándolos inversamente proporcional a la raíz cuadrada de la suma del cuadrado de los anteriores gradientes [15]. Esto regula los cambios de cada parámetro, asignando un pequeño ratio de aprendizaje a los que conllevan una derivada parcial muy grande y mientras que los que tienen unas derivadas no tan grandes no ven casi disminuido su ratio de aprendizaje. Teóricamente este algoritmo posee buenas propiedades, pero en la práctica la acumulación del cuadrado del gradiente desde el principio resulta en un prematuro y excesivo descenso del ratio de aprendizaje, dificultando así el aprendizaje.

[5]Es decir, como el producto de Hadamard, como por ejemplo $\boldsymbol{g}_t^2$ sería $\boldsymbol{g}_t$ con cada elemento al cuadrado.

En este algoritmo se define un ratio de aprendizaje global constante, ϵ, y un pequeño constante $\delta \sim 10^{-7}$ que servirá para dar estabilidad numérica.

$$\begin{aligned} \mathbf{r}_t &= \mathbf{r}_{t-1} + \boldsymbol{g}_t^2 \\ \Delta\boldsymbol{\theta}_t &= -\frac{\epsilon}{\delta + \sqrt{\mathbf{r}_t}}\boldsymbol{g}_t \\ \boldsymbol{\theta}_{t+1} &= \boldsymbol{\theta}_t + \Delta\boldsymbol{\theta}_t \end{aligned} \tag{4.3}$$

4.2.2. RMSProp

RMSProp arregla el problema de la acumulación excesiva del gradiente a lo largo de la iteraciones, multiplicando a la acumulación anterior un parámetro ρ que hará que su peso se reduzca a lo largo de las iteraciones exponencialmente, de esta forma no se reducirá tan drásticamente el ratio de aprendizaje. AdaGrad funciona bien para funciones convexas, pero si la función de error no es convexa (que por lo general no lo es) entonces podría ser que durante el aprendizaje se atravesaran diferentes estructuras hasta al final llegar a una zona convexa, pero para entonces AdaGrad ya habría reducido tanto el ratio que no podría seguir avanzando. Más como RMSProp olvida las contribuciones del pasado lejano, al llegar a esa zona convexa convergería rápidamente hacia el mínimo. Se podría decir que RMSProp es lo mismo que AdaGrad pero inicializado en diferentes momentos del entrenamiento, y se ha demostrado que empíricamente es uno de los algoritmos de optimización más efectivos y prácticos.

Como hemos indicado, se define un ratio de declive ρ, y como en AdaGrad se define también ϵ y $\delta \sim 10^{-6}$.

$$\begin{aligned} \mathbf{r}_t &= \rho\mathbf{r}_{t-1} + (1-\rho)\boldsymbol{g}_t^2 \\ \Delta\boldsymbol{\theta}_t &= -\frac{\epsilon}{\sqrt{\delta + \mathbf{r}_t}}\boldsymbol{g}_t \\ \boldsymbol{\theta}_{t+1} &= \boldsymbol{\theta}_t + \Delta\boldsymbol{\theta}_t \end{aligned} \tag{4.4}$$

4.2.3. *Nesterov momentum*

Sutskever y otros [16] introdujeron una variante del algoritmo con momento inspirado en el método NAG (*Nesterov Accelerated Gradient*). En éste, se le suma una porción (determinada por el hiperparámetro α) del anterior gradiente a la 'velocidad' o momento, se utiliza esta cantidad para actualizar los parámetros de la red. La mayor diferencia con respecto a NAG es que el gradiente se evalúa una vez aplicada la 'velocidad' actual, por lo que el método *Nesterov momentum* se entiende como una corrección del método estándar de NAG.

Los hiperparámetros de este algoritmo son el ratio de aprendizaje ϵ y el parámetro de momento α.

$$\begin{aligned} \boldsymbol{g}_t &= \frac{1}{m'}\sum_{i=1}^{m'} \nabla_{\boldsymbol{\theta}_t} L(\mathbf{x}^i, y^i, \boldsymbol{\theta}_t + \alpha\boldsymbol{v}_t) \\ \boldsymbol{v}_{t+1} &= \alpha\boldsymbol{v}_t - \epsilon\boldsymbol{g}_t \\ \boldsymbol{\theta}_{t+1} &= \boldsymbol{\theta}_t + \boldsymbol{v}_{t+1} \end{aligned} \tag{4.5}$$

4.2.4. Adam

Se puede entender Adam [17] como una combinación de RMSProp con el concepto de momento, con alguna distinciones. Adam añade la acumulación del gradiente (primer momento,$\mathbf{s}$) al contrario de RMSProp, el cual solo acumulaba el cuadrado del gradiente (segundo momento,$\mathbf{r}$), y también utiliza otro hiperparámetro ρ_1 para regularlo como en RMSProp. Por último, Adam hace una corrección de cada momento dividiéndolo por $(1-\rho^t)$. Se dice que Adam es un algoritmo robusto respecto a la decisión de los valores de los hiperparámetros, aunque el ratio de aprendizaje global suele tener que cambiarse en vez utilizar el valor por defecto.

Los valores recomendados para cada hiperparámetro son: $\epsilon = 0{,}001$, $\rho_1 = 0{,}9$, $\rho_2 = 0{,}999$ y $\delta = 10^{-8}$.

$$\begin{aligned}
\mathbf{s}_t &= \rho_1 \mathbf{s}_{t-1} + (1-\rho_1)\boldsymbol{g}_t \\
\mathbf{r}_t &= \rho_2 \mathbf{r}_{t-1} + (1-\rho_2)\boldsymbol{g}_t^2 \\
\hat{\mathbf{s}}_t &= \frac{\mathbf{s}_t}{1-\rho_1^t} \\
\hat{\mathbf{r}}_t &= \frac{\mathbf{r}_t}{1-\rho_2^t} \\
\Delta\boldsymbol{\theta}_t &= -\epsilon \frac{\hat{\mathbf{s}}_t}{\delta + \sqrt{\hat{\mathbf{r}}_t}} \\
\boldsymbol{\theta}_{t+1} &= \boldsymbol{\theta}_t + \Delta\boldsymbol{\theta}_t
\end{aligned} \tag{4.6}$$

4.3. Cómo minimizar el *overfitting*

Al entrenar redes neuronales, uno de los mayores problemas a los que nos enfrentamos es el denominado *overfitting* (sobreentrenamiento), se le denomina así al momento en el que la red ha aprendido a adaptarse 'demasiado bien' a los datos de entrenamiento. Por lo tanto, obtiene una gran parte de la salida deseada cuando es inferida con una entrada del corpus de entrenamiento, aunque al inferir información nueva no obtenemos un buen resultado. En esta sección hablaremos de tres métodos para deshacernos o minimizar el *overfitting*: el *dropout*, los regularizadores y el *early-stopping*.

1. Dropout: Con el *dropout* lo que hacemos es anular la actividad de ciertos nodos de la red al azar durante la inferencia. Esto ayuda a la red a no especializarse demasiado con los datos de entrenamiento y a obtener un entendimiento más abstracto del *input*. *Keras* implementa este método en sus capas, por ejemplo, simplemente añadiendo el parámetro *dropout=0.1* haremos que el 10 % de los nodos de la capa anulen su actividad aleatoriamente.

2. Regularizadores: Este método trata de penalizar pesos y umbrales con valores absolutos demasiado grandes, sumando a la función de error otra función que dependa de dichos pesos. Un par de ejemplos son los regularizadores *Lasso Regression* y *Ridge regression* (implementables en *Keras* como regularizadores l1 y l2 respectivamente), el primero suma el valor absoluto de todos los pesos a la función de error multiplicado por un parámetro λ[6], el cual determina la importancia que tendrán los pesos en la función de error total, y el segundo suma el cuadrado de los pesos.

$$\begin{aligned}
L1: &\quad f'(\mathbf{y},\mathbf{a}) = f(\mathbf{y},\mathbf{a}) + \lambda \sum_i |w_i| + \mu \sum_j |\beta_j| \\
L2: &\quad f'(\mathbf{y},\mathbf{a}) = f(\mathbf{y},\mathbf{a}) + \lambda \sum_i w_i^2 + \mu \sum_j \beta_j^2
\end{aligned} \tag{4.7}$$

[6] En esta notación w_i serán los pesos y β_j los umbrales. Podemos aplicar regularizadores a ambos por separado, por lo que denominaremos λ y μ a los parámetros de cada uno respectivamente, pues no tienen por que tener el mismo valor.

