Universidad Zaragoza

MINERÍA DE TEXTOS

Trabajo de Fin de Máster

-Máster en Modelización e Investigación matemática, Estadística y Computación-

Universidad de Zaragoza

-Facultad de Ciencias-

Autora:
Rocío Aznar Gimeno

Directores:
Luis Mariano Esteban Escaño
Gerardo Sanz Sáiz
Jorge Vea-Murguía Merck

25 Noviembre 2016

Índice general

Motivación y objetivos

El trabajo fin de máster surge de una propuesta de beca por parte del Instituto Tecnológico de Aragón (*Itainnova*), de una duración de 6 meses.

Itainnova es un centro tecnológico de carácter público dependiente del Departamento de Innovación, Investigación y Universidad del Gobierno de Aragón, fundado en 1984. Su motivación es aportar a las empresas e instituciones servicios de investigación y desarrollo de innovación tecnológica.

El desarrollo tecnológico de los últimos años ha llevado a que el concepto de *Big Data* haya tenido cada vez más popularidad en el mundo actual. Esto es, el mundo de hoy crea y almacena información en grandes volúmenes de datos, gracias, en su mayor parte, al avance de las TICs. Itainnova participa en el reto del tratamiento de estos grandes volúmenes de datos (Big Data).

Para la explotación de estos datos, la modelización matemática y estadística, junto con técnicas de inteligencia artificial, juega un papel importante. Es decir, es necesario el uso de modelos matemáticos y técnicas estadísticas que permitan analizar los datos y poder así explotar la información de ellos.

Dentro de este marco se desarrolla el trabajo fin de máster, profundizándose en técnicas propias de lo que se denomina *Deeplearning* o aprendizaje profundo.

El objetivo general del trabajo fin de máster es diseñar un proceso que permita evaluar y categorizar un conjunto de textos, mediante técnicas propias de Minería de textos. En concreto, crear un modelo matemático que permita encontrar y clasificar las entidades nombradas (nombres propios) en un texto. Los modelos que se usan son redes neuronales, en particular, el perceptrón multicapa y las redes recurrentes LSTM (*Deeplearning*).

La motivación que me llevó a realizar este trabajo fin de máster está inspirada en tres factores principales:

- Estudiar técnicas de Machine Learning con grandes conjuntos de datos textuales. Es decir, adentrarme en la disciplina de Minería de textos, casi desconocida por mi en ese momento, emergente y con un gran impacto a día de hoy.
- Trabajar en un entorno de trabajo multidisciplinar.
- Aplicar las técnicas de Machine Learning con tecnologías nuevas y punteras.

La realización del trabajo fin de máster no podría haber sido posible sin la ayuda del personal de Itainnova con el que he trabajado durante este periodo.

Agradecer a Rosa Montañés por haber estado a mi lado sobre todo frente a la lucha contra Hamilton, a Enrique Meléndez por amenizarme los días y a mis tutores Jorge Vea-Murguía

y Rafael del Hoyo. Hacer mención especial también a mis tutores de la universidad Luis Mariano Esteban y Gerardo Sanz.

Capítulo 1

Introducción a la Minería de textos

" La idea detrás de los computadores digitales puede explicarse diciendo que estas máquinas están destinadas a llevar a cabo cualquier operación que pueda ser realizada por un equipo humano "

Alan Turing

En este primer capítulo se presenta una breve introducción a la Minería de textos, explicando conceptos básicos que serán necesarios para el entendimiento de los sucesivos capítulos. Algunos términos propios del campo que se mencionan a lo largo del trabajo son definidos en el Glosario al final del documento.

Aunque tradicionalmente la búsqueda de conocimiento se ha realizado sobre datos almacenados en bases de datos numéricos, donde la información está estructurada, la mayoría de la información que disponemos hoy en día se encuentra en formato textual. Esto se debe, en gran parte, al uso masivo de internet. Por poner un ejemplo concreto, se estima que cada minuto se envían alrededor de 204 millones de correos electrónicos y 100 mil tweets.

Debido a la necesidad que tiene la sociedad actual de manejar esta elevada cantidad de información en grandes volúmenes de datos textuales, surge lo que se denomina *Minería de textos*. Se entiende por Minería de textos la rama de la lingüística computacional cuyo objeto es la obtención de información que no se encuentra de forma explícita en un conjunto de textos (documentos no estructurados). Esta disciplina surge por analogía a la Minería de datos pero con un problema latente, la dificultad en la codificación de la información.

La tarea de la extracción de la información, como puede ser el resumen de un texto, en grandes conjuntos de textos, llevaba a cabo por una persona, tardaría un gran tiempo en realizarse. De ahí que surja la necesidad de 'enseñar' al ordenador a realizar dichas tareas del lenguaje humano. Así es como surge lo que se denomina *Procesamiento del lenguaje natural (PLN)*. El procesamiento del lenguaje natural es una disciplina de la inteligencia artificial que se encarga de la formulación e investigación de mecanismos computacionales para la comunicación entre personas y máquinas mediante el uso del lenguaje humano (lenguaje natural).

Gracias a estas disciplinas, la información textual que disponemos puede ser explotada y, tareas que un humano tardaría gran tiempo en realizar, pueden ser realizadas en un tiempo razonable.

Estas disciplinas abarcan multitud de aplicaciones que van desde tareas sencillas como puede ser la corrección ortográfica a tareas más complejas como puede ser la traducción

automática de textos de un lenguaje a otro. Otras de sus aplicaciones pueden ser, por ejemplo, la detección de frases, el tokenizado, el reconocimiento de entidades nombradas, el Part-Of-Speach Tagger (POS Tagger), el chunking, el parsing, la construcción de un filtro de spam, el análisis de sentimientos o la clasificación de textos, entre otras.[1]

Reconocimiento y clasificación de entidades nombradas

Dentro de las aplicaciones de la Minería de textos, la tarea que se aborda en el trabajo es el *reconocimiento y clasificación de entidades nombradas* mediante el uso de redes neuronales.

Para entender correctamente esta tarea se debe definir bien al principio lo que entendemos por *entidad nombrada*. Borrega et al. [3], diferencian entre las que denominan entidades nombradas fuertes y débiles. Las primeras se refieren a nombres propios, además de expresiones alfanuméricas y fechas, y las segundas, además de lo anterior, a toda la estructura del sintagma nominal. Por ejemplo, en el siguiente fragmento de frase: "un municipio de Zaragoza" se puede considerar como entidad nombrada todo el sintagma nominal o solamente el nombre propio "Zaragoza". En nuestro caso, cuando hablemos de entidad nombrada nos referiremos a nombres propios.

De esta forma, cuando hablemos de reconocimiento y clasificación de entidades nos referiremos a la tarea de la extracción de la información textual que consiste en localizar los nombres propios de un texto y clasificarlos en categorías. Por tanto, nuestra tarea formaría parte de las tareas de aprendizaje supervisado y nuestras observaciones (lo que clasificaremos) serán lo que se denominan ***tokens*** del texto, que son una cadena de caracteres separada de otra por un delimitador. En nuestro caso, el delimitador será un espacio en blanco.

En el trabajo que se desarrolla, se estudiará en un primer momento el uso de algoritmos de redes neuronales para la detección de estas entidades nombradas en un texto para después ir más allá abarcando, además de la detección de estos nombres propios, la clasificación de ellos según sean de persona, organización, localización o miscelánea, en el caso de que no pertenezcan a ninguna de las tres clases anteriores. El primer caso es el que se conoce con el nombre de *reconocimiento de entidades nombradas (NER)* que es un problema de clasificación binaria y el segundo caso, que será nuestro principal caso de estudio, con el de *reconocimiento y clasificación de entidades nombradas (NERC)*, que es un problema de clasificación multinomial.

Etapas de la Minería de textos

Para el descubrimiento de este conocimiento textual, es necesario pasar por diferentes etapas, que ilustramos de forma esquemática en la siguiente figura:[2]

[1] Los conceptos mencionados pueden consultarse en el Glosario.

[2] La figura se refiere, en concreto, al proceso de generación del modelo que nos permitirá realizar la tarea. Una vez se dispusiera del modelo, las etapas serían las mismas, excepto la etapa de validación, y el texto de entrada no contendría etiquetas.

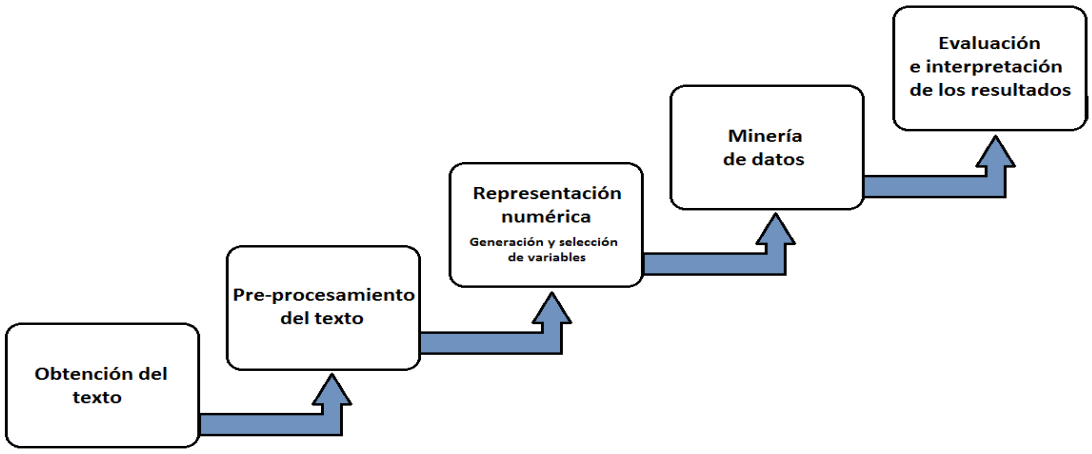


Figura 1.1: Etapas de la Minería de Textos

En primer lugar, nótese que, para poder crear un modelo que nos permita realizar en un futuro la tarea concreta de Minería de textos, es necesario disponer de un texto adecuado para dicha tarea (conjunto de entrenamiento). Luego el primer paso en la Minería de textos sería la **obtención del texto**. La obtención de este texto depende de la finalidad de nuestra aplicación. Así, si nuestro objetivo es el análisis de sentimientos en twitter, nuestros textos serán los tweets, o si, por otro lado, la finalidad es el reconocimiento de entidades nombradas en prensa, nuestros textos serán las noticias.

En tareas de aprendizaje supervisado, este texto deberá además informarnos acerca de sus variables de salida (o *outcomes*). Por ejemplo, en nuestra tarea de reconocimiento de entidades nombradas, los textos que nos permitirán entrenar y validar el modelo deberán tener implícitos una etiqueta (o *tag*) que nos informe sobre qué palabras son nombres propios. El conjunto de textos que contienen dicha información recibe el nombre de *Corpus*. Los dos corpus que se utilizan en el trabajo son el `Ancora` [1] y el `CONLL2002` [14], que explicaremos con detalle en el siguiente capítulo (capítulo 2).

Remarcar que, si no tuviesemos a nuestro alcance un corpus adecuado para la tarea en cuestión, deberíamos generarlo nosotros mismos de forma manual, y puesto que para las tareas del procesamiento del lenguaje natural se necesitan grandes conjuntos de datos para crear un modelo adecuado, esto nos llevaría bastante tiempo de trabajo.

Una vez tenemos a disposición el corpus adecuado, que será nuestro conjunto de entrenamiento, la siguiente etapa sería la encargada del **pre-procesamiento del texto**.

El pre-procesamiento del texto es la fase encargada de la "limpieza" del texto, entendiéndose por limpieza la eliminación de información no relevante (ruido) o sobreinformación para el objetivo final. Los tokens del texto que no aportan información relevante reciben el nombre de *stop words*. Stop words pueden ser, por ejemplo, los números. No obstante, los stop words dependen de la aplicación a realizar pues, en algunas tareas los números, por ejemplo, pueden ser información innecesaria y en otras tareas ser necesaria. Otros ejemplos de pre-procesamiento del texto podrían ser el reemplazamiento de una palabra por su sinónimo o el lematizado de palabras, consiguiendo así unificar los sinónimos y/o palabras con el mismo lema de un texto. Por tanto, esta fase, además de eliminar ruido en el modelo, reduce la dimensionalidad del problema, que es un problema muy presente en Minería de textos.

Una vez tengamos nuestro corpus preprocesado, la siguiente etapa sería la encargada de la **representación numérica** de nuestra información textual, para poder así aplicar el algoritmo matemático.

Existen diversas formas de representación vectorial de los tokens. Introducimos de forma breve dos ejemplos. Una forma simple de representar un token en forma numérica es la denominada codificación *one-hot* donde cada token se representa como un vector en $\{0,1\}^{|V|}$, siendo V el vocabulario total del texto, y donde cada coordenada del vector corresponde con el token del vocabulario ordenado. Así, cada token del texto se representará como un vector con el valor 1 en su índice correspondiente y 0 en el resto. Por ejemplo, supongamos el vocabulario ordenado {"Hoy", "lunes", "es"} , entonces la palabra "Hoy" estaría representada por el vector $[1,0,0]$. Sin embargo, esta codificación tiene bastantes limitaciones puesto que, además de tener el problema de la dimensionalidad, todos los vectores resultan equidistantes, y se pierde así la noción de similaridad. Por otro lado, la herramienta *Word2Vec*, propuesta por Mikolov [16], consigue una codificación de palabras de tipo continuo que sí tiene en cuenta la similaridad de las palabras, además de almacenar información sintáctica y semántica de ellas. Esta herramienta se explicará en el capítulo 3 con más detalle.

Otra forma de codificar el texto sería considerar unas determinadas variables que nos aporten la información del texto que consideremos importante y suficiente para nuestro objetivo, y asignarles unos determinados valores numéricos. Por ejemplo, en el análisis de sentimientos, donde se clasifica una parte del texto, como pueden ser las frases, según tengan una valoración positiva o negativa, las variables que podrían considerarse son los propios tokens y los bigramas (parejas de tokens seguidos) que aparecen en el texto, y el valor que toman estas variables la frecuencia tf-idf.

Nótese que, para extraer esta información del texto en forma de variables numéricas, además de para la etapa anterior de pre-procesamiento del texto, es necesario hacer uso de la programación informática. El lenguaje de programación que usaremos a lo largo del trabajo es el lenguaje de programación orientado a objetos `Java`, que es uno de los lenguajes más extendidos en este área.

Siguiendo con el ejemplo anterior de análisis de sentimientos, para la generación de sus variables se debería programar un código determinado que permita extraer de forma automática los tokens que aparecen en el texto y sus bigramas, además de obtener la frecuencia de éstos en las frases.

Existen librerías de `java` que permiten, a partir de un corpus dado como input, realizar tareas propias del procesamiento del lenguaje natural y, por tanto, implícitamente, generar unas determinadas variables para la tarea en concreto. Nótese que, si queremos hacer uso de esta generación implícita de las variables que nos ofrece la librería, como será nuestro caso, deberemos transformar nuestro corpus al formato concreto que admite la librería como input. Este proceso y las aportaciones realizadas en este trabajo fin de máster se explica con más detalle en el capítulo siguiente (capítulo 2).

Una vez generadas las variables que caracterizan lo que quiero clasificar, ya sea el token, frase o documento, el siguiente paso sería la *selección de variables*. Esta fase es interesante pues el exceso de variables puede derivar en el posible problema conocido como sobreentrenamiento y/o conducir a un posible aprendizaje muy lento. El sobreentrenamiento (*overfitting*) se produce cuando el modelo se ajusta en exceso a las particularidades de los datos del conjunto de entrenamiento, perdiendo la capacidad de generalizar a otros conjuntos de datos distintos.

En nuestro caso de estudio, las variables que consideraremos serán variables binarias que tomarán el valor 1 cuando se produzca el suceso en cuestión y 0 en caso contrario. Una forma de reducir la dimensionalidad en este caso, y que usaremos en nuestra aplicación, sería establecer un valor n para el *cutoff* (o poda) de forma que se eliminasen todas las variables cuya suma sobre todas sus observaciones tome un valor menor que n. Dicho de otra forma, establecer un valor de n para el cutoff equivaldría a eliminar aquellas variables que no hayan sido visitadas al menos n veces o, equivalentemente, cuyo suceso en concreto se presenta menos de n veces en el total del texto. Por ejemplo, si la variable fuese aquella que tomase el valor

1 cuando el token fuese inicio de sentencia y 0 en caso contrario, si el *cutoff* se estableciese en $n = 5$ y el texto constase de 4 sentencias, dicha variable quedaría eliminada y no entraría a formar parte del modelo.

Completada la fase de construcción y selección de variables y teniendo una representación numérica de nuestros datos textuales, estaríamos en condiciones de aplicar algoritmos matemáticos propios de la **Minería de datos** que hiciesen posible la tarea en cuestión. Es decir, estaríamos en la etapa donde se genera el modelo.

El propósito de nuestro estudio será aplicar algoritmos de redes neuronales, en particular, el perceptrón multicapa y las redes recurrentes LSTM[3]. Explicamos estos algoritmos con detalle en el capítulo 3.

Una vez se ha generado el modelo, la siguiente y última etapa es la **validación e interpretación del modelo**. Esta es la etapa encargada de evaluar la capacidad clasificatoria del modelo entrenado, esto es, valorar si se ajusta relativamente bien a la realidad. Esta evaluación del modelo se realiza con las métricas de evaluación sobre el conjunto de validación.

Recuérdese que, tanto para la fase de entrenamiento (la etapa anterior) como para esta fase de validación, se necesita disponer de un documento etiquetado que nos informe sobre la variable de salida (corpus). Para una correcta validación del modelo es necesario que el texto que usemos para la validación (conjunto de validación) sea distinto al de entrenamiento, con el fin de poder conocer si el modelo es generalizable.

Existen diferentes métricas que nos permiten evaluar la eficiencia de un clasificador. Antes de introducir algunas de ellas, debemos definir los siguientes parámetros. Denotaremos por tp, tn, fp y fn al número de verdaderos positivos, verdaderos negativos, falsos positivos y falsos negativos, respectivamente. Para el uso correcto de estos parámetros, es necesario establecer al principio la clase que consideraremos como "positiva". La clase "negativa" quedaría definida como la complementaria de la "positiva".

Por ejemplo, en nuestro primer caso de estudio de clasificación binaria de nombres propios tomaremos como clase positiva la clase de nombres propios. Así, en este caso, tp denotaría el número de nombres propios clasificados correctamente como tal y fp el número de 'no nombres propios' clasificados como nombres propios.

Bajo dicha notación, introducimos a continuación diferentes métricas de evaluación:

- Accuracy: Esta métrica viene dada por la siguiente expresión

$$accuracy = \frac{tp + tn}{tp + tn + fp + fn},$$

  esto es, indica la proporción de buenos clasificados.
  Nótese que el *accuracy* no es una métrica fiable para el rendimiento real de un clasificador en el caso en que el conjunto de datos esté desequilibrado en cuanto al número de clases, es decir, cuando el número de observaciones en las diferentes clases varía en gran medida, como es nuestro caso. Supongamos que el porcentaje de nombres propios en nuestro texto es del 5 % y que nuestro clasificador clasifica todos los tokens en la clase negativa, esto es, en la clase de 'no nombres propios'. En este caso, el valor del *accuracy* será alto, en concreto 0.95, y, sin embargo, no es un buen clasificador para los nombres propios.
  En esos casos, la métrica que se debería usar para medir el rendimiento real del clasificador sería la denominada *F-measure* o *F-score*.

[3]Siglas en inglés de memoria a corto y largo plazo

- F-measure: La métrica *f-measure* viene dada por la siguiente expresión

$$F = 2 \times \frac{precision \cdot recall}{precision + recall} \in [0, 1],$$

donde *precision* y *recall* son las métricas de evaluación que explicamos a continuación.

- Recall: Se distingue entre recall para la clase positiva y para la clase negativa. Indica, en nuestra tarea de clasificación binaria, la proporción de nombres propios (no nombres propios) que han sido correctamente identificado como tales, esto es,

$$\text{Recall (clase positiva)} = \frac{tp}{tp + fn}$$
$$\text{Recall (clase negativa)} = \frac{tn}{tn + fp}.$$

- Precision: Se distingue entre precision para la clase positiva y para la clase negativa. Estas métricas vienen dadas por las siguientes expresiones:

$$\text{Precision (clase positiva)} = \frac{tp}{tp + fp}$$
$$\text{Precision (clase negativa)} = \frac{tn}{tn + fn}$$

En nuestra tarea de clasificación binaria la *precision*, mediría, de entre los clasificados como nombres propios (no nombres propios), la proporción que realmente son nombres propios (no nombres propios).

Para la clasificación binaria, los parámetros tp, tn, fp y fn son fijos pues solo se dispone de dos clases. Sin embargo, si queremos generalizar a más clases, el significado de estos parámetros variará en función de la clase que consideremos de referencia como 'positiva'. Para nuestra principal tarea de estudio de clasificación multiclase si definimos, por ejemplo, como clase positiva la clase de nombres propios de localización, el parámetro tp denotará el número de nombres propios de localización correctamente clasificados. Sin embargo, si consideramos como clase positiva la clase de nombres propios de persona, tp denotará el número de nombres propios de persona correctamente clasificados.

Así, se tendrán 5 *precision*, *recall* y *f-measure*, una para cada clase. La *f-measure* medirá el rendimiento del clasificador para cada clase.

La métrica de evaluación que se considera a la hora de evaluar el modelo depende del problema concreto que se esté modelizando. Por ejemplo, si estamos modelizando el problema de clasificar pacientes con cáncer de próstata, puede que la métrica que más nos interese sea la *precision*. Es decir, se puede pensar que lo más importante sea que haya el menor número de pacientes sanos clasificados como pacientes con cáncer o viceversa.

En nuestro trabajo nos interesa, no solo que los clasificados como nombres propios sean, en un mayor porcentaje posible, realmente nombres propios (mayor *precision*), sino que el porcentaje de nombres propios que se deje sin clasificar como tal sea el menor posible (mayor *recall*). Con este propósito, a la hora de evaluar y elegir nuestro mejor modelo, nos fijaremos en los resultados que nos proporciona la métrica de evaluación *f-measure* para las clases de nombres propios, que es una métrica que tiende a englobar a las dos anteriores.

En los siguientes capítulos se exponen, de forma más amplia, las diferentes etapas por las que pasamos hasta llegar a nuestro objeto de estudio particular: el reconocimiento y

clasificación de entidades nombradas. En el capítulo 2 se aborda toda la etapa del procesado del texto para su representación numérica, introduciendo las variables que consideramos en el modelo. En el capítulo 3 se explican los diferentes algoritmos de redes neuronales que se usan en la aplicación, en particular, redes de propagación hacia adelante y redes recurrentes LSTM. Se explica también una herramienta, basada en redes neuronales, que permite la codificación de las palabras en un vector numérico continuo y que tiene su aplicación en la parte práctica. En los capítulos 4 y 5 se presentan los resultados de nuestra aplicación. El capítulo 4 está dirigido a nuestra primera tarea de estudio de clasificación binaria y el capítulo 5 a nuestra tarea de clasificación multiclase, que es nuestro principal caso de estudio. Por último, en el capítulo 6 se recogen las conclusiones de nuestro estudio práctico y se presentan posibles mejoras para un trabajo futuro.

Capítulo 2

Representación de la información

"Sólo podemos ver poco del futuro, pero lo suficiente para darnos cuenta de que hay mucho que hacer"
Alan Turing

En este capítulo presentamos los corpus sobre los que trabajamos e introducimos las variables que consideramos para nuestra aplicación y, por tanto, la representación de esta información en forma numérica, lo que nos permitirá la posterior aplicación de modelos de redes neuronales. Este capítulo engloba a las tres primeras etapas que aparecen en la figura 1.1.

2.1. Corpus

Los corpus que utilizamos para nuestras tareas en cuestión, como ya se había adelantado en el capítulo anterior, son los denominados `Ancora` y `CONLL2002` que explicamos a continuación.

Ancora

El primer corpus (`Ancora` [1]) se utiliza para nuestra primera tarea de clasificación binaria de reconocimiento de entidades nombradas (NER).

`Ancora` es un corpus del castellano y del catalán, formado principalmente por textos periodísticos, con diferentes niveles de anotación, y en cuya anotación han participado diferentes personas especializadas en el tema de la lingüística.

