ESCUELA TÉCNICA SUPERIOR DE INGENIERÍA INFORMÁTICA

Grado en Ingeniería Informática

Curso Académico 2018-2019

Trabajo Fin de Grado

Estudio práctico sobre modelos de secuencias con redes neuronales recurrentes para la generación de texto

Autor

Guillem García Subies

Tutores

Javier Martínez Moguerza

Álvaro Barbero Jiménez

RESUMEN

La redes neuronales, en concreto las redes neuronales recurrentes, han obtenido un alto nivel de popularidad en los últimos años. Y esto viene dado, entre otras cosas, por el gran abanico de problemas que consiguen solucionar con unos resultados mucho mejores a todo lo que existía antes.

En concreto, un área del aprendizaje automático todavía por explotar y que está cobrando una gran importancia hoy en día, son los modelos del lenguaje para el procesado de secuencias de texto. Gracias a las redes neuronales hay una gran variedad de propuestas para la creación de modelos del lenguaje polivalentes que pueden servir desde para resumir textos, a generar noticias falsas en base a un solo titular, pasando por la comprensión lectora y la respuesta a preguntas en base a un texto.

Dadas las grandes dimensiones de este tema y sus dos grandes ramas; la teórica y la técnica, se ha optado por hacer dos Trabajos de Fin de Grado con la misma temática. Uno enfocado en la práctica, para el Grado en Ingeniería Informática y otro enfocado en las teorías subyacientes que permiten usar técnicas de procesado de secuencias de texto con herramientas software, para el Grado en Matemáticas.

En este trabajo se ha puesto el foco en aprender las técnicas y metodologías de programación de soluciones de aprendizaje automático, ahondando en las redes neuronales, el procesamiento del lenguaje natural y cómo juntar todas las partes usando únicamente software libre.

ÍNDICE GENERAL

ÍNDICE DE FIGURAS

ÍNDICE DE TABLAS

1. INTRODUCCIÓN

1.1. Motivación del trabajo

Este estudio viene motivado por el auge de los métodos de procesado del lenguaje natural (PLN) que cada vez están más presentes en la vida de la gente, por ejemplo podemos encontrarlos en asistentes inteligentes como Alexa de Amazon, en los teclados predictivos de los teléfonos móviles actuales, las respuesta automáticas [1] y un largo etcétera. Los métodos más usados en esta disciplina son las redes neuronales recurrentes, en concreto las redes neuronales de larga memoria a corto plazo (LSTM, por sus siglas en inglés) [2], sin embargo hay empresas que no desvelan completamente todos los métodos que usan [3].

Dado el claro interés por parte de la industria en estos métodos y su proyección futura, es importante poder replicarlos de forma abierta y replicable, con software libre, para que el conocimiento no esté oculto, con las posibles implicaciones que eso tendría (creación de las llamadas "fake news" de forma masiva y automática, creación masiva de críticas hacia personajes públicos en redes sociales con el fin de desprestigiarlos, robos de identidad, etc.).

1.2. Objetivos generales

En este trabajo se ha puesto el foco en el estudio de los modelos de lenguaje, en concreto en la generación de texto. La principal motivación es que un modelo que puede predecir la siguiente palabra de un texto, en el fondo, es un modelo que ha aprendido mucho sobre gramática y semántica, de forma que puede ser utilizado para una gran cantidad de aplicaciones prácticas. Por ejemplo, los teclados inteligentes que predicen la siguiente palabra del discurso o los bots de conversación [4], generadores de noticias en base a frases simples con los datos relevantes [5], etc. Se pretende estudiar las diferentes de soluciones que el estado del arte propone para estos problemas e implementarlas para comprobar su eficacia.

Sin embargo, uno de los principales problemas a los que se puede afrontar un investigador que intente recrear los resultados de las grandes multinacionales, es el poco poder de cómputo o la dificultad de encontrar soluciones de código abierto para replicar los algoritmos y demás procesos que se suelen describir en los *papers*. De este modo, pues, se van a estudiar arquitecturas de redes neuronales recurrentes con *embeddings* y con capas LSTM [6] y GRU (más fáciles de entrenar) [7] [8] usando software de código abierto y gratuito.

Por otra parte, se va a intentar, también, usar los modelos de secuencias con redes neu-

ronales recurrentes para obtener una métrica de complejidad del texto que vaya más allá de las métricas simples tradicionales, que solo se basan en qué palabras se han utilizado.

Por lo tanto, el principal objetivo será explorar las técnicas del estado del arte en generación automática de texto y modelado de lenguajes e implementar modelos sencillos acorde a las limitaciones hardware y software libre. El objetivo secundario se centrará en la creación de una métrica de complejidad del texto.

2. MARCO TEÓRICO

A continuación se presenta un breve resumen sobre el estudio llevado a cabo en el Trabajo de Fin de Grado de Matemáticas [9] con respecto a los métodos de Procesado del Lenguaje Natural basados en modelos de secuencias y redes neuronales para la generación de texto.

2.1. Procesado del lenguaje natural

El procesado del lenguaje natural es la disciplina, dentro de la inteligencia artificial, que se encarga de estudiar las interacciones entre ordenadores y humanos mediante el lenguaje natural (ya sea oral o escrito).

Desde los años 50, casi en paralelo a la aparición de las computadoras modernas, ya se especulaba con la posibilidad de tener máquinas parlantes que fueran indistinguibles de un humano [10]. Obviamente, poco o nada distaba esto de la filosofía por aquella época. Sin embargo, con el paso de los años y la mejora de capacidad de cálculo de los ordenadores, estamos presenciando una grandísima evolución de esta disciplina. Tal es dicha evolución que organizaciones como OpenAI aseguran que no hacen pública la totalidad de sus modelos porque están preocupados por los posibles usos maliciosos que se les dé [3] [11].

Actualmente, este campo es muy interdisciplinario ya que agrupa a lingüistas computacionales, ya que aun hay muchas reglas del lenguaje que serían difíciles de aprender para una máquina sin ayuda externa; a informáticos, para optimizar los procesos y disminuir los altos costes computacionales de procesar lenguaje natural, y a matemáticos, para crear nuevos algoritmos y teoremas que aprovechen todo el poder computacional actual.

2.2. Aprendizaje automático

Goodfellow [12] explica que “un algoritmo de aprendizaje automático es aquél que puede aprender de los datos”. Sin embargo, esta definición se queda corta pues hay que explicar qué significa aprender. Tanto Goodfellow como Mitchell [13] concuerdan en que para que un algoritmo aprenda se necesita una experiencia E, una tarea a desempeñar T, una métrica para cuantificar el rendimiento P que mejora con la experiencia E.

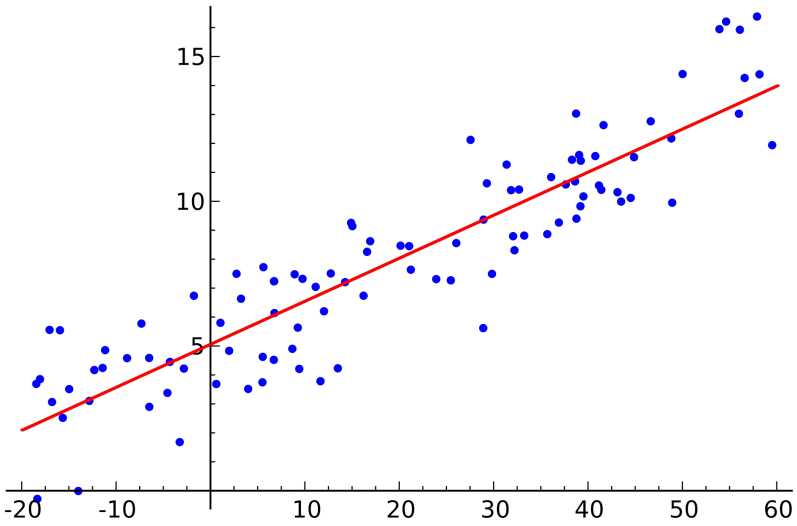

Fig. 2.1. Ejemplo de una regresión lineal [14]

Ejemplos de problemas que pueden resolver los algoritmos de aprendizaje automático son los de regresión 2.1 o *clustering* 2.2, entre otros.