3. Early-Stopping: Para entender este método primero debemos hacer una breve explicación de los datos de entrenamiento y validación. Mientras entrenamos la red, tendremos separado nuestro corpus en dos grupos de datos, el de entrenamiento, el cual suele ser un 70 % del corpus total, y el resto siendo el de validación. El bloque de validación servirá para calcular el error de validación en cada época, ya que en cada época los pesos y umbrales de la red irán actualizándose y nos interesa saber con que precisión se esta ajustando la red a los datos ajenos a los del entrenamiento. Por otro lado, iremos calculando en cada época el error que ha producido con los datos de entrenamiento. En el gráfico 4.2 podemos apreciar la evolución típica de dichos errores durante el entrenamiento:

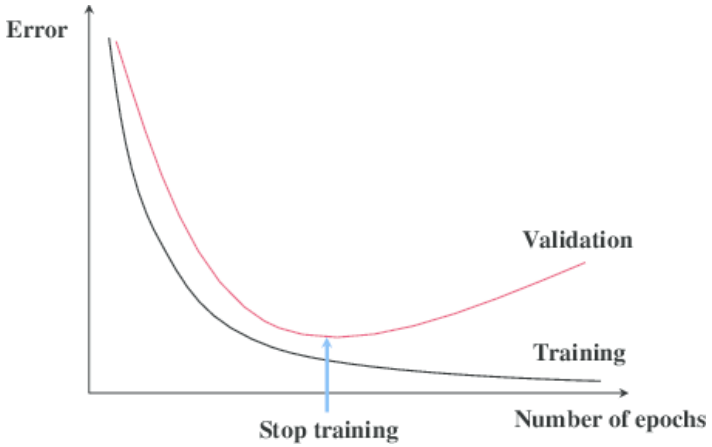


Figura 4.2: Error de entrenamiento y de validación a lo largo de las épocas de entrenamiento. Ref: [18].

En el gráfico vemos dos zonas, la primera es en la que el error de validación baja a medida que avanza el entrenamiento, y la segunda en la que comienza a aumentar. A estas zonas se les denomina como zonas de *underfitting* y *overfitting* respectivamente, y como nuestro objetivo es que nuestra red no se encuentre en ninguna de esas dos zonas, el *early-stopping* simplemente trata de quedarse con la red que ha obtenido el mínimo de error de validación (lo que en el gráfico esta marcado con una flecha azul). Por lo que, mientras entrenemos a la red, iremos guardando los pesos actualizados de cada época, y localizaremos aquellos más óptimos para la red.

4.4. Búsqueda local

Tras entrenar nuestra red existen distintas formas de optimizar los resultados, una de ellas es utilizar un algoritmo de búsqueda local. La salida de la red será una distribución de probabilidad, de la cual consideraremos que la clase de la frase de entrada será la correspondiente a la probabilidad máxima, sin embargo, podemos modificar dicha salida sumando un vector con la misma dimensión que la salida, al que denominaremos $\boldsymbol{\delta}$. Con esto básicamente haremos un *shift* diferente en cada probabilidad, y tal vez pueda mejorar las predicciones. El objetivo del algoritmo será buscar el valor del vector $\boldsymbol{\delta}$ que optimice la métrica que utilizaremos para evaluar la red, en nuestro caso será el *F1 Macro Score* (5.3).

Primero se asignará un valor inicial al vector $\boldsymbol{\delta}$, por no escoger un vector nulo asignaremos el vector que representa las frecuencias invertidas $\boldsymbol{w}$ de cada clase:

$$\delta_i = w_i = \frac{f_i^{-1}}{\sum_j f_j^{-1}} \tag{4.8}$$

Después se crearán otros vectores $\boldsymbol{\delta}'$ los cuales se sumarán a los vectores de salida, y como hemos explicado anteriormente compararemos los *F1 Score* obtenidos de cada vector. Para obtener estos nuevos vectores cogeremos distintos pares de coordenadas, y a uno de ellos le sumaremos un valor aleatorio (ξ) y al otro le restaremos lo mismo, con esto pretendemos conservar la propiedad $\sum_i \delta'_i = 1$, el cual nos servirá para acotar los valores de las coordenadas de $\boldsymbol{\delta}'$. Al encontrar un vector que dé un resultado mejor que el anterior asignaremos este nuevo vector a $\boldsymbol{\delta}$, y volveremos a repetir el procedimiento durante N iteraciones.

A continuación mostramos el diagrama de flujo de este algoritmo, hay que destacar que la salida $\mathbf{o}$ es una lista de vectores, ya que representa los softmax para distintas frases de entrada, por lo que cuando escribimos $\mathbf{o}+\boldsymbol{\delta}$ significa que sumamos $\boldsymbol{\delta}$ para cada vector softmax. Lo mismo sucede con $\boldsymbol{\delta}'$, por lo que cuando escribimos $\boldsymbol{\delta}+\boldsymbol{\xi}$ significa que creamos una serie de nuevos vectores aleatorios con el procedimiento mencionado anteriormente.

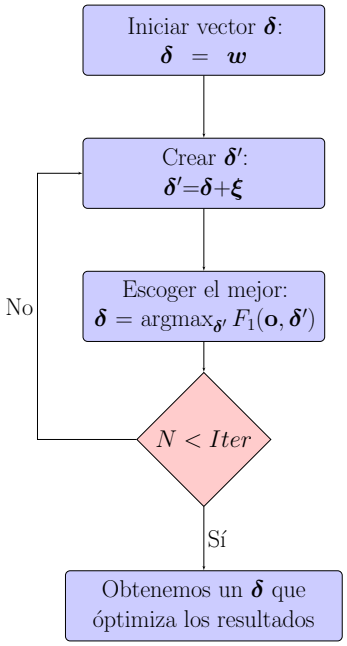


Figura 4.3: Diagrama de flujo del algoritmo de búsqueda local.

Durante este trabajo utilizamos 0,8 como valor máximo de ξ, y buscamos el $\boldsymbol{\delta}$ más óptimo durante $N = 10^5$ iteraciones.

Capítulo 5

Análisis del corpus

Como hemos mencionado en el capítulo 1, el corpus para este trabajo se nos ha sido proporcionado por el Proyecto Europeo *EMPHATIC*, el cual consta de unas 7.500 frases[1] etiquetadas. En este capítulo hablaremos sobre el objetivo de dicho proyecto, también haremos un analisis exhaustivo del corpus y estudiaremos la manera adecuada de determinar la eficiencia un clasificador. En la siguiente tabla mostramos la cantidad exacta de diálogos y frases que disponemos, a la vez que cuantas palabras diferentes existen en todo el corpus y el tamaño del vocabulario que utilizaremos para los *wordvectors*, ya que ignoraremos las palabras que aparezcan menos de 2 veces en todo el corpus.

Diálogos	Frases	Vocabulario completo	Vocabulario *wordvector*
142	7572	4448	2274

5.1. Qué corpus, con qué objetivo

El objetivo principal del proyecto *EMPHATIC* es desarrollar un asistente virtual, el cual hará la función de un *coach*. El *coaching* es un método que consiste en acompañar e instruir a una persona, con el objetivo de conseguir cumplir metas o de desarrollar hábitos específicos. En el caso de *EMPHATIC* estas metas o hábitos serán en el ámbito de la salud, ya que el asistente estará orientado para tratar con personas mayores, que necesiten una supervisión o motivación en los temas de nutrición, deporte, relaciones, etc.

Un asistente virtual consta de distintas partes las cuales ejecutan funciones especializadas. En la imagen 5.1 podemos ver un esquema con dichas partes y como están organizadas. Primero, se obtiene el *input*, el cual habrá que convertirlo a texto si se trata de un audio. Después, se lleva a cabo la comprensión del texto de entrada, este proceso tratará de dar una representación más abstracta de los datos de entrada, o bien devolviendo una representación numérica o bien como en nuestro caso etiquetando la frase según el tema, intención o polaridad. El siguiente módulo podría llamarse el çerebro"del asistente virtual, ya que es el gestor de diálogo el que procesa la frase de entrada e ingenia una frase de salida. Esta salida puede también ser una representación abstracta, por lo tanto el generador de respuesta se encarga de crear la frase *per se*. Por último, si el *output* debe ser en formato audio entonces se procesará la frase de salida por un sintetizador de voz.

[1] En realidad lo que se etiqueta son subfrases, es decir, pequeñas secciones de lo que respondía el usuario, ya que en cada sección podía hablar de temas distintos. Aunque sean subfrases durante el trabajo nos referiremos a ellos como frases.

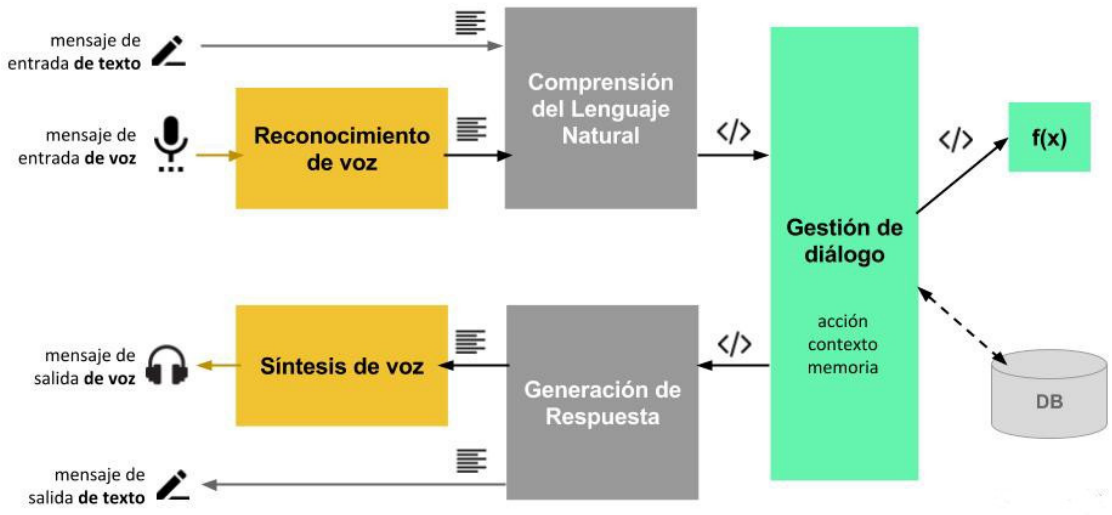


Figura 5.1: Esquema de asistente virtual con opción de tener audio o texto como *input* y *output*, por Nieves Ábalos.