Un ejemplo de un fragmento de este corpus en castellano es el siguiente:

```
Según    según    _    s    _    postype=preposition    _    15    _    cc    _    _    _    _    _
el    el    _    d    _    postype=article|gen=m|num=s    _    3    _    spec    _    _    _    _
informe    informe    _    n    _    postype=common|gen=m|num=s    _    1    _    sn    _    _    _
,    ,    _    f    _    punct=comma    _    1    _    f    _    _    _    _    _
el    el    _    d    _    postype=article|gen=m|num=s    _    6    _    spec    _    (date    _    _
19_de_mayo    19_de_mayo    _    w    _    _    _    15    _    cc    _    date)    _    _    _
las    el    _    d    _    postype=article|gen=f|num=p    _    8    _    spec    _    _    _    _    _
reservas    reserva    _    n    _    postype=common|gen=f|num=p    _    15    _    suj    _    _    _
de    de    _    s    _    postype=preposition    _    8    _    sp    _    _    _    _    _
oro    oro    _    n    _    postype=common|gen=m|num=s    _    9    _    sn    _    _    _    _    _
y    y    _    c    _    postype=coordinating    _    8    _    coord    _    _    _    _    _
divisas    divisa    _    n    _    postype=common|gen=f|num=p    _    8    _    grup.nom    _    _    _
del    del    _    s    _    postype=preposition|gen=m|num=s|contracted=yes    _    8    _    sp    _
Banco_Central    Banco_Central    _    n    _    postype=proper    _    13    _    sn    _    (org)
eran    ser    _    v    _    postype=semiauxiliary|num=p|person=3|mood=indicative|tense=imperfect    _
```

```
de    de    _    s    _    postype=preposition    _    15    _    atr    _    _    _    _    _
18.300    18300    _    z    _    _    _    18    _    spec    _    (number    _    _    _
millones    millón    _    n    _    postype=common|gen=m|num=p    _    16    _    sn    _    _    _
de    de    _    s    _    postype=preposition    _    18    _    sp    _    _    _    _    _
dólares    dólar    _    z    _    postype=currency    _    19    _    grup.nom    _    number)    _
.    .    _    f    _    punct=period    _    15    _    f    _    _    _    _    _
```

Cada línea aporta información sobre cada token del corpus. En el primer lugar aparece el propio token y el resto de la línea ofrece diferentes niveles de anotación tanto sintácticos, semánticos como morfológicos (en la web oficial de `Ancora` [1] se ofrece documentación sobre las diferentes etiquetas). Nosotros prestaremos atención a la etiqueta que nos indica cuando un token es nombre propio. El cuarto elemento de cada línea del corpus nos informa sobre la morfología de cada token sobre el que se proporciona la información. La letra `n` indica que es un nombre. Si su subclasificación es de nombre propio, aparecerá en lo siguiente la etiqueta `postype=proper`, como ocurre en línea catorce con 'Banco Central'.

CONLL2002

El segundo corpus (`CONLL2002` [14]) se utiliza para nuestra última y principal tarea de estudio de clasificación multiclase, que es el reconocimiento y clasificación de entidades nombradas (NERC).

`CONLL2002` nos proporciona un corpus en castellano con etiquetas de postaggeado y de entidades nombradas, construido a partir de una colección de noticias publicadas por la Agencia de Noticias EFE desde Mayo de 2000.

El corpus está formado por tres columnas separadas por un único espacio en blanco. En la primera columna de cada línea se presenta el token, en la segunda columna la etiqueta del postagger y en la tercera columna la etiqueta de la entidad nombrada, que es la que nos interesa.

Un ejemplo de un fragmento de este corpus es el siguiente:

```
Por SP O
su DP O
parte NC O
, Fc O
el DA O
Abogado NC B-PER
General AQ I-PER
de SP O
Victoria NC B-LOC
, Fc O
Rob NC B-PER
Hulls AQ I-PER
, Fc O
indicó VMI O
que CS O
no RN O
hay VAI O
nadie PI O
que PR O
controle VMS O
que CS O
las DA O
informaciones NC O
contenidas AQ O
en SP O
CrimeNet NC B-MISC
son VSI O
veraces AQ O
. Fp O
```

La letra `B` indica que es la primera palabra de la entidad nombrada y la letra `I` que le antecede otra palabra que forma parte de la entidad nombrada. Por el contrario, la letra `O` indica que el token no es una entidad nombrada. Las etiquetas `PER, LOC, ORG` y `MISC` indican, respectivamente, que las entidades nombradas corresponden a personas, localizaciones, organizaciones o misceláneas.

Estos corpus contienen la información necesaria para nuestra tarea de aprendizaje supervisado, esto es, nos informan acerca del valor de la variable de salida. Por tanto, habríamos completado la etapa 1 de la figura 1.1, al disponer de corpus adecuados para nuestra aplicación.

Una vez seleccionado el corpus, que es nuestro conjunto de entrenamiento, la siguiente etapa sería la que hemos denominado *pre-procesamiento del texto*.

Aunque esta etapa puede resultar interesante en algunas aplicaciones, en nuestro caso hemos decidido no realizarla por lo motivos que se exponen a continuación. Se podría haber pensado, en un primer momento, eliminar, por ejemplo, los signos de puntuación. Sin embargo, nuestro objeto de estudio es el reconocimiento de nombres propios y se pensó que dichos signos, más que constituir ruido en el modelo, podrían aportar todo lo contrario. Por ejemplo, en el fragmento de `CONLL2002`, las comas sirven para especificar la entidad nombrada sobre la que se habla. Por otro lado, se podría haber considerado la opción de lematizar las palabras. Sin embargo, se pensó que no era buena opción pues en el estudio interesa, no solo la información semántica, sino la sintáctica también. Por poner un ejemplo, en la frase 'Miguel es un estudiante que estudia en Zaragoza', aunque 'estudiante' y 'estudia' tengan la misma raíz, la información que aportan es diferente pues la palabra 'estudiante' hace referencia a 'Miguel' (entidad nombrada de persona), mientras que la forma verbal 'estudia' hace referencia a 'Zaragoza' (entidad nombrada de localización). A grandes rasgos, estas son las consideraciones por las que se decidió no pre-procesar el texto.

2.2. Representación numérica

Como se ha adelantado en el capítulo anterior, hacemos uso de una librería de `Java`, que hemos tenido que modificar y adaptar a nuestro caso, para la representación numérica de nuestros corpus.

2.2.1. Librería openNLP

La librería que se utiliza es la denominada `openNLP` [24]. Esta librería soporta la mayoría de tareas para el procesamiento del lenguaje natural como es el NERC. Aunque el algoritmo que incluye para la resolución de estas tareas es el de máxima entropía [20], que no es el algoritmo propósito de estudio, aprovecharemos, para la tarea del NERC, la parte del código de la librería donde genera las variables.

Para poder llevar a cabo cada una de estas tareas, es necesario que el texto esté en un formato específico que admite la librería como input. Un ejemplo del formato opennlp para el reconocimiento y clasificación de nombres propios es el siguiente:

```
Por su parte , el <START:person> Abogado General <END> de <START:location> Victoria
<END> , <START:person> Rob Hulls <END> , indicó que no hay nadie que controle que
las informaciones contenidas en <START:misc> CrimeNet <END> son veraces .
```

Cada token debe ir separado por un espacio en blanco y cada frase debe ir en una línea diferente. Nótese que en nuestro caso se consideran tokens también los signos de puntuación,

luego irán también separados por un espacio en blanco. El texto contiene etiquetas que nos indican cuando uno o varios tokens son entidades nombradas. En el caso de clasificación binaria (NER), debe aparecer la etiqueta `<START>` antes de la entidad nombrada, y la etiqueta `<END>` después. En el caso de la clasificación multiclase, las etiquetas que se consideran son `<START:person>`, `<START:location>`, `<START:organization>` o `<START:misc>` según sean los nombres propios de persona, localización, organización o miscelánea, respectivamente. Esta marca informa al programa del valor de la variable de salida y, como se ha dicho en el primer capítulo, es necesaria tanto para el texto de entrenamiento como para el de validación. Es importante que la tokenización del texto de entrenamiento, de validación y el de entrada (cuando se quiera predecir) sea idéntica. Por tanto, una vez se tenga el modelo generado, el texto de entrada sobre el que se quiera aplicar el modelo debería procesarse para que la tokenización y segmentación por frases así sea.

Como puede verse en los ejemplos de los corpus `Ancora` y `CONLL2002`, sus formatos no corresponden con el formato opennlp (2.2.1). Por tanto, como se anunció en el capítulo anterior, es necesario crear una clase en `Java` para la transformación de estos corpus al formato opennlp.

Afortunadamente, la librería permite transformar el formato del corpus `CONLL2002` a su formato original, por lo que, en este caso, no es necesario un trabajo adicional para ello. Sin embargo, para el corpus `Ancora`, sí es necesario crear una clase en `Java` que nos permita transformar el corpus en formato opennlp. El código de la clase y un resumen de él se incluye en los Anexos.

Una vez tenemos los corpus en el formato opennlp, el siguiente paso sería realizar nuestras tareas NER o NERC con la librería `openNLP` para generar así, implícitamente, las variables que nos permitirán la representación numérica de la información para aplicar después los algoritmos de redes neuronales.

Para la aplicación de los algoritmos de redes neuronales se utilizó la librería `deeplearning4j` [4] de `Java`. Esta librería exige como input una matriz de datos donde el elemento (i, j) de la matriz sea el valor que tome la variable j en la observación i. Por tanto, se tuvo que analizar el código fuente de la librería `openNLP` e implementar código en él que permitiera, en el momento en el que se "generan" las variables para máxima entropía, extraer esta información y obtener la matriz de datos numérica, es decir, la representación numérica del texto. En los Anexos se muestra el código implementado y algunas consideraciones que se han debido tener en cuenta como parte de la realización del trabajo.

A diferencia de la Minería de datos, en la Minería de textos, debido al tipo de dato con el que se trabaja (datos textuales), todo este proceso comentado hasta ahora, es necesario y ha supuesto una dificultad adicional en el trabajo.

2.2.2. Variables

Como se adelantó en el capítulo anterior, nuestras variables, que se generan con la librería `openNLP`, serán variables binarias que tomarán el valor 1 o 0 correspondiendo con la presencia o ausencia, respectivamente, de un determinado suceso.

Para facilitar la explicación de las variables que consideramos en nuestro estudio, introducimos el siguiente ejemplo:

```
Este es mi trabajo fin de máster
```

Para cada token del texto, las variables que se generan son las siguientes:

1. Ventana de tamaño 2: Genera 5 variables que serían las siguientes: el token actual (`w`), los dos tokens anteriores (`p1w` y `p2w`) y los dos tokens posteriores (`n1w` y `n2w`), en minúscula. Así, para la palabra 'trabajo' de nuestro ejemplo, las variables que se generarían serían las siguientes:

   `w=trabajo, p2w=es, p1w=mi, n1w=fin, n2w=de.`

   Para un determinado token (observación), estas variables tomarían el valor 1 cuando el token sea la palabra 'trabajo', el token que ocupa dos posiciones anteriores la palabra 'es', el token anterior la palabra 'mi', el token posterior la palabra 'fin' y el token que ocupa dos posiciones por delante la palabra 'de', respectivamente.

2. Ventana de tamaño 2 con clase: Genera otras 5 variables en base a la información de la ventana, como en el caso anterior, pero en este caso con la información de la clase de cada token. La librería distingue las siguientes clases:

   - lc - lowercase alphabetic
   - 2d - two digits
   - 4d - four digits
   - an - alpha-numeric
   - dd - digits and dashes
   - ds - digits and slashes
   - dc - digits and commas
   - dp - digits and periods
   - num - digits
   - sc - single capital letter
   - ac - all capital letters
   - ic - initial capital letter
   - other - other

   Figura 2.1: Clases de los tokens. OpenNLP.

   Por tanto, para la palabra 'trabajo' de nuestro ejemplo, las 5 variables que se generarían serían las siguientes

   `wc=lc, p2wc=lc, p1wc=lc, n1wc=lc, n2w=lc,`

   pues toda la ventana son palabras en minúscula (`lc`). Además de estas variables, se generan otras 5 que son la unión de las 10 anteriores (el propio token además de la información de su clase):

   `w&c=trabajo,lc , p2w&c=es,lc , p1w&c=mi,lc , n1w&c=fin,lc , n2&w=de,lc.`

3. Bigrama: Genera dos variables que se corresponderían con el bigrama constituido por el token actual y el anterior y el bigrama constituido por el token actual y el siguiente. De la misma manera que antes, genera otras dos variables (bigramas) pero con la información de sus clases. Así, para la palabra 'trabajo' de nuestro ejemplo, las 4 variables que se generarían serían las siguientes:

   `w,nw=trabajo,fin , pw,w=mi,trabajo , pwc,wc=lc,lc , wc,nwc=lc,lc`

4. Inicio de sentencia: La última variable que se considera es la que nos indica si el token es inicio de sentencia o no. Como la palabra 'trabajo', en nuestro ejemplo, no es inicio de sentencia, dicha variable tomaría, para esta observación, el valor 0.

No es de extrañar que, para un corpus dado, el número de variables que se generen, con respecto al número de observaciones, se dispare. Nótese que, como mínimo, se generarán 20 variables y como máximo 20 por el número de observaciones (tokens). En el ejemplo que hemos introducido se generarían en total 84 variables y es fácil prever que para un corpus más grande el número de variables crezca exponencialmente.

Por ello, es necesario reducir la dimensión de estas variables. Como se comentó en el capítulo anterior, el método que se usa es el *cutoff* dado un parámetro fijo n. El valor de *cutoff* que se considera en la aplicación es de 10. Así, las variables cuyo suceso no se haya experimentado al menos 10 veces en el texto de entrenamiento, no entrarán a formar parte del modelo. Esto hace posible reducir considerablemente la dimensión a la par que eliminamos variables específicas del conjunto de entrenamiento que, posiblemente, hubiesen afectado al modelo aportando ruido, previniendo así el sobreentrenamiento.

Volviendo al ejemplo anterior, vamos a considerar para su representación matricial, por comodidad, un valor de 2 para el *cutoff*. Para este valor de *cutoff*, la matriz numérica resultante sería la que sigue

	p2wc=lc	p1wc=lc	wc=lc	n1wc=lc	n2wc=lc
Este	0	0	0	1	1
es	0	0	1	1	1
mi	0	1	1	1	1
trabajo	1	1	1	1	1
fin	1	1	1	1	1
de	1	1	1	1	0
máster	1	1	1	0	0

Un texto bien estructurado contiene la información de forma secuencial, esto es, el significado de una determinada palabra depende del contexto de la frase. Las variables que consideramos guardan parte de esta información (mediante las ventanas y bigramas), además de la información de la clase (Figura 2). Es por eso por lo que se consideró tratar con ellas, pues se pensó que estas variables nos ofrecían información relevante para nuestra tarea en cuestión.

Este capítulo se podría resumir como sigue. El propósito de nuestro estudio es la aplicación de redes neuronales para la tarea de reconocimiento y clasificación de entidades nombradas. La librería `deeplearning4j` de `Java` nos permite la aplicación de estos algoritmos. Sin embargo, para poder hacer uso de dicha librería, es necesario aportar al programa la información en forma de matriz numérica. Para la creación de esta matriz numérica se ha tenido que modificar el código fuente de la librería `openNLP` de `Java` para que así lo hiciese. Esta librería admite un determinado formato como input por lo que hemos tenido que transformar nuestro corpus al formato opennlp, para lo cual ha sido necesario la construcción de una clase en `Java` que así lo hiciese.

Una vez se tiene la representación en forma numérica, la siguiente etapa sería la aplicación de los algoritmos matemáticos. En el siguiente capítulo, se explican los algoritmos que utilizamos en nuestra aplicación.

Capítulo 3

Redes neuronales artificiales

"Cuando las computadoras igualen la capacidad de cálculo del cerebro humano, necesariamente lo superarán."
Raymond Kurzweil

En el capítulo anterior se han presentado los corpus sobre los que trabajamos y se han introducido las variables que consideramos en nuestro estudio.

Una vez disponemos de la información en forma numérica, el siguiente paso sería la estimación de los modelos matemáticos. En este capítulo se explican los modelos de redes neuronales artificiales que usamos en nuestra aplicación, en concreto, el perceptrón multicapa y las redes recurrentes LSTM, haciendo hincapié en los parámetros que utilizamos en sus entrenamientos.

Las redes neuronales artificiales (RNA) son algoritmos matemáticos que surgen, como el propio nombre indica, de la idea de imitar el comportamiento de las redes neuronales del cerebro humano. Del mismo modo que un cerebro humano está constituido por neuronas, una red neuronal artificial será el conjunto de neuronas artificiales (o nodos), que son las encargadas de realizar el cálculo, distribuidas en diferentes capas, interconectadas entre sí, siguiendo una configuración determinada.

El comportamiento de este elemento básico de la red (neurona artificial) se puede describir de la siguiente manera:

- La neurona recibe una entrada (o input) desde otras neuronas. Cada conexión de estos inputs a la neurona lleva asociada un 'peso', que equivaldría a la eficiencia sináptica de una neurona biológica, esto es, estos pesos asignarían la importancia de cada input.
- Se realiza la suma ponderada de los inputs mediante sus pesos asociados, más un valor de sesgo (bias). Esta suma ponderada equivaldría a la 'activación' de la neurona.
- La salida de la neurona será el resultado de aplicar una función, llamada *función de activación o de transferencia*, a la 'activación' de la neurona. Esta función limita el rango de valores que puede tomar la salida.

Matemáticamente, lo anterior puede resumirse con la siguiente expresión:

$$y_i = f\left(\sum_{j=1}^{n} w_{ji}x_j + u_i\right), \tag{3.1}$$

donde n es el número de entradas, w_{ji} los pesos asociados al input x_j, u_i el sesgo asociado a la neurona i e y_i su salida.

Nota 1. *Por comodidad, es habitual considerar el sesgo u_i como el resultado de la multiplicación de un peso w_{0i} por una entrada ficticia $x_0 = 1$. Bajo dicha consideración, la expresión (3.1) quedaría:*

$$y_i = f\left(\sum_{j=0}^{n} w_{ji}x_j\right). \tag{3.2}$$

Como se ha dicho, una red neuronal artificial estará formada por varias capas de neuronas conectadas entre sí. Las neuronas de la capa de entrada serán nuestras variables del modelo (*inputs*) y las neuronas de la capa de salida nuestras variables de salida (*outcomes*). Si existen, el resto de capas reciben el nombre de *capas ocultas*.

Respecto a la configuración de la red, cuando la salida de una neurona solo puede ser entrada de una neurona de la capa siguiente, las redes se denominan de *propagación hacia adelante o feedforward* (FNN, siglas en inglés). Si, por otro lado, la salida de una neurona sí puede ser entrada de neuronas de capas anteriores o de la misma capa, incluyendo ser entrada de ella misma, entonces se dice que la red es *recurrente* (RNN, siglas en inglés). La siguiente figura ilustra dos ejemplos de estas configuraciones de red:

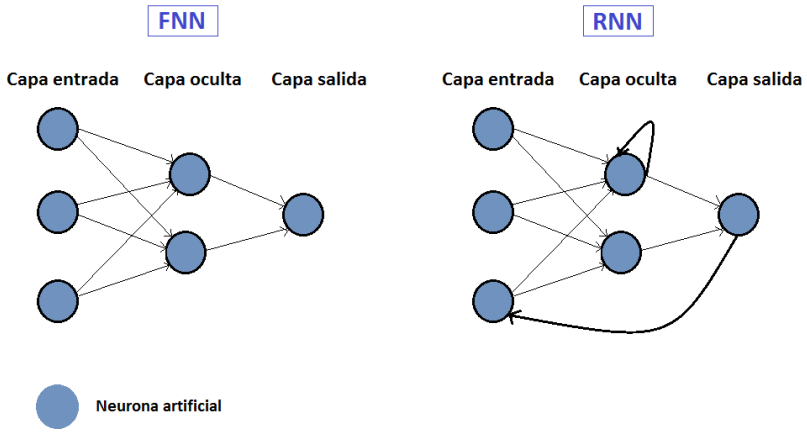


Figura 3.1: Red feedforward y recurrente

Abordaremos en el trabajo estos dos tipos de redes en tareas de clasificación. Más detalle sobre redes neuronales y sus aplicaciones en diversos problemas pueden verse en [2] y [9].

3.1. Redes neuronales *feedforward*. Perceptrón multicapa

La red neuronal más sencilla que se puede considerar es aquella que no tiene capas ocultas. El primer modelo de este tipo de redes fue el denominado *Perceptrón Simple*, introducido por Rosenblatt [22], cuya función de activación es la función escalón:

$$y_i = \begin{cases} 1, & \sum_{j=0}^{n} w_{ji}x_j \geq 0 \\ 0, & \sum_{j=0}^{n} w_{ji}x_j < 0. \end{cases}$$

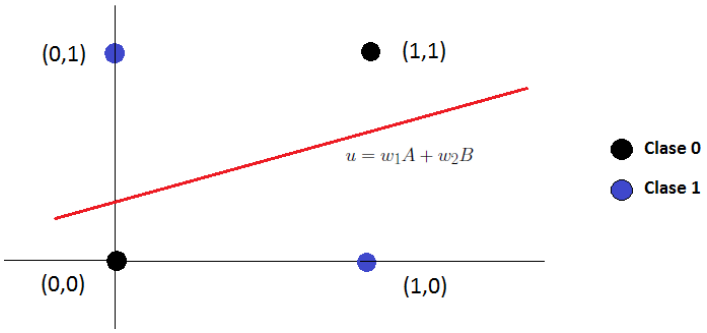
Obsérvese que el Perceptrón Simple discrimina clases linealmente separables, esto es, mediante un hiperplano, sin embargo, no puede resolver correctamente problemas no linealmente separables, como es el conocido *compuerta XOR*, que ilustramos en el siguiente ejemplo.

Ejemplo XOR

El problema XOR puede verse como una adición de módulo 2. Luego, suponiendo que tenemos dos variables de entrada A y B tomando los valores 0 o 1, el output será 0, en el caso de que las dos variables de entrada tomen el valor 0 o el valor 1, y 1 en caso contrario:

A	B	A XOR B
0	0	0
0	1	1
1	0	1
1	1	0

Si representamos gráficamente los datos observamos que, en efecto, no existe ningún hiperplano que sea capaz de separar las dos clases:



Por tanto, el Perceptrón Simple (red monocapa) tiene limitaciones, pudiendo solo establecer fronteras lineales entre las regiones. Si queremos clasificar estructuras más complejas, no linealmente separables, será necesario introducir más capas en la red neuronal (capas ocultas). Así es como surge la red neuronal denominada *Perceptrón Multicapa*.

Perceptrón Multicapa (MLP)

El perceptrón multicapa es una red neuronal compuesta por una capa de entrada, una (o más) capas ocultas y una capa de salida con una conexión completa hacia adelante de las neuronas y con funciones de activación no lineales.

Como se ha comentado anteriormente, el perceptrón multicapa surge para resolver problemas más complejos, no linealmente separables, permitiendo así abordar problemas de clasificación de gran envergadura.

En el caso de aproximación de funciones, Funahashi demostró (Teorema de Funahashi [5]) que un perceptrón multicapa con al menos una capa oculta permite aproximar hasta el nivel deseado, dentro de un conjunto compacto, cualquier función continua, es decir, que el perceptrón multicapa actúa como un aproximador universal de funciones. Este teorema de aproximación universal es un 'teorema de existencia' en el sentido de que asegura que una red neuronal con una capa oculta es suficiente para aproximar cualquier función continua arbitraria, pero no dice que sea lo óptimo ni cómo construirla.

Supongamos que el perceptrón multicapa tiene una sola capa oculta. Entonces es fácil

deducir de (3.2) que la salida de una neurona i tendrá la siguiente expresión:

$$y_i = \widetilde{f}\left(\sum_{j=0}^{m} w_{ji}^{(2)} f\left(\sum_{k=0}^{n} w_{kj}^{(1)} x_k\right)\right), \tag{3.3}$$

donde $w_{ji}^{(2)}$ representa el peso asociado a la conexión de la neurona j de la segunda capa con la neurona i de la tercera capa (la capa de salida), $w_{ki}^{(1)}$ el peso asociado a la conexión de la neurona x_k de la primera capa (capa de entrada) con la neurona j de la segunda capa y f y $\widetilde{f}$ las funciones de activación de la capa oculta y la capa de salida, respectivamente. Bajo esta notación, n representaría el número de neuronas en la capa de entrada (variables del modelo) y m el número de neuronas en la capa oculta.

Generalizando a $C \geq 3$ capas, el modelo de perceptrón multicapa (3.3) se reescribiría matemáticamente con la siguiente expresión recursiva:

$$y_i = f\left(\sum_{j=0}^{n_{C-1}} w_{ji}^{C-1} x_j^{C-1}\right), \quad i = 1, \ldots, n_C \tag{3.4}$$

$$x_j^{C-1} = f\left(\sum_{k=0}^{n_{C-2}} w_{kj}^{C-2} x_k^{C-2}\right), \quad j = 1, \ldots, n_{C-1} \tag{3.5}$$

donde n_C y n_{C-1} representan el número de neuronas en la capa de C y el número de neuronas en la capa $C-1$, respectivamente, w_{ji}^{C-1} el peso asociado a la neurona j de la capa $C-1$ con la neurona i de la capa C y x_j^{C-1} la salida de la neurona j de la capa $C-1$. Sin pérdida de generalidad, se ha denotado por f a la función de activación de las neuronas para todas las capas. Sin embargo, la función de activación f no tiene por qué ser la misma en todas las capas de la red. Usaremos esta notación en lo que sigue.