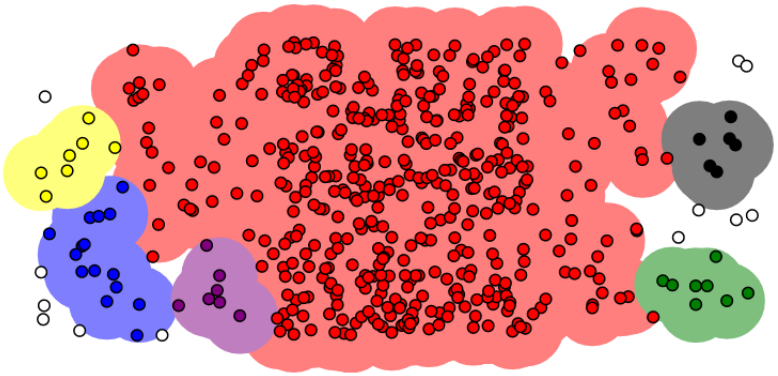

Fig. 2.2. Ejemplo de clustering en dos dimensiones [15]

2.3. Redes neuronales artificiales

Pese a que mucha gente piense que las redes neuronales funcionan igual que el cerebro, realmente no es así; solo tienen una vaga inspiración en las neuronas.

2.3.1. La neuronas

Una neurona biológica es una célula que responde a impulsos eléctricos y que se comunica con otras neuronas usando unos conexiones llamadas sinapsis. Estos impulsos entran por las dendritas, liberando neurotransmisores, los cuales hacen que cambie el voltaje dentro de la neurona (en el soma). Si estos cambios pasan un cierto umbral, la neurona produce una señal eléctrica que se transmite a otras neuronas a través del axón.

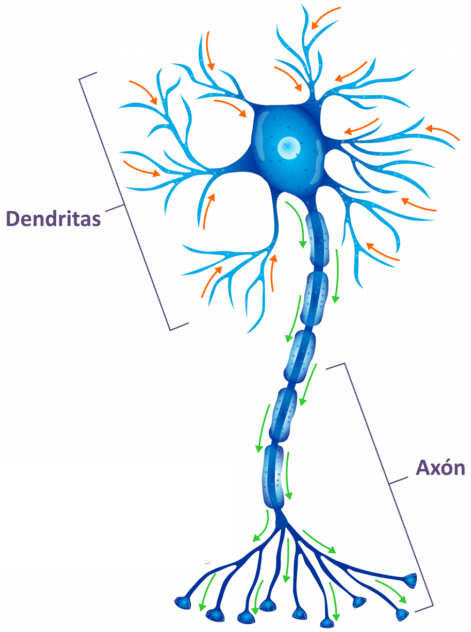

Fig. 2.3. La neurona biológica [16]

2.3.2. El perceptrón

El perceptrón [17] es la base de las redes neuronales. Se podría asemejar a lo que es una neurona. Básicamente el perceptrón es un algoritmo que recibe unos *inputs* y, clasifica las muestras de los datos de forma binaria. Para aprender a clasificar, el perceptrón tiene una serie de pesos que se van ajustando a medida que se entrena usando el llamado algoritmo de convergencia del perceptrón.

Sin embargo el perceptrón es una estructura demasiado simple y, entre otras cosas, no tiene la convergencia asegurada cuando las dos clases no son linealmente separables.

2.3.3. El perceptrón multicapa

El perceptrón multicapa es una red cuyos nodos son perceptrones, como podemos ver en 2.4 (donde cada punto gris representaría un perceptrón). Esta red consta de una capa de *input* y otra de *output* y una o varias capas intermedias u ocultas.

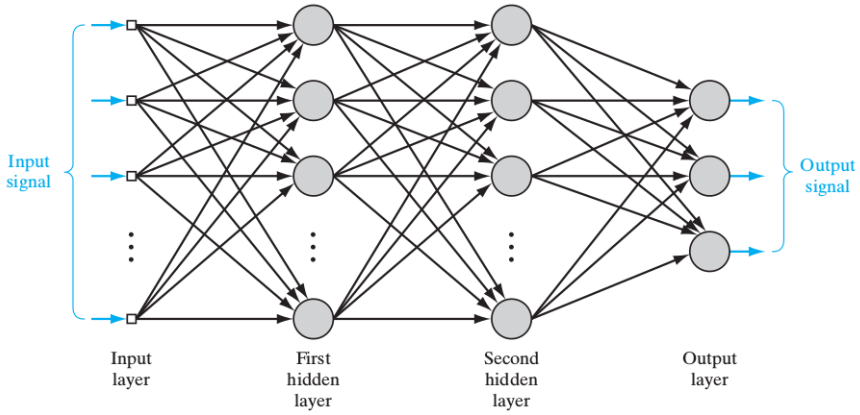

Fig. 2.4. La arquitectura de un perceptrón multicapa [18]

Gracias al perceptrón multicapa, se consiguen solucionar muchos otros problemas que el perceptrón no podía.

Además, el perceptrón multicapa se puede ver como un problema en el que hay que minimizar una función de error que compara el *output* de la red con la solución real. Esto es muy útil ya que hace que se puedan usar algoritmos de optimización para entrenar las redes neuronales y se puede hacer de forma distribuida. Si a esto se le suma el algoritmo de *backpropagation* que sirve para calcular eficientemente todos los gradientes necesarios para llevar a cabo estas optimizaciones, se consigue que las redes neuronales sean altamente escalables.

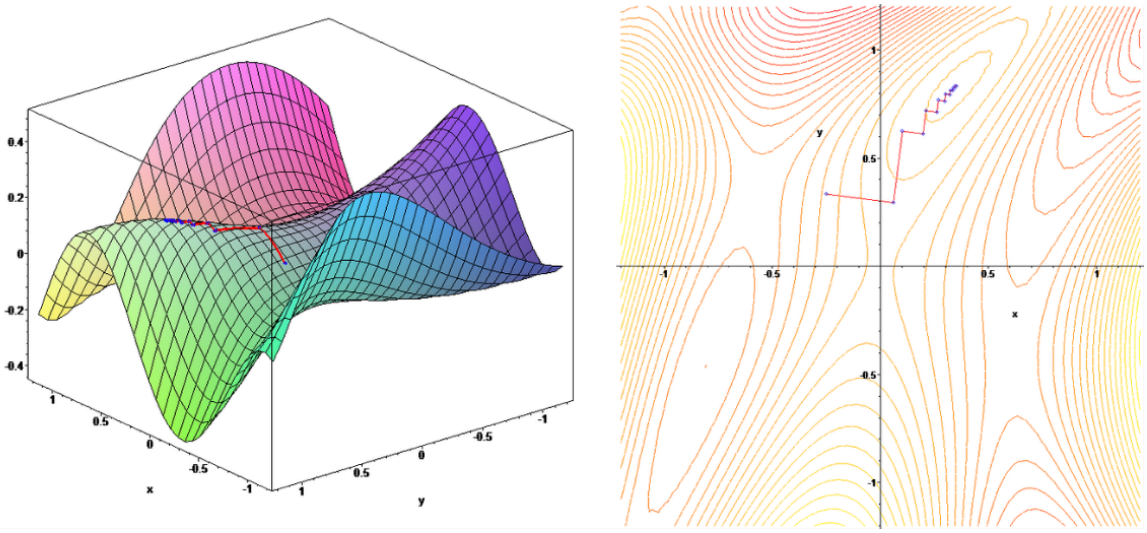

Fig. 2.5. Ilustración del descenso por gradiente, usado para entrenar redes neuronales [19]

2.3.4. Redes neuronales recurrentes

Las redes neuronales recurrentes [20] [21] se centran en intentar procesar datos con forma de secuencias como, por ejemplo, listas de palabras o series temporales.