El corpus se obtuvo con el experimento Mago de Oz, el cual se basa en hacer pensar a una persona que esta hablando con un robot, pero que en realidad se trata de una persona, entonces se entabla una conversación con ella y se guardan dichas conversaciones. Primero se les hacia preguntas sobre aficiones, para así poder relajarse, ya que nadie está acostumbrado a charlar con un robot. Entonces, una vez relajada la persona, el mago hacia preguntas sobre salud, hábitos, deporte, etc. Siempre intentando concretar un objetivo, motivar a la persona y definir un plan de acción para mejorar su situación actual, siguiendo el modelo *GROW*(*Goals* o metas, *Reality* o realidad, *Options* u opciones y *Will* o plan).

Una vez obtenidos los diálogos, se etiquetaron las frases por tema, intención del usuario y polaridad (positiva,negativa o neutra). En los diagramas del Apéndice A vemos las ramas de las etiquetas relacionadas con el tema e intención de la frase y sus frecuencias [19].

5.2. Estructura del corpus

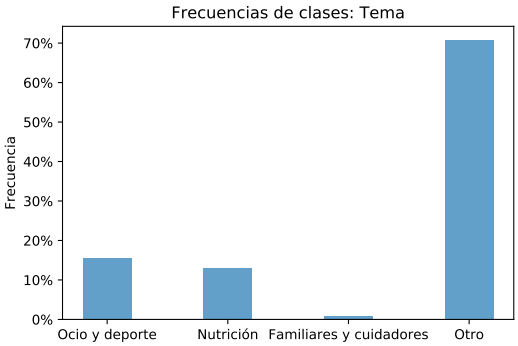
El corpus consta de una gran cantidad de frases etiquetadas de esta forma:

COACH:	¿Dónde comerías sola, entonces, en esos momentos que, bueno, pues no has podido?
USUARIO:	Ah, pues yo por ejemplo antes tenía una pauta que me gustaba pero no lo hacía toda las semanas, ¡eh!
POLARIDAD:	Neutra.
SUBFRASE 1:	antes tenía una pauta que me gustaba pero no lo hacía toda las semanas
ETIQUETA TEMA:	tema, nutrición
ETIQUETA INTENCIÓN:	intención, informar, hábito
ENTIDADES:	fechas relativas: todas las semanas

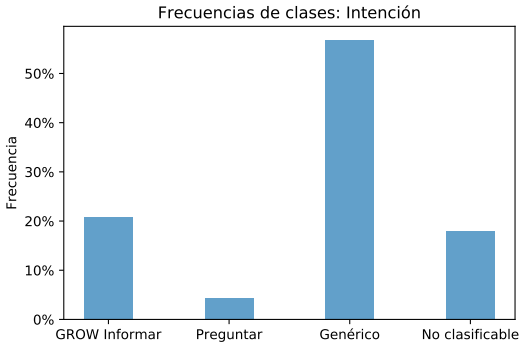
COACH:	Y entonces, para priorizar eso, ¿qué podrías hacer?
USUARIO:	Pues hombre, se me ocurre que podría hacer ejercicio más a menudo y salir a más conciertos de los que me gustan.
POLARIDAD:	Neutra.
SUBFRASE 1:	podría hacer ejercicio más a menudo
ETIQUETA TEMA:	tema, ocio y deporte, deportes, frecuencia
ETIQUETA INTENCIÓN:	intención, informar, plan, posible/no definitivo
ENTIDADES:	cantidades: más; fechas relativas: a menudo; acción: hacer ejercicio
SUBFRASE 2:	salir a más conciertos de los que me gustan
ETIQUETA TEMA:	tema, ocio y deporte, eventos, espectador
ETIQUETA INTENCIÓN:	intención, informar, plan, posible/no definitivo
ENTIDADES:	cantidades: más; afición: conciertos

La primera línea es la frase de entrada, la segunda corresponde a etiquetas de entidades, de la cual no nos preocuparemos. Por último, tenemos la frase seguida de sus etiquetas correspondientes en este orden: tema, intención y polaridad.

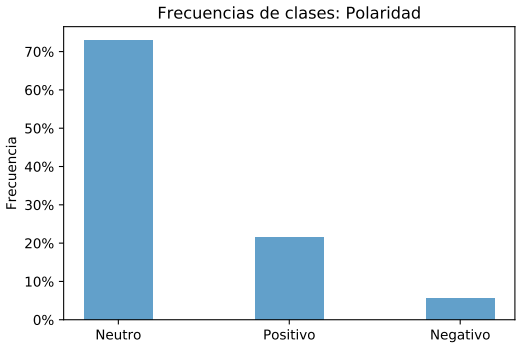
Ya que desarrollaremos un clasificador de clases, nos interesa saber con qué frecuencia aparecen las clases en el corpus, ya que durante el entrenamiento es muy común que la red se acostumbre a predecir solamente la clase dominante. En un principio pensábamos hacer que el clasificador sólo etiquetara los nodos principales de los diagramas (es decir, en temas solo etiquetar si se habla de deporte, nutrición, familiares u otro), por lo que calculamos con que frecuencia aparecían cada clase, y de ahí obtenemos los gráficos 5.2.



(a)



(b)



(c)

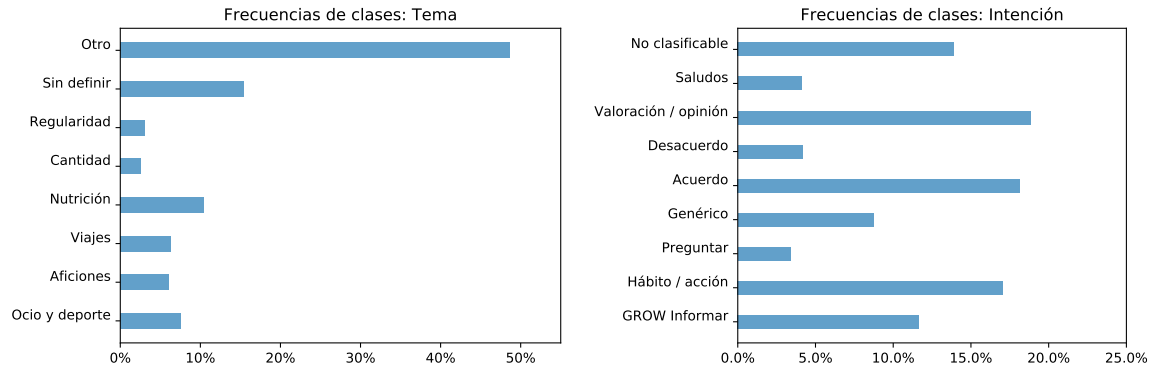
Figura 5.2: (a) En las etiquetas relacionadas con el tema la clase dominante es claramente Otro. (b) Relacionadas con la intención sería la clase Genérico. (c) Y en la polaridad la clase Neutro.

Observando estos gráficos la primera conclusión que obtenemos es que la clase *Familiares y Cuidadores* aparece con tan poca frecuencia que no merece la pena tenerla en cuenta, por lo que no la consideraremos en el futuro. Por otro lado, la clase *Otro* predomina con un 70 % de frecuencia, observando por encima el corpus nos dimos cuenta de que una gran parte de las frases son simplemente respuestas 'si/no' o 'muy bien' y por ello eran etiquetadas con *Otro*. Por lo que hemos hecho un filtrado del corpus, y a las frases etiquetadas como *Otro* y que tengan una longitud relativamente pequeña (unos 4 caracteres) los hemos etiquetado como *Sin definir*, ya que no existe un tema definible si solo se trata de una respuesta corta. Haciendo este procedimiento hemos separado el 70 % de *Otro* en un 18 % *Sin definir* y 52 % *Otro*.

Sin embargo, si utilizáramos solo las clases mostradas arriba, las clases dominantes al ser tan frecuentes harían que la red solo predijera dichas clases, por lo que decidimos en aumentar el número de clases a considerar, para así tener mejor balance entre las clases. El criterio que seguimos fue bajar un nivel más en los diagramas de etiquetado (véase el Apéndice A) y solamente considerar las clases que tengan una frecuencia superior al 2 %.
Con nuestro criterio finalmente éstas serán las clases que consideraremos[2]:

- Temas: Ocio y deporte, Aficiones, Viajes, Nutrición, Cantidad, Regularidad, Sin definir, Otro.
- Intención: GROW informar, Hábito / acción, Preguntar, Genérico, Acuerdo, Desacuerdo, Valoración / opinión, Saludos, No clasificable.

[2] Ahora las clases superiores consideradas anteriormente representarán el resto de sus subclases que no hemos considerado, como por ejemplo *Nutrición* será la representación de la clase descartada *Variedad*.



(a) Tema (b) Intención

Figura 5.3: Las frecuencias de las clases que tendremos en cuenta. Como en polaridad solo existen 3 clases, no cambia nada con respecto al gráfico en 5.2. Obsérvese que aún así la clase *Otro* sigue siendo al rededor del 50 % de las clases en *Tema*.