Mostramos a continuación, a modo ilustrativo, el esquema del modelo de perceptrón multicapa (3.4-3.5) con una capa oculta, $n_3 = 1, n_2 = 2$ y $n_1 = 5$:

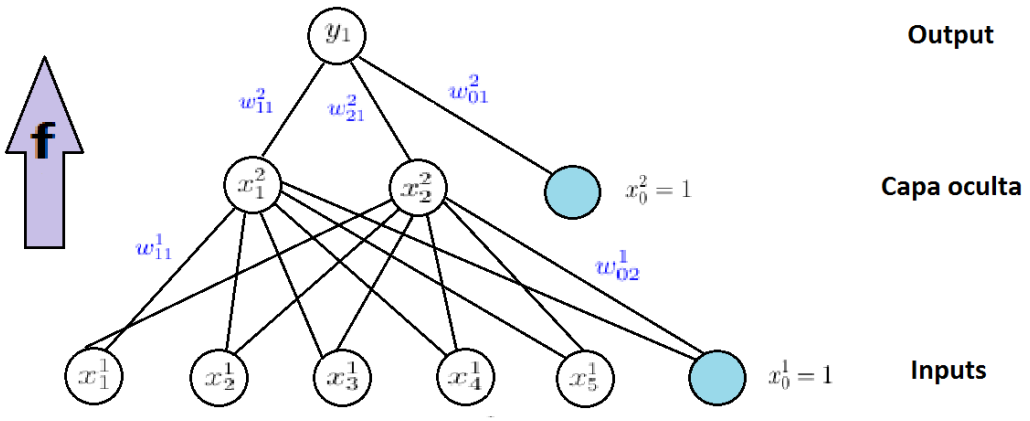


Figura 3.2: MLP de una capa oculta

Nuestro objetivo será estimar el modelo de clasificación (3.4)-(3.5), o equivalentemente, estimar los pesos w_{ji}. El proceso de estimación de los pesos se denomina proceso de 'aprendizaje' o 'entrenamiento' de la red. El algoritmo de aprendizaje del perceptrón multicapa es

el denominado *algoritmo de backpropagation.* Al igual que las personas, las redes neuronales artificiales aprenden mediante el ejemplo, que en este caso es el conjunto de entrenamiento.

3.1.1. Algoritmo de backpropagation

El algoritmo de backpropagation (o de retropropagación) es un algoritmo de aprendizaje por el cual se van adaptando y modificando los parámetros (pesos) de la red neuronal. Esta adaptación de los pesos se consigue minimizando una función de error o *función de pérdida* que mide el error que se comete con la estimación de la salida con respecto a la salida real.

El algoritmo constaría básicamente de dos pasadas (una hacia adelante y otra hacia atrás) a través de las capas de la red neuronal[1]. En la pasada hacia adelante se le aplica el vector de entrada de la observación del conjunto de entrenamiento y su efecto se propaga a través de la red de la forma (3.4)-(3.5). Esto da como resultado un vector de salida $[\hat{y}_1, \ldots, \hat{y}_{n_C}]$ que corresponde con la salida actual. La pasada hacia atrás es la encargada de ajustar los pesos de forma que el error cometido sea mínimo. Así, el algoritmo de backpropagation original no es más que un problema de optimización cuya función objetivo es una función de error E:

$$\min_{\mathbf{w}} E$$

donde

$$E = \frac{1}{N}\sum_{n=1}^{N} E_n,$$

siendo $\mathbf{w}$ los pesos, N el número de observaciones de nuestro conjunto de entrenamiento y E_n el error cometido en la n-ésima observación. Nótese que la minimización en $\mathbf{w}$ pone de relieve la dependencia de los errores E_n con los pesos $\mathbf{w}$.

Se pueden considerar diferentes funciones de error. Sin pérdida de generalidad, consideramos para la explicación del algoritmo el error cuadrático:

$$E_n = \frac{1}{2}\sum_{i=1}^{n_C} (y_i(n) - \hat{y}_i(n))^2 \tag{3.6}$$

donde $y = (y_1(n), \ldots, y_{n_C}(n))$ es el vector de salida real de la red e $\hat{y} = (\hat{y}_1(n), \ldots, \hat{y}_{n_C}(n))$ el vector de salida estimado (3.4), para la observación n.

El método que utiliza el algoritmo de backpropagation original para la minimización del error es el método iterativo de primer orden conocido como *gradiente descendente*, que consigue una adaptación de los pesos siguiendo la dirección de búsqueda negativa del gradiente de la función del error E en el espacio de pesos, de ahí el nombre, y que determina la zona donde la disminución del error es más rápida. De esta forma, la actualización de los pesos tendrá la siguiente expresión general:

$$\mathbf{w}(n) = \mathbf{w}(n-1) - \eta \nabla E(\mathbf{w}(n-1)), \tag{3.7}$$

siendo $\mathbf{w}(n)$ el vector de pesos en la iteración $n \geq 1$, donde $\mathbf{w}(0)$ es el vector de pesos iniciales desde donde se parte, $\nabla E(\mathbf{w}(n-1))$ el gradiente de E en $\mathbf{w}(n-1)$ y $\eta > 0$ un factor denominado *tasa de aprendizaje*. Así, cuando hablemos de *iteración* nos referiremos a cada actualización de los pesos.

De la expresión anterior (3.7) se deduce la siguiente expresión

$$\Delta w(n) = w(n) - w(n-1) = -\eta \frac{\partial E}{\partial w}[n], \tag{3.8}$$

[1]El nombre del algoritmo viene, de hecho, de esta última pasada hacia atrás.

donde w es el peso a estimar y $\frac{\partial E}{\partial w}[n]$ denota la derivada parcial de E en la iteración n.

Esta actualización se realiza de forma iterativa hasta cumplir un determinado criterio de parada.

Tipos de entrenamiento

Se distinguen tres tipos de entrenamiento en base al momento en el que se realiza la actualización en los pesos.

- Por una parte, la *estrategia batch (o por lotes u off-line)* actualiza los pesos después de pasar por la red todos los datos de entrenamiento (pasada hacia adelante). De esta forma, se minimiza directamente el error total, esto es, la media de los errores producidos por todas las observaciones de entrenamiento. Aunque teóricamente hablando esta estrategia sería la correcta, para grandes conjuntos de datos no resulta factible en la práctica pues la convergencia es muy lenta, requiriéndose mucho tiempo computacional, además de memoria para su implementación. Esta estrategia resulta útil para conjuntos de datos pequeños.

- La *estrategia on-line*, en cambio, realiza la actualización de los pesos después de pasar por la red cada observación del conjunto de entrenamiento, acelerando así la convergencia con respecto a la estrategia anterior. Esta estrategia, por tanto, para conjuntos de datos más grandes, está más extendida en la práctica. Sin embargo, la estrategia *on-line* no permite realizar cálculos en paralelo, como sí permite la estrategia *batch*, pues es necesario el resultado obtenido después de procesar una observación antes de procesar la siguiente.

- La estrategia que usamos en el trabajo es una solución intermedia de las dos estrategias anteriores y recibe el nombre de *estrategia por minibatch*. Como el propio nombre indica, se basa en la estrategia *batch* pero sobre conjuntos de datos más pequeños (subconjuntos), esto es, divide el conjunto de entrenamiento en grupos del mismo tamaño (*minibatchs*) y actualiza los pesos tras pasar por la red cada grupo. De esta forma, conservamos la ventaja de la estrategia *on-line* además de permitirnos realizar cálculos en paralelo.

Cada pasada de la red sobre todo el conjunto de datos recibe el nombre de *época*. Nótese que, en el caso de la estrategia *batch*, la definición de iteración y época es equivalente.

Por simplicidad, desarrollamos a continuación el algoritmo de backpropagation para el caso de entrenamiento *on-line* con función de error (3.6). En este caso, el método que se usa recibe el nombre de *gradiente descendiente estocástico*. Distinguiremos la actualización de los pesos conectados de la capa oculta $C-1$ a la capa de salida C y el resto.

Bajo la misma notación de (3.4)-(3.5), para el primer caso, usando el método del gradiente descendente estocástico, se tiene que

$$w_{ji}^{C-1}(n) = w_{ji}^{C-1}(n-1) - \eta \frac{\partial E_n}{\partial w_{ji}^{C-1}},$$

donde n es la iteración actual y

$$\frac{\partial E_n}{\partial w_{ji}^{C-1}} = -\left(y_i(n) - \hat{y}_i(n)\right) \frac{\partial \hat{y}_i(n)}{\partial w_{ji}^{C-1}},$$

en virtud de la expresión (3.6).
Aplicando la regla de la cadena, por la ecuación (3.4), se tiene que

$$\frac{\partial \hat{y}_i(n)}{\partial w_{ji}^{(C-1)}} = \frac{\partial f(\sum_{j=0}^{n_{C-1}} w_{ji}^{C-1} x_j^{C-1})}{\partial w_{ji}^{C-1}} \times x_j^{C-1}.$$

Si definimos el siguiente término $\delta_i^C(n)$, asociado a la neurona i de la capa de salida C y la observación n, como sigue

$$\delta_i^C(n) = -\left(y_i(n) - \hat{y}_i(n)\right) \frac{\partial f(\sum_{j=0}^{n_{C-1}} w_{ji}^{C-1} x_j^{C-1})}{\partial w_{ji}^{C-1}},$$

entonces la expresión de la actualización de los pesos queda finalmente de la forma siguiente:

$$w_{ji}^{C-1}(n) = w_{ji}^{C-1}(n-1) - \eta \delta_i^C(n) x_j^{C-1} \quad j = 0, 1, 2, \ldots, n_{C-1},\ i = 1, 2, \ldots, n_C. \tag{3.9}$$

Para el caso de los pesos de la capa c a la capa $c+1$ con $c = 1, 2 \ldots, C-1$, el razonamiento es análogo al anterior, pero aplicando ahora la regla de la cadena $c+1$ veces. Desarrollamos el caso para $c = 1$.
Usando el método del gradiente descendente estocástico, tenemos que

$$w_{kj}^{C-2}(n) = w_{kj}^{C-2}(n-1) - \eta \frac{\partial E_n}{\partial w_{kj}^{C-2}}$$

donde

$$\frac{\partial E_n}{\partial w_{kj}^{C-2}} = -\left(y_i(n) - \hat{y}_i(n)\right) \frac{\partial \hat{y}_i(n)}{\partial w_{kj}^{C-2}}.$$

Aplicando la regla de la cadena dos veces, de las ecuaciones (3.4)-(3.5), se tiene que

$$\frac{\partial \hat{y}_i(n)}{\partial w_{kj}^{C-2}} = \frac{\partial f\left(\sum_{j=0}^{n_{C-1}} w_{kj}^{C-1} x_j^{C-1}\right)}{\partial w_{kj}^{C-2}} \times x_j^{C-1} \times \frac{\partial x_j^{C-1}}{\partial w_{kj}^{C-2}},$$

donde

$$\frac{\partial x_j^{C-1}}{\partial w_{kj}^{C-2}} = \frac{\partial f\left(\sum_{j=0}^{n_{C-2}} w_{kj}^{C-2} x_j^{C-2}\right)}{\partial w_{kj}^{C-2}} \times x_j^{C-2}$$

Denotando por $\delta_j^{C-1}(n)$ la siguiente expresión

$$\delta_j^{C-1}(n) = \frac{\partial f\left(\sum_{j=0}^{n_{C-2}} w_{kj}^{C-2} x_j^{C-2}\right)}{\partial w_{kj}^{C-2}} \sum_{i=1}^{n_C} \delta_i^C(n) w_{ji}^{C-1},$$

se tiene la siguiente expresión de actualización de los pesos:

$$w_{kj}^{C-2}(n) = w_{kj}^{C-2}(n-1) - \eta \delta_j^{C-1}(n) x_j^{C-2} \quad k = 0, 1, 2, \ldots, n_{C-2},\ j = 1, 2, \ldots, n_{C-1}. \tag{3.10}$$

3.1.2. Parámetros

Para el entrenamiento de la red neuronal será necesario fijar una función de error a minimizar, además de establecer un valor para la tasa de aprendizaje η. Por otro lado, nótese que para poder aplicar el algoritmo de backpropagation, en particular el método del gradiente descendente, será necesario que la función de error sea diferenciable, en particular las funciones de activación, además de requerir un valor inicial en los pesos. Se deberá también establecer un criterio de parada que, en ocasiones, suele ser un número de épocas fijado. Además de esto, deberemos fijar el número de neuronas para las capas ocultas de la red.

Debido a la estructura de los datos y/o a la configuración de la red y/o los parámetros elegidos, el entrenamiento de la red neuronal puede sufrir unos determinados problemas. Para evitar o atenuar estos posibles problemas, se suelen considerar otros parámetros adicionales para el entrenamiento como veremos a continuación.

Así, a la hora de definir el entrenamiento y la arquitectura de la red en su totalidad, se requerirá la elección de multitud de 'parámetros'. Encontrar el mejor modelo clasificatorio requerirá explorar diferentes valores de dichos parámetros.

Los posibles problemas en los que puede derivar el entrenamiento de la red, así como la elección de sus parámetros se explican a continuación.

Función objetivo

Anteriormente, hemos explicado el algoritmo de backpropagation usando, por simplicidad, el error cuadrático como función de error.

Para problemas de clasificación, la función de pérdida que suele considerarse es (menos) el logaritmo de la verosimilitud, o equivalentemente, la denominada entropía cruzada, cuya expresión para dos clases es la siguiente ([2])

$$E = -\sum_x [y \ln \hat{y} + (1-y)\ln(1-\hat{y})],$$

donde y la salida real de la red e $\hat{y}$ la salida estimada, para cada observación x. Más adelante se justifica que la expresión está bien definida, por el uso de la función *softmax*.

Variantes del algoritmo

Hemos visto en el algoritmo de backpropagation que nos vamos acercando al mínimo de la función de error E mediante el descenso del gradiente.

Así, la tasa de aprendizaje η está relacionada con la trayectoria en el espacio de los pesos mediante este método (3.7), influyendo en la variación de los pesos, siendo mayor esta variación a valores más grandes de η. Una tasa de aprendizaje pequeña puede conllevar a una trayectoria más "suave", pero también a un aprendizaje más lento. Por otro lado, una tasa de aprendizaje alta lleva a un entrenamiento más rápido pero puede ocasionar oscilaciones en superficies del error de alta curvatura o incluso puede provocar el estancamiento en un mínimo local y no global.

Momento estándar

Para atenuar estos posibles problemas, además de acelerar la convergencia, Rumelhart et al. [23] introducen en la ecuación de la actualización de los pesos (3.8) un término, denominado *momento*, de la siguiente forma:

$$w(n) = w(n-1) - \eta \frac{\partial E}{\partial w}[n] + \mu \Delta w(n-1), \tag{3.11}$$

donde $\mu > 0$ es el correspondiente parámetro del momento[2]. Este método recibe el nombre de *método del momento estándar.*
La innovación de la inclusión del término $\mu\Delta w(n-1)$ es considerar la información del gradiente obtenida en iteraciones anteriores. De esta forma, se tiende a acelerar el descenso en direcciones similares en iteraciones consecutivas (gradientes consistentes) y, por otro lado, estabiliza en el caso de que se tengan en iteraciones consecutivas direcciones con oscilaciones de signos. Por tanto, se evita oscilaciones (inestabilidad) en "valles" de la superficie de error a la par que acelera la convergencia en regiones con poca pendiente. Así, esta idea de actualización de los pesos se basa en el gradiente para modificar la 'velocidad' del vector de pesos en vez de su 'posición', como hace el algoritmo original. Mostramos a continuación estas ideas de forma analítica.

Reescribiendo la expresión (3.11) se tiene, en este caso, la siguiente expresión equivalente

$$\Delta w(n) = w(n) - w(n-1) = -\eta \sum_{k=0}^{n} \mu^{n-k} \frac{\partial E}{\partial w}[k] \tag{3.12}$$

$$= -\eta \frac{\partial E}{\partial w}[n] - \eta \sum_{k=0}^{n-1} \mu^{n-k} \frac{\partial E}{\partial w}[k], \tag{3.13}$$

Así, la expresión del método del momento estándar se forma incluyendo en la expresión del gradiente descendente original (3.8) un sumando que es una serie de gradientes en iteraciones anteriores. Para que la serie converja, el parámetro momento μ tiene que cumplir que $|\mu| < 1$.

Si el término $\partial E/\partial w$ tiene el mismo signo en iteraciones sucesivas, entonces se acelera el descenso de manera estable hacia el mínimo, debido al sumando de la serie que acumula el valor del gradiente en iteraciones anteriores. En caso contrario, si el signo del gradiente varía en iteraciones consecutivas (trayectorias en zig-zag), entonces la serie desempeña el papel de estabilizador, evitando las posibles oscilaciones.

Si x en un mínimo local, entonces el gradiente en el punto x es cero. En dicho caso, con el algoritmo original (3.8) no se produciría actualización en los pesos, permaneciendo estancado en ese punto en las iteraciones sucesivas. El momento estándar (3.13) evita este estancamiento gracias al sumatorio de los gradientes acumulados.

Momento Nesterovs

Aunque hemos visto que el método del momento estándar mejora la versión original del algoritmo de backpropagation, existe todavía una versión que mejora al método del momento estándar. Esta versión mejorada introducida por Nesterovs [18] recibe el nombre de *Nesterovs Momentum* (Momento Nesterovs). Esta variante del algoritmo es la que usamos en nuestra aplicación.

La idea mejorada que subyace en este método es la de "corregir el error después de haberlo realizado". Esto es, en vez de calcular primero el gradiente y realizar después la actualización, en este método se da primero un salto en la dirección del gradiente acumulado $\mu\Delta\mathbf{w}(n-1)$, se calcula el gradiente en el punto donde ha terminado y se realiza después la actualización. De esta forma, la actualización de los pesos podría expresarse de la siguiente manera

$$\mathbf{w}(n) = \mathbf{w}(n-1) + \mu\Delta\mathbf{w}(n-1) - \eta\nabla E\left(\mathbf{w}(n-1) + \mu\Delta\mathbf{w}(n-1)\right), \tag{3.14}$$

donde $\nabla E\left(\mathbf{w}(n) + \mu\Delta\mathbf{w}(n-1)\right)$ es ahora el gradiente de E en el punto $(\mathbf{w}(n-1) + \mu\Delta\mathbf{w}(n-1))$. Hinton et al. [12] introducen estos métodos de momento (momento estándar y Nesterovs) con una notación equivalente y estudian sus relaciones.

[2]Para $\mu = 0$ la actualización de pesos sería la original (3.8).

La siguiente figura ilustra la actualización de los pesos con el método del momento estándar y Nesterovs:

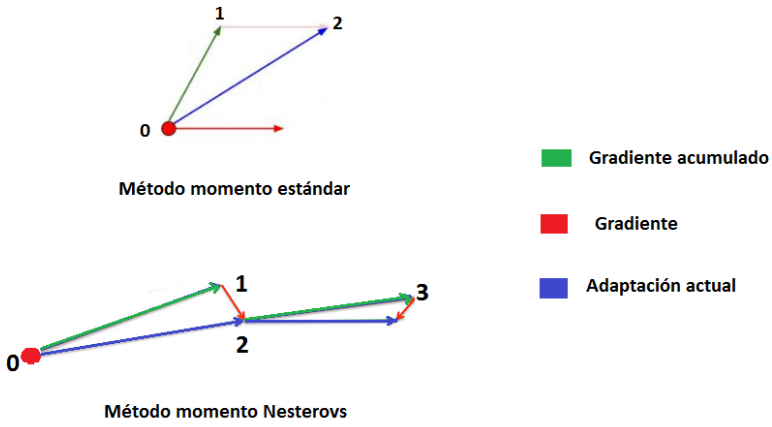


Figura 3.3: Método del Momento estándar vs Nesterovs

El punto rojo representa el peso que se va a actualizar. En el método del Momento estándar, se calcula el gradiente en dicho punto y se suma al gradiente acumulado. En el método Nesterovs, primero se da un salto en la dirección del gradiente acumulado (fecha verde) y después se calcula el gradiente (fecha roja). El nuevo peso será la suma de dicho gradiente con el gradiente acumulado.

Tasa de aprendizaje y término de momento

No existe una regla explícita que determine los valores óptimos para la tasa de aprendizaje η y el término momento μ, sino que su valor óptimo depende del problema a modelizar. Suelen establecerse por prueba y error. Sin embargo, algunos autores ofrecen un rango de valores para estos parámetros que pueden servir de guía. Por ejemplo, Bonifacio Martín del Brío y Alfredo Sanz [19] exponen que se suele tomar en la práctica un valor de μ próximo a uno ($\mu \approx 0.9$), que es el valor que usamos en nuestra aplicación, y la tasa de aprendizaje η un valor entre 0 y 1. Por otro lado, los autores de la librería `deeplearning4j` [4] recomiendan el intervalo $[10^{-6}, 0.1]$ como rango de valores para la tasa de aprendizaje η. En el trabajo se han entrenado diferentes redes neuronales con diferentes tasas de aprendizaje comprendidas en dicho intervalo.

Métodos con tasas de aprendizaje adaptativas

Hasta ahora, los métodos que se han expuesto mantienen una tasa de aprendizaje constante η.

Sin embargo, existen métodos, variantes del algoritmo de backpropagation original, que lo mejoran bajo la idea de utilizar tasas de aprendizaje adaptativas según la evolución del error en cada iteración, sin necesidad de incluir el término momento.

Uno de estos métodos, que es el que aplicamos en la práctica, es el denominado *Rmsprop*, que fue propuesto por Geoff Hinton en [11], para el entrenamiento por *mini-batchs*. Este método introduce un parámetro α que Hinton sugiere fijarlo en 0.95, que es el valor que tomamos en la aplicación.

Se basa en la normalización del gradiente dividiendo por un término en función de los gradientes anteriores, de forma que reduce la tasa de aprendizaje si el gradiente alto y vi-

ceversa. De esta manera, evita los posibles problemas en el aprendizaje comentados. Es por tanto, otra alternativa de aprendizaje más a las anteriores.

Funciones de activación

Capa de salida

En la práctica, en la mayoría de los problemas de clasificación y, en particular en nuestro caso, la función de activación que se utiliza para la capa oculta es la *función softmax:*

$$f(x_j) = \frac{e^{x_j}}{\sum_i^n e^{x_i}}.$$

El uso extendido de esta función de activación se debe al hecho de que proporciona valores entre 0 y 1 y la suma sobre todos los x_j da 1, es decir, se puede ver como una probabilidad. De esta manera, las salidas de la red podemos interpretarlas como probabilidades de pertenencia a una clase.

Consideraciones

En el caso de clasificación en más de dos clases, la red neuronal clasificará la observación i en la clase j si el valor en la neurona de salida y_j es mayor que el resto de neuronas $y_k,\ k \neq j$. Esto es, la probabilidad de pertenecer a esa clase supera a la de pertenencia del resto de clases.

Para problemas de clasificación binaria, esta consideración no suele aplicarse en la práctica, sino que se suele establecer un *punto de corte* p de forma que la red neuronal clasifique la observación en la clase positiva (clase 1) siempre que el valor en la neurona de salida y_1 de la clase positiva cumpla que $y_1 > p$.

Nótese que, en los casos donde las clases no están equilibradas, como es el nuestro, donde hay muchos más tokens que no son nombres propios que tokens que sí lo son, la probabilidad de pertenencia a la clase mayoritaria puede tender a ser mayor en las observaciones. Por tanto, si se siguiera la misma dinámica que en el caso anterior, posiblemente la tendencia sería clasificar los tokens en la clase mayoritaria. Es por ello por lo que se debe optimizar un punto de corte (que puede ser distinto a 0.5) para la clasificación de las observaciones. Se estudia la capacidad de la red para clasificar en el conjunto de validación para distintos puntos de corte y se suele elegir el punto de corte que mejores resultados haya obtenido.

Capa oculta

Tradicionalmente, las funciones de activación que se han considerado para la capa oculta eran las funciones no lineales sigmoide (o logística) y tangente hiperbólica, cuyas expresiones, respectivamente, son las que siguen:

$$f_{sig}(x) = \frac{1}{1+e^{-x}}.$$
$$f_{tanh}(x) = \frac{e^x - e^{-x}}{e^x + e^{-x}}.$$

Sin embargo, estas funciones de activación suelen llevar al problema conocido como *"vanishing problem"* o *problema del desvanecimiento*, en español, como se expone a continuación.

Problema del vanishing

El problema de vanishing ocurre cuando el entrenamiento de la red neuronal se dificulta con métodos basados en el gradiente y backpropagation.