La novedad que introducen este tipo de redes es el uso compartido de parámetros a

través de diferentes partes del modelo. Este parámetro actúa como si fuera una “memoria” de la red en puntos clave que, de otra forma, probablemente no conseguiría retener.

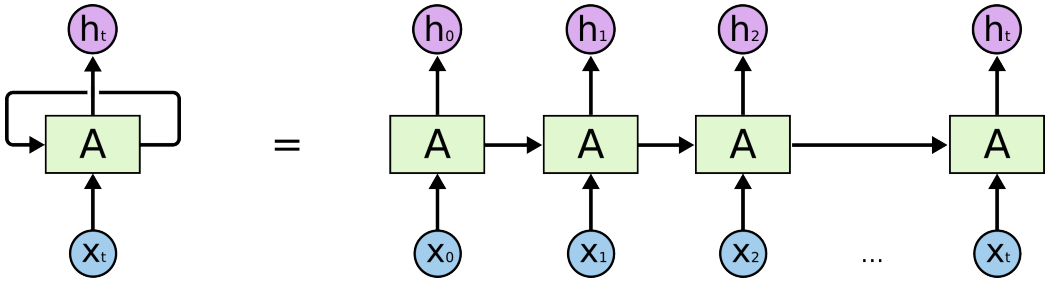

Fig. 2.6. Red neuronal recurrente y su desenrollado [22]

2.3.5. LSTM

Las redes de larga memoria de corto plazo (LSTM, por sus siglas inglesas Long Short Term Memory) [6] son un tipo especial de redes recurrentes diseñadas para recordar dependencias a largo plazo e intentar paliar el problema de los gradientes desvanecientes.

Abajo podemos ver una imagen de que hay dentro de una red recurrente; solo una función, sin embargo, las LSTM tienen un comportamiento más complejo.

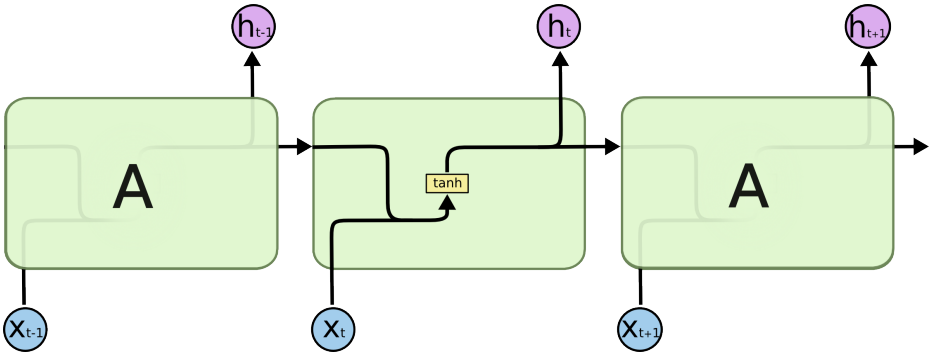

Fig. 2.7. Red neuronal recurrente por dentro [22]

En la figura 2.8 podemos ver que, en vez de tener una simple capa de activación, hay cuatro capas distintas.

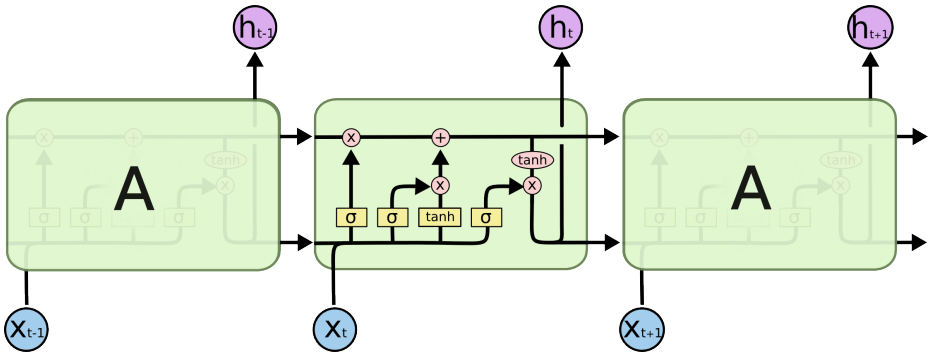

Fig. 2.8. LSTM por dentro [22]

Este tipo de redes se han usado con éxito para crear textos con el estilo de Cervantes, Lovecraft o J.K. Rowling, entre otros. [23]. También pueden incluso aprender a hacer código fuente en C o *papers* en Latex [24]. Obviamente, ni los relatos tienen mucho sentido a la larga ni el código que se genera es útil. Sin embargo, esto sirve para demostrar como las LSTM pueden aprender complejas estructuras sintácticas.

También han surgido variantes de las LSTM, como las redes GRU [7] [8], que intentan mejorar su rendimiento cuando hay pocos datos simplificando en gran medida su estructura.

2.3.6. Redes bidireccionales

Las redes recurrentes bidireccionales [25] son simplemente dos redes neuronales recurrentes juntas. Los *inputs* de una van en orden normal mientras que los de la otra van en orden inverso. Después, los *outputs* de ambas se concatenan.

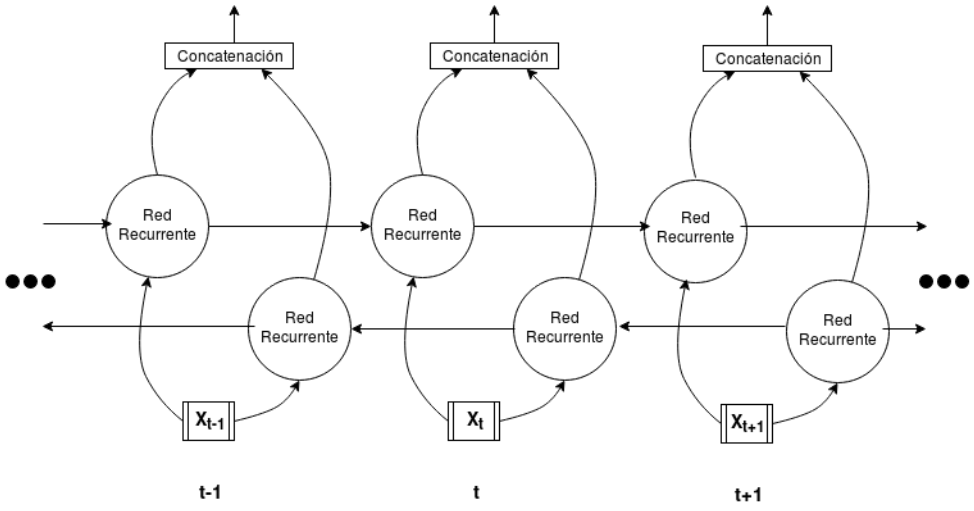

Fig. 2.9. Estructura de una red bidireccional

Esta estructura permite a las redes tener información tanto hacia adelante en la secuencia de texto como hacia atrás. Esta es la razón por las que también se usan bastante en el tipo problema que queremos a abordar.

2.4. Métodos para el procesado de lenguaje natural

Como hemos visto anteriormente, los métodos de aprendizaje automático (en particular las redes neuronales), no aceptan texto como *input*, por lo tanto hay que preprocesarlo antes de usar este tipo de métodos.