	Tema	Intención	Polaridad
M	8	9	3

Cuadro 5.1: Ámbitos de etiquetado y la dimensión M de sus clasificadores, es decir, su número de clases.

5.3. Cómo evaluar los resultados (F1 score)

Al tratar con clasificadores, debemos familiarizarnos con el concepto de la matriz de confusión, la cual es una herramienta que permite la visualización del desempeño de un clasificador. Cada columna de la matriz representa el número de predicciones de cada clase, mientras que cada fila representa a las instancias en la clase real. Mostramos a continuación dos ejemplos, los cuales utilizaremos para discutir el mejor método para evaluar si un clasificador es eficiente o no.

Real\Predicho	A	B	C
A	700	50	5
B	50	40	0
C	20	10	5

(a)

Real\Predicho	A	B	C
A	560	100	95
B	15	70	5
C	5	0	30

(b)

Cuadro 5.2: Ejemplos de dos matrices de confusión, estos datos no son resultados reales, simplemente los utilizaremos para explicar la manera óptima de analizar un clasificador.

Lo primero que se nos podría venir a la cabeza es determinar la viabilidad con la exactitud, la cual se calcula simplemente dividiendo el número total de aciertos entre el número total de predicciones que se ha hecho. Si representamos la matriz de confusión como C, esto vendría dado por[3]:

[3]Donde $sum(C)$ se refiere a la suma de todos los elementos de la matriz C, y $Tr(C)$ es la suma de los elementos diagonales de la matriz.

$$Exactitud = \frac{Tr(C)}{sum(C)} \equiv E \tag{5.1}$$

Con esta definición obtenemos para nuestros ejemplos $E_1 = 84\,\%$ *y* $E_2 = 75\,\%$, por lo que podríamos concluir que el modelo correspondiente al Cuadro 5.2a es el más adecuado. Sin embargo, este criterio es bastante engañoso en el caso de que nuestros datos no estén balanceados, es decir, como hemos visto en la Figura 5.2 tenemos una enorme cantidad de etiquetas con la clase A, por lo que es normal que los modelos tiendan a predecir esta clase. Aun así, el hecho de que prediga tantas veces la clase A hace que la exactitud sea un mal criterio a la hora de evaluar los modelos, ya que si por ejemplo, nuestro modelo solamente predijera la clase dominante (A), al ser tan frecuente nos parecería que el modelo trabaja bien porque tendrá una gran exactitud.

Nuestro interés es que el modelo sea capaz de predecir todas las clases, independientemente de lo frecuentes que sean. Para ello, existe otro criterio llamado el *F1 Score* [20]. Se basa en calcular la precisión y exhaustividad (*Recall*) de cada clase para después hacer la media, y por último calcular el *Score* total del clasificador. Existen varias maneras de calcular la media, en nuestro caso trabajaremos con el *F1 Macro Score*.

Para calcular la precisión y exhaustividad de cada clase se hace una distinción binaria, es decir, se considera por ejemplo la clase A como clase O y el resto de clases serán una sola clase $\overline{O}$. De esta forma utilizaremos los términos falso positivo, falso negativo, verdadero positivo y verdadero negativo como en un clasificador binario.

	O	$\overline{O}$
O	VP	FN
$\overline{O}$	FP	VN

Cuadro 5.3: Matriz de confusión de la clasificación binaria. Donde las abreviaciones son : falso positivo(FP), falso negativo(FN), verdadero positivo(VP) y verdadero negativo(VN).

En el caso binario la precisión y exhaustividad se definen así:

$$\begin{aligned} Precision &= \frac{VP}{VP + FP} \equiv P \\ Exhaustividad(Recall) &= \frac{VP}{VP + FN} \equiv R \end{aligned} \tag{5.2}$$

En cambio, si generalizamos esto a un problema de multiclases, entonces P y R para cierta clase se calculan teniendo en cuenta que VP es el valor diagonal de dicha clase, mientras que FP será la suma de de los elementos de la columna (excluyendo la diagonal) y FN la suma de los elementos de la fila (excluyendo también la diagonal). Usamos el ejemplo del Cuadro 5.2b para enseñar cuales serían estos valores en el caso de la clase B[4]:

[4]En este ejemplo utilizaremos una indexación que comenzará desde el 1, por lo que la clase B será la clase 2.

Real\Predicho	A	B	C
A	560	100	95
B	15	70	5
C	5	0	30

Cuadro 5.4: En esta matriz de confusión mostramos para la clase A su verdadero positivo (Azul), falso positivo (Verde) y falso negativo (Rojo). Con ello obtenemos los valores: $VP_2 = 70$, $FP_2 = 100$, $FN_2 = 20$.

Teniendo los elementos de matriz C_{ij}, siendo j el indice de cada columna e i el de cada fila, podemos generalizar las ecuaciones 5.2 para cada clase[5]:

$$\begin{aligned} VP_k &= C_{kk} \\ FP_k &= \sum_{j \neq k} C_{kj} \\ FN_k &= \sum_{i \neq k} C_{ik} \end{aligned} \tag{5.3}$$

$$\begin{aligned} P_k &= \frac{VP_k}{VP_k + FP_k} \\ R_k &= \frac{VP_k}{VP_k + FN_k} \end{aligned} \tag{5.4}$$

Por último, como hemos mencionado anteriormente, se calcula la P y R de todo el clasificador haciendo la media (siendo M el número de clases), y el *F1 Macro Score* es la media armónica de P y R:

$$\begin{aligned} P &= \frac{1}{M} \sum_{k=1}^{M} P_k \\ R &= \frac{1}{M} \sum_{k=1}^{M} R_k \\ F1 &= \left(\frac{P^{-1} + R^{-1}}{2} \right)^{-1} = 2 \frac{P \cdot R}{P + R} \end{aligned} \tag{5.5}$$

Ahora que sabemos cómo calcular el *F1 Macro Score*, podemos comparar los resultados de los dos ejemplos. Usando las ecuaciones 5.4 y 5.5 obtenemos $F1_1 = 52\,\%$ y $F1_2 = 58\,\%$, es decir, el ejemplo 2 muestra un mejor rendimiento según este criterio. Ésto tiene sentido ya que si comparamos los primeros cuadros 5.2a y 5.2b, vemos que aunque el ejemplo 1 tenga una exactitud mayor, el ejemplo 2 ha predicho mucho mejor las clases minoritarias B y C. La gran exactitud de 2 se debe a que predice muchas veces la clase dominante, lo cual no es un resultado interesante si a costa de ello se ignoran las clases minoritarias.

Con este ejemplo concluimos que para un *dataset* muy desequilibrado, como el nuestro, es más conveniente utilizar el *F1 Macro Score* para determinar la eficiencia de los clasificadores que vayamos a desarrollar.

[5]En cada suma i y j van de 1 a M, siendo M el número de clases.

Capítulo 6

Modelo de red

Una vez establecido el corpus, podemos desarrollar la estructura de la red. En este capítulo describiremos la arquitectura de la red, resumiremos la funcionalidad de las diferentes redes principales que utilizaremos, ya que probaremos la misma arquitectura pero utilizando tres tipos de redes, después expondremos la forma de los datos de entrada y de salida, para terminar hablando sobre la partición de los datos del corpus.

6.1. Arquitectura de las redes

La arquitectura que hemos escogido para nuestro modelo de red se basa en la presentada en la referencia [21]. Al tratarse de una estructura bastante sencilla y sin ninguna funcionalidad especial, lo hemos desarrollado con *Keras* [22], ya que, como hemos mencionado en 2.3, no será necesario crear unas celdas con propiedades concretas, por lo que *Keras* puede ser una mejor opción que *Pytorch*. .

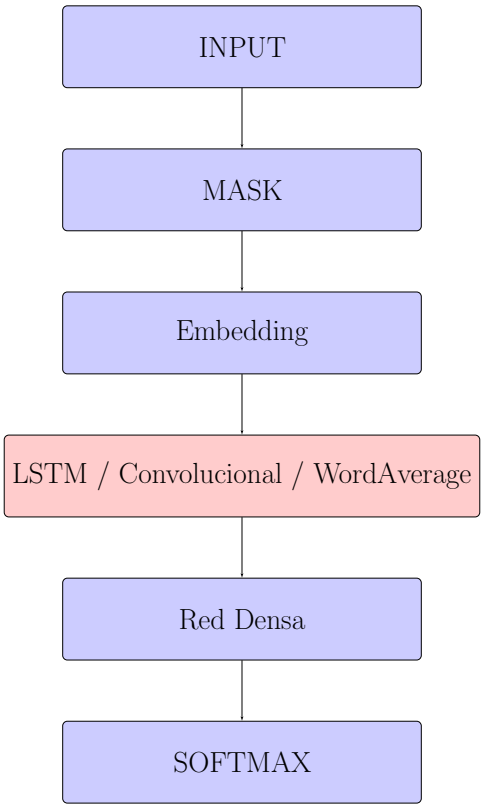


Figura 6.1: Arquitectura del modelo de red neuronal.

Aquí mostramos los parámetros y funciones de cada capa de la red:

1. INPUT: Toma la entrada que constará de una serie de números enteros, los cuales representarán el índice que tiene cierta palabra en nuestro diccionario. Si nuestro diccionario tiene N palabras, entonces las palabras que no se encuentran dentro del diccionario se les asignará el número $N + 1$.