La gráfica de la función sigmoide es la siguiente

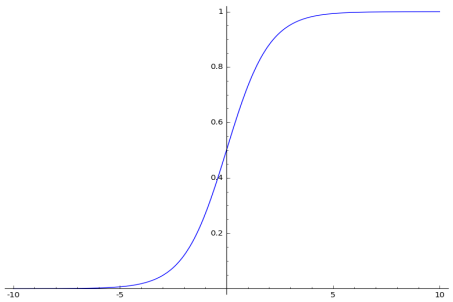


Figura 3.4: Función sigmoide

y su derivada tiene la siguiente expresión $f'_{sig}(x) = f_{sig}(x)(1 - f_{sig}(x))$.

Es decir, para valores de x cada vez más grandes, la función se acerca cada vez más a 1, y para valores de x muy pequeños, la función se acerca a 0. Por tanto, en estos casos, su derivada estará próxima a cero.

Debido a la regla de la cadena, para la actualización de los pesos en el algoritmo de backpropagation es necesario realizar el producto de las derivadas de las funciones de activación. Luego, para estos casos donde los x son relativamente grandes o pequeños, la actualización de los pesos puede ser minúscula, dando lugar a que los pesos no varíen prácticamente nada, provocando un aprendizaje lento en la red. Además, a medida que la red neuronal tiene más capas ocultas, este problema se vuelve peor pues la actualización de los pesos requiere de un mayor número de factores de estos pequeños números (derivadas de la función de activación); por lo que el gradiente decrece exponencialmente con el número de capas. Lo mismo ocurre para la función tangente hiperbólica, pues su derivada:

$$f'_{tanh}\left(\frac{x}{2}\right) = 2f'_{sig}(x).$$

Para solucionar este problema en el entrenamiento de la red, se hace uso de la función de activación denominada *RELU (rectified linear unit)*:

$$f_{relu}(x) = max(0, x) \tag{3.15}$$

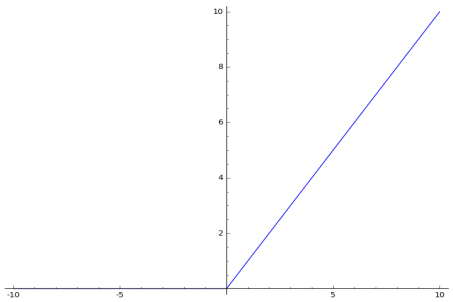


Figura 3.5: Función Relu

Su derivada es 1 si $x > 0$ y 0 si $x < 0$. Aunque teóricamente, la función relu no tiene derivada en el cero, la mayoría de las librerías tienen implementada la derivada en $x = 0$ como cero.

La ventaja del uso de esta función de activación, con respecto a las anteriores, es la forma de su derivada que evita el problema anterior, acelerando la convergencia del algoritmo.

Sin embargo, tienen el inconveniente de que para valores $x \leq 0$ la función de activación es cero y, por tanto, las neuronas no se activan.

Para evitar este posible problema de la función relu en los valores negativos, se puede considerar otra función, variante de ésta, denominada *leakyrelu*, que no toma el valor cero para todos los $x < 0$, si no que permite una pequeña pendiente negativa. Su expresión es la siguiente:

$$f_{lrelu}(x) = \begin{cases} \alpha x & \text{si } x < 0 \\ x & \text{si } x \geq 0 \end{cases}$$

donde $\alpha \approx 0{,}01$.

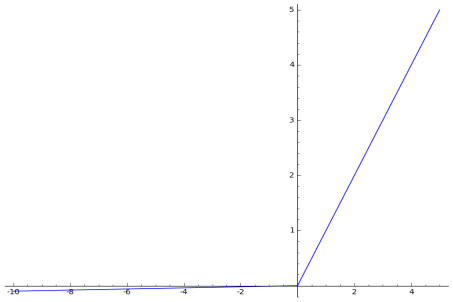


Figura 3.6: Función Leakyrelu

Otro problema que puede ocurrir es el caso contrario al problema del vanishing, esto es, que el gradiente tome valores muy altos, conocido como el *"exploding problem"*.

El método de tasas adaptativas Rmsprop, introducido antes, protege al entrenamiento de este problema, debido a la normalización que hace sobre los gradientes, así como del problema del vanishing, sin necesidad de introducir en la red las funciones relu o leakyrelu anteriores, como deduciremos en la aplicación práctica.

Este método, por tanto, tiene una gran aplicación en redes neuronales complejas, cuya tendencia a estos problemas es mayor y casi trivial, sobre todo en redes recurrentes LSTM.

Inicialización de los pesos

Para la aplicación del algoritmo de backpropagation, es necesario partir de unos valores iniciales en los pesos. La mala elección de estos pesos puede ser causa del conocido problema de *vanishing* y/o el problema de *exploding*. Por tanto, lo ideal es que los pesos no sean ni demasiado grandes ni demasiado pequeños.

Glorot y Bengio [6] propusieron una inicialización en los pesos denominada *Xavier*[3], muy extendida en la práctica, que evita los posibles problemas de vanishing y exploding (desvanecimiento o explosión en los gradientes, en español).

[3]Se debe al nombre del autor Xavier Glorot.

En la librería `deeplearning4j` se refiere a pesos Xavier como aquellos pesos w que son inicializados usando una distribución normal de media cero y varianza

$$Var(w) = \frac{1}{n_o + n_i},$$

donde n_i y n_o es el número de neuronas de la capa anterior y de la capa actual, respectivamente. En la inicialización Xavier original la varianza es $Var(w) = 2/(n_i + n_o)$.

Aunque esta inicialización de pesos funcione bien en la práctica, incluso con funciones de activación relu o leakyrelu, en [10] se propone una inicialización de pesos, que denominamos *relu*, que es una extensión de la fórmula de inicialización de pesos Xavier para las funciones relu.

Otras inicializaciones pueden ser, por ejemplo, una inicialización de pesos a valor 0, inicialización de pesos uniforme y normalizados.

Número de neuronas

La elección del número de neuronas óptimo suele ser por prueba y error. No existen reglas teóricas que nos permitan establecer el número determinado de neuronas para las capas ocultas, pero sí existen diversas reglas aproximadas que nos permiten seleccionar a priori un número de neuronas en las capas ocultas y que han proporcionado buenos resultados en la práctica. La que utilizamos en el trabajo es la siguiente:

- *Regla de la pirámide geométrica:* Se basa en la suposición de que el número de neuronas en la capa oculta debe ser menor que el número de neuronas en la capa de entrada pero mayor al número de neuronas de la capa de salida. Si denotamos por n al número de neuronas en la capa de entrada, m al número de neuronas en la capa de salida, entonces, para el caso de las redes neuronales con una sola capa oculta, la fórmula que proporciona un número de neuronas en la capa oculta, h, es la siguiente

  $$h = \sqrt{nm}.$$

  Para las redes neuronales con dos capas ocultas, si denotamos por r a la siguiente expresión $\sqrt[3]{\frac{n}{m}}$, por h_1 al número de neuronas para la primera capa oculta y por h_2 al número de neuronas para la segunda capa oculta, las expresiones para el número de neuronas son las siguientes:

  $$h_1 = mr^2$$
  $$h_2 = mr$$

Una mala elección del número de capas y neuronas puede conducir a problemas en el entrenamiento.
Si la red neuronal que se considera cuenta con un número demasiado pequeño de neuronas en la capa oculta, puede ocurrir que la red no aprenda demasiado de los datos de entrenamiento (*underfitting*). Pero, si por el contrario, cuenta con un número excesivo de neuronas, puede producirse el problema contrario conocido como *overfitting* o sobreentrenamiento.

Regularización

El sobreentrenamiento puede ocurrir cuando el número de parámetros es excesivo o el número de épocas es elevado. Este problema se puede solucionar con diferentes medidas que denominamos *medidas de regularización*.

En [7] los autores definen como *regularización* a "cualquier modificación que hagamos a un algoritmo de aprendizaje que tenga la intención de reducir su error de generalización pero no su error de entrenamiento".

Estas medidas de regularización pueden afectar directamente a la función del error, a la relación entre las neuronas o al error en el conjunto de validación. Exponemos a continuación las medidas de regularización que consideramos en la aplicación.

Regularización L1-L2

Esta regularización busca evitar el sobreajuste añadiendo a la función de error un sumando de pesos. Esto equilibra el valor de los pesos en función del valor en la función del error, resultando a menudo en una penalización de pesos grandes.

La regularización L1 añade a la función de error el siguiente sumando:

$$\lambda_1 \sum_i |w_i|$$

donde el parámetro λ_1 representa la importancia de este sumando y w_i el peso.

Por otro lado, la regularización L2 añade a la función del error el siguiente sumando:

$$\lambda_2 \sum_i w_i^2$$

La regularización L1 es más tolerante a pesos lejanos del cero que la L2.

Los valores más comunes que suelen considerarse en la práctica para los parámetros λ_1 y λ_2 están en el intervalo $[10^{-6}, 10^{-3}]$.

Dropout

Aunque esta técnica trabaja de forma diferente a las dos anteriores, el objetivo es el mismo: evitar el sobreentrenamiento. Sin embargo, esta técnica no afecta directamente a la función objetivo sino a la estructura de la red.

Consiste en la eliminación de las neuronas de la red, al menos temporalmente, forzando a la red a trabajar con parte de sus neuronas. De esta manera las neuronas se convierten menos dependientes unas de otras.

Se eliminan de forma aleatoria con una probabilidad p, que se suele fijar en 0,5, aunque puede variar en función de la capa que se considere. Por ejemplo, se sugiere probabilidades más altas en las capas ocultas que en la de entrada y de salida. En nuestro caso, no se considera esta medida en las capas de entrada y de salida.

Más detalles de estas técnicas pueden consultarse en [7].

Early-stopping

El gradiente descendente es un método iterativo que termina con un criterio de parada establecido. El criterio de parada que se suele establecer en el entrenamiento es un número determinado de épocas. Si el número de épocas es suficientemente grande se puede producir

sobreentrenamiento, es decir, a partir de un número de épocas, el error en el conjunto de entrenamiento sigue disminuyendo pero en el de validación comienza a aumentar. El algoritmo conocido como *early-stopping* evita el sobreentrenamiento tomando un número de épocas óptimo, que será cuando el error en el conjunto de validación es mínimo.

Así, el algoritmo base de early-stopping se puede describir de la siguiente manera: Se fija un número de épocas n. Para cada época, se evalúa el error en el conjunto de validación y se guarda el modelo generado. Una vez transcurridas las n épocas se termina el entrenamiento. El modelo resultante será el que menor error haya cometido en el conjunto de validación. De esta forma, se elige el modelo en el punto de menor error. En la siguiente imagen se muestra la idea de forma ilustrativa:

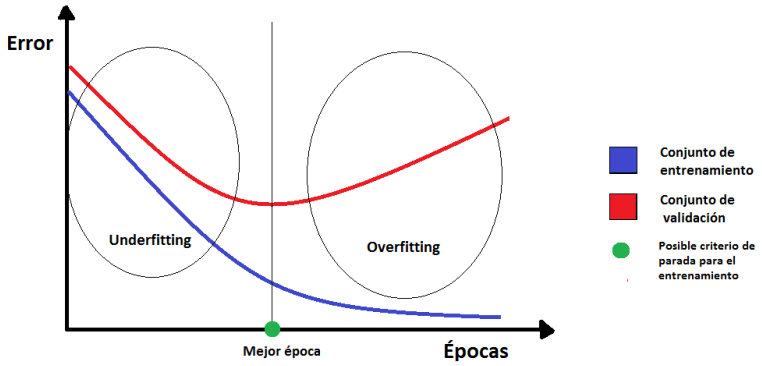


Figura 3.7: Early-stopping

Este algoritmo permite variaciones como decidir cada cuándo evaluar el conjunto de validación, terminar el algoritmo cuando hayan transcurrido k épocas sin que el error en el conjunto de validación haya disminuido o terminar el algoritmo después de un tiempo transcurrido fijado.

Esta técnica nos permite tener una idea acerca de las épocas que son necesarias para un entrenamiento adecuado de la red neuronal, es decir, puede ayudarnos a elegir el número de épocas óptimo.

A lo largo de esta sección se han explicado los parámetros que influyen en el entrenamiento del modelo de red neuronal. En la aplicación del trabajo se ha considerado la selección ideal de dichos parámetros a través del entrenamiento con distintos valores, como veremos en los capítulos siguientes.

3.2. Redes recurrentes

Los humanos no comenzamos a pensar desde cero cada segundo, sino que nuestros pensamientos perduran en el tiempo. Con la idea de imitar esta "memoria" que tiene el cerebro humano surgen las que se denominan *redes neuronales recurrentes*. Estas redes tienen su aplicación sensata en datos que presentan cierta secuencialidad, como pueden ser las palabras de un texto bien organizado, donde el significado de una palabra depende del contexto de la frase.

Como se comentó al inicio del capítulo, las redes neuronales recurrentes, a diferencia de las redes de propagación hacia adelante, como el perceptrón multicapa, permiten que las salidas

de las neuronas sean también entradas de capas anteriores o incluso de su propia capa. Estos bucles son los que permiten que la información persista, consiguiendo en la red esa capacidad de "memoria".

Un ejemplo de red recurrente estándar es el que se ilustra en la siguiente figura, donde la salida de la capa oculta para la observación t es variable de entrada para la observación $t+1$.

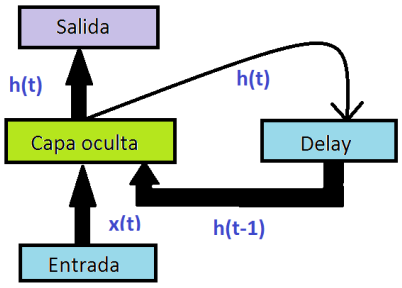


Figura 3.8: Red recurrente simple

Aunque pueda parecer que las redes recurrentes están lejos de las redes neuronales de propagación hacia adelante, pueden verse como múltiples copias de estas redes conectadas entre sí:

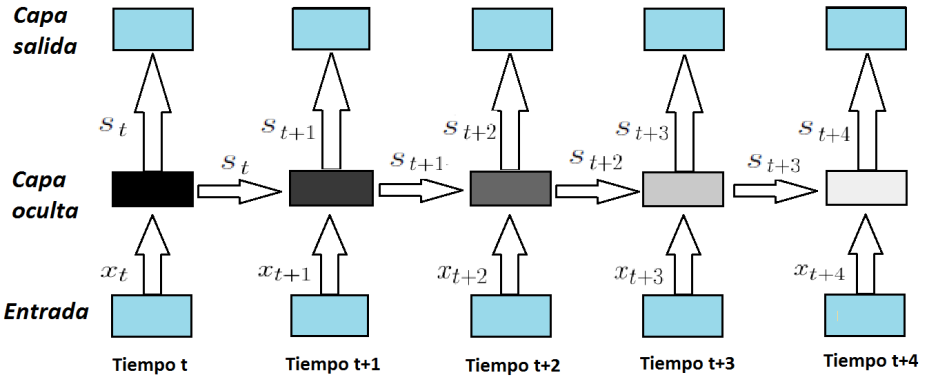


Figura 3.9: Red recurrente: MLP en paralelo

Por tanto, la única diferencia con respecto al perceptrón multicapa es el cálculo de la capa oculta, donde será necesario introducir la variable adicional 'tiempo' t.[4] La salida de la capa oculta en el tiempo t se calcularía de la siguiente manera:

$$s_t = f(Ux_t + Ws_{t-1}) \quad t = 1, \ldots, n, \tag{3.16}$$

siendo n el número de observaciones, x_t la salida de la capa anterior, U y W las matrices de pesos a estimar y s_{t-1} la salida de la capa oculta en el tiempo anterior $(s_0 = 0)$.

Sin embargo, aunque las redes recurrentes convencionales tengan esta capacidad de "memoria", en la práctica, estas redes tienen la limitación de no aprender dependencias a largo plazo, es decir, se pierde la información de tiempos pasados en la red a largo plazo.

Lo que ocurre se ilustra en la Figura 3.9. El sombreado indica la sensibilidad de la red a las entradas en el tiempo t, siendo más oscuras en los tiempos más cercanos. La sensibilidad

[4]En nuestro caso, la variable t denotará cada token del texto.

disminuye a medida que se introducen nuevas entradas, sobreescribiendo las activaciones de la capa oculta, "olvidando" así las primeras entradas.

Con la idea de conseguir esta dependencia a largo plazo, Hochreiter y Schmidbuber [13] introducen las denominadas *redes recurrentes LSTM* (siglas en inglés de memoria a corto y largo plazo), que son las que utilizamos en el trabajo práctico.

3.2.1. Redes recurrentes LSTM

Una red neuronal recurrente LSTM puede verse como una red recurrente estándar donde los nodos de la capa oculta son reemplazados por lo que se denominan *celdas de memoria*, que son las unidades que permiten que se almacene la información a corto y largo plazo, siendo este último el gran avance de estas redes.

Para estas redes, la función de activación en la capa oculta que suele considerarse, y que consideramos en nuestra aplicación, es la función de activación tangente hiperbólica y el método de aprendizaje el método de tasas adaptativas Rmsprop.

El cálculo de la unidad oculta en una capa oculta LSTM podría interpretarse como un mecanismo de compuertas que deciden qué información "dejan pasar" y qué información "olvidan". En estos términos, se consideran 3 puertas: la puerta de entrada, la puerta de 'olvido' y la puerta de salida. Las expresiones para las tres puertas en el tiempo $t \geq 1$ son las siguientes:

$$i_t = \sigma\left(U^i x_t + W^i s_{t-1}\right) \tag{3.17}$$

$$f_t = \sigma\left(W^i x_t + W^f s_{t-1}\right) \tag{3.18}$$

$$o_t = \sigma\left(U^o x_t + W^o s_{t-1}\right) \tag{3.19}$$

donde U, W son las matrices de pesos que deberemos estimar, x_t el vector de entrada a la capa, s_{t-1} el vector de salida de la capa en el tiempo $t-1$ y σ representa a la función sigmoide. Así, la puerta de entrada i_t, la de olvido f_t y la de salida o_t tomarán valores entre 0 y 1.

La puerta de entrada es la que decide qué información al estado oculto deja pasar, la puerta de olvido es la que decide qué parte de la "memoria" no olvida y la puerta de salida es la que decide qué información deja salir de la capa oculta. En otras palabras, la puerta de entrada es la que permite el acceso a la capa oculta, la puerta de olvido la que permite la permanencia en la capa oculta y la puerta de salida la que permite el acceso de salida. Con este mecanismo de compuertas, la red permite que la información perdure en el tiempo.

Junto con las ecuaciones anteriores (3.17)-(3.19), las siguientes expresiones representarían el comportamiento general de las capas ocultas LSTM:

$$g_t = \tanh\left(U^g x_t + W^g s_{t-1}\right) \tag{3.20}$$

$$c_t = c_{t-1} * f_t + g_t * i_t \tag{3.21}$$

$$s_t = \tanh(c_t) * o_t \tag{3.22}$$

donde c_t representa la celda de memoria interna en el tiempo t, g_t la 'posible' entrada al estado oculto y s_t la salida del estado oculto en el tiempo t, donde $c_0 = s_0 = 0$. Denotamos por $*$ a la multiplicación componente a componente de vectores.

Para facilitar la comprensión, ilustramos el comportamiento de la capa oculta LSTM en la siguiente figura:

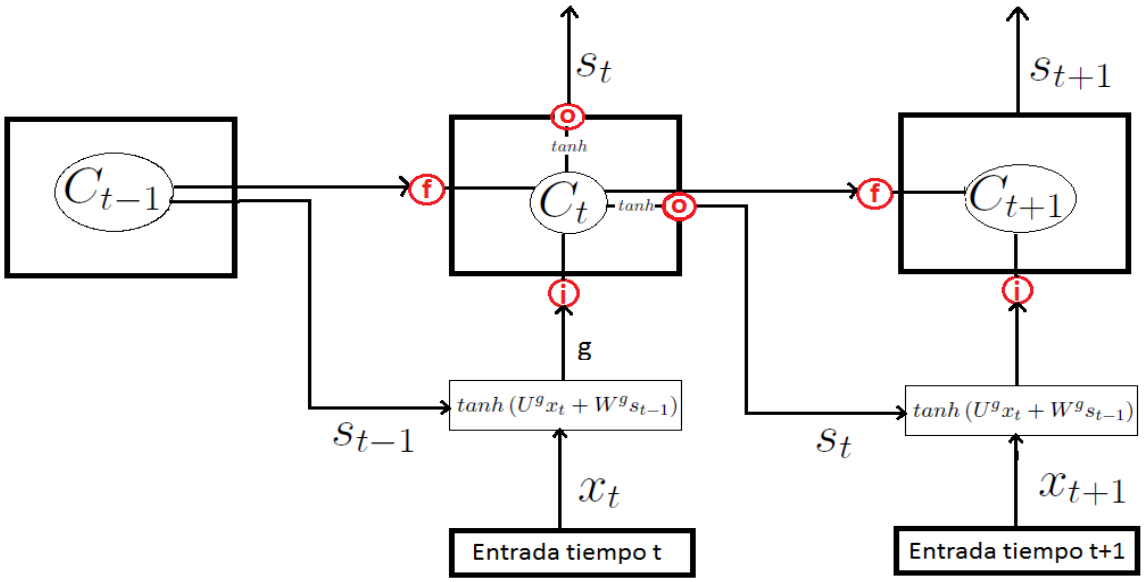


Figura 3.10: Cálculo celda de memoria interna

Los círculos rojos representan el sistema de compuertas (3.17)-(3.19) y g la 'posible' entrada a la capa oculta, que viene dada por la expresión (3.20). La 'entrada' g se multiplicará elemento a elemento por la puerta de entrada i que reducirá su valor a un rango entre 0 y 1. El resultado de esta multiplicación se podría interpretar como 'la información que deja entrar' de g.
La celda de memoria interna C_t es la que almacena la información en el tiempo. Se formará con parte de la información de tiempos anteriores (memoria) y de la nueva información que ha dejado pasar. La celda de memoria interna en el tiempo anterior C_{t-1} se multiplica elemento a elemento por la puerta de olvido f. El resultado de esta multiplicación se podría interpretar como 'la memoria que perdura', es decir, la memoria que deja que siga permaneciendo en la celda de memoria. La nueva celda de memoria interna C_t será el resultado de sumar las dos cantidades anteriores: la información que perdura de tiempos pasados y la información actual que ha dejado pasar (ecuación (3.21)). Hasta ahora tendríamos calculada la nueva celda de memoria interna C_t y nos faltaría calcular la salida de la capa.
A esta celda C_t se le aplicará la función tangente. El resultado se multiplicará, elemento a elemento, por la puerta de salida o y esta información será la que deja pasar a capas superiores, que se denota por s_t. Nótese que la salida de la capa oculta s_t formará parte también de la nueva entrada al estado oculto en el tiempo siguiente, como se ilustra en la Figura 3.10 y como ocurre en la red recurrente estándar (3.16).

Como se ha dicho, el gran avance de estas redes es la propiedad de memoria a largo plazo y esto se consigue gracias al sistema de puertas (3.17)-(3.19). Consideremos el siguiente ejemplo extremo. Diremos, por comodidad, que una puerta está totalmente abierta o totalmente cerrada cuando toma el valor 1 y 0, respectivamente.

Supongamos que la puerta de olvido está totalmente abierta en los tiempos $t = 1, \dots, 4$ y las puertas de entrada y de salida solo se abren totalmente en el tiempo $t = 1$ y $t = 4$, respectivamente. Entonces, la información completa de la entrada a la capa oculta en el tiempo $t = 1$ perdurará hasta el tiempo $t = 4$.

La equivalencia de estas redes a las redes recurrentes estándar sería el caso extremo de considerar las puertas de entrada y de salida totalmente abiertas y la puerta de olvido

totalmente cerrada.

Retropropagación a través del tiempo

El algoritmo de entrenamiento para las redes recurrentes es el denominado algoritmo de Retroprogación a través del tiempo o *Backpropagation Through Time (BPTT)*, en inglés. Extiende al algoritmo de backpropagation original adaptándose a la arquitectura de las redes recurrentes. Los detalles del algoritmo pueden verse en [8] y [25].

3.3. Word2Vec

A diferencia de los algoritmos presentados en las secciones anteriores, cuyo objetivo es la clasificación de nombres propios (reconocimiento de entidades nombradas), en esta sección introducimos la herramienta *Word2vec* que permite la codificación de las palabras en un vector continuo. Haremos uso de esta herramienta en la aplicación de nuestra principal tarea de estudio (capítulo 5), donde dicha codificación de las palabras formará parte de los inputs de las redes neuronales.

Como bien se ha adelantado, Word2vec es una herramienta, introducida por Mikolov en 2013 [16], para el cálculo de representaciones continuas de palabras.