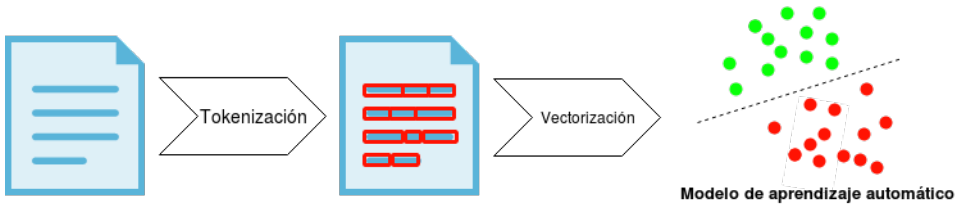

Fig. 2.10. Estructura típica de una solución de PLN

Antes que nada hay que simplificar el texto. Según la tarea que deseemos realizar, nos interesará eliminar tildes, mayúsculas, signos de puntuación, etc. De esta forma se podría hacer, por ejemplo, que el modelo pueda entender que *casa* es lo mismo que *Casa*. Obviamente, utilizando este tipo de métodos, se pierde alguna información sobre el texto.

Una vez normalizado, hay que dividir el texto en tokens. Generalmente los tokens solo son palabras normalizadas, sin embargo se pueden crear tokens especiales que delimiten el final de una frase, por ejemplo, para que el modelo tenga la posibilidad de aprender a crear frases de principio a fin.

A continuación, hay que transformar los tokens en vectores de igual longitud de forma para que un modelo pueda aprender de ellos. Uno de los métodos más utilizados para esto son los *embeddings* a partir de modelos del lenguaje. Estos se explican en el TFG de Matemáticas [9].

3. DESCRIPCIÓN DEL PROBLEMA

Actualmente hay muchísimas grandes empresas trabajando en la generación de texto de forma automática. Ya sea para creación de noticias, ayuda a la escritura, chatbots, etc. Sin embargo la mayoría de ellas ocultan sus métodos. Además disponen de un poder de cómputo mucho mayor al que puede tener un investigador de una universidad.

Por lo tanto, el problema a abordar es conseguir encontrar recursos software y matemáticos para poder generar texto que sea creíble a partir de una secuencia dada usando métodos de código abierto para el procesado del lenguaje natural y el aprendizaje automático.

3.1. Objetivos específicos

Dado el objetivo general de obtener un método que pueda resolver el problema propuesto satisfactoriamente, podemos desgranarlo en los siguientes

Objetivos técnicos:

- Obtención un modelo de PLN y redes neuronales capaz de generar texto de forma automática.
- Elaboración de una librería de software que permita replicar los experimentos fácilmente y que cumpla con los siguientes requisitos mínimos:
  - Tener una API ajustada a los estándares de aprendizaje automático.
  - Tener una documentación fácil de entender y útil.
  - Ser fácil de desplegar en cualquier sistema operativo.
  - Ser de código abierto y pública.
  - Tener una estructura y arquitectura del código que la haga mantenible y eficiente.
  - Usar estructuras de datos adecuadas para tanto textos de gran tamaño como de pequeño tamaño.
- Utilización de herramientas de software del estado del arte para optimizar al máximo el proceso del tratamiento del texto.
- Utilización de métodos matemáticos y algorítmicos del estado del arte para obtener los mejores resultados posibles.
- Creación de una métrica de complejidad del texto avanzada.

Objetivos académicos:

- Aprendizaje de las técnicas punteras de PLN.
- Aprendizaje de las técnicas punteras de redes neuronales, redes neuronales profundas y redes neuronales recurrentes.
- Aprendizaje de las metodologías de desarrollo de proyectos de aprendizaje automático.
- Aprendizaje de uso de las principales librerías y lenguajes de programación para aprendizaje automático, redes neuronales y tratamiento de datos.
- Aprendizaje sobre la cultura del software de código abierto y sobre como crear un proyecto de código abierto.
- Aprendizaje de patrones de diseño de software orientados al aprendizaje automático.
- Aprendizaje sobre métricas de complejidad de textos.

3.2. Solución propuesta

Desde el punto de vista informático, la solución pasa por conseguir un código fácil de usar y, sobre todo, de mantener. Dado que el campo del aprendizaje automático está en continua evolución, las librerías de código abierto están en constante cambio, lo que complica la mantenibilidad del código. Por esta razón, es imprescindible que se sigan metodologías concretas de desarrollo y buenas prácticas de programación.

Por lo tanto, la solución (que se explicará en el capítulo siguiente), se va a basar en:

- Uso del *Cross Industry Standard Process for Data Mining* como metodología principal de desarrollo.
- Uso de las principales librerías de aprendizaje automático y redes neuronales de código abierto.
- Creación de una librería de código abierto de aprendizaje automático que cumpla con todas las necesidades científicas del problema, descritas en el Trabajo de Fin de Grado de Matemáticas [9].

4. IMPLEMENTACIÓN INFORMÁTICA

4.1. Metodología de desarrollo

Se ha elegido seguir la metología CRISP-DM [26] o *Cross Intustry Standard Process for Data Mining*, por sus siglas en inglés. Esta elección viene motivada por la filosofía cíclica y de constante aprendizaje de la metodología (que se explica abajo) y porque es la más usada en el sector [27] [28] [29] [30].

Esta metodología divide los procesos de ciencia de datos en seis fases principales y se basa en que para obtener modelos de calidad, muchas veces hay que hacer muchas iteraciones de selección y entrenamiento de modelos y análisis de los datos.

Las fases son las siguientes:

- Entendimiento del negocio (*Business Understanding*): Se basa en definir el problema de negocio concreto a resolver y una métrica adecuada para optimizar.
- Entendimiento de los datos (*Data Understanding*): Obtener datos de calidad y relevantes para crear la solución al problema de negocio.
- Preparación de los datos (*Data Preparation*): Limpieza (imputación de valores, estandarización de formatos, etc.) y tratamiento de los datos previos al modelado. A parte de estandarizar los datos, es muy importante quitar datos redundantes y crear datos de valor en base a los datos existentes. Este proceso es conocido como ETL y se explicará en el apartado de la implementación-
- Modelado (*Modeling*): Entrenamiento de modelos predictivos usando técnicas como, por ejemplo, la validación cruzada para elegir el mejor modelo.
- Evaluación (*Evaluation*): Pruebas de la solución desarrollada con el fin de comprobar si se han conseguido los objetivos de la métrica de negocio elegida en la fase de *Business Understanding*.
- Despliegue (*Deployment*): Despliegue del modelo en un entorno real de producción donde sea usado para el fin con el que fue concebido, siempre y cuando se haya conseguido una evaluación positiva.

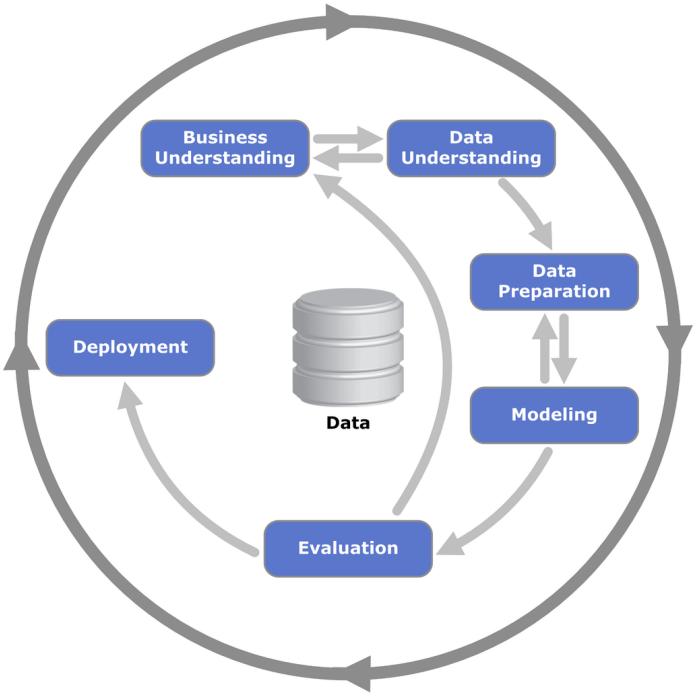

Fig. 4.1. Fases del ciclo de CRISP-DM [31]

Las flechas que podemos ver en el diagrama superior no son estrictas, esto significa que esos son los movimientos más comunes en el ciclo, pero no los únicos posibles. Eso es porque después del despliegue, siempre se pueden aplicar los nuevos conocimientos adquiridos para mejorar el modelo o tener un mejor entendimiento global del negocio.