2. MASK: Ignora todo índice nulo. Esto sirve para poder trabajar con frases de longitud variable.

3. Embedding: Teniendo la matriz que relaciona cada índice con su correspondiente *wordvector* devolverá la representación de cada palabra de entrada.

4. Red variable: Utilizaremos tres celdas distintas, y compararemos los resultados que obtenemos de cada uno de ellos.

5. Red Densa: Una red densa estándar que hará un segundo procesamiento con la tangente hiperbólica como función de activación.

6. SOFTMAX: Por último, como nuestro objetivo es clasificar, la salida será un vector de dimensión M, siendo M la cantidad de clases que queremos clasificar, y cada coeficiente del vector vendrá dada por la función *softmax*, por lo que la salida representará la distribución de probabilidad de la clase a la que pertenezca la frase de entrada.

En la cuarta capa utilizaremos las siguientes redes:

1. **LSTM**: Se tratará de un red LSTM(3.2) con la tangente hiperbólica como función de salida, y como trataremos con pocos datos no necesitaremos demasiados parámetros y estableceremos que devuelva de salida un vector de dimensión 10. Teniendo como entrada vectores de dimensión 300 esto resulta en 12.440 parámetros entrenables de la red LSTM.

2. **Convolucional**: Utilizaremos un red convolucional, aunque será una *Conv1D*, es decir, hará la convolución a cada frase entera siendo ésta un dato unidimensional en contraste con las imágenes que son bidimensionales. Estableceremos que haga la convolución por cada dos palabras y que tenga una salida de dimensión 10, además en ésta arquitectura añadiremos un *GlobalMaxPooling1D* el cual se encargará de deshacer la parte temporal(la longitud de la frase) dando en cada salida solamente el valor máximo obtenido, con estas especificaciones obtenemos unos 6.000 parámetros entrenables, bastante menos que en el caso de la LSTM, con ello esperamos que el entrenamiento se lleve a cabo más rápidamente.

3. **WordAverage**: En este caso usaremos una red que hará la media de los wordvector de las palabras para obtener una representación de la frase, es decir, trabajaremos a nivel de frases esta vez, pero obtendremos dicha representación a partir de los wordvectors. Esta metodología se analiza en detalle en [23].

6.2. Datos de entrenamiento

Los datos para entrenar esta red constarán de una serie de entradas y salidas que tendrán esta forma:

1. **Entrada**: Como hemos mencionado anteriormente, de entrada tendremos una seguida de números enteros que representarán la frase, para que con estos índices la red relacione el *wordvector* correspondiente *wordvector*[1]. Podemos ver un sencillo ejemplo:

Frase en el corpus	**Entrada como secuencia de índices**
Hola PERSON . ¿ Qué tal estás ?	[109 , 38 , 1 , 32 , 73 , 336 , 270 , 31]

   Sin embargo, la entrada no será exactamente igual que el ejemplo mostrado ahora mismo. Ya que utilizaremos el *padding* para poder hacer que en cada bloque de entrenamiento, todas las frases tengan la misma longitud. El *padding* simplemente se trata de añadir tantos 0-s como sean necesarios a cada seguida de entrada para que tengan todas la misma longitud, siendo ésta la longitud máxima de entre todas las frases del bloque. Todos esos 0-s serán ignorados gracias a la capa *MASK*.

2. **Salida**: La salida será la distribución de probabilidades de que la frase pertenezca a cada clase, por lo que el *output* que utilizaremos para entrenar la red serán vectores *one-hot*[2]. Siendo la salida de la red una *softmax*, el error más conveniente a optimizar es la *Categorical crossentropy*. Esta función de error se define de esta forma:

$$f(\mathbf{y}, \mathbf{a}) = -\sum_{i=1}^{N} y_i \log(a_i) \tag{6.1}$$

   Siendo $\mathbf{y}$ el vector (de dimensión N) de salida deseado y $\mathbf{a}$ la salida una vez inferida la entrada. Al tratarse de vectores *one-hot*, la ecuación 6.1 se simplifica en $-\log(a_j)$, siendo j el índice en el que el vector $\mathbf{y}$ no es nulo. Gracias al signo negativo nos aseguramos de que la función sea siempre mayor que 0, ya que al tratarse de la salida de una *softmax* consolidamos que a_j esté acotado entre 0 y 1, por tanto la función de error tomará valores entre 0 e infinito, por lo que minimizando el error conseguiremos que a_j se acerque al valor 1 que es precisamente la salida que deseamos.

Además de eso, en la sección 5.2 hemos visto como hay un desequilibrio importante en cuestión de la cantidad de datos que tenemos para las diferentes clases. Sin embargo, habiendo escogido unas cuantas clases más como hemos hecho al final de dicha sección las frecuencias se han balanceado un poco más. Aun así, podría suceder que la red se acostumbrara a predecir solo las clases predominantes, para evitar esto podemos penalizar las clases mayoritarias multiplicando en la función de error a cada término la frecuencia de la correspondiente clase (f_i). La ecuación 6.1 quedaría así:

$$f(\mathbf{y}, \mathbf{a}) = -\sum_{i=1}^{N} f_i y_i \log(a_i) \; ; \; \sum_{i=1}^{N} f_i = 1 \tag{6.2}$$

[1] Los *wordvectors* que utilizaremos los hemos obtenido con la herramienta *fastText*, la cual tomando como entrada todas la frases del corpus entrena unos *wordvectors* pre-entrenados, siendo éstos de dimensión 300. Para evitar ruido hemos ignorado palabras que aparezcan menos de 2 veces.

[2] Los vectores *one-hot* son simplemente vectores de N dimensiones con $N-1$ coeficientes nulos y uno solo con valor 1.

6.3. *Train, validation* y *test*

Como hemos mencionado en en la sección 4.3 separaremos nuestro grupos de datos más pequeños. Más concretamente, partiremos los datos en un 70 % para el entrenamiento, 10 % para la validación y 20 % para el testeo. Construiremos estas particiones tomando los datos originales, los cuales estarán guardados en una tabla y serán accesibles usando el índice de la fila en la que se encuentra el dato. Crearemos una lista con todos los índices en orden, los desordenaremos aleatoriamente y después separaremos la lista en tres listas con las proporciones mencionadas anteriormente. Esto resulta en unas 5.300 frases de entrenamiento, 700 de validación y 1.500 de testeo.

Los datos de entrenamiento y de validación se utilizarán durante el entrenamiento de cada modelo, para poder detectar si sucede el *overfitting*. Después se utilizarán los datos de validación para optimizar la métrica[3], con el método de búsqueda local explicada en la sección 4.4. Por último, los datos de testeo servirán para comparar la eficiencia de los modelos con el F1 Score, al tratarse éstos de frases que la red nunca ha 'visto' serán los más apropiados para examinar los modelos.

Dentro del método *Cross validation*, la validación que utilizaremos será la aleatoria. Sin embargo, existen otras maneras de validar los datos, una de ellas es el *Leave one out*. Éste consiste en entrenar varias redes, en las cuales se utilizarán los datos del corpus pero en cada uno el grupo de entrenamiento constará de todos los datos menos un dato, el cual se utilizará como validación, este último dato será distinto para cada red[4]. De esta forma nos aseguramos de que todos los datos que poseemos serán utilizados en el entrenamiento, y una vez entrenados escogeremos el más óptimo.

Como es de esperar, éste método consume demasiado tiempo ya que se entrenan varias redes en vez de una, por ese motivo hemos escogido utilizar el método de Validación cruzada aleatoria descrita al principio de esta sección.

[3] *F1 Macro Score.*

[4] Una generalización de este método es la Validación cruzada de K iteraciones, en la cual se procede de la misma forma pero esta vez se toman K datos como muestra de validación.

Capítulo 7

Resultados

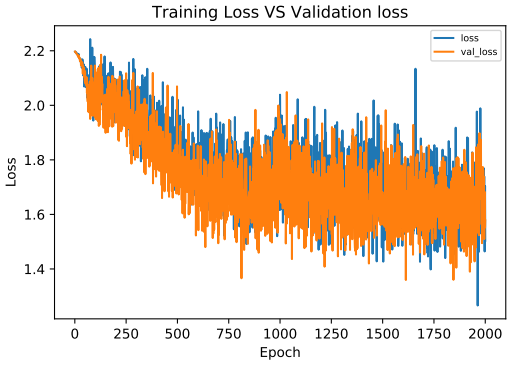
En este capítulo expondremos los resultados obtenidos durante el trabajo, primero compararemos los *F1 Macro Score* obtenidas por cada red (LSTM, Convolucional y *WordAverage*) y después tomaremos la red con el mejor rendimiento e intentaremos mejorarlo aún más, probando distintos métodos de optimización como los optimizadores, reguladores o el *dropout*.