Los modelos que introduce Mikolov para la representación de las palabras son los denominados *Skip-gram* y *CBOW* (bolsa continua de palabras). Estos modelos son modelos de redes neuronales de una sola capa oculta.

La idea que subyace en ambos modelos es similar: definir una ventana simétrica (contexto) en torno a una palabra central y plantear un problema de optimización basado en predecir la palabra central dado el contexto o viceversa. En la siguiente figura se muestra la arquitectura de ambos modelos para una ventana de tamaño 2.

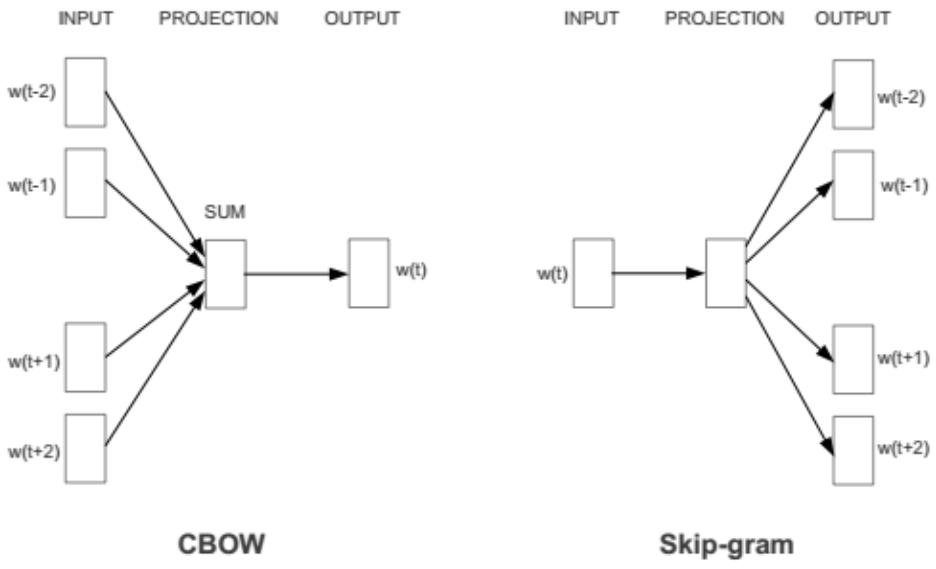


Figura 3.11: Representación gráfica de los modelos Skip-gram y CBOW

Sin pérdida de generalidad, explicaremos en lo que sigue el modelo de Skip-gram que nos permitirá la representación de los tokens de nuestros conjuntos de datos.

3.3.1. Skip-gram

Sean $w_1, w_2, \ldots, w_T$ los T tokens del texto. El objetivo del skip-gram será maximizar la siguiente expresión:

$$\frac{1}{T}\sum_{t=1}^{T}\left[\sum_{j=-k}^{k}\log p(w_{t+j}|w_t)\right], \tag{3.23}$$

donde k es el tamaño de la ventana.

Supongamos que tomamos N neuronas para la capa oculta y un tamaño de ventana $k = 2$, esto es, dada una palabra central w_i tomamos las dos palabras anteriores (w_{i-2}, w_{i-1}) y las dos palabras posteriores (w_{i+2}, w_{i+1}) del texto como output.

El input de la red neuronal es la codificación *one-hot* de la palabra central. Así, suponiendo que la palabra central w_i es aquella que ocupa el k-ésimo lugar en el vocabulario ordenado, el input de la red neuronal será un vector de dimensión $T \times 1$ con el valor 0 en todas las componentes excepto en la k-ésima, donde tomará el valor 1:

$$w_i^t = [0, \ldots, \overbrace{1}^{(k)}, \ldots, 0].$$

Denotamos por $\mathbf{W}_I$ a la matriz de pesos de la capa de entrada y la capa oculta, de dimensiones $T \times N$ y por $\mathbf{h}$ al vector de la capa oculta de dimensión $N \times 1$. Dado que la función de activación de la capa oculta es la identidad, el vector resultante $\mathbf{h}$ será la k-ésima fila de la matriz de pesos $\mathbf{W}_I$, que denotamos por $\mathbf{u}_{w_i}$:

$$\mathbf{h} = \mathbf{W}_I^t w_i = \mathbf{u}_{w_i}^t.$$

La función de activación de la capa oculta es la función softmax que nos permitirá obtener la probabilidad condicionada de (3.23). Denotamos por $\mathbf{W}_O$ a la matriz de pesos de la capa oculta y la capa de salida, de dimensiones $N \times T$, matriz que comparten todos los outputs del contexto. De esta forma, denotando por $\mathbf{v}_j$ la j-ésima columna de la matriz $\mathbf{W}_O$, se tiene que la j-ésima componente del vector de salida será el resultado de aplicar la función softmax a la siguiente expresión:

$$\nu_j = \mathbf{v}_j^t \mathbf{h}.$$

Así,

$$y_{i-1,j} = p(w_{i-1}|w_i) = \frac{e^{\nu_j}}{\sum_{i=k}^{T} e^{\nu_k}},$$

denotaría la probabilidad de que la palabra que ocupa el lugar j-ésimo del vocabulario ordenado sea la palabra anterior a w_i en el texto (w_{i-1}).

Debido a que el cálculo de la función softmax se realiza sobre la suma de todo el vocabulario T, esto puede causar problemas en la eficiencia computacional. En [17] Mikolov et al. consideran dos alternativas a la función softmax con una mejora computacional (*softmax jerárquico* y *negative sampling*).

Tras optimizar la función objetivo obtendríamos el valor estimado de los pesos. Las filas de la matriz de pesos estimada $\mathbf{W}_I$ se corresponderían con la codificación continua Word2vec de las T palabras del texto, cuyas componentes (columnas de la matriz) son el número de neuronas que han fijado en la capa oculta.

Más detalles sobre el algoritmo Skip-gram y el CBOW pueden verse en [21].

La codificación continua de palabras Word2vec es capaz de capturar la similitud de palabras de manera que dos palabras similares estarán cercanas en el espacio n-dimensional de palabras. Además, es capaz de capturar información semántica y sintáctica de ellas. En [15] y en [17] muestran resultados empíricos que demuestran que esta representación de palabras captura dicha información.

Por tanto, esta representación continua de palabras ofrecería información útil de las palabras en función del contexto.

En resumen, la herramienta Word2vec permite representar cada palabra en un espacio $n-$dimensional, obtenido de los pesos asociados a las neuronas ocultas en función de su contexto, capturando información sintáctica y semántica de ellas.

Capítulo 4

Clasificación binaria de nombres propios

" Mientras los filósofos discuten si es posible o no la inteligencia artificial, los investigadores la construyen."
C.Frabetti

Este capítulo está dirigido a la primera de nuestras tareas de estudio que es el reconocimiento de entidades nombradas (NER). Es una tarea de clasificación binaria que consiste en clasificar los tokens de un texto en dos clases: la clase de nombres propios y la clase de 'no nombres propios'.

Denotamos por clase positiva (clase 1) a la clase de los nombres propios y a la clase negativa (clase 0) a la clase de los 'no nombres propios'.

Los algoritmos de redes neuronales que se aplican en este primer estudio de clasificación binaria son las redes neuronales *feedforward* (redes monocapa y perceptrón multicapa), introducidas en el capítulo anterior (capítulo 3).

La librería que usamos tanto en este capítulo como en el siguiente, que nos permite la implementación de los algoritmos de redes neuronales, como se dijo en el capítulo 2, es la librería `deeplearning4j` con las modificaciones que comentamos a continuación. El código implementado de su clase principal de `Java` puede verse en Anexos.

El programa tomaba para la clasificación binaria, como punto de corte por defecto el 0.5, no consideraba otras opciones, pudiendo infravalorar los valores de clasificación óptimos. Se modificó el código fuente de la librería para que proporcionase las métricas considerando diferentes puntos de corte y el mejor punto de corte según la métrica que se considerase. Por otro lado, las métricas que proporcionaba la librería en su origen eran la media de la *precision*, la media del *recall* y la media de la *f-measure* sobre todas sus clases. Se modificó el código para que proporcionase estas métricas para las distintas clases y poder así interpretar por separado los resultados en cada clase.

La métrica que se decidió considerar para la validación de los modelos y selección de los más adecuados, como se argumentó en el capítulo 1, es la ***f-measure*** de la clase positiva. No consideramos el estudio sobre la *f-measure* de la clase negativa (no nombres propios) puesto que sus valores en ella son significativamente altos, cercanos al 1, y muy similares en los

distintos modelos entrenados. Es decir, los modelos clasifican muy bien la clase negativa.

Presentamos a continuación las características del servidor que utilizamos para la aplicación de los algoritmos, tanto para este capítulo como para el siguiente:

Número de CPU	28
Tamaño de caché	35 MB.
Número de cores	14
Memoria	386 GB.
Velocidad CPU	1200 MHz.

Tabla 4.1: Características del servidor

El capítulo se organiza como sigue. Primero se presentan los datos sobre los que se realiza el estudio. Después se introducen los diferentes parámetros, funciones y algoritmos que se han considerado para el entrenamiento de las diferentes redes neuronales *feedforward*, y se presentan aquellas que mejores resultados han obtenido para el punto de corte óptimo en el conjunto de validación. Se interpretan los resultados y, al final del capítulo, se exponen unas consideraciones generales sobre el análisis realizado.

Presentación de los datos

Nuestro conjunto de entrenamiento esta compuesto de 100 frases del corpus `Ancora` (corpus introducido en el capítulo 2), y el conjunto de validación de otras 100 frases de este corpus, distintas a las anteriores.

Lo ideal, para el análisis de los datos, es que las proporciones de las clases estén aproximadamente equilibradas en el conjunto de entrenamiento y de validación, o no varíen demasiado de un conjunto a otro. En nuestro caso, el porcentaje de nombres propios en el conjunto de entrenamiento es de casi el 10 % y en el conjunto de validación de algo más del 8 %, luego no varía demasiado.

Para cada token, se generaron las variables introducidas en el capítulo 2 y, tras aplicar un *cutoff* de 10, se dispone de 467 variables. El número de tokens en nuestro conjunto de entrenamiento de 100 frases es de 4102, luego disponemos de una matriz numérica de dimensión 4102×468.

4.0.2. Entrenamiento

Recuérdese que el tipo de entrenamiento que usamos en nuestra aplicación, como se comentó en el capítulo anterior (capítulo 3), es el algoritmo de backpropagation por estrategia de *mini-batch*. Por tanto, es necesario, además de los parámetros y funciones propias de la red neuronal, fijar un tamaño para estos *mini-batch*. El tamaño que se ha fijado para el conjunto de entrenamiento y el de validación es de 100, esto es, cada *mini-batch* constará de 100 tokens.

El entrenamiento de la red termina una vez que se haya completado un número de épocas, que fijamos en 20. Remarcar que, tanto el tamaño del *mini-batch* como el criterio de parada (número de épocas) son algunos de los parámetros que deciden en gran medida el tiempo computacional del entrenamiento. Nótese también que la dimensión de nuestra matriz de datos es considerable, luego el entrenamiento de las redes neuronales necesita de un número elevado de cálculos.

Se empezaron considerando diferentes configuraciones para redes neuronales sin capa oculta, con una capa oculta y con dos capas ocultas. Los diferentes parámetros, funciones y algoritmos que se consideraron para las configuraciones de las diferentes redes neuronales que entrenamos se dan en la siguiente tabla:

Parámetros	
Función de pérdida	-Logaritmo de la verosimilitud
Algoritmo de optimización	-Nesterovs ($\mu = 0{,}9$)
Ratio de aprendizaje	η=0.05, 0.08, 0.1
Función de activación	-Capa oculta: Relu, Sigmoide -Capa salida: Softmax
Medidas de regularización	-$l_2 = 10^{-3}, 10^{-4}$ -Dropout ($p = 0{,}5$)
Pesos iniciales	-Xavier -Ceros -Distribución normal -Distribución uniforme
Nº de neuronas en las capas ocultas	5, 7, 10

Tabla 4.2: Parámetros para las diferentes redes *feedforward* entrenadas

El tiempo computacional empleado en el entrenamiento de los diferentes modelos entrenados se encuentra en el rango de 12 y 24 horas.

4.0.3. Validación

En esta sección se presenta la validación de los modelos generados. Los resultados de los mejores modelos obtenidos para el punto de corte óptimo, así como la configuración de la red para los diferentes tipos de redes (monocapa, una capa oculta y dos capas ocultas) se presentan a continuación.

Red monocapa

Las siguientes tablas muestran la configuración de la red monocapa que mejor resultado ha obtenido en la métrica *f-measure* de la clase positiva, así como los valores de ésta para los diferentes puntos de corte probados.

Configuración de la red monocapa	
Función de pérdida	-Logaritmo de la verosimilitud
Algoritmo de optimización	-Nesterovs ($\mu = 0{,}9$)
Ratio de aprendizaje	η=0.05
Función de activación	-Capa salida: Softmax
Medidas de regularización	Ninguna
Pesos iniciales	-Xavier

Tabla 4.3: Configuración de la red monocapa con mejores resultados

PtoCorte	0.1	0.2	0.3	0.4	0.5	0.6	0.7	0.8	0.9
F-measure(1)	0.8215	0.859	0.8762	0.8971	0.8938	**0.8972**	0.8911	0.88205	0.8415

Tabla 4.4: Métrica *f-measure* para los diferentes puntos de corte

Aunque, en general, todos los puntos de corte presentan valores de *f-measure* que son satisfactorios, el punto de corte óptimo es el 0.6. El punto de corte se interpreta como la probabilidad de pertenencia a la clase positiva. Si ésta es mayor de 0.6, el token se clasifica en dicha clase.

Nótese, por otro lado, que el mejor modelo clasificatorio se ha obtenido sin utilizar ninguna medida de regularización en el entrenamiento. Esto puede ser debido a que la red neuronal no tiene un excesivo número de parámetros (pesos) y que se han tomado una tasa de aprendizaje y un número de épocas no excesivamente altos.

En la siguiente tabla se muestra la matriz de confusión para el punto de corte óptimo 0.6:

		Estimada	
		Clase 0	Clase 1
Real	Clase 0	2277	12
	Clase 1	29	179

Figura 4.1: Matriz de confusión red monocapa

El modelo clasifica 12 tokens que no son nombres propios como nombres propios (falsos positivos) y, por otro lado, 'pierde' 29 nombres propios que clasifica como no nombres propios (falsos negativos).

Aunque a la vista de los resultados podríamos decir que esta red ofrece un modelo clasificatorio adecuado, se analizó si la incorporación de más capas ocultas (perceptrón multicapa) mejoraba los resultados en nuestro conjunto de validación.

Red con una capa oculta (MLP)

La configuración de la red del perceptrón multicapa con una capa oculta que mejor resultado ha proporcionado en la métrica *f-measure*, así como los valores de ésta para los diferentes puntos de corte, se muestra a continuación:

Configuración del MLP (una capa oculta)	
Función de pérdida	-Logaritmo de la verosimilitud
Algoritmo de optimización	-Nesterovs ($\mu = 0{,}9$)
Ratio de aprendizaje	η=0.1
Función de activación	-Capa oculta: Sigmoide -Capa salida: Softmax
Medidas de regularización	$l2 = 10^{-4}$
Pesos iniciales	-Xavier
Nº de neuronas en las capas ocultas	7

Tabla 4.5: Configuración del MLP con una capa oculta con mejores resultados

PtoCorte	0.1	0.2	0.3	0.4	0.5	0.6	0.7	0.8	0.9
F-measure(1)	0.8604	0.8862	0.8949	**0.9015**	0.8983	0.8945	0.8939	0.8814	0.8624

Tabla 4.6: Métrica *f-measure* para los diferentes puntos de corte

En general, los resultados de la *f-measure* de la clase positiva para los diferentes puntos de corte, es superior a los obtenidos con la red monocapa. El mejor valor de la métrica, en este caso, se da con el punto de corte 0.4.

Nótese que en este tipo de redes, los mejores valores sí que se dan con un entrenamiento que usa una medida de regularización, a diferencia del caso anterior. Esto no nos debería sorprender pues, la red es más compleja, con más parámetros, luego puede tener más problemas de sobreajuste.

En la siguiente tabla se muestra la matriz de confusión para el punto de corte óptimo 0.4:

		Estimada	
		Clase 0	Clase 1
Real	Clase 0	2274	15
	Clase 1	25	183

Figura 4.2: Matriz de confusión MLP con una capa oculta

Aunque hayan aumentado en 3 los falsos positivos (3 no nombres propios clasificados como nombres propios), hay 4 verdaderos positivos de más (nombres propios clasificados como tal) que con el modelo monocapa, proporcionando un mayor valor en la *f-measure*.

Una vez analizado el modelo con una capa oculta, se consideraron modelos con dos capas ocultas.

Red dos capas ocultas (MLP)

La configuración del perceptrón multicapa con dos capas ocultas que proporcionó un mayor valor en la *f-measure* para la clase positiva, así como los valores de ésta para los diferentes puntos de corte, se muestran en las siguientes tablas:

Configuración del MLP (dos capas ocultas)	
Función de pérdida	-Logaritmo de la verosimilitud
Algoritmo de optimización	-Nesterovs ($\mu = 0{,}9$)
Ratio de aprendizaje	η=0.1
Función de activación	-Capas ocultas: Sigmoide -Capa salida: Softmax
Medidas de regularización	$l2 = 1e^{-3}$
Pesos iniciales	-Xavier
Nº de neuronas en las capas ocultas	-Primera capa oculta: 10 -Segundo capa oculta: 5

Tabla 4.7: Configuración del MLP con dos capas ocultas una capa oculta con mejores resultados

PtoCorte	0.1	0.2	0.3	0.4	0.5	0.6	0.7	0.8	0.9
F-measure(1)	0.863	0.8867	0.891	0.8927	0.8971	**0.9037**	0.8955	0.8928	0.8861

Tabla 4.8: Métrica *f-measure* para los diferentes puntos de corte

El valor de la *f-measure* para el punto de corte óptimo es mayor que en los dos casos anteriores. Este valor óptimo se da en el punto de corte 0.6.

En la siguiente tabla se muestra la matriz de confusión para este punto de corte óptimo

		Estimada	
		Clase 0	Clase 1
Real	Clase 0	2274	14
	Clase 1	25	183

Figura 4.3: Matriz de confusión

donde se observa que el número de falsos positivos se reduce en uno, con respecto al modelo anterior.

4.0.4. Consideraciones generales

Exponemos a continuación las consideraciones generales del estudio realizado:

- En general, cabe destacar que los resultados obtenidos para las diferentes configuraciones de la red son muy buenos y, aunque el perceptrón multicapa con dos capas ocultas sea el modelo que mejores resultados ha obtenido, los resultados en los diferentes modelos son muy similares.
- Se validó el modelo que mejor resultado proporcionó (MLP con dos capas ocultas) con un texto de 1000 frases del corpus `Ancora`, distinto a los anteriores. El valor de la *f-measure* para la clase positiva en este conjunto es de **0.9098**. Esto es, obtiene un valor satisfactorio, incluso mejor que con la validación de 100 frases, demostrando así la generabilidad del modelo.
- Como conclusión final de este estudio podríamos decir que, con las variables consideradas, la tarea de clasificación binaria de reconocimiento de entidades nombradas es una tarea modelizable con algoritmos de redes neuronales, proporcionando buenos resultados de clasificación en nuestro conjunto de validación.

Este estudio nos da una primera aproximación a nuestra tarea principal de estudio, que es la clasificación multiclase de los nombres propios, que es una tarea más compleja. En el siguiente capítulo veremos cómo trabajan las redes neuronales para este problema de clasificación multiclase.

Capítulo 5

Clasificación multiclase de nombres propios

"No hay enigmas. Si un problema puede plantearse, también puede resolverse."
Ludwig Wittgenstein

En el capítulo anterior se han construido modelos que clasifican, de un texto dado, los tokens según sean nombres propios (NP) o no. En este capítulo, ampliamos nuestro estudio a una tarea más compleja de clasificación multiclase: detectamos los nombres propios clasificándolos en 4 categorías. Esta es la tarea principal de nuestro trabajo y, por tanto, es a la que más tiempo se ha dedicado. La librería y el servidor que se han utilizado para ella son los descritos en el capítulo anterior (capítulo 4).

Definimos las clases de nuestro estudio de la siguiente manera:

- Clase 0: No nombre propio.
- Clase 1: NP Organización.
- Clase 2: NP Miscelánea.
- Clase 3: NP Persona.
- Clase 4: NP Localización.

La clase de nombres propios del capítulo anterior se ramifica en cuatro clases (organización, miscelánea, persona y localización). En este capítulo, el estudio lo realizamos sobre la clasificación de estas cuatro clases.

Como en el estudio anterior, la métrica de evaluación que consideramos es la ***f-measure*** de las clases de nombres propios. Se realizó un estudio por separado de esta métrica para cada clase de nombres propios y para la selección de los modelos se consideró la media de ésta sobre todas estas clases. Así, decimos que un modelo obtuvo mejores resultados que otro cuando tuvo un mayor valor en la media de la *f-measure* sobre las clases de nombres propios.

El capítulo se organiza como sigue. Primero se presentan los datos sobre los que se realiza el estudio. Después, se introducen los parámetros de las redes neuronales *feedforward* entrenadas, se presentan aquellas que mejores resultados hayan obtenido y al final se exponen

unas consideraciones generales sobre el estudio realizado. Lo mismo para las redes recurrentes LSTM. Por último, se introducen 5 variables adicionales en el modelo, que corresponden con la codificación Word2vec continua de los tokens (introducida en el capítulo 3) y se presenta el modelo que mejores resultados ha obtenido.

Presentación de los datos

Para esta tarea de clasificación multiclase se utiliza el corpus `CONLL2002`, como se adelantó en el capítulo 2. Nuestro conjunto de entrenamiento está compuesto de 500 frases de este corpus y el de validación de otras 200 frases del corpus, distintas a las anteriores. En total, se tienen 18551 observaciones (tokens) en el conjunto de entrenamiento y 7648 en el de validación. En este caso disponemos, tras aplicar un cutoff de 10, de 1509 variables.

Nótese que se manejan conjuntos de datos de tamaños muy elevados y que esto tiene un efecto en el tiempo computacional del entrenamiento. Este tiempo de entrenamiento de los modelos oscila entre un día y una semana.

El porcentaje de nombres propios en el conjunto de entrenamiento y de validación está en torno al 12 %. En el conjunto de entrenamiento, de este 12 % de nombres propios, el 38,3 % está etiquetado como nombre propio de organización, el 16, 7 % misceláneo, el 23 % de persona y el 22 % de localización. Las clases de nombres propios de persona y localización se encuentran, aproximadamente, en la misma proporción y la clase organización destaca por ser la más abundante; por el contrario, la clase miscelánea es la más escasa. Para el conjunto de validación, las proporciones de las clases son aproximadamente equivalentes y siguen el mismo patrón que en el conjunto de entrenamiento.

El entrenamiento que se sigue, como en el capítulo anterior, es el entrenamiento por *mini-batch* y el tamaño del *mini-batch* que se ha fijado es de 100. Para esta tarea se han establecido diferentes valores para el número de épocas en función del entrenamiento. El máximo número de épocas que se ha considerado es de 500.

5.1. Redes *feed-fordward*. Perceptrón multicapa

Siguiendo el mismo esquema que en el capítulo anterior, se empezó entrenando modelos de redes *feedforward* para ninguna capa oculta, una y dos capas ocultas.

Se presentan los parámetros que mejores resultados numéricos en las *f-measure* hayan obtenido para las redes monocapa, las redes multicapa con una capa oculta y las redes multicapa con dos capas ocultas.

No obstante, se entrenaron diferentes redes neuronales considerando diferentes configuraciones. Los diferentes parámetros, funciones y algoritmos que se consideraron para el entrenamiento de las diferentes redes neuronales se muestran en la siguiente tabla:

Parámetros	
Función de pérdida	-Logaritmo de la verosimilitud
Algoritmo de optimización	-Nesterovs ($\mu = 0{,}9$) -Rmsprop ($\alpha = 0{,}95$)
Ratio de aprendizaje	$\eta \in [10^{-3}, 0{,}1]$
Función de activación	-Capa oculta: Relu, Leakyrelu, Sigmoide, Tanh -Capa salida: Softmax
Medidas de regularización	-$l_2, l_1 \in [10^{-6}, 10^{-3}]$ -Dropout ($p = 0{,}5$) -Early-stopping
Pesos iniciales	-Xavier -Ceros -Relu
Nº de neuronas en las capas ocultas	$m \in [5, 200]$

Tabla 5.1: Parámetros para las diferentes redes *feedforward* entrenadas

Red monocapa

La configuración de la red neuronal *feedforward* monocapa que mejores resultados obtuvo, así como sus resultados en las clases de nombres propios, se muestran en las siguientes tablas:

Configuración de la red monocapa	
Función de pérdida	-Logaritmo de la verosimilitud
Algoritmo de optimización	-Rmsprop ($\alpha = 0{,}95$)
Ratio de aprendizaje	η=0.1
Función de activación	-Capa salida: Softmax
Medidas de regularización	-Early-Stopping (50 épocas)
Pesos iniciales	-Xavier

Tabla 5.2: Configuración de la red monocapa con mejores resultados

Resultados	
F-measure (clase 1)	0.6713
F-measure (clase 2)	0.1763
F-measure (clase 3)	0.724
F-measure (clase 4)	0.4417

Tabla 5.3: Resultados de la métrica *f-measure* de las clases de NP

En primer lugar, a la vista de los resultados observamos, como era fácil prever, que la tarea que trabajamos en este capítulo es mucho más compleja de modelizar que la del capítulo anterior.