En el caso particular de este problema, y una vez pasado por las primeras fases del ciclo, se han realizado muchas iteraciones desde la evaluación al modelado y viceversa hasta que se ha conseguido un resultado satisfactorio.

4.2. Diseño

4.2.1. Tecnologías utilizadas

Python

Se ha optado por el lenguaje de programación Python. Python es un lenguaje de programación de código libre y de propósito general muy versátil (gracias a, entre otras cosas, soportar distintos paradigmas de programación como la funcional, la imperativa o la orientada a objetos), potente y que dispone de una gran cantidad de *frameworks* para el aprendizaje automático y las redes neuronales, de hecho, las tres librerías más usadas por los usuarios de Kaggle (Scikit-learn, TensorFlow y Keras) son exclusivas para Python [32].

Esta decisión viene dada por las características explicadas anteriormente y viene respaldada por el uso y crecimiento que ha ido teniendo Python en los últimos años. Como

se puede ver en la encuesta anual de Kaggle de 2018 [33], el 83 % de los más de 20,000 los participantes en sus competiciones de aprendizaje automático respondió que usaba Python.

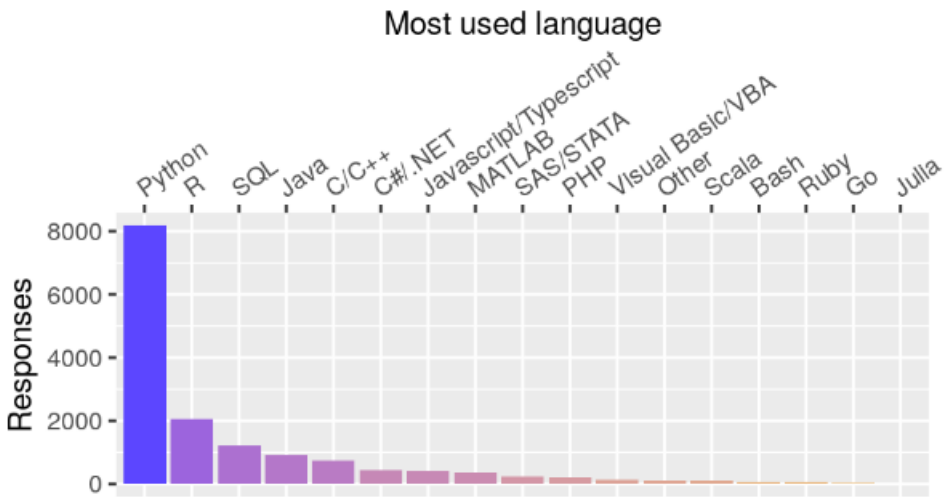

Fig. 4.2. Lenguajes más usados por los participantes de Kaggle [32]

En la encuesta anual de Stack Overflow de 2019 (donde respondieron más de 72,000 desarrolladores activos) se puede observar que Python es el cuarto lenguaje de programación más usado entre los desarrolladores profesionales [34]. Además, hay que destacar que los únicos lenguajes más usados que Python son SQL (para bases de datos) y HTML/CSS y JavaScript (para diseño y desarrollo web respectivamente).

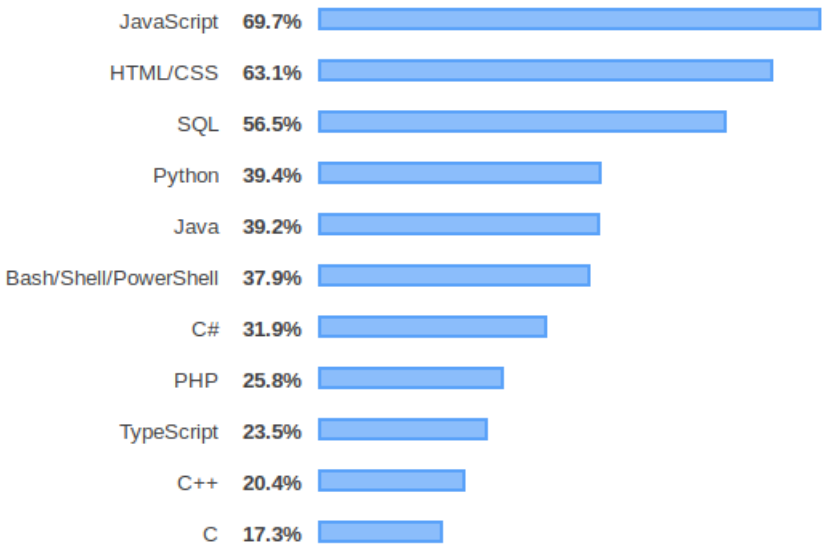

Fig. 4.3. Lenguajes más usados por los usuarios de Stack Overflow [34]

Puede que no sea el lenguaje más usado en términos absolutos, sin embargo, que sea el lenguaje de propósito general más usado por los usuarios de Stack Overflow significa que hay mucha documentación sobre el mismo. Esto, sumado a que la sintaxis de Python

intenta asemejarse al lenguaje natural de una forma simplificada pero intuitiva, implica que los nuevos usuarios tienen muchas facilidades para aprenderlo, fomentando así, su crecimiento como lenguaje.

Anaconda

Anaconda es una distribución de Python gratuita de código abierto enfocada para aplicaciones científicas. Su mayor ventaja es que sirve también como gestor de paquetes de Python y gestiona todas las dependencias de manera que evita los conflictos de versiones de forma muy robusta.

Asimismo, Anaconda dispone de una funcionalidad llamada Conda que sirve de gestora de entornos de trabajo. De este modo se pueden tener entornos de trabajo que sirven en cualquier sistema operativo y con una robusta gestión de paquetes. Este aspecto ha sido crucial en el desarrollo del trabajo ya que para poder entrenar redes neuronales en GPUs hay que instalar versiones muy concretas de los drivers de nvidia y cudatoolkit.

Tensorflow y Keras

Para las redes neuronales en sí se ha usado Keras. Keras es una librería de código abierto para Python cuya función es hacer de enlace entre las librerías de muy bajo nivel de redes neuronales y el usuario final. La principal razón para este hecho es que dichas librerías están pensadas para la investigación y añaden muchas capas de complejidad que Keras gestiona de forma transparente para el usuario. Keras, además es usada por una gran cantidad de empresas referentes en el sector como Netflix, Uber, Google, Amazon y Microsoft y también centros de investigación como el CERN o la NASA [35].

De entre las librerías de redes neuronales que funcionan como *backend* de Keras, se ha elegido TensorFlow ya que es, también, de código abierto; está respaldada por Google [36] y es una de las más utilizadas por la comunidad. Además TensorFlow da soporte oficial a Keras [37] como su *frontend*, lo que significa que es más fácil hacer cambios personalizados en las redes de Keras usando el *backend* de TensorFlow.