7.1. Comparación de las redes

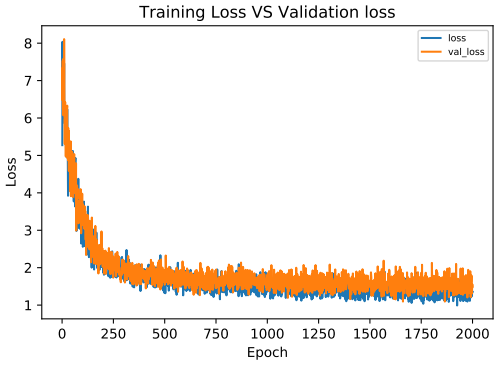
En esta sección hemos tomado cada red y hemos utilizado el optimizador Adam con cada uno de ellos, no hemos utilizado reguladores (4.3) ni *dropout*, excepto en la LSTM en la que hemos establecido *dropout*= 0,2. La red densa (6.1) tiene una salida de dimensión 30 para cada red, y como hemos mencionado anteriormente la *softmax* tendrá una dimensión variable M que será el número de clases que se van a predecir. Hemos iniciado los parámetros de estas últimas dos capas utilizando una distribución aleatoria Gaussiana, con una desviación estándar $\sigma = 1$.

Durante el entrenamiento hemos utilizado *minibatches* de tamaño 54 para el set de entrenamiento y de 13 para el de validación, ya que disponemos de pocos datos. Por ello las redes también tienen una cantidad de parámetros entrenables relativamente pequeña, teniendo la LSTM, Convolucional y el *WordAverage* un total de 12.539, 6.433 y 9.030 parámetros entrenables respectivamente. Habiendo probado antes cuánto tiempo le tomaba a cada red un entrenamiento de 2.000 épocas (iteraciones), hemos visto que duran 2 minutos (LSTM), 40 segundos (Convolucional) y 30 segundos (*WordAverage*) en cada entrenamiento. Se ve claramente que la LSTM necesita mucho más tiempo ya que tiene muchos parámetros y es una celda mucho más compleja que las demás.

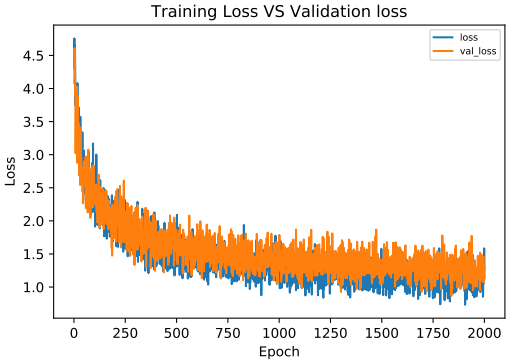
Una vez entrenadas las redes, hemos tomado algunos gráficos en las que se representan la función de error de entrenamiento y de validación para compararlas:



(a) LSTM



(b) Convolucional



(c) *WordAverage*

Figura 7.1: Función de error de entrenamiento y validación para los clasificadores de *Intención* en función de las épocas.

Como vemos estos gráficos se asemejan a la Figura 4.2, aunque se observa un ruido bastante pronunciado. Esto se puede deber a que entrenamos en lotes muy pequeños, por lo que en cada época de entrenamiento al ver muchos datos 'no antes vistos' el error aumenta mucho respecto a la iteración anterior o que sucede todo lo contrario y el error baja considerablemente, dando una figura muy ruidosa. Otra gran diferencia importante que notamos respecto a la Figura 4.2 es que en las 2.000 épocas de entrenamiento no sucede el fenómeno de *overfitting*, por lo que no necesitamos utilizar métodos para evitarlo como el *early stopping*. Por otro lado, vemos que (b) y (c) llegan a disminuir el error muy rápidamente en las primeras 500 épocas, aunque esto no sucede en el caso (a) en el que vemos que la función es prácticamente ruido, por lo que esperamos que la LSTM no tendrá un buen rendimiento al etiquetar las *Intenciones*.

Una vez visto como se ha desarrollado el entrenamiento, mostraremos el rendimiento de cada red comparando algunas tablas de confusión. Debemos aclarar que las tablas que mostramos a continuación están calculadas después de la optimización de búsqueda local explicada en la sección 4.4.

Real\Predicho	Ocio y deporte	Aficiones	Viajes	Nutrición	Cantidad	Regularidad	Sin definir	Otro
Ocio y deporte	12	39	10	12	4	2	1	26
Aficiones	12	36	10	8	3	3	0	24
Viajes	7	44	9	9	3	3	0	17
Nutrición	9	46	13	23	15	3	0	26
Cantidad	3	14	5	8	2	1	0	10
Regularidad	2	20	5	11	3	0	0	8
Sin definir	0	0	0	0	1	0	220	23
Otro	21	104	45	38	18	7	5	512

Cuadro 7.1: Tabla de confusión de la red Convolucional, clasificando las etiquetas de *Tema*.

Real\Predicho	Ocio y deporte	Aficiones	Viajes	Nutrición	Cantidad	Regularidad	Sin definir	Otro
Ocio y deporte	17	17	4	8	2	6	0	52
Aficiones	8	46	2	4	1	3	0	32
Viajes	4	18	24	9	0	1	0	36
Nutrición	4	6	0	60	9	20	0	36
Cantidad	0	1	0	21	7	5	0	9
Regularidad	1	2	0	16	4	22	0	4
Sin definir	3	1	0	1	0	0	205	34
Otro	8	36	13	26	12	18	8	629

Cuadro 7.2: Tabla de confusión de la red *WordAverage*, clasificando las etiquetas de *Tema*.

Vemos en estas tablas una clara diferencia entre ellas, y es que en 7.2 hay una mejor clasificación ya que los elementos diagonales son mayores, dando una exactitud de 53 % y 66 % (ver Cuadro 7.4) para el Convolucional y *WordAverage* respectivamente. Pero como hemos explicado en la sección 5.3 la métrica más apropiada para evaluar un clasificador con varias clases y estando éstas muy desequilibradas es el *F1 Macro Score*. Calculando dicha métrica resulta que el *WordAverage* sigue dando un mejor resultado de 45 % frente al 28 % del Convolucional, esto se puede entender por ejemplo fijándonos en que la red convolucional predice muchas veces la clase *Ocio y deporte* erróneamente, además no predice correctamente ninguna vez la clase *Regularidad*, esto hace que en la media el *F1 Score* se reduzca mucho y de ahí la diferencia con el modelo *WordAverage*.

Real\Predicho	Neutro	Positivo	Negativo
Neutro	589	381	0
Positivo	163	321	0
Negativo	42	19	0

(a) *WordAverage*

Real\Predicho	Neutro	Positivo	Negativo
Neutro	0	869	101
Positivo	0	410	74
Negativo	0	45	16

(b) LSTM

Cuadro 7.3: Tabla de confusión de las redes *WordAverage* (a) y LSTM (b), clasificando las etiquetas de *Polaridad*.

En este caso vemos que la red *WordAverage* se ha inclinado a predecir solamente las clases dominantes, olvidándose completamente de la clase *Negativo*. Este comportamiento es algo esperado al tener unas clases tan desbalanceadas, sin embargo, con el LSTM sucede algo muy curioso, y es que no predice ni una sola vez la clase dominante. Ésto se puede deber a que el vector de optimización $\boldsymbol{\delta}$ (4.4) ha disminuido tanto el valor de la probabilidad de la etiqueta *Neutro* que ha pasado a ser ignorada, ya que recordemos que el vector $\boldsymbol{\delta}$ básicamente hace un *shift* a la probabilidad de cada etiqueta para intentar mejorar el resultado. Con ello obtenemos un *F1 Score* del 40 % para (a) y 19 % para (b).

Por último, en esta sección compararemos los distintos *F1 Scores*, al igual que las exactitudes de todas las redes que hemos entrenado. En las tablas siguientes distinguiremos cuatro datos: primero mostraremos la exactitud antes de optimizar (E), después la exactitud tras la optmización ($E(\boldsymbol{\delta})$), luego el *F1 Score* antes de optimizar (F_1) y por último el *F1 Score* tras usar el algoritmo de búsqueda local ($F_1(\boldsymbol{\delta})$).

	Intención		
	LSTM	*Convolucional*	*WordAverage*
E	37 %	48 %	57 %
$E(\boldsymbol{\delta})$	39 %	46 %	57 %
F_1	20 %	50 %	**60** %
$F_1(\boldsymbol{\delta})$	27 %	52 %	**59** %

	Tema		
	LSTM	*Convolucional*	*WordAverage*
E	50 %	62 %	66 %
$E(\boldsymbol{\delta})$	49 %	53 %	66 %
F_1	10 %	23 %	42 %
$F_1(\boldsymbol{\delta})$	26 %	28 %	45 %

	Polaridad		
	LSTM	*Convolucional*	*WordAverage*
E	64 %	62 %	64 %
$E(\boldsymbol{\delta})$	28 %	54 %	60 %
F_1	26 %	36 %	40 %
$F_1(\boldsymbol{\delta})$	19 %	41 %	40 %

Cuadro 7.4: Tablas de confusión de cada ámbito de etiquetado, en el que se muestra la Exactitud y el *F1 Score* de la red sin mejorar y mejorada para los tres tipos de modelos utilizados.

Claramente el modelo basado en *WordAverage* tiene un mayor rendimiento en los tres ámbitos de clasificación, siendo el de *Intención* el más óptimo de todos, ésto se debe a que las clases están más equilibradas que en *Tema* y *Polaridad*, como hemos podido observar en la Figura 5.3. Por lo que éste será la red que intentaremos mejorar con los métodos presentados en el capítulo 4, sobre ello trataremos en la siguiente sección. Por último, vemos que la red LSTM es la peor de todas en todos los ámbitos, ésto se puede deber a que el modelo tenga demasiados parámetros entrenables para los pocos datos de entrenamiento que poseemos, no llegando éstos a ajustarse demasiado bien para obtener un resultado satisfactorio.