Por otro lado, a la vista del resultado en el *f-measure* de la clase miscelánea, observamos que el modelo no es capaz de clasificar bien dicha clase, hecho que no sorprende pues nótese que, además de ser la clase que en menor proporción se presenta en los datos, es la clase más ambigua de nuestro estudio, en el sentido de que todos los nombres propios que no pertenecen al resto de clases se clasifican en ella.

Se podría haber pensado en no considerar la clase miscelánea en nuestra clasificación. Sin embargo, nos interesó que apareciese, aunque fuese con un papel de 'basurero', pues recordemos que el primer objetivo del trabajo era el reconocimiento de nombres propios y después su clasificación en diferentes clases. Si obviáramos dicha clase, podría ocurrir que muchos nombres propios se clasificaran como no nombres propios y los 'perderíamos'. Esto es, preferimos, por ejemplo, que un nombre propio de organización se clasifique en la clase miscelánea que se clasifique en la clase de no nombre propio.

Las clases 1 y 3 (organización y persona) destacan por ser las clases con los valores más satisfactorios. Por otro lado, destaca el valor de 0.4417 en la clase 4, lo que indica que el modelo no es muy adecuado en la clasificación de la clase de localización. La proporción de nombres propios de persona y localización en los datos es muy similar, sin embargo, el modelo clasifica bastante mejor la clase de persona. Es decir, la clase de persona es más fácil de aprender que la de localización. Esto puede deberse a que la clase de localización tiende a ser más ambigua en un contexto periodístico. El corpus sobre el que se trabaja está constituido por textos periodísticos y en ellos suele ser común que un nombre propio pertenezca a organización o localización en función del contexto. Por ejemplo, en la frase "EEUU prohibe...", EEUU sería un nombre propio de organización, sin embargo, en la frase "El presidente de EEUU..." sería nombre propio de localización. Además, la proporción de nombres propios de localización es inferior a la de organización, que es la clase que se presenta con mayor proporción en los datos. Todo lo anterior podría explicar los valores que se dan en las métricas.

MLP con una capa oculta

Con la idea de mejorar el modelo anterior, sobre todo para las clases miscelánea y localización, se aplicaron diferentes modelos de perceptrón multicapa con una capa oculta.

El modelo de perceptrón multicapa con una capa oculta que mejores resultados proporcionó se obtuvo con un criterio de parada de 50 épocas. Su configuración, así como sus resultados, se muestran a continuación:

Configuración del MLP con una capa oculta	
Función de pérdida	-Logaritmo de la verosimilitud
Algoritmo de optimización	-Rmsprop ($\alpha = 0{,}95$)
Ratio de aprendizaje	η=0.1
Función de activación	-Capa oculta: Sigmoide -Capa salida: Softmax
Medidas de regularización	-$l_2 = 10^{-4}$
Pesos iniciales	-Xavier
Número neuronas capa oculta	10

Tabla 5.4: Configuración del MLP con una capa oculta con mejores resultados

Resultados	
F-measure (clase 1)	0.6962
F-measure (clase 2)	0.198
F-measure (clase 3)	0.7166
F-measure (clase 4)	0.4677

Tabla 5.5: Resultados de la métrica *f-measure* de las clases de NP

Observamos que el perceptrón multicapa consigue mejores resultados en las clases de NP en general. Esto es, introduciendo una capa oculta hemos conseguido mejores valores en las métricas. Sin embargo, sobre todo la clase miscelánea, sigue ofreciendo valores muy pequeños. En la siguiente sección se presentan entrenamientos de redes con una capa oculta más.

MLP con dos capas ocultas

Con el fin de seguir mejorando el modelo, se probaron diferentes configuraciones de red con dos capas ocultas.

El modelo que mejores resultados proporcionó se obtuvo con un criterio de parada de 30 épocas. Su configuración, así como sus resultados, se muestran en las siguientes tablas:

Configuración del MLP con dos capas ocultas	
Función de pérdida	-Logaritmo de la verosimilitud
Algoritmo de optimización	-Rmsprop ($\alpha = 0{,}95$)
Ratio de aprendizaje	η=0.1
Función de activación	-Capas ocultas: Sigmoide -Capa salida: Softmax
Medidas de regularización	-$l_2 = 10^{-3}$
Pesos iniciales	-Xavier
Número neuronas capa oculta	-Primera capa: 100 -Segunda capa: 20

Tabla 5.6: Configuración del MLP con dos capas ocultas con mejores resultados

Resultados	
F-measure (clase 1)	0.6865
F-measure (clase 2)	0.2394
F-measure (clase 3)	0.7339
F-measure (clase 4)	0.4685

Tabla 5.7: Resultados de la métrica *f-measure* en las clases de NP

Cabe destacar el valor en la *f-measure* para la clase miscelánea, cuyo valor ha aumentado. Se podría decir que aumentando la complejidad de la red hemos conseguido una mejor capacidad de aprendizaje en esta clase más ambigua.

En media, ofrece un mejor resultado en la *f-measure* de las clases de nombres propios que el modelo anterior y, consecuentemente, que el modelo de red monocapa, como se ilustra en el siguiente diagrama resumen:

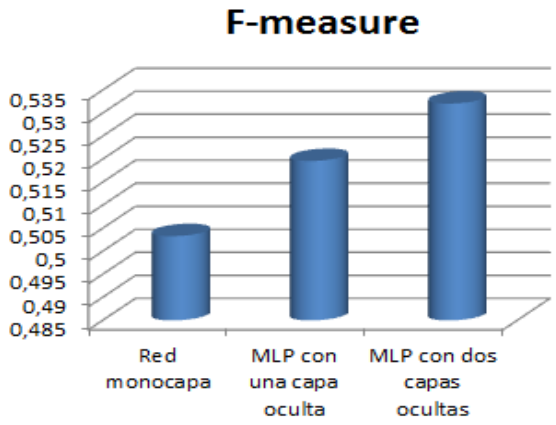


Figura 5.1: Diagrama media de la *f-measure* en las clases positivas

Consideraciones generales

Hasta ahora se han presentado los mejores valores de validación obtenidos de los tres modelos de redes neuronales con unos parámetros determinados. No obstante, remarcamos algunas consideraciones generales sobre la influencia de los parámetros en los diferentes entrenamientos que se han probado, que concuerdan con los resultados teóricos del capítulo 3:

- Se probaron primero modelos con Nesterovs como algoritmo de aprendizaje. Después se entrenaron modelos considerando el algoritmo de aprendizaje Rmsprop y los resultados, en general, mejoraban de forma notable. De hecho, a medida que la red era más compleja (más capas ocultas), la mejora era más pronunciada.

- Utilizando como algoritmo de aprendizaje el método Nesterovs, se observó que los modelos de redes neuronales con funciones de activación sigmoide o tangente hiperbólica, sobre todo al aumentar el número de capas ocultas, conducían al problema del vanishing de aprendizaje lento. Esto se solucionó considerando funciones relu, mejorando los resultados en gran medida. Refinando lo anterior, se consideró la función leakyrelu, mejorando los resultados todavía más.

- El problema del vanishing también se solucionó sin necesidad de considerar este tipo de funciones, usando el método de tasa adaptativa Rmsprop para el entrenamiento.

- Con respecto a la inicialización de los pesos, la *Xavier* es la que, en general, mejores resultados ha proporcionado en los modelos.

- Sobre todo para redes neuronales complejas, se comprobó que, en general, era necesario la incorporación de una medida de regularización en el modelo para evitar el sobreentrenamiento. Dentro de las medidas de regularización que consideramos, la l_2, en general, ha sido la más favorable. Por otro lado, remarcar que la medida de regularización early-stopping ha servido de guía a la hora de establecer el número de épocas óptimo para el entrenamiento, tanto para esta sección como para las siguientes.

- Los modelos con método de aprendizaje Rmsprop frente a los modelos con el método de aprendizaje Nesterovs obtuvieron mejores resultados en las clases de organización, persona y localización, aunque peor en la clase miscelánea (clase más ambigua).

5.2. Redes recurrentes LSTM

En la sección anterior se han validado e interpretado los mejores resultados de redes neuronales *feedforward*. En esta sección se muestran los mejores resultados proporcionados por modelos de redes recurrentes LSTM.

Nótese que en esta tarea de clasificación multiclase de nombres propios podría tener una gran influencia el contexto del texto, esto es, la secuencialidad del mismo. Como se explicó en el capítulo 3, las redes recurrentes tienen la propiedad de almacenar esta información del contexto gracias a su capacidad de 'memoria'. Por tanto, son modelos aconsejables para este tipo de tareas donde la secuencialidad del texto juega un papel fundamental. Es por ello, con la idea de mejorar nuestra tarea de clasificación, por lo que se aplicaron modelos de redes recurrentes LSTM, que son redes recurrentes que tienen la capacidad de almacenar información a corto y largo plazo.

Los parámetros, funciones y algoritmos que se han considerado para el entrenamiento de las diferentes redes recurrentes LSTM son los siguientes:

Parámetros	
Función de pérdida	-Logaritmo de la verosimilitud
Algoritmo de optimización	-Rmsprop ($\alpha = 0{,}95$)
Ratio de aprendizaje	$\eta \in [10^{-3}, 0{,}1]$
Función de activación	-Capa oculta: Tanh -Capa salida: Softmax
Medidas de regularización	-$l_2, l_1 \in [10^{-6}, 10^{-3}]$ -Dropout ($p = 0{,}5$) -Early-stopping
Pesos iniciales	-Xavier -Ceros
Nº de neuronas en las capas ocultas	$m \in [50, 200]$

Tabla 5.8: Parámetros para las diferentes redes recurrentes LSTM entrenadas

El modelo que proporcionó los mejores resultados se obtuvo con un criterio de parada de 25 épocas. La configuración de esta red recurrente, así como los resultados en la métrica *f-measure* se muestran en las siguientes tablas:

Configuración de la red LSTM	
Función de pérdida	-Logaritmo de la verosimilitud
Algoritmo de optimización	-Rmsprop ($\alpha = 0{,}95$)
Ratio de aprendizaje	η=0.1
Función de activación	-Capa oculta: Tanh -Capa salida: Softmax
Medidas de regularización	-$l_2 = 0{,}001$
Pesos iniciales	-Xavier
Número de capas ocultas	Dos capas ocultas LSTM
Número de neuronas capas ocultas	50

Tabla 5.9: Configuración de la red recurrente LSTM con mejores resultados

Resultados	
F-measure (clase 1)	0.6657
F-measure (clase 2)	0.2622
F-measure (clase 3)	0.7529
F-measure (clase 4)	0.4856

Tabla 5.10: Resultados de la métrica *f-measure* en las clases de NP

Con respecto a los valores de la Tabla 5.7, observamos que, aunque baja un poco el valor de la *f-measure* en la clase 1 (organización), mejoran el resto de clases. Los valores obtenidos en el conjunto de validación con este modelo de red recurrente LSTM superan en media a los de los modelos de redes *feedforward*.

En la tarea de reconocimiento y clasificación de entidades nombradas, el contexto de la frase puede ser importante (recuérdense las frases introducidas en la sección anterior sobre la entidad 'EEUU'). Se ha mostrado que los modelos de redes recurrentes LSTM son los modelos que consiguen resultados más satisfactorios en nuestro conjunto de validación.

Consideraciones generales

Como en la sección anterior, exponemos las consideraciones más generales de nuestro estudio del entrenamiento de modelos de redes LSTM.

- En general, los modelos entrenados que consideraban esta arquitectura de red (redes recurrentes LSTM) ofrecían valores más satisfactorios en el conjunto de validación que los modelos de perceptrón multicapa.
- En el entrenamiento de estas redes se comprobó que, en general, era necesario una medida de regularización que evitara el sobreentrenamiento.
- Se entrenaron redes neuronales con un mayor número de neuronas en la capa oculta (mayor que 50), consiguiendo peores resultados.
- Al disminuir la tasa de aprendizaje η era necesario un mayor número de épocas para el entrenamiento y los resultados mejoraban en las clases de organización, persona y localización, aunque no en la clase miscelánea.

Aunque en la validación o evaluación de los modelos se decidió seguir el criterio de mayor valor en la media de la *f-measure* de las clases de nombres propios, se podría haber realizado un estudio bajo un criterio diferente. Por ejemplo, supongamos que nos interesase más un modelo que consiga un valor de la *f-measure* significativamente mayor en alguna de las clases de organización, persona y localización, aunque esto implique un valor significativamente peor en la clase miscelánea, y la media de la *f-measure* sobre todas estas clases fuese inferior. Esto es, que importase más una mejora sobre las clases de organización, localización y persona que sobre todas ellas. Bajo dicha suposición, el mejor modelo se consiguió con una tasa de aprendizaje de $\eta = 0{,}01$ y con un entrenamiento de 350 épocas. Los resultados para las clases organización, persona y localización fueron satisfactorios, sobre todo para la clase de localización con respecto a los resultados anteriores, cuyo valor superaba ahora el 0.5:

Resultados	
F-measure (clase 1)	0.6715
F-measure (clase 2)	0.1105
F-measure (clase 3)	0.7213
F-measure (clase 4)	0.5907

5.3. Word2vec

En las secciones anteriores se han mostrado resultados de modelos de redes *feedforward* (perceptrón multicapa) y hemos visto como, al considerar otra arquitectura de red (redes recurrentes LSTM), los resultados mejoraban en nuestro conjunto de validación.

Con el objetivo de mejorar los modelos, pensamos ahora hacia otra dirección que no tiene que ver con la arquitectura de la red en sí, sino con las variables que introducimos en el modelo. Un modelo puede mejorar porque los parámetros que se consideran en su entrenamiento conducen a mejores resultados o porque las variables que consideramos consiguen explicar mejor el modelo.

Como se expuso en el capítulo 3, la herramienta Word2vec de codificación continua de las palabras consigue almacenar información sintáctica y semántica de ellas en función del contexto del texto. Es por ello por lo que se pensó que considerar la codificación Word2vec de los tokens de nuestro texto como variables podría influir de forma positiva en nuestro modelo de clasificación.

Se generó un modelo Word2vec, con una ventana de tamaño 5, que proporcionaba la codificación de los tokens de nuestros textos en un vector de dimensión 5×1, que se corresponde con las 5 variables adicionales en el modelo.

Para la creación de esta nueva matriz numérica con las 5 variables adicionales proporcionadas por el modelo Word2vec, se tuvo que crear una clase en `Java`, cuyo código puede verse en Anexos.

Se entrenaron, considerando estas nuevas variables, los diferentes modelos (perceptrón multicapa y redes LSTM) con las distintas configuraciones de red entrenadas anteriormente.

El modelo que mejores resultados obtuvo fue el perceptrón multicapa con una capa oculta con un criterio de parada de 30 épocas. Su configuración, así como sus resultados, se muestran a continuación:

Configuración de la red	
Función de pérdida	-Logaritmo de la verosimilitud
Algoritmo de optimización	-Nesterovs ($\mu = 0{,}9$)
Ratio de aprendizaje	$\eta = 0{,}08$
Función de activación	-Capa oculta: Sigmoide -Capa salida: Softmax
Pesos iniciales	-Xavier
Número de neuronas capa oculta	10

Tabla 5.11: Configuración de la red con Word2vec con mejores resultados

Resultados	
F-score (clase 1)	0.7103
F-score (clase 2)	0.2672
F-score (clase 3)	0.7247
F-score (clase 4)	0.496

Tabla 5.12: Resultados de la métrica *f-measure*

En general, los resultados de la *f-measure* en las clases de nombres propios son más satisfactorios que en los modelos anteriores, sobrepasando el valor 0.7 en las clases de organización y persona y cercano al 0.5 en la clase de localización.

Podríamos decir que este modelo es el que mejores resultados en la métrica de *f-measure* proporciona.

Consideraciones generales

Hasta ahora se ha presentado el modelo que mejores valores en la métrica ha conseguido, con la inclusión de la codificación de los tokens como nuevas variables. Sin embargo, se entrenaron diferentes modelos con la incorporación de estas nuevas variables. A continuación comentamos las consideraciones generales de este estudio.

Se comprobó que, en general, la inclusión de estas nuevas variables en los modelos de perceptrón multicapa mejoraban los resultados. Sin embargo, no ocurría lo mismo en las redes recurrentes LSTM cuya mejora no era apreciable, proporcionando resultados similares a los modelos recurrentes LSTM originales.

Una explicación de todo lo anterior podría ser la siguiente. La arquitectura de la red del perceptrón multicapa no tiene la capacidad de 'memoria' que tienen las redes recurrentes LSTM. Por otro lado, la codificación Word2vec permite codificar los tokens de un texto, cuya codificación almacena información sintáctica y semántica en función del contexto. Si introducimos esta codificación como variables en el modelo del perceptrón multicapa, los resultados mejoran, posiblemente porque le añadimos información adicional al modelo sobre el contexto, información importante que no es posible con la propia arquitectura de la red. Sin embargo, como se ha dicho, esta idea de almacenar la información del 'contexto' sí que es posible con las redes recurrentes LSTM. Al introducir las variables del Word2vec en estas redes neuronales, los resultados no mejoran, posiblemente porque la información que introduzco en el modelo no explica más allá de lo que puede hacer la propia red neuronal.

Capítulo 6

Conclusiones de los resultados y posibles mejoras

"Si buscas resultados distintos
no hagas siempre lo mismo."
Albert Einstein

6.1. Conclusiones

A continuación se exponen las conclusiones más generales de los resultados de nuestro estudio práctico.

El trabajo realizado abarca, por un lado, la clasificación binaria de nombres propios, y por otro, la clasificación multiclase de nombres propios mediante el uso de algoritmos de redes neuronales.

En el primer caso de estudio, aunque el perceptrón multicapa con dos capas ocultas, con un punto de corte óptimo de 0.6, y cuya configuración viene dada en la Tabla 4.7, haya sido el que mejores resultados haya obtenido en el conjunto de validación, los resultados en la métrica para los diferentes modelos entrenados fueron satisfactorios y muy similares. Por tanto, podríamos concluir diciendo que, considerando las variables introducidas en la capítulo 2 y nuestros conjuntos de datos, esta tarea de clasificación binaria es una tarea modelizable, con buenos resultados de clasificación, bajo modelos de redes neuronales *feedforward*.

Sin embargo, el problema de clasificación multiclase ha mostrado ser una tarea mucho más compleja de modelizar.

En el entrenamiento de los diferentes modelos se comprobó, apoyado por la teoría, que una elección adecuada de los parámetros mejoraba los resultados de clasificación.

Los modelos entrenados para esta tarea de clasificación multiclase proporcionaban valores en la métrica *f-measure* inferiores a los valores obtenido en el caso anterior. Las clases de nombres propios de localización y miscelánea han sido las clases que peor se clasificaban, tomando unos valores en torno a 0.4 y 0.2, respectivamente. Además de ser las clases que en menor proporción se presentaban en los datos, son las clases más ambiguas y su clasificación depende del contexto del texto.

Con esta idea de la importancia del contexto para la clasificación, esto es, la secuencialidad del texto, se entrenaron modelos de redes recurrentes LSTM. Los valores de la métrica *f-measure* de los nombres propios mejoraban, sobre todo en estas clases más ambiguas, con respecto a los modelos de redes *feedforward*, cuya arquitectura (conexión total hacia adelante) no tiene esta capacidad de 'memoria' que tiene la arquitectura de las redes recurrentes.

Siguiendo con esta idea pero intentando mejorar el modelo no por la arquitectura de la red sino por las variables que introducimos en él, se consideró la codificación Word2vec de los tokens. Estudios empíricos demuestran que dicha codificación almacena información sintáctica y semántica del token en función de su contexto.

En los modelos de redes *feedforward*, la incorporación de estas nuevas variables mejoró los resultados. En cambio, en las redes recurrentes LSTM, la inclusión de estas nuevas variables no provocó mejora apreciable en los resultados de la métrica. Esto puede ser debido a que la arquitectura de la propia red neuronal es suficiente para almacenar esta información sobre el contexto del texto.

El modelo que mejores resultados obtuvo, considerando estas variables adicionales, fue el perceptrón multicapa de una capa oculta, cuyo valor de media superaba al modelo de red recurrente LSTM original y al modelo de red *feedforward* original, como se ilustra en el siguiente diagrama:

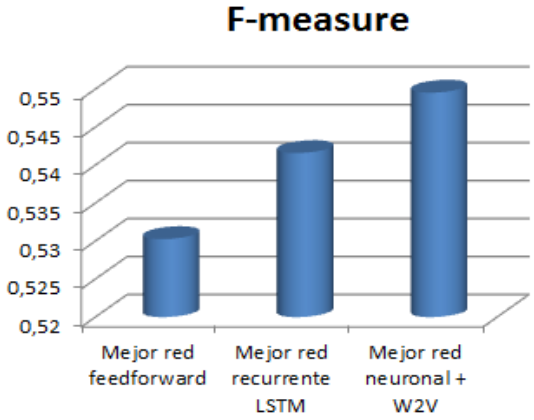


Figura 6.1: Media del f-measure sobre las clases positiva de los mejores modelos obtenidos

Por tanto, podríamos concluir diciendo que, bajo nuestro criterio de validación (media de la *f-measure* de las clases positivas), incorporar como variables del modelo la codificación Word2vec influye en una mejora del modelo en la clasificación multiclase de nombres propios.

6.2. Limitaciones y posibles mejoras

En esta sección se comentan las posibles mejoras en el modelo de clasificación para un futuro trabajo.

Debido a las limitaciones en la práctica, provocadas por el problema del tiempo computacional, se eligió un corpus para la tarea de clasificación multiclase de 500 frases. Un modelo de clasificación puede mejorar por el refinamiento de los parámetros, las variables del modelo, su arquitectura o por el conjunto de entrenamiento. Se piensa que considerar un corpus más grande como conjunto de entrenamiento puede mejorar la clasificación de manera favorable.

Otra de las posibilidades que se piensa que podrían mejorar el modelo es considerar un tamaño del vector Word2vec mayor.

La tarea de clasificación binaria de nombres propios es un caso concreto de la tarea de POS-Tagger (tarea que consiste en asignar a cada palabra del texto su categoría gramatical). Sin embargo, la clasificación multiclase de nombres propios no busca solo reconocer los nombres propios sino clasificarlos en diferentes categorías. Se piensa que si incluyésemos como variable la etiqueta del *postagger* de cada token, el modelo de clasificación multiclase mejoraría considerablemente. Esta idea se puede ver más clara con los siguientes ejemplos introducidos en el capítulo anterior (capítulo 5): "EEUU prohibe..." y "El presidente de EEUU...". La etiqueta gramatical de 'verbo' (prohibe) del token posterior informa sobre que EEUU se contextualiza como nombre propio de organización y, por otro lado, la etiqueta gramatical de 'preposición' del token anterior (de) informa sobre que EEUU, en ese caso, es nombre propio de localización. Siguiendo esta línea, esto podría ser un futuro trabajo que conllevaría la adicional estimación de un modelo de POS-Tagger.

6.3. Posibles aplicaciones

La tarea de Minería de textos sobre el reconocimiento de entidades nombradas puede aplicarse a diversas situaciones reales, facilitando el trabajo humano de hoy en día.

Una de las posibles aplicaciones que puede tener el reconocimiento de entidades nombradas es en el análisis de sentimientos. En él, se clasifican fragmentos de un texto, como pueden ser las frases, según tengan una valoración positiva o negativa. Sin embargo, para sacar conclusiones de estas opiniones, será necesario conocer sobre qué entidad nombrada se está hablando. Para ello será necesario la aplicación de modelos de reconocimiento de entidades sobre las frases.

Una de las aplicaciones con más éxito del reconocimiento de entidades nombradas está en la catalogación supervisada de contenido audiovisual de forma automatizada. Este trabajo es muy importante, por ejemplo, para los documentalistas.