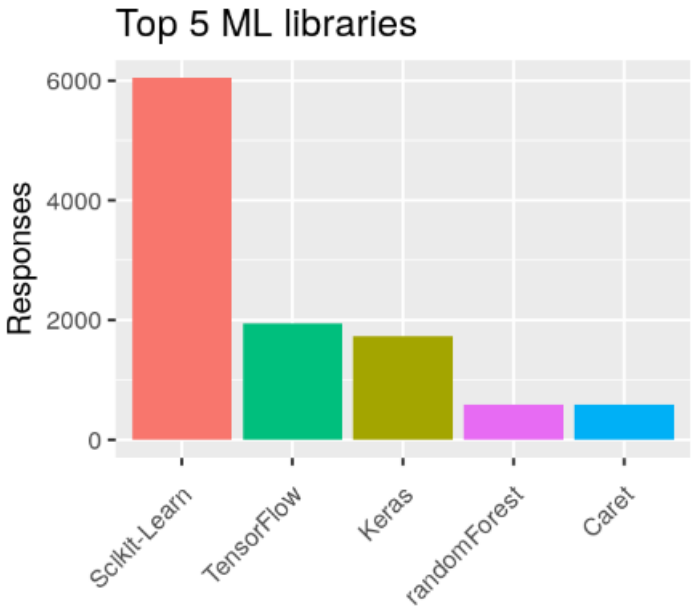

Fig. 4.4. Principales librerías usadas por los participantes de Kaggle [32]

En la encuesta anual de Kaggle también se puede apreciar (gráfica 4.4) como de entre las librerías de aprendizaje automático, después de Scikit-learn (más enfocada a un uso general y no tanto a redes neuronales), las más usadas son TensorFlow y Keras. Asimismo, se puede observar en 4.5 que también lo son dentro de las los *frameworks* de redes neuronales.

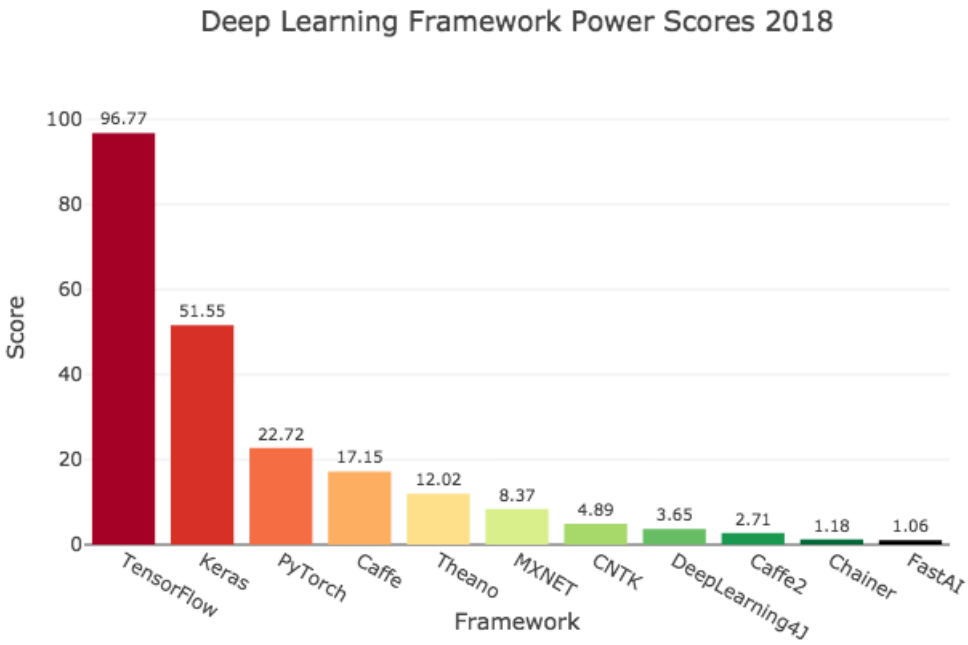

Fig. 4.5. Principales librerías usadas para deep learning [35]

Notebooks de Jupyter

Python es un lenguaje de programación interpretado, lo que significa que se puede interactuar con él de forma dinámica por línea de comandos, lo que permite que sea muy fácil hacer distintas pruebas con el lenguaje sin tener que compilar un programa principal.

Jupyter, de código abierto, combina el intérprete de IPython con la posibilidad de ejecutar código por bloque e intercalar bloques de texto incluso texto con formato como Markdown en un mismo archivo llamado *notebook*. La principal ventaja de usar Notebooks de Jupyter en la ciencia de datos es que son una forma muy fácil de preparar y compartir modelos. Esto, a la hora de encontrar el mejor modelo y los mejores parámetros es crucial ya que las celdas de los notebooks se pueden ejecutar independientemente todas las veces que se quiera.

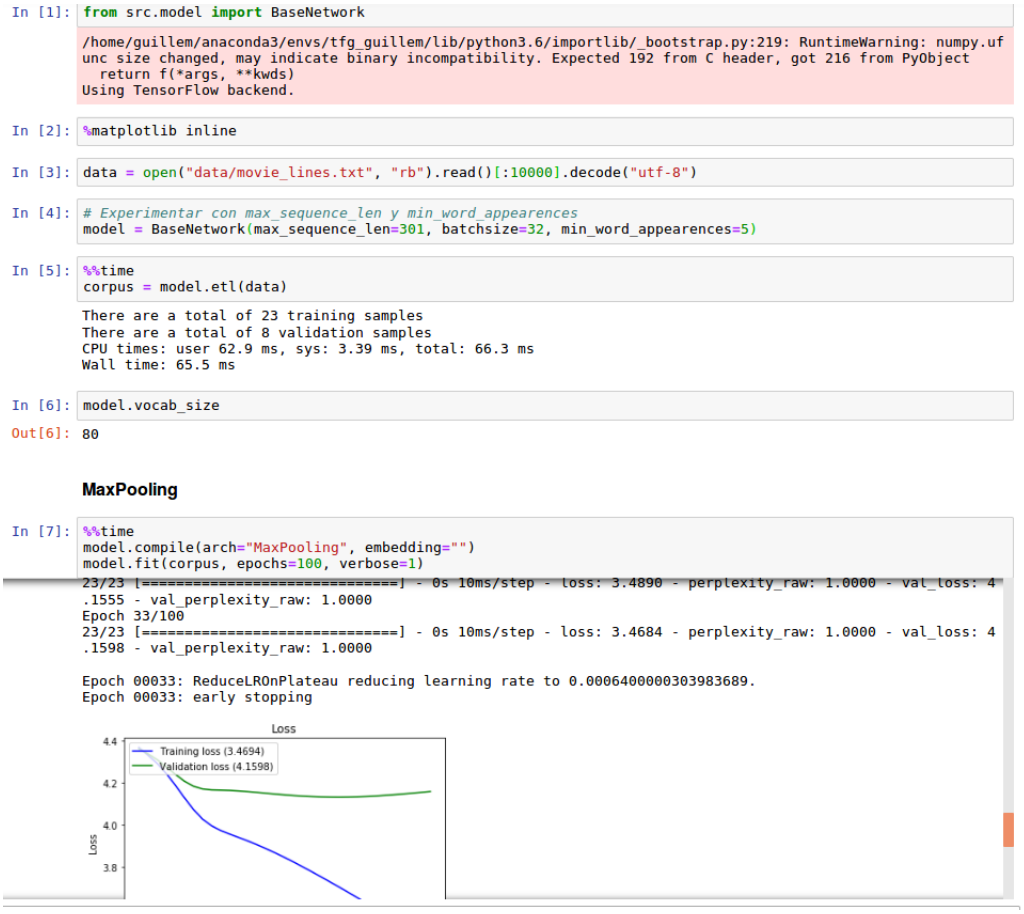

Fig. 4.6. Ejemplo de un notebook de Jupyter

4.2.2. Arquitectura de la solución

Se ha elegido diseñar una módulo principal de Python que contenga todos los métodos necesarios para realizar los entrenamientos de modelos y las predicciones. De este modo, y desde un Notebook de Jupyter, resulta muy fácil hacer pruebas hasta obtener el modelo ideal y se pueden visualizar con facilidad estadísticas sobre los datos.