Por otra parte, hay un fenómeno curioso que *a priori* parecería que no tiene sentido, y es que en la tabla de *Polaridad* la red LSTM 'mejorada' tiene un *F1 Score* más que pequeño con respecto a la red sin mejorar. ¿Cómo puede suceder esto? ¿Al haber realizado el algoritmo de búsqueda local no debería el *F1 Score* aumentar? Pues bien, esto simplemente se debe a que para optimizar el *F1 Score* hemos utilizado el set de validación (un 10 % de los datos), pero para comparar los resultados de cada red hemos utilizado el set de testeo (20 %), es decir, los datos que las redes nunca han 'visto' durante el entrenamiento. Por lo que los vectores $\boldsymbol{\delta}$ han sido calculados para optimizar un resultado que venía de los datos de validación, y la mejora de dichos resultados no tiene por que asegurar que con otro set de datos (el de testeo) fuera a existir una mejora. De hecho, este fenómeno también es ligeramente apreciable en el caso de la red más óptima (marcado en negrita).

7.2. Optimización de la red

Para optimizar la red clasificadora de *Intención* con el modelo *WordAverage* probamos añadiendo regularizadores a los pesos como el L1 y L2 (4.3), y por otro lado también comprobamos si había alguna mejora al utilizar el *Nesterov momentum* (4.2) en vez de Adam a la hora de actualizar los parámetros.

A los pesos (controlado por λ siguiendo la notación de la ecuación 4.7) y umbrales (μ) de las capas densa y *softmax*, serán a los que les aplicaremos los reguladores L1 y L2. Y los hiperparámetros del optimizador *Nesterov* son el ratio de aprendizaje ϵ y el parámetro de momento α. En todos los casos hemos entrenado la red durante 3.000 épocas, ya que al introducir un término más en la función de error como en la ecuación 4.7 consume más tiempo reducirla. En la tabla se especifica si se ha utilizado un regularizador o el optimizador *Nesterov* y los valores de los hiperparámetros están representados como : $L1(\lambda, \mu)$ y $N(\epsilon, \alpha)$.

	***WordAverage*: Intención**					
	$L1(10^{-2}, 10^{-2})$	$L1(0, 10^{-3})$	$L2(10^{-4}, 10^{-4})$	$N(10^{-1}, 10^{-3})$	$N(1, 10^{-3})$	$N(1, 10^{-3})\|L2(10^{-4}, 10^{-4})$
E	31 %	59 %	60 %	53 %	**60** %	62 %
$E(\boldsymbol{\delta})$	35 %	59 %	59 %	52 %	**61** %	62 %
F_1	11 %	62 %	64 %	55 %	**63** %	65 %
$F_1(\boldsymbol{\delta})$	28 %	62 %	63 %	54 %	**65** %	64 %

Cuadro 7.5: *F1 Score* para los distintos intentos de mejorar la red *WordAverage* al etiquetar *Intención*.

En el primer intento de utilizar el regularizador $L1$ los hiperparámetros eran tan grandes que los pesos contribuían demasiado a la función de error, haciendo así que durante el entrenamiento se centrara más en reducir la magnitud de dichos pesos que en minimizar el error, de ahí el mal resultado obtenido. En el segundo hemos visto una mejora significativa respecto al anterior intento, en este caso sólo hemos penalizado a los umbrales. Después hemos probado el $L2$ con unos hiperparámetros más pequeños que en el primer caso, ya que con $L2$ se consideran los cuadrados de los parámetros de la red (ecuación 4.7), por lo que pueden contribuir aún más a la función de error, habiendo obtenido un resultado un poco mejor que el anterior. Una vez probados los regularizadores hemos querido ver si obteníamos un mejor *Score* entrenando a la red con *Nesterov momentum*. En el primer caso hemos obtenido un resultado un poco peor que el conseguido con Adam, en cambio, en el segundo el *Score* es un poco mejor. Finalmente, hemos querido combinar las técnicas que han sido más eficaces, de ahí que hayamos probado *Nesterov* combinado con $L2$, sin embargo, el resultado ha sido exactamente el mismo que con *Nesterov*. Por lo que hemos decidido mostrar una última tabla de confusión en la que vemos el mejor desempeño obtenido:

Real\Predicho	GROW Informar	Hábito / acción	Preguntar	Genérico	Acuerdo	Desacuerdo	Valoración / opinión	Saludos	No clasificable
GROW Informar	45	54	0	7	2	1	38	1	13
Hábito / acción	15	173	0	4	5	1	18	1	26
Preguntar	0	0	53	0	0	0	0	0	0
Genérico	10	11	4	58	4	1	24	0	26
Acuerdo	0	5	0	3	241	0	14	0	7
Desacuerdo	3	1	0	3	1	51	2	0	0
Valoración / opinión	27	37	1	11	21	3	177	3	21
Saludos	0	1	1	1	1	0	3	67	2
No clasificable	28	39	3	23	14	3	29	4	69

Cuadro 7.6: Tabla de confusión con el mejor *F1 Score* (65 %) obtenido. Modelo *WordAverage* con optimizador *Nesterov*(1,10^{-3}).

Capítulo 8

Conclusiones

Aunque no se haya mencionado demasiado durante el trabajo, el paquete *Keras* ha sido totalmente útil para el desarrollo de éste. La variedad de capas y funciones que ofrece han sido de gran ayuda, ya que como habíamos dicho anteriormente, al no necesitar ninguna capa con una funcionalidad fuera de lo normal, *Keras* cubre prácticamente todo tipo de capas estándar. Por otro lado, su base *TensorFlow* permite paralelizar los cálculos, con lo cual hemos podido entrenar las redes sin necesidad de utilizar un superordenador, ya que nuestras redes poseían no muchos parámetros.

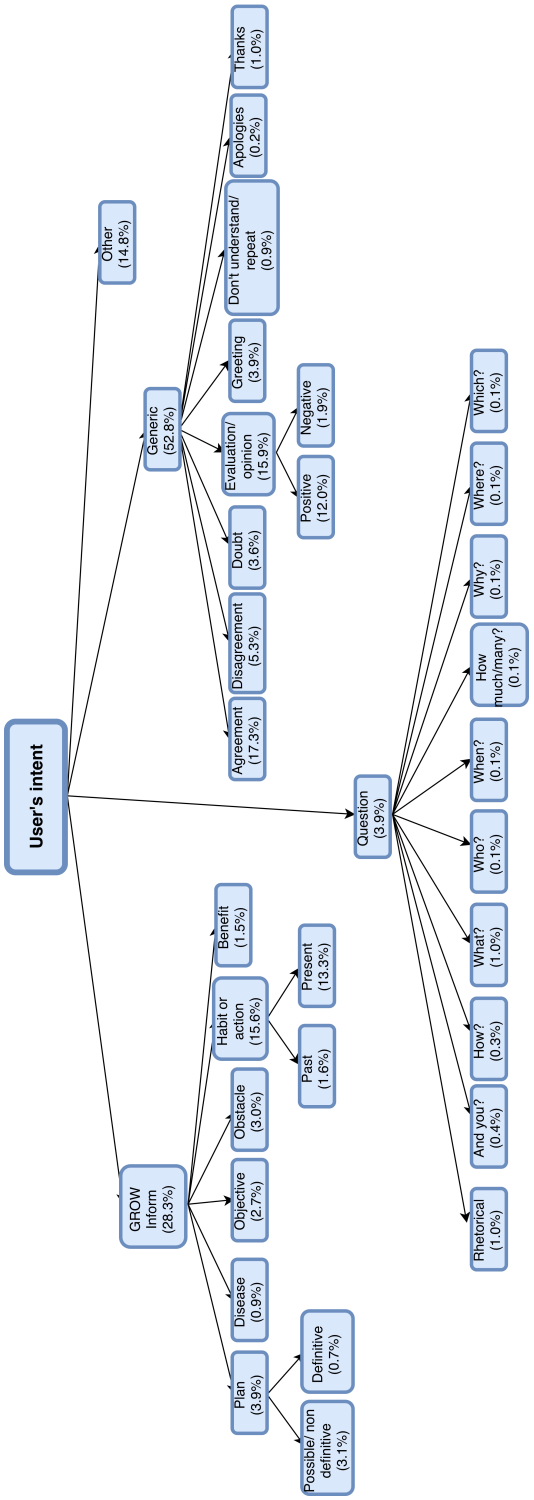
Los resultados muestran un hecho bastante sorprendente, y es que el simple hecho de hacer la media de los *Wordvectors* ha funcionado mejor que una red basada en una capa Convolucional o LSTM, esto se puede deber a la sencillez de la arquitectura propuesta. Por otro lado, hemos visto que el desequilibrio en la cantidad de cada etiqueta es determinante para el funcionamiento óptimo de un clasificador, ya que en el ámbito de *Intención* se han obtenido los mejores resultados y era claramente el más equilibrado en cuanto a frecuencias se refiere. En el ámbito de *Tema* el mayor problema venía de la etiqueta *Otro*, por que a parte de ser mayoritario era una etiqueta que no aporta mucho al entendimiento de la frase, al tratarse de un etiqueta muy general.