Gracias a la tarea del reconocimiento de entidades nombradas se puede extraer la información de un audio o vídeo, etiquetando las entidades nombradas sobre las que se está hablando. De esta forma, se permite la clasificación o la búsqueda automática de los vídeos o audios bajo dichas palabras clave (las entidades nombradas). Así, por ejemplo, si un documentalista quiere buscar un vídeo en el que aparezcan dos personajes públicos protagonistas de un noticia en concreto, buscaría en la base de datos esas dos entidades nombradas de personas y, de manera automática, se le proporcionarían los vídeos donde aparecen dichas personas.

Así, la tarea que se ha abordado en el trabajo (reconocimiento y clasificación de entidades nombradas) es un problema con gran aplicación en el mundo real.

Glosario

Definimos a continuación, por orden alfabético, algunos de los términos básicos del procesamiento del lenguaje natural y la Minería de textos que se nombran a lo largo del trabajo. Algunas de estas definiciones ya vienen definidas a lo largo del trabajo.

- **Análisis de sentimientos:** El análisis de sentimientos se refiere al uso del procesamiento del lenguaje natural, análisis de textos y lingüística computacional para identificar y extraer información subjetiva de unos recursos. Esto es, determinar la actitud de un interlocutor o un escritor con respecto a algún tema de un documento.
- **Bigrama:** Es un grupo de dos letras, dos sílabas o dos palabras. La generalización para n letras, sílabas o palabras se denomina **N-grama.**
- **Chunking:** Es el proceso que consiste en dividir un texto en partes sintácticamente correlacionadas de palabras, como son los grupos nominales, grupos verbales, etc., pero no especifica su estructura interna, ni su papel en la oración principal.
- **Corpus:** En lingúistica, se denomina corpus a un conjunto amplio y estructurado de textos (etiquetado) de acuerdo a unos ciertos criterios que, normalmente, suele ser almacenado y procesado automáticamente. Estos documentos seon parte de nuestro conjunto de entrenamiento y de validación.
- **Lematización:** Es el proceso lingüístico que consiste en sustituir una palabra por su lema, equivalentemente, reducir una palabra a su raíz.
- **Lenguaje natural:** Es la lengua o idioma hablado o escrito por humanos para propósitos generales de comunicación.
- **Outcome:** Es el valor de la variable de salida del modelo o etiqueta.
- **Parsing (o analizador sintáctico):** Es el proceso que consiste en analizar sintácticamente una frase.
- **POS-Tagger (o etiquetado gramatical):** Es el proceso de asignar o etiquetar a cada una de las palabras del texto su categoría gramatical (sustantivo, nombre, verbo, etc.).
- **Reconocimiento de entidades:** Es una tarea de la extracción de la información que localiza y clasifica nombres de entidades en unas categorías predefinidas.
- **Stop words:** Son palabras sin poder discriminatorio, que no aportan significado.
- **Token:** En un texto, se denomina token a cada cadena de caracteres separada de otra por un delimitador. Ejemplos de tokens pueden ser, por ejemplo, las palabras.

- **Tf-idf:** El tf-idf es una medida numérica que expresa cuán relevante es una palabra para un documento en una colección de ellos. Un peso alto en tf-idf se alcanza con una elevada frecuencia de término (en el documento dado) y una pequeña frecuencia de ocurrencia del término en la colección completa de documentos. Más información en: https://es.wikipedia.org/wiki/Tf-idf
- **Tokenización:** En el análisis léxico, la tokenización es el proceso que consiste en separar un texto en tokens.
- **Ventana:** Denotaremos por ventana al conjunto de palabras en la vecindad de una dada. Por ejemplo, en la frase "Este es mi trabajo fin de máster", una ventana de tamaño 2 de la palabra 'fin' estará constituida por sus dos palabras anteriores y siguientes, esto es: [mi, trabajo, de, máster].

Bibliografía

[1] *AnCora Corpus*. Fecha de consulta: noviembre de 2016. Disponible en: http://clic.ub.edu/corpus/es/ancora

[2] Bishop, Christopher M. (2003). *Neural Networks for Pattern Recognition.* Departament of Computer Science and Applied Mathematics. Aston University. Birmingham, UK.

[3] Borrega, O., Martí, M. A., Taulé, M. (2007) *'What do we mean when we talk about Named Entities?'*, Corpus Linguistics, Birmingham (UK).

[4] *Deep Learning for Java*. Fecha de consulta: noviembre de 2016. Disponible en: https://deeplearning4j.org/

[5] Funahaski, K. (1989) *On the approximate realization of continuous mapping by neural networks.* Neural Networks, 2(3): 183-192.

[6] Glorot, X. and Bengio, Y. (2010). *Understanding the difficulty of training deep feedforward neural networks*. In Proceedings of the International Conference on Artificial Intelligence and Statistics (AISTATS10). Society for Artificial Intelligence and Statistics.

[7] Goodfellow, I., Bengio, Y. and Courville, A. *Deep Learning* (2016). Libro no publicado. En preparación para MIT Press. http://www.deeplearningbook.org.

[8] Guo, Jiang. (2013). *BackPropagation Through Time*. Harbin Institute of Technology.

[9] Haykin, S. (2005). *Neural Networks - A Comprehensive Foundation* (2nd. ed.). Prentice-Hall, Upper Saddle River, NJ.

[10] He, J., Zhang, X., Ren, S. and Sun, J. (2015). *Delving Deep into Rectifiers:Surpassing Human-Level Performance on ImageNet Classification.*

[11] Hinton, G. *Neural Networks for Machine Learning. Lecture 6a Overview of mini-batch gradient descent.* Fecha de consulta: noviembre de 2016. Disponible en: http://www.cs.toronto.edu/~tijmen/csc321/slides/lecture_slides_lec6.pdf

[12] Hinton, G., Sutskever, I. Martens, J. and Dahl, G. (2013). *On the importance of initialization and momentum in deep learning.* In Proceedings of the 30th international conference on machine learning (ICML-13), pages 1139-1147.

[13] Hochreiter and Schmidbuber (1997). *LONG-SHORT MEMORY*. Neural Computation 9(8): 1735-1780.

[14] *Language-Independent Named Entity Recognition (I)*. Fecha de consulta: noviembre de 2016. Disponible en: http://www.cnts.ua.ac.be/conll2002/ner/

[15] Mikolov, T., Yih, W. and Zweig, G. (2013). *Lingüistic regularities in continuous space word representations.* Proceedings of NAACL-HLT 2013. Pgs 746-751.

[16] Mikolov, T., Chen, K., Corrado, G., and Dean, J. (2013). *Efficient estimation of word representations in vector space.* arXiv preprint arXiv:1301.3781.

[17] Mikolov, T., Sutskever, I., Chen, K., Corrado, G. S., and Dean, J. (2013). *Distributed representations of words and phrases and their compositionality.* In Advances in Neural Information Processing Systems, pgs 3111-3119.

[18] Nesterov, Y. (1983). *A method for unconstrained convex minimization problem with the rate of convergence* $O(1/k^2)$. In Soviet Mathematics Doklady, volume 27, pgs 372–376.

[19] Martin del Brío, B. y Sanz, A. (2001). *Redes neuronales y sistemas borrosos: introducción, teoría y práctica.* 2ª Edición ampliada y revisada. Editorial Ra-Ma. Pg 44. Pgs 69-70.

[20] Ratnaparkhi, Adwait. (1996). *A Maximum Entropy Model for Part-Of-Speech Tagger.* University of Pennsylvania. Dept. of Computer and Information Science.

[21] Rong, X. (2014). *word2vec Parameter Learning Explained.* arXiv:1411.2738 .

[22] Rosenblatt, F. (1962). *Principles of Neurodynamics: Perceptrons and the Theory of Brain Mechanisms.* Washington DC: Spartan.

[23] Rumelhart, D. E., Hinton G. E. and Williams, R. J. (1986). *Learning representations by back-propagating errors.* Nature, 323. Pgs 533–536.

[24] The Apache Software Foundation. *Apache OpenNLP.* Fecha de consulta: noviembre de 2016. Disponible en: https://opennlp.apache.org/

[25] Werbos, P. J. (1990). *Backpropagation through time: what it does and how to do it.* Proc. IEEE, 78. Pgs 1550-1560.

Anexos

Transformación formato Ancora-formato openNLP

Resumen

El código se resumiría como sigue (en vista del formato Ancora): Leeríamos cada línea del fichero de entrada (corpus Ancora). Escribiríamos, de todas las frases, el primer elemento en un fichero de salida, que es el que corresponde con el propio token. Si el elemento que ocupa la posición 4 de la línea es la letra `n` (que nos indica que es nombre) y el que ocupa la posición 6 corresponde a la marca `postype=proper` (que indica que es nombre propio) escribiríamos en el fichero la etiqueta `<START>` antes del propio token y `<END>` al final. Nótese que los nombres propios de más de una palabra aparecen unidos con un guión bajo, que deberemos reemplazar por un espacio en blanco para que contribuyan en el texto como dos tokens diferentes, que es lo que son. Cada vez que leamos una nueva línea del corpus, que corresponde a cada token del texto, escribiremos un espacio en blanco en el fichero de salida para así conservar la tokenización. Cuando el primer elemento de la línea del corpus sea un punto, escribiremos en fichero un salto de línea, para así tener cada frase separada en líneas diferentes, como en el formato opennlp. De esta forma, conseguimos convertir la información que nos interesa del corpus Ancora en el formato adecuado de opennlp.

Código en Java

```
package unizar.es;

/*
* @author raznar
*/

import java.io.BufferedReader;
import java.io.File;
import java.io.FileReader;
import java.io.FileWriter;
import java.io.IOException;


public class CambiandoFormato4 {

FileWriter fichero1, fichero2;
StringBuilder aux = new StringBuilder();
String bfRead;
String bfRead1;
String bfRead2;
final String STR_END = " <END> ";
final String STR_START = "<START> ";
```

```
String[] tokenss;
int count = 0;

public String nombreFicheroDestino1;
public String nombreFicheroDestino2;


public CambiandoFormato4(String nombreFicheroDestino1, String
   nombreFicheroDestino2) {
        this.nombreFicheroDestino1 = nombreFicheroDestino1;
        this.nombreFicheroDestino2 = nombreFicheroDestino2;

}

public String procesoUnaCarpeta(String direccion) throws IOException {
    //direccion del archivo

    String texto = " ";

    fichero1 = new FileWriter(nombreFicheroDestino1);
    fichero2 = new FileWriter(nombreFicheroDestino2);


    File folder = new File(direccion);
    File[] listOfFiles = folder.listFiles();


    for (int i = 0; i < listOfFiles.length; i++) {

        if (listOfFiles[i].isFile()) {
            System.out.println("File " + listOfFiles[i].getName());
            BufferedReader bf = new BufferedReader(new FileReader(
               direccion + "\\" + listOfFiles[i].getName()));
            aux.append("");
            while ((bfRead = bf.readLine()) != null) { //lee cada
               linea del fichero
                if (!bfRead.trim().isEmpty()) {
                    tokenss = bfRead.split("\t"); //separa por
                       tabulador para obtener los tokens
                    if ("n".equals(tokenss[4]) && "postype=proper".
                       equals(tokenss[6])) { //si es nombre propio
                        aux.append("<START> ");
                        aux.append(tokenss[1].replaceAll("_", " "));
                        aux.append(" <END>");
                    } else if (!tokenss[1].equals("_")) { //si es un
                       guion
                    aux.append(tokenss[1].replaceAll("_", " "));
                    } else {
                    //nada
                    }
                    aux.append(" "); //tokenizo por espacio en blanco
                    if (".".equals(tokenss[1])) { // si es un punto,
                       salto de linea
                        aux.append("\n");
                    }
                }
```

```
            }

            fichero1.write(aux.toString()); //escribo en fichero
            aux = new StringBuilder();

        }
    }

    fichero1.close();

    BufferedReader bf1 = new BufferedReader(new FileReader(
       nombreFicheroDestino1));
    aux.append("");

    while ((bfRead1 = bf1.readLine()) != null) { //quitar espacios en
       blanco al inicio de la sentencia
        count++;
        if (bfRead1.startsWith(" ")) {
            aux.append(bfRead1.replaceFirst(" ", "").replaceAll("  ",
               " "));
            aux.append("\n");
            fichero2.write(aux.toString());
            aux = new StringBuilder();
        } else {
            aux.append(bfRead1.replaceAll("  ", " "));
            aux.append("\n");
            fichero2.write(aux.toString());
            aux = new StringBuilder();
        }

    }
    fichero2.close();

    //El orden de los ficheros
    for (File listOfFile : listOfFiles) {
        System.out.println("File " + listOfFile.getName());
    }

    return texto;

}

}
```

Generación matriz de datos en openNLP

Mostramos la implementación de código de la clase principal `GISTrainer.java`, marcado por $*$. Dependiendo si se quiere generar la matriz del conjunto de entrenamiento o de validación, se comentará una parte u otra del código.

Código

```
outcomeLabels = di.getOutcomeLabels();
outcomeList = di.getOutcomeList();
numOutcomes = outcomeLabels.length;
predLabels = di.getPredLabels();
```

```
prior.setLabels(outcomeLabels, predLabels);
numPreds = predLabels.length;

//*************************************************************



// ROCIO: MATRIZ DE ENTRENAMIENTO //

StringBuilder aux = new StringBuilder();
int b[] = new int[numPreds];
FileWriter fichero1;
fichero1 = new FileWriter(fichero_csv_Entrenamiento);

//las 5 siguientes lineas de codigo comentar cuando no quiero cabecera
for (String predLabel : predLabels) { //los predicados contextuales
   que genera en la cabecera
    aux.append("#"); //comprobar que no hay # en el texto
    aux.append(predLabel);
}
aux.append("\n");

for (int i = 0; i < numPreds; i++) { //inicializo a cero
    b[i] = 0;
}


if (contexts != null) {

    for (int x = 0; x < contexts.length; x++) {// para cada token

       for (int y = 0; y < contexts[x].length; y++) { //pongo 1 en las
           posiciones que corresponden a los    predicados
          contextuales que visita
           b[contexts[x][y]] = 1;
       }

        //valores en la variable de salida
        if (outcomeList[x] == 0) {
            outcomeList[x] = 0; //para que deje 0 NNP y 1 NP
        } else if (outcomeList[x] == 2 || outcomeList[x] == 1) {
            outcomeList[x] = 1;
        }

        aux.append("" + outcomeList[x]); //pongo el outcome el primero
        for (int i = 0; i < numPreds; i++) {
            aux.append("," + b[i]);  //el valor para las variables
               separado de coma (formato .csv)
        }

        aux.append("\n"); //paso a la siguiente linea para el
           siguiente token
        fichero1.write(aux.toString()); //escribo en fichero
        aux = new StringBuilder(); //limpio el auxiliar

        for (int i = 0; i < numPreds; i++) { //e inicializo a cero
```

```
                b[i] = 0;
            }

        }

} else {
            //nada
            }

fichero1.close(); //cierro el fichero


//Para guardar la cabecera de la prediccion. NOTAR: Habra que cambiar
    el directorio.
FileWriter fichero2;
fichero2 = new FileWriter(fichero_csv_Cabecera);
StringBuilder aux2 = new StringBuilder();
String s1;
BufferedReader bf2 = new BufferedReader(new FileReader(
    direccion_fichero_csv_Entrenamiento));
s1 = bf2.readLine();
String[] predicados;
predicados = s1.split("#"); //obtengo los tokens de la cabecera que
    estaban separados por #

int cuenta = 0;

for (int i = 1; i < predicados.length; i++) { //empiezo en el 1 (me
    dejo el outcome)
     cuenta++;
     System.out.println("predicados->" + predicados[i]);
     aux2.append(predicados[i]); //escribo cada token, en este caso,
         cada predicado en un nuevo fichero
     aux2.append("#");
     fichero2.write(aux2.toString());
     aux2 = new StringBuilder();
}
System.out.println("Numero de variables es--> " + cuenta);
fichero2.close(); //cierro el fichero



//ROCIO: MATRIZ DE PREDICCION/VALIDACION //

StringBuilder aux3 = new StringBuilder();
FileWriter fichero3;
fichero3 = new FileWriter(fichero_csv_Validacion); //fichero de
    prediccion

//pongo la misma cabecera que el conjunto de entrenamiento, guardada
    antes
BufferedReader bf3 = new BufferedReader(new FileReader(
    direccion_fichero_csv_Cabecera));
String s11;
s11 = bf3.readLine(); //leo la primera fila
String[] predicados3;
```

```
predicados3 = s11.split("#"); //separo por #
System.out.println("variables-->" + predicados3.length);


//el siguiente vector de enteros inicializado a 0 sera cada fila de la
    matriz de datos
int bb[];
bb = new int[predicados3.length];
for (int i = 0; i < predicados3.length; i++) { //inicializo a cero
    bb[i] = 0;
}


//Rellenamos la matriz de datos
if (contexts != null) {
        for (int x = 0; x < contexts.length; x++) { //para cada token
            for (int y = 0; y < contexts[x].length; y++) { //para cada
                variable generada de cada token
                for (int z = 0; z < predicados3.length; z++) { //para
                   cada variable de la cabecera del cjto de
                   entrenamiento
                    if (predicados3[z].equals(predLabels[contexts[x][y
                       ]])) { //si la variable de la cabecera
                       coincide con la variable generada del token a
                       predecir se le pone un 1. En caso contrario
                       sera cero por la inicializacion anterior.
                        bb[z] = 1;
                    }
                }
            }

            //valores en la variable de salida
            if (outcomeList[x] == 0) {
                outcomeList[x] = 0; //para que deje 0 NNP y 1 NP
            } else if (outcomeList[x] == 2 || outcomeList[x] == 1) {
                outcomeList[x] = 1;
            }


            aux3.append("" + outcomeList[x]); //para el conjunto test
            for (int i = 0; i < predicados3.length; i++) {
                aux3.append("," + bb[i]);
            }
            aux3.append("\n"); //y paso a la siguiente fila y lo mismo
                para el siguiente token
            fichero3.write(aux3.toString()); //escribo en fichero
            aux3 = new StringBuilder(); //limpio el auxiliar

            //antes de pasar al siguiente token, los vectores de 0's y
                1's los inicializo a cero y vuelvo a repetir los
               mismos pasos
            for (int i = 0; i < predicados3.length; i++) { //
               inicializo a cero
                bb[i] = 0;
            }
        }
```

```
} else {
        System.out.println("Error");
}
fichero3.close(); //cierro el fichero


//COMPROBACION //

BufferedReader bf33 = new BufferedReader(new FileReader(
   fichero_csv_Entrenamiento/Validacion));
String s111;
s111 = bf33.readLine(); //leo la primera fila
String[] predicados33;
predicados33 = s111.split(","); //separo por ,
System.out.println("variables-->" + predicados33.length + " (tiene que
    ser uno mas por el outcome que he metido) ");


//************************************************************


display("\tNumber of Event Tokens: " + numUniqueEvents + "\n");
display("\t    Number of Outcomes: " + numOutcomes + "\n");
display("\t  Number of Predicates: " + numPreds + "\n");
```

Observaciones

- En el código implementado se hace diferencia si el corpus que introduzco en la librería es el de entrenamiento o el de validación. Esto se hace pues se generan todos las variables para cada texto introducido pero nuestro conjunto de entrenamiento y validación tiene que tener lógicamente las mismas variables. De esta forma, guardaremos la cabecera con las variables resultantes tras aplicar el *cutoff* en nuestro conjunto de entrenamiento para así, cuando introduzcamos el texto de validación y se generen sus variables, introduzcamos solo aquellas que formen parte de dicha cabecera.

- Por otro lado, nótese que la librería no toma un valor fijo para las variables de salida sino que empieza a asignar valores ordenados en vista de lo que se vaya encontrando al leerlo. Así, si el primer token del corpus es nombre propio, el valor 0 en la variable de salida denotará que el token es nombre propio, pero si por el contrario el primer token del Corpus no es nombre propio, el valor 0 denotará en este caso que el token pertenece a la clase complementaria. Por esta razón, antes de generar nuestra matriz de datos a partir del texto etiquetado, deberemos estar seguros de que los valores en el outcome en la matriz de entrenamiento y en la de validación se corresponden con las mismas clases. Si no es así, se debería crear código para que así lo hiciese.

- Nótese también que en el caso del conjunto de entrenamiento hacemos uso de un seleccionador de variables, esto es, fijamos un *cutoff* en la librería. Sin embargo, para el conjunto de validación, como las variables son las generadas en el conjunto de entrenamiento, no se deberá aplicar ningún tipo de *cutoff* para que se generen todas las variables posibles del texto de validación y así elegir las que forman parte en el conjunto de entrenamiento y así no eliminar información. Esta consideración se deberá reflejar en el test de esta clase.