Este diseño ha servido, también, para que el código sea fácilmente reutilizable en otros proyectos de redes neuronales y lo puedan usar tanto usuarios con poca experiencia en el campo como usuarios experimentados.

Asimismo, se ha descartado hacer un programa por línea de comandos con un *main* de Python ya que eso complicaría tanto la mantenibilidad como la usabilidad del código.

Por otro lado, se usan entornos de anaconda con las dependencias necesarias para que la instalación sea rápida y sin conflictos. De hecho, con solo dos comandos por línea de comandos se puede desplegar completamente el servicio. Además, existe la posibilidad de elegir entre un entorno con las dependencias necesarias para usar GPUs (y así minimizar el tiempo de entrenamiento de las redes neuronales) o no, lo que da aun más flexibilidad de cara al usuario.

4.3. Implementación

Todo el código generado durante la elaboración de este trabajo se encuentra disponible con licencia MIT en GitHub [38]. Además, están disponibles todos los sets de datos salvo los que podían incumplir los derechos de autor de terceros.

El código está estructurado en un módulo principal (*model.py*) y varios módulos complementarios independientes.

Se ha usado una clase principal (*BaseNetwork*), siguiendo el paradigma orientado a objetos para poder estructurar la gran cantidad de parámetros y objetos Python que va a tener nuestro modelo. Esta, hereda de la clase *BasesEstimator* de Scikit-learn [39], la cual implementa *setters* y *getters* para los modelos, entre otras cosas:

```
class BaseNetwork(BaseEstimator):
    """Class to generate new text with neural networks"""

    def __init__(
        self,
        tokenizer=None,
        max_sequence_len=301,
        min_word_appearences=None,
        vocab_size=None,
        batchsize=32,
        **kwargs,
    ):

        self.vocab_size = vocab_size
        self.min_word_appearences = min_word_appearences
        if self.vocab_size and self.min_word_appearences:
            raise AttributeException(
                "You must specify only vocab_size or
                    min_word_appearences, not both."
            )
```

```
        self.tokenizer = (
            Tokenizer(num_words=vocab_size, oov_token=None)
            if tokenizer is None
            else tokenizer
        )
        self.max_sequence_len = max_sequence_len
        self.batchsize = batchsize
        # Other kwargs, this is used in the load_model method
        self.__dict__.update(kwargs)
```

En el método de inicialización se puede elegir qué tokenizador se va a usar para el texto, si no se elige, se asigna el tokenizador por defecto de Keras. El tokenizador es el paso intermedio entre el texto bruto y los *embeddings*, se encarga de pasar a minúsculas las palabras y a crear diccionarios para mapear cada palabra con un número.

También se pueden definir otros parámetros en relación a cómo se procesará el texto: máxima longitud (en palabras) de las muestras para entrenar, cuántas palabras forman el vocabulario de la red (se puede hacer que la red solo tenga en cuenta las palabras que han aparecido más de *n* veces) Y tamaño del batch. Este tiene que definirse aquí ya que se usará en el proceso de transformación de los datos.

```
    def etl(self, data, mask=None):

        # Prepare masks
        self.mask = mask or [True, True, True, False]
        self.testmask = [not x for x in self.mask]

        # Basic cleanup
        corpus = data.lower().split("\n")

        # Tokenization
        self.tokenizer.fit_on_texts(corpus)
        if self.min_word_appearences:
            low_count_words = [
                word
                for word, count in self.tokenizer.word_counts.items
                    ()
                if count < self.min_word_appearences
            ]
            for word in low_count_words:
                del self.tokenizer.word_index[word]
                del self.tokenizer.word_docs[word]
                del self.tokenizer.word_counts[word]
            self.vocab_size = len(self.tokenizer.word_index) + 1

        # Total samples
        self.num_train_samples = len(
            list(
                self.patterngenerator(
```

```
                        corpus, batchsize=self.batchsize, count=True,
                            mask=self.mask
                    )
                )
            )
            self.num_test_samples = len(
                list(
                    self.patterngenerator(
                        corpus, batchsize=self.batchsize, count=True,
                            mask=self.testmask
                    )
                )
            )

            print(f"There are a total of {self.num_train_samples}
                training samples")
            print(f"There are a total of {self.num_test_samples}
                validation samples")

            return corpus
```

El método *ETL* (del inglés Extract, transform and load; extraer, transformar y cargar) se encarga de tomar los datos en bruto (en formato string) y tranformarlos en un iterable de strings en minúsculas y preparar al tokenizador.

Además para crear nuestro set de datos, necesitamos que tengan alguna estructura de secuencia (que permita a la red neuronal aprender lo que viene después). De este modo se usa el método *patterngenerator* (que se explicará posteriormente) para que todas las muestras de datos tengan la misma longitud. De esta forma las que sean más cortas se rellenarán con un carácter especial que servirá para denotar el vacío y las que sean más largas, se truncarán.

Una vez tengamos eso, crearemos muestras en las que solo aparezca la primera palabra de la frase y el objetivo sea la siguiente. La siguiente muestra tendría las dos primeras palabras de la frase y el objetivo sería la tercera. Y la secuencia continuaría hasta acabar la frase.

Este método también se encarga de calcular el tamaño del vocabulario en el caso de que hayamos elegido que solo aparezcan las palabras que se repiten más de *n* veces en el texto.

Para dividir en entrenamiento y validación se pasan unas máscaras que sirven para decir cada cuantas muestras, se guarda una para validación. Esta implementación es muy flexible ya que, según como sean los datos, nos puede interesar una combinación u otra. Por defecto, cada cuatro muestras, la última se guarda para validación.

A continuación se encuentra el método *compile*, cuyo comportamiento es análogo al del método compile de Keras, salvo que también se encarga de crear la estructura de la

red neuronal que se vaya a utilizar.

```
    def compile(
        self,
        activation="softmax",
        kind="AveragePooling",
        embedding=None,
        embedding_output_dim=64,
        gpu=False,
        loss="categorical_crossentropy",
        metrics=None,
        optimizer="adam",
        arch=None,
        dropout=None,
        **kwargs,
    ):

        arch = arch or [64]
        metrics = metrics or [perplexity]
        self.net = Sequential()
```

compile en Keras se encarga de configurar cómo aprenderá la red neuronal, a este método se le llama siempre una vez definida la arquitectura de la red. Los parámetros que se le pasan son el optimizador ("adam" en este caso), una función de pérdida y una métrica. Para el problema de generación de texto se suelen usar la entropía y la perplejidad, como se explica en el TFG de Matemáticas [9].

A parte de llamar al *compile* de Keras, este *compile* también construye la estructura de la red neuronal en base a unos pocos parámetros introducidos por el usuario, de esta forma se consigue hacer más fácil su uso.