Si observamos la última tabla de confusión 7.6, vemos que la red ha aprendido a distinguir prácticamente a la perfección las preguntas, es decir, que ha aprendido que los caracteres '?¿' determinan las preguntas, algo bastante evidente para nosotros pero que es deseable que la red aprenda. Además, también distingue razonablemente bien el *Acuerdo* y *Desacuerdo*, esto es de suma importancia ya que en el proyecto *EMPATHIC* al estar tratando con personas reales es crucial entender cuándo hay acuerdo o desacuerdo por parte del usuario. Lo mismo sucede con los *Saludos* prendiciéndolos bastante bien, al contrario que con *No clasificable*, la cual vemos que es de las que peor se predicen, aunque también hay que comprender que esta etiqueta es muy ambigua y difícil de determinar incluso para un humano.

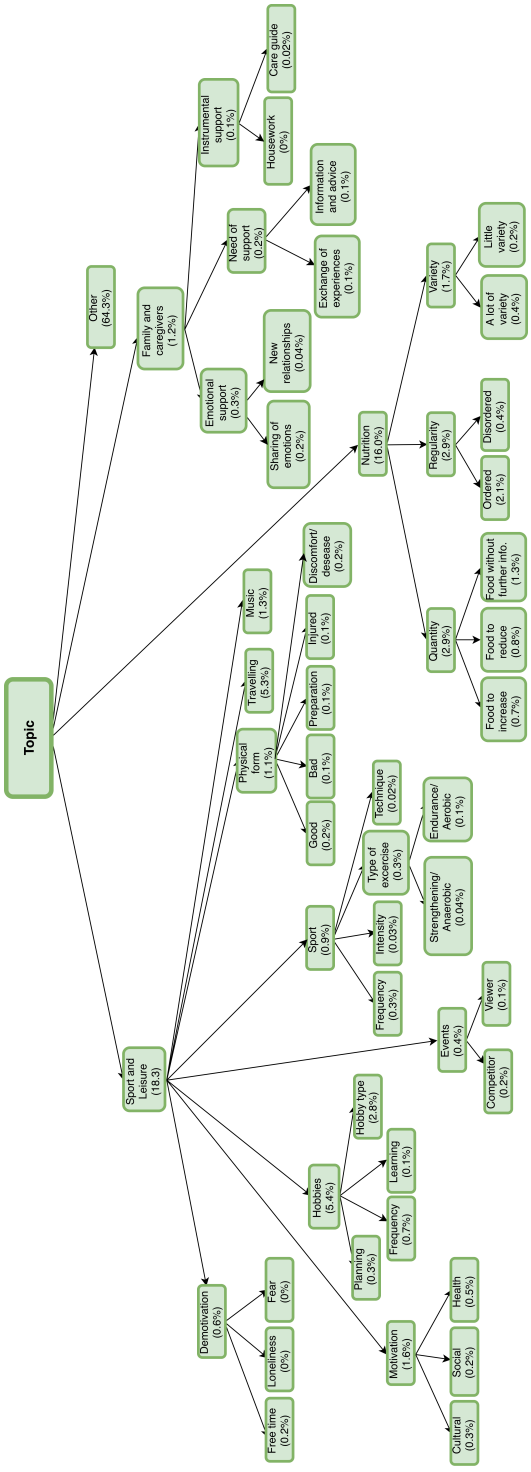
Por otra parte, proponemos algunas mejoras a realizar para obtener mejores resultados aún. Primero se podría descartar la etiqueta *Neutro* en el ámbito de *Polaridad*, y desarrollar un clasificador binario con unos datos de entrenamiento más equilibrados. Así mismo se podría implementar el *Leave-one-out* (6.3) en el proceso de entrenamiento, para de esta forma asegurar que se utilizan todos los datos que poseemos durante el entrenamiento. Este método no lo hemos llevado a cabo por cuestión de que consumiría demasiado tiempo. Igualmente, se podría experimentar con muchos más valores para los hiperparámetros($\epsilon, \lambda, \alpha$...) que hemos utilizado durante el trabajo, y así poder determinar el set de hiperparámetros que mejor se ajuste a este problema de clasificación.

En este trabajo hemos aprendido las bases del funcionamiento de las redes neuronales, cómo se construyen y los métodos para entrenarlos. Al igual que la importancia de los *wordvectors*, sin los cuales una red no podría procesar el lenguaje. Asimismo hemos interiorizado una métrica diferente a la exactitud, que es el *F1 Score*, con el cual hemos sabido determinar lo eficaz que es un clasificador. Y por último, hemos podido observar cómo se trabaja dentro de un grupo de investigación, del cual sus miembros han supuesto un gran apoyo para nosotros, no solamente porque nos han aportado datos para el corpus, sino porque también proponían ideas y mejoras, las cuales han ayudado mucho en la realización de este trabajo.

Apéndice A



(a) Etiquetas de *Intención*



(b) Etiquetas de *Tema*

Bibliografía

[1] Mike Tyka Alexander Mordvintsev, Christopher Olah. Google deep dream. https://deepdreamgenerator.com/, 2015.

[2] Asier López-Zorrilla, Mikel de Velasco-Vázquez, Jon Irastorza Manso, Javier Mikel Olaso Fernández, Raquel Justo Blanco, and María Inés Torres. EMPATHIC: empathic, expressive, advanced virtual coach to improve independent healthy-life-years of the elderly. *Procesamiento del Lenguaje Natural*, 61:167–170, 2018.

[3] Steven Bird, Ewan Klein, and Edward Loper. *Natural Language Processing with Python*. O'Reilly Media, Inc., 1st edition, 2009.

[4] Christopher Manning, Mihai Surdeanu, John Bauer, Jenny Finkel, Steven Bethard, and David McClosky. The Stanford CoreNLP natural language processing toolkit. In *Proceedings of 52nd Annual Meeting of the Association for Computational Linguistics: System Demonstrations*, pages 55–60, Baltimore, Maryland, June 2014. Association for Computational Linguistics.

[5] Jenny Rose Finkel, Trond Grenager, and Christopher Manning. Incorporating non-local information into information extraction systems by gibbs sampling. In *Proceedings of the 43rd Annual Meeting on Association for Computational Linguistics*, ACL '05, pages 363–370, Stroudsburg, PA, USA, 2005. Association for Computational Linguistics.

[6] Pranoy Radhakrishnan. Introduction to recurrent neural network. https://towardsdatascience.com/introduction-to-recurrent-neural-network-27202c3945f3, 2017.

[7] Sepp Hochreiter and Jürgen Schmidhuber. Long short-term memory. *Neural computation*, 9(8):1735–1780, 1997.

[8] Klaus Greff, Rupesh Kumar Srivastava, Jan Koutník, Bas R. Steunebrink, and Jürgen Schmidhuber. Lstm: A search space odyssey. 2015.

[9] Denny Britz. Recurrent neural network tutorial, part 4 : Implementing a gru/lstm rnn with python and theano. http://www.wildml.com/2015/10/recurrent-neural-network-tutorial-part-4, 2015.

[10] Junyoung Chung, Caglar Gulcehre, KyungHyun Cho, and Yoshua Bengio. Empirical evaluation of gated recurrent neural networks on sequence modeling, 2014.

[11] Datascience, stackexchange. https://datascience.stackexchange.com/questions/23183/why-convolutions-always-use-odd-numbers-as-filter-size?answertab=oldest#tab-top, 2017.

[12] Yoon Kim. Convolutional neural networks for sentence classification, 2014.

[13] Ian Goodfellow, Yoshua Bengio, and Aaron Courville. *Deep Learning*. MIT Press, 2016. http://www.deeplearningbook.org.

[14] Conor McDonald. Machine learning fundamentals (i): Cost functions and gradient descent. https://towardsdatascience.com/machine-learning-fundamentals-via-linear-regression-41a5d11f5220, 2017.

[15] John Duchi, Elad Hazan, and Yoram Singer. Adaptive subgradient methods for online learning and stochastic optimization. *Journal of Machine Learning Research*, 12(Jul):2121–2159, 2011.

[16] Ilya Sutskever, James Martens, George Dahl, and Geoffrey Hinton. On the importance of initialization and momentum in deep learning. In *International conference on machine learning*, pages 1139–1147, 2013.

[17] Diederik P. Kingma and Jimmy Ba. Adam: A method for stochastic optimization, 2014.

[18] Konstantin. The mystery of early stopping. `http://fouryears.eu/2017/12/06/the-mystery-of-early-stopping/`, 2017.

[19] Javier Mikel Olaso Roberto Santana Raquel Justo Jose A. Lozano Cesar Montenegro, Asier López Zorrilla and María Inés Torres. A dialogue-act taxonomy for a virtual coach designed to improve the life of elderly. *Multimodal Technologies and Interactions*, Special Issue "Semantics of Multimodal Social Interaction", 2019.

[20] Nancy Chinchor. Muc-4 evaluation metrics. In *Proceedings of the 4th Conference on Message Understanding*, MUC4 '92, pages 22–29, Stroudsburg, PA, USA, 1992. Association for Computational Linguistics.

[21] Ronan Collobert and Jason Weston. A unified architecture for natural language processing: Deep neural networks with multitask learning. In *Proceedings of the 25th international conference on Machine learning*, pages 160–167. ACM, 2008.

[22] François Chollet et al. Keras. `https://keras.io`, 2015.

[23] Sanjeev Arora, Yingyu Liang, and Tengyu Ma. A simple but tough-to-beat baseline for sentence embeddings. 2016. `openreview.net`.