Deeplearning4j

Código

Se expone el código creado de la clase principal de Java. Según se quiera entrenar o validar, se deberá comentar una parte u otra del código.

```
package org.deeplearning4j.examples.feedforward.classification;

import java.io.BufferedReader;
import org.datavec.api.records.reader.RecordReader;
import org.datavec.api.records.reader.impl.csv.CSVRecordReader;
import org.datavec.api.split.FileSplit;
import org.deeplearning4j.datasets.datavec.RecordReaderDataSetIterator
    ;
import org.nd4j.linalg.dataset.api.iterator.DataSetIterator;
//import org.deeplearning4j.eval.Evaluation;
import org.deeplearning4j.examples.feedforward.classification.
    Evaluation.Evaluation; //clase modificada para permitir diferentes
     puntos de corte
import org.deeplearning4j.nn.conf.MultiLayerConfiguration;
import org.deeplearning4j.nn.multilayer.MultiLayerNetwork;
import org.deeplearning4j.optimize.listeners.ScoreIterationListener;
import org.nd4j.linalg.api.ndarray.INDArray;
import org.nd4j.linalg.dataset.DataSet;
import java.io.File;
import java.io.FileInputStream;
import java.io.FileNotFoundException;
import java.io.FileOutputStream;
import java.io.FileReader;
import java.io.FileWriter;
import java.io.IOException;
import java.time.Duration;
import java.time.Instant;
import java.util.ArrayList;
import java.util.Arrays;
import java.util.Collections;
import java.util.List;
import java.util.Map;
import java.util.Objects;
import java.util.concurrent.TimeUnit;
import org.datavec.api.split.InputSplit;
import org.deeplearning4j.earlystopping.EarlyStoppingConfiguration;
import org.deeplearning4j.earlystopping.EarlyStoppingModelSaver;
import org.deeplearning4j.earlystopping.EarlyStoppingResult;
import org.deeplearning4j.earlystopping.saver.LocalFileModelSaver;
import org.deeplearning4j.earlystopping.scorecalc.
    DataSetLossCalculator;
import org.deeplearning4j.earlystopping.termination.
    MaxEpochsTerminationCondition;
import org.deeplearning4j.earlystopping.termination.
    MaxTimeIterationTerminationCondition;
import org.deeplearning4j.earlystopping.trainer.EarlyStoppingTrainer;
import org.deeplearning4j.util.ModelSerializer;
//import org.nd4j.jita.conf.CudaEnvironment;
```

```
public class MLPClassifierMoon {

public static final String CAL_ACCURACY = "ACCURACY";
public static final String CAL_YOUDEN = "YOUDEN";
public static final String CAL_F1 = "F1";
public static final String CAL_PPV1 = "PPV1";
public static final String CAL_SENSITIVITY = "SENSITIVITY";


int batchSize;
int nEpochs;
int iterations;
int horas;
double tol;

RecordReader rr = new CSVRecordReader();
RecordReader rrTest = new CSVRecordReader();
InputSplit isRR;
InputSplit is2RRTest;
DataSetIterator trainIter;
DataSetIterator testIter;

double b[] = {0.1, 0.2, 0.3, 0.4, 0.5, 0.6, 0.7, 0.8, 0.9};
String modelo;
String modelo2;
String yaml;
String texto;
String nombrespropios;
String counts2;
String direction;

List<Integer> ptoscortey = new ArrayList();
List<Integer> ptoscortea = new ArrayList();
List<Integer> ptoscortef1 = new ArrayList();
List<Integer> ptoscorteppv1 = new ArrayList();
List<Integer> ptoscorteSensitivity = new ArrayList();

int numOutputs;

public MLPClassifierMoon(int batchSize, int nEpochs, InputSplit isRR,
    InputSplit is2RRTest, String modelo, int numOutputs, int
    iterations, int horas, String modelo2, double tol, String yaml,
    String texto, String nombrespropios, String counts2, String
    direction) throws IOException, InterruptedException {

this.iterations = iterations;
this.horas = horas;
this.batchSize = batchSize;
this.nEpochs = nEpochs;
this.isRR = isRR;
this.is2RRTest = is2RRTest;
rr.initialize(isRR);
trainIter = new RecordReaderDataSetIterator(rr, batchSize, 0, 2);
rrTest.initialize(is2RRTest);
testIter = new RecordReaderDataSetIterator(rrTest, batchSize, 0, 2);
this.modelo = modelo;
this.modelo2 = modelo2;
```

```
this.numOutputs = numOutputs;
this.tol = tol;
this.yaml = yaml;
this.texto = texto;
this.nombrespropios = nombrespropios;
this.counts2 = counts2;
this.direction = direction;

}


//*************ENTRENAMIENTO***************

//Entrenamiento dependiente de la configuracion obtenida en la clase
   NetworksConfiguration.java
public MultiLayerNetwork entrenar(MultiLayerConfiguration conf) throws
    FileNotFoundException, IOException {

MultiLayerNetwork model = new MultiLayerNetwork(conf);
model.init();
model.setListeners(new ScoreIterationListener(1)); //imprime el error
   de cada iteracion del entrenamiento

//Entrenamos el modelo hasta que se alcance el primero de estos 3
   criterios: error<tolerancia, numero de epocas o tiempo
   computacional.
int n = 0;
Instant start = Instant.now(); //empieza a contar el tiempo desde
   ahora

//Mientras no sobrepase el numero de epocas sobrefijado
while (n < nEpochs) {
    model.fit(trainIter);  //entrenamos el modelo
    System.out.println("En la epoca--> " + n);
    FileOutputStream fos = new FileOutputStream(modelo2);
    ModelSerializer.writeModel(model, fos, true);  //escribimos el
       modelo creado en esta epoca
    FileInputStream fis = new FileInputStream(modelo2); //lo abrimos
       para comprobar si se da alguno de los criterios de parada
    MultiLayerNetwork network = ModelSerializer.
       restoreMultiLayerNetwork(fis);

    List<Double> a = new ArrayList();
    int conteo = 0;
    while (testIter.hasNext()) {

        DataSet t = testIter.next();
        System.out.println("a[conteo]-->" + network.score(t));

        a.add(network.score(t)); //add los scores de cada batch en la
           epoca actual del conjunto test
        conteo++; //cuenta el numero de batchs que hay para hacer
           luego la media
    }
    testIter.reset(); //reseteamos el test para la siguiente epoca

    //Calculamos la media del error de todos los batchs en cada epoca
```

```
    double result = 0.0;
    for (int i = 0; i < conteo; i++) {
        result += a.get(i);
    }
    if (result / conteo > tol) { //si la media de cada epoca de todos
       los batchs en mayor que una cierta tolerancia sigo
        Instant end = Instant.now(); //Calculo el tiempo que ha pasado
            hasta terminan esta epoca
        Duration timeElapsed = Duration.between(start, end);
        if (timeElapsed.toDays() < 9) { //si no supera los 9 dias sigo
            (paso a la siguiente etapa y vuelvo a proceder a los
           pasos anteriores)
            System.out.println("Time taken in the epoch " + n + " is:
               " + timeElapsed.toMillis() + " milliseconds");
            n++;
        } else { //si el tiempo supera los 9 minutos termina el
           entrenamiento
            n = nEpochs;
        }
    } else { //si la media del error de todos los batchs en la epoca
       actual supera la tolerancia puesta termina el entrenamiento
        n = nEpochs;
    }

}

return model; //me devuelve el modelo creado verificando los 3
   criterios de parada

}


//Entrenamiento en funcion de un .yaml
public MultiLayerNetwork entrenaryaml() throws FileNotFoundException,
   IOException {

//Abro el archivo.yaml y creo la configuracion de la red neuronal a
   partir de el
BufferedReader leo = new BufferedReader(new FileReader(yaml));
String leo1;
StringBuilder auxiliar = new StringBuilder();
while ((leo1 = leo.readLine()) != null) {
    auxiliar.append(leo1 + "\n");
}

MultiLayerConfiguration conf = MultiLayerConfiguration.fromYaml(
   auxiliar.toString());


//A partir de aqui es lo mismo que la funcion de entrenamiento
   anterior
MultiLayerNetwork model = new MultiLayerNetwork(conf);
model.init();
model.setListeners(new ScoreIterationListener(1));
//model.setListeners(new HistogramIterationListener(1));
```

```
//Entrenamos el modelo hasta que se alcance el primero de estos 3
   criterios: error<tolerancia, numero de epocas o tiempo
   computacional.
int n = 0;
Instant start = Instant.now(); //empieza a contar el tiempo desde
   ahora

FileOutputStream fos = new FileOutputStream(modelo2);
//FileInputStream fis = new FileInputStream(modelo2); //lo abrimos
   para comprobar si se da alguno de los criterios de parada

//Mientras no sobrepase el numero de epocas sobrefijado
while (n < nEpochs) {
    model.fit(trainIter);  //entrenamos el modelo
    //wrapper.fit(trainIter);
    System.out.println("En la epoca--> " + n);

    ModelSerializer.writeModel(model, fos, true);  //escribimos el
       modelo creado en esta epoca

    List<Double> a = new ArrayList();
    int conteo = 0;
    while (testIter.hasNext()) {
        DataSet t = testIter.next();
        System.out.println("a[conteo]-->" + model.score(t));
        a.add(model.score(t)); //add los scores de cada batch en la
           epoca actual del conjunto test
        conteo++; //cuenta el numero de batchs que hay para hacer
           luego la media
    }
    testIter.reset(); //reseteamos el test para la siguiente epoca

    //Calculamos la media del error de todos los batchs en cada epoca
    double result = 0.0;
    for (int i = 0; i < conteo; i++) {
        result += a.get(i);
    }
    if (result / conteo > tol) { //si la media de cada epoca de todos
       los batchs en mayor que una cierta tolerancia sigo
        Instant end = Instant.now(); //Calculo el tiempo que ha pasado
            hasta terminan esta epoca
        Duration timeElapsed = Duration.between(start, end);
        if (timeElapsed.toHours() < 50) { //si no supera los 9 minutos
            sigo (paso a la siguiente etapa y vuelvo a proceder a los
            pasos anteriores)
            System.out.println("Time taken in the epoch " + n + " is:
               " + timeElapsed.toMillis() + " milliseconds");
            n++;
        } else { //si el tiempo supera los 9 minutos termina el
           entrenamiento
            n = nEpochs;
        }
    } else { //si la media del error de todos los batchs en la epoca
       actual supera la tolerancia puesta termina el entrenamiento
        n = nEpochs;
    }
```

```
}

return model; //me devuelve el modelo creado verificando los 3
    criterios de parada

}


//Entrenamiento con earlystopping con configuracion de la clase
    NetworksConfiguration.java
public void earlystopping(MultiLayerConfiguration conf, String modelo)
     {

//La configuracion de EarlyStopping
EarlyStoppingConfiguration esConf = new EarlyStoppingConfiguration.
    Builder()
.epochTerminationConditions(new MaxEpochsTerminationCondition(
    iterations)) //maximo de epocas
.iterationTerminationConditions(new
    MaxTimeIterationTerminationCondition(horas, TimeUnit.HOURS)) //
    maximo de horas
.scoreCalculator(new DataSetLossCalculator(testIter, true)) //calcula
    el error del dataset test
.evaluateEveryNEpochs(1)
.modelSaver(new LocalFileModelSaver(modelo)) //guarda el modelo (se
    guarda con el nombre de bestModel.bin)
.build();

EarlyStoppingTrainer trainer = new EarlyStoppingTrainer(esConf, conf,
    trainIter); //entreno el modelo

EarlyStoppingResult result = trainer.fit();

//Print out the results:
System.out.println("Termination reason: " + result.
    getTerminationReason());
System.out.println("Termination details: " + result.
    getTerminationDetails());
System.out.println("Total epochs: " + result.getTotalEpochs());
System.out.println("Best epoch number: " + result.getBestModelEpoch())
    ;
System.out.println("Score at best epoch: " + result.getBestModelScore
    ());

//Get the best model:
MultiLayerNetwork model;
model = (MultiLayerNetwork) result.getBestModel();

System.out.println("Ya se ha entrenado el modelo");
}


//Entrenamiento con earlystopping con .yaml
public MultiLayerNetwork earlystoppingYaml() throws
    FileNotFoundException, IOException { //multilayernetwork por void
//Abro el archivo.yaml y creo la configuracion de la red neuronal a
    partir de el
```

```
BufferedReader leo = new BufferedReader(new FileReader(yaml));
String leo1;
StringBuilder auxiliar = new StringBuilder();
while ((leo1 = leo.readLine()) != null) {
    auxiliar.append(leo1).append("\n");
}
MultiLayerConfiguration conf = MultiLayerConfiguration.fromYaml(
   auxiliar.toString());


EarlyStoppingModelSaver saver = new LocalFileModelSaver(direction);

//La configuracion de EarlyStopping
EarlyStoppingConfiguration esConf = new EarlyStoppingConfiguration.
   Builder()
.epochTerminationConditions(new MaxEpochsTerminationCondition(
   iterations)) //maximo de epocas
.iterationTerminationConditions(new
   MaxTimeIterationTerminationCondition(horas, TimeUnit.HOURS)) //
   maximo de horas
.scoreCalculator(new DataSetLossCalculator(testIter, true)) //calcula
   el error del dataset test
.evaluateEveryNEpochs(1)
.modelSaver(saver)
.build();

EarlyStoppingTrainer trainer = new EarlyStoppingTrainer(esConf, conf,
   trainIter); //entreno el modelo

EarlyStoppingResult result = trainer.fit();

//Print out the results:
System.out.println("Termination reason: " + result.
   getTerminationReason());
System.out.println("Termination details: " + result.
   getTerminationDetails());
System.out.println("Total epochs: " + result.getTotalEpochs());
System.out.println("Best epoch number: " + result.getBestModelEpoch())
   ;
System.out.println("Score at best epoch: " + result.getBestModelScore
   ());


Map<Integer,Double> scoreVsEpoch = result.getScoreVsEpoch();
List<Integer> list = new ArrayList<>(scoreVsEpoch.keySet());
Collections.sort(list);
System.out.println("Score vs. Epoch:");
for( Integer i : list){
    System.out.println(i + "\t" + scoreVsEpoch.get(i));
}
//Get the best model:
MultiLayerNetwork model;
model = (MultiLayerNetwork) result.getBestModel();

model.init();
model.setListeners(new ScoreIterationListener(10));
```

```
System.out.println("Ya se ha entrenado el modelo");
return model;
}



//*************VALIDACION***************

//Crea listas de puntos de corte para 3 metricas consideradas donde el
    ultimo lugar de la lista indica el punto de corte que mejor
   metrica ha obtenido
// Por la forma en la que se ha programado, si hay empate entre
   diferentes puntos de corte, dara como resultado de ``mejor'' el
   mas pequeno
public void prediccion() throws FileNotFoundException, IOException {

List<Double> youden = new ArrayList(); //lista con los valores de
   youden para los diferentes puntos de corte
List<Double> accuracy = new ArrayList(); //lista con los valores de
   accuracy para los diferentes puntos de corte
List<Double> f1 = new ArrayList();  //lista con los valores de f1-
   score para los diferentes puntos de corte
List<Double> ppv1 = new ArrayList();
List<Double> sensitivity = new ArrayList();
//Inicializo la lista con un valor double 0.0
youden.add(0.0);
accuracy.add(0.0);
f1.add(0.0);
ppv1.add(0.0);
sensitivity.add(0.0);


for (int i = 0; i < b.length; i++) {//para cada posicion del array de
   puntos de corte (b)
    FileInputStream fis = new FileInputStream(modelo);
    MultiLayerNetwork network = ModelSerializer.
       restoreMultiLayerNetwork(fis);
    Evaluation eval = new Evaluation(numOutputs);

    while (testIter.hasNext()) {
        DataSet t = testIter.next();
        INDArray features = t.getFeatureMatrix();
        INDArray labels = t.getLabels();
        INDArray predicted1 = network.output(features, false);
        eval.eval1(labels, predicted1, b[i]); //eval1 es el que te
           permite decidir el punto de corte, si no coge punto de
           corte 0.5
    }

    youden.add(eval.youden()); //add el valor de youden para cada
       punto de corte
    accuracy.add(eval.accuracy()); //add el valor de accuracy para
       cada punto de corte
    f1.add(eval.f1(1)); //add el valor de f1-score para cada punto de
       corte
    ppv1.add(eval.precision(1));
    sensitivity.add(eval.recall(1));
```

```
    System.out.println("Punto de corte-->" + b[i] + " accuracy--> " +
       eval.accuracy() + " youden--> " + eval.youden() + " f1-score
       --> " + eval.f1(1) + " sensitivity--> " + eval.recall(1) + "
       ppv1--> " + eval.precision(1));

    if (eval.youden() > youden.get(i)) //si la actual es mayor que la
       anterior
    {
        ptoscortey.add(i); //add esa posicion del punto de corte
    }
    if (eval.accuracy() > accuracy.get(i)) //si la actual es mayor que
        la anterior
    {
        ptoscortea.add(i); //add esa posicion del punto de corte
    }
    if (eval.f1(1) > f1.get(i)) //si la actual es mayor que la
       anterior
    {
        ptoscortef1.add(i); //add esa posicion del punto de corte
    }
    if (eval.precision(1) > ppv1.get(i))
    {
        ptoscorteppv1.add(i);
    }
    if (eval.recall(1) > sensitivity.get(i))
    {
        ptoscorteSensitivity.add(i);
    }

    testIter.reset(); //resetea el conjunto test para el siguiente
       punto de corte
}

//Puede ocurrir que la lista de putosdecorte (son aquellos puntos de
   corte que han superado al anterior) este vacia
//Esto ocurre cuando las metricas son para todos puntos de corte igual
    a 0.0, en este caso, acumulo, por ejemplo, el valor 100 a la
   lista para que no este vacia
//Ese 100 me indicara, ademas de que toma el valor 0 para todos los
   puntos de corte, que es indistinto que punto de corte elegir
if (ptoscortey.size() == 0.0) {
    ptoscortey.add(100);
} else if (ptoscortea.size() == 0.0) {
    ptoscortea.add(100);
} else if (ptoscortef1.size() == 0.0) {
    ptoscortef1.add(100);
} else if (ptoscorteppv1.size() == 0.0) {
    ptoscorteppv1.add(100);
}else if (ptoscorteSensitivity.size() == 0.0) {
    ptoscorteSensitivity.add(100);
}else {//nada

}

}
```

```
//Da las metricas para el mejor punto de corte en funcion de la
    metrica que desee
public void function(String type) throws FileNotFoundException,
    IOException {

List<Integer> ptoscorte = null; //Lista de puntos de corte
String message = "";
if (type == CAL_ACCURACY) {
    message = "Seccion de accuracy";
    ptoscorte = ptoscortea;
} else if (type == CAL_F1) {
    ptoscorte = ptoscortef1;
    message = "Seccion de f1";
} else if (type == CAL_YOUDEN) {
    ptoscorte = ptoscortey;
    message = "Seccion de Youden";
} else if (type == CAL_PPV1) {
    ptoscorte = ptoscorteppv1;
    message = "Seccion de PPV1";
} else if (type == CAL_SENSITIVITY) {
    ptoscorte = ptoscorteSensitivity;
    message = "Seccion de Sensitivity";
}

FileInputStream fiss = new FileInputStream(modelo); //Abro el modelo
    entrenado anteriormente
MultiLayerNetwork networkk = ModelSerializer.restoreMultiLayerNetwork(
    fiss);
Evaluation evall = new Evaluation(numOutputs);


while (testIter.hasNext()) {
    DataSet t = testIter.next();
    INDArray features2 = t.getFeatureMatrix();
    INDArray labelss = t.getLabels();
    INDArray predictedd = networkk.output(features2, false);
    if (ptoscorte.get(0) == 100) { //es cuando no meto ningun pto de
        corte porq el 'type' no supera el 0.0
        evall.eval1(labelss, predictedd, b[0]); //entonces lo evaluo
            para el punto de corte 0.0, por ejemplo, que es el primero
             de la lista
    } else {
        evall.eval1(labelss, predictedd, b[ptoscorte.get(ptoscorte.
            size() - 1)]); //en otro caso, lo evaluo con el mejor
            punto de corte calculado en prediccion
    }
}

System.out.println("
    ****************************************************");
if (ptoscorte.get(0) == 100) {
    System.out.println("El valor " + type + " toma para todos puntos
        de corte el valor 0");
} else {
    System.out.println("Punto de corte-->" + b[ptoscorte.get(ptoscorte
        .size() - 1)]);
}
```

```
System.out.println("Mejor " + message);

System.out.println(evall.stats());

testIter.reset();

}



//*************PREDICCION***************

public void predictionlabels(org.deeplearning4j.nn.multilayer.
    MultiLayerNetwork model) {

while (testIter.hasNext()) {
     DataSet t = testIter.next();
     System.out.println("PREDICCIONEEEES--> " + model.predict(t));
}

}


public static void main(String[] args) throws Exception {

Instant start = Instant.now();
int numOutputs = 2;

String default1[] = {
"C:\\Users\\raznar\\Documents\\NetBeansProjects\\dl4j-examples-master
    \\dl4j-examples\\src\\main\\resources\\classification\\
    matrixAncora100NPEntr.csv", //matriz_Entrenamiento
"C:\\Users\\raznar\\Documents\\NetBeansProjects\\dl4j-examples-master
    \\dl4j-examples\\src\\main\\resources\\classification\\
    matrixAncora100NPVali.csv", //matriz_Validacion
"C:\\Users\\raznar\\Documents\\NetBeansProjects\\ActualizandoFormato
    \\100documentoAncoraNP.txt", //textoValidacion
"C:\\Users\\raznar\\Documents\\NetBeansProjects\\dl4j-examples-master
    \\dl4j-examples\\counts1.txt",
"C:\\Users\\raznar\\Documents\\NetBeansProjects\\dl4j-examples-master
    \\dl4j-examples\\counts2.txt",
"C:\\Users\\raznar\\Documents\\NetBeansProjects\\dl4j-examples-master
    \\dl4j-examples\\0hiddenlayers7.yaml", //configuracion yaml
"bestModel.bin", //nombre del modelo
"100", "2", //batchs y epocas
"C:\\Users\\raznar\\Documents\\NetBeansProjects\\dl4j-examples-master
    \\dl4j-examples", //donde guarda el nombre del modelo de early
    stopping
"2", "12", // epocas y tiempo
"modelo0", //el modelo que va entrenando y guardando
"0.05", //tolerancia
"nombrespropios.txt",
"counts2.txt"};
if (args.length == 0) {
args = default1;
} else if (args.length != 9) {
```

```
new Exception("Necesitamos 9 parametros");
}



MLPClassifierMoon entrenamiento = new MLPClassifierMoon(
Integer.parseInt(args[7]),
Integer.parseInt(args[8]),
new FileSplit(new File(args[0])),
new FileSplit(new File(args[1])),
args[6],
numOutputs,
Integer.parseInt(args[10]),
Integer.parseInt(args[11]),
args[6],
Double.parseDouble(args[13]),
args[5],
args[2],
args[14],
args[15],
args[9]
);

//Entrenamiento sin yaml
MultiLayerConfiguration conf = new NetworksConfiguration(numOutputs).
   getConfiguration();
String conf1 = conf.toYaml();
System.out.println(conf1);

MultiLayerNetwork model = entrenamiento.entrenar(conf);
FileOutputStream fos = new FileOutputStream(args[6]);
ModelSerializer.writeModel(model, fos, true);
System.out.println("Ya se ha entrenado el modelo");


//Entrenamiento con yaml
MultiLayerNetwork model = entrenamiento.entrenaryaml();
FileOutputStream fos = new FileOutputStream(args[6]);
ModelSerializer.writeModel(model, fos, true);
System.out.println("Ya se ha entrenado el modelo");


//Entrenamiento con earlystopping
MultiLayerNetwork model = entrenamiento.earlystoppingYaml();
FileOutputStream fos = new FileOutputStream(args[6]);
ModelSerializer.writeModel(model, fos, true);
System.out.println("Ya se ha entrenado el modelo");



RecordReader rrTest = new CSVRecordReader();
rrTest.initialize(new FileSplit(new File(args[1])));
DataSetIterator testIter1 = new RecordReaderDataSetIterator(rrTest,
   Integer.parseInt(args[7]),0,2);


FileInputStream fis = new FileInputStream(args[6]);
```

```
MultiLayerNetwork model1 = ModelSerializer.restoreMultiLayerNetwork(
    fis);


int contamos=0;
System.out.println("Evaluate model....");
Evaluation eval = new Evaluation(numOutputs);
while(testIter1.hasNext()){
contamos++;
DataSet t = testIter1.next();
INDArray features = t.getFeatureMatrix();
INDArray lables = t.getLabels();
INDArray predicted = model1.output(features,false);

eval.eval(lables, predicted);
System.out.println("EL BATCH--> " + contamos);

}

System.out.println(eval.stats());



//Validacion. Mejor punto de corte
entrenamiento.prediccion();
entrenamiento.function(MLPClassifierMoon.CAL_ACCURACY);
entrenamiento.function(MLPClassifierMoon.CAL_F1);
entrenamiento.function(MLPClassifierMoon.CAL_YOUDEN);
entrenamiento.function(MLPClassifierMoon.CAL_SENSITIVITY);
entrenamiento.function(MLPClassifierMoon.CAL_PPV1);



Instant end = Instant.now();
Duration timeElapsed = Duration.between(start, end);

System.out.println("Time taken is: " + timeElapsed.toMillis() + "
    milliseconds");



}
}
```

Matriz con Word2vec

El modelo word2vec devuelve un fichero .txt donde en cada línea aparece un determinado token y los valores en sus coordenadas. La matriz de datos original no dispone de identificador de cada token, luego se deberá leer el fichero de texto token a token y buscar el determinado token en el fichero .txt del word2vec. Las coordenadas de su vector se añadirán a la matriz correspondiente de datos. El código implementado es el siguiente:

Código

```
package unizar.es;

import java.io.BufferedReader;
import java.io.FileReader;
import java.io.FileWriter;
import java.io.IOException;

/*
* @author raznar
*/
public class WordToVec {

    FileWriter fichero;
    FileWriter fichero2;
    public String nombreFicheroDestino;
    public String nombreFicheroDestino2;
    StringBuilder aux = new StringBuilder();
    String bfRead1;
    String bfRead2;
    String bfRead3;
    String bfRead4;

    public WordToVec(String nombreFicheroDestino2) {
        this.nombreFicheroDestino2 = nombreFicheroDestino2;
    }

    public String CompletandoMatriz(String direccion1, String
       direccion2, String direccion3, String direccion4) throws
       IOException { //direccion del archivo

        String texto = " ";

        fichero2 = new FileWriter(nombreFicheroDestino2);

        BufferedReader bf2 = new BufferedReader(new FileReader(
           direccion2));
        BufferedReader bf3 = new BufferedReader(new FileReader(
           direccion3));
        BufferedReader bf4 = new BufferedReader(new FileReader(
           direccion4));

        String[] tokens2;
        String[] tokens4;
        int count = 0;
        int conteo = 0;

        // Esta parte solo si se pone el texto limpio sin tags en una
           sola linea
        /*
        while ((bfRead1 = bf1.readLine()) != null) {//el texto limpio
            tokens1 = bfRead1.split(" ");
            for (int j = 0; j < tokens1.length; j++) {
                aux.append(tokens1[j]).append(" ");
                fichero.write(aux.toString());
                aux = new StringBuilder();
            }
        }
```

```
            fichero.close();  //todo el texto en la misma linea
            */

            while ((bfRead4 = bf4.readLine()) != null) {//el texto limpio
                en la misma linea
                tokens4 = bfRead4.split(" "); //todos los tokens que
                    aparecen en el texto
                while ((bfRead3 = bf3.readLine()) != null) { //la matriz
                    de datos
                    bf2 = new BufferedReader(new FileReader(direccion2));
                    while ((bfRead2 = bf2.readLine()) != null) { //el
                        texto con wordToVec
                        tokens2 = bfRead2.split(" ");
                        if ((tokens2[0].trim().equalsIgnoreCase(tokens4[
                            count].trim()))) { //siempre que el token
                            determinado del word2vec corresponda con el
                            token determinado del texto
                            conteo++;
                            System.out.println(conteo);

                            aux.append(bfRead3).append(","); //guardo la
                                linea de la matriz de datos y add las
                                coordenadas del vector word2vec (variables
                                 adicionales)
                            fichero2.write(aux.toString());
                            aux = new StringBuilder();
                            for (int i = 1; i < 6; i++) { //dimensiones
                                del word2vec
                                if (i == 5) { //si es la ultima coordenada
                                    , guardarla y salto de linea
                                    aux.append(tokens2[i]).append("\n");
                                    fichero2.write(aux.toString());
                                    aux = new StringBuilder();
                                } else { //guardar las coordenadas del
                                    vector
                                    aux.append(tokens2[i]).append(",");
                                    fichero2.write(aux.toString());
                                    aux = new StringBuilder();
                                }
                            }
                            fichero2.write(aux.toString());
                            aux = new StringBuilder();
                            break;
                        }

                    }
                    count++;
                    fichero2.write(aux.toString());
                }
                fichero2.close();
            }
            System.out.println("conteooo-->" + count);
            return texto;
        }

    }
```