Para ello, el usuario tiene que elegir qué tipo de *embeddings* usar para procesar el texto, que tipo de neuronas y, mediante una lista de enteros, cuantas capas y con cuantas neuronas.

```
        # Embedding layer
        output_dim = embedding_output_dim
        trainable = True
        weights = None
        if "fastText" in embedding:
            if embedding == "fastText":
                fastText_file = "crawl-300d-2M.vec"
                zip_fastText_file = f"{fastText_file}.zip"
                url = f"https://dl.fbaipublicfiles.com/fasttext/
                    vectors-english/{zip_fastText_file}"
            else:
                fastText_file = "cc.es.300.vec"
                zip_fastText_file = f"{fastText_file}.gz"
                url = f"https://dl.fbaipublicfiles.com/fasttext/
```

```
                    vectors-crawl/{zip_fastText_file}"
            try:
                embeddings = self.load_vectors_words(fastText_file)
                print("Embedding file loaded sucessfully!")
            except:
                print(
                    "No embedding file found, downloading it... (
                        this will take a while)"
                )
                check_call(f"curl -L# '{url}'", shell=True)
                with zipfile.ZipFile(f"{zip_fastText_file}", "r") as
                     file:
                    file.extractall("./")
                embeddings = self.load_vectors_words(fastText_file)
                print("Embedding file loaded sucessfully!")
            finally:
                embedding_matrix = self.create_embedding_matrix(
                    embeddings)
                output_dim = embedding_matrix.shape[1]
                trainable = False
                weights = [embedding_matrix]

        self.net.add(
            Embedding(
                input_dim=self.vocab_size,
                output_dim=output_dim,
                input_length=self.max_sequence_len - 1,
                trainable=trainable,
                weights=weights,
            )
        )
```

Como se puede observar arriba, solo con elegir el tipo de *embeddings*, todo lo demás lo hace el propio *compile*. En el caso de querer usar una capa común de *embeddings* de Keras, simplemente se añade esa capa y se pone el parámetro *trainable* a verdadero ya que si no, no serviría de nada porque ese parámetro sirve para bloquear ciertas capas de una red neuronal y que no cambien sus pesos nunca. En cambio, si se quieren usar los embeddings preentrenados de fastText, el usuario solo tiene que elegir el idioma de los mismos (español o inglés) y el código se encarga de descargarlos, descomprimirlos, cargarlos en memoria y crear la matriz de *embeddings* que se le pasará a la capa de *embeddings* de Keras.

```
    def load_vectors_words(self, fname):

        data = {}
        vocab = tuple(self.tokenizer.word_index.keys())[: self.
            vocab_size - 1]
        with open(fname) as fin:
```

```
            next(fin)  # Skip first line, just contains embeddings
                size data
            for line in fin:
                tokens = line.rstrip().split(" ")
                word = tokens[0]
                if word in vocab:
                    data[word] = np.array(list(map(float, tokens
                        [1:])))
        return data
```

Arriba podemos ver como se cargan los vectores de fastText en memoria, esta operación suele ser pesada y ocupar mucha RAM, de modo que se ha optimizado cargando solo los vectores para las palabras que están en los datos que se van a utilizar.

A continuación podemos ver como se crea la matriz de *embeddings* en la que, si una palabra está en el vocabulario pero no en los *embeddings*, se le imputa un valor procedente de una distribución normal con media, la media de los vectores originales y varianza, la varianza de los vectores originales. De este modo se consigue que, aunque las palabras no estén en los *embeddings*, sí se puedan procesar por la red neuronal.

```
    def create_embedding_matrix(self, embeddings):

        # Compute mean and standard deviation for embeddings
        all_embs = np.stack(embeddings.values())
        emb_mean, emb_std = all_embs.mean(), all_embs.std()
        # If we are using fastText, this is 300
        embedding_size = len(list(embeddings.values())[0])
        embedding_matrix = np.random.normal(
            emb_mean, emb_std, (self.vocab_size, embedding_size)
        )
        vocab = tuple(self.tokenizer.word_index.items())[: self.
            vocab_size - 1]
        for word, i in vocab:
            embedding_vector = embeddings.get(word)
            if embedding_vector is not None:
                embedding_matrix[i] = embedding_vector

        return embedding_matrix
```

Ahora, según el tipo de neuronas que quiera tener el usuario en la red, se añade solo una capa en el caso de las más simples y, si se quieren neuronas de tipo “GRU” o “LSTM”, se llama al método *Complex_network* que crea la arquitectura de la red con ese tipo de neuronas dada las especificaciones del usuario.

```
        # Core layers
        if kind == "Flatten":
            self.net.add(Flatten())
        elif kind == "MaxPooling":
```

```
            self.net.add(GlobalMaxPooling1D())
        elif kind == "AveragePooling":
            self.net.add(GlobalAveragePooling1D())
        elif kind == "LSTM" or kind == "GRU":
            self.Complex_Network(kind=kind, gpu=gpu, arch=arch)
        else:
            raise Exception("Unknown network architecture")
```

Abajo podemos ver el funcionamiento del método *Complex_network*. En él, si se ha desplegado el servicio en el entorno de GPU y el usuario quiere usarla, se elige las versiones de GPU de las redes recurrentes "CuDNNGRU" o "CuDNNLSTM" en vez de las normales. De esta forma se consigue mejorar sustancialmente la velocidad de entrenamiento con los mismos resultados de calidad.

Después de eso, se añade una capa bidireccional con la red elegida y, al final, se añaden las capas ocultas acorde con la arquitectura definida por el usuario.

```
    def Complex_Network(self, kind, gpu, arch):

        if kind == "LSTM":
            layer = CuDNNLSTM if gpu else LSTM
        elif kind == "GRU":
            layer = CuDNNGRU if gpu else GRU

        # Bidirectional layer
        bidirect = layer(
            arch.pop(0),
            input_shape=(self.max_sequence_len,),
            return_sequences=False if len(arch) == 0 else True,
        )
        self.net.add(Bidirectional(bidirect, merge_mode="concat"))

        # Hidden layers
        for i, elem in enumerate(arch):
            self.net.add(
                layer(
                    elem,
                    input_shape=(self.max_sequence_len,),
                    return_sequences=True if i < len(arch) - 1 else
                        False,
                )
            )

        return self
```

Finalmente, se añade una capa de *dropout*, si lo elige el usuario, se añade una capa densa al final, se llama al método *compile* de Keras y se imprime por pantalla un resumen de la arquitectura de la red creada y sus parámetros.

La capa de *dropout* sirve para "anular" de forma aleatoria algunos flujos de información de la red, de forma que no toda la información pase de un paso al siguiente y se disminuyan las probabilidades de hacer sobreajuste en el entrenamiento. Sin embargo, y como veremos en el apartado de experimentos, esta estrategia no ha resultado en una mejora de los resultados en este caso.

```
        # Dropout layer
        if dropout:
            self.net.add(Dropout(dropout))
        # Final layer
        self.net.add(Dense(self.vocab_size, activation=activation))
        self.net.compile(loss=loss, optimizer=optimizer, metrics=
            metrics, **kwargs)

        print(self.summary())
```

El tercer método principal de la clase *BaseNetwork* es el *fit* que se encarga de entrenar la red neuronal creada en el *compile*. Este método se puede personalizar con una gran serie de parámetros. Aun así, hay una gran cantidad de parámetros que tienen valores por defecto de forma que se maximice la usabilidad.

Muchos de ellos funcionan como *callbacks*, lo que significa que hacen algo al final de cada iteración, en este caso al final de cada época de entrenamiento.

```
    def fit(
        self,
        corpus,
        callbacks=None,
        checkpoints=True,
        dynamic_lr=True,
        earlystop=True,
        epochs=200,
        verbose=1,
        plot=True,
        restore_best_weights=True,
        **kwargs,
    ):

        callbacks = callbacks or []
```

El primero de esos *callbacks* es *dynamic_lr* y sirve para que el ritmo de aprendizaje vaya disminuyendo a medida que la red lo vaya necesitando. Esta necesidad se debe a que ritmos muy altos, cuando la red ya lleva un rato entrenando, hacen que la red no pueda encontrar el mínimo de la función que está optimizando ya que lo pasan de largo.

```
        if dynamic_lr:
            callbacks.append(
                ReduceLROnPlateau(
```

```
                        monitor="val_perplexity",
                        factor=0.8,
                        patience=8,
                        verbose=verbose,
                        mode="min",
                    )
                )
```

A continuación el*earlystop* o parada temprana se encarga de dejar de entrenar la red neuronal (aunque no se hayan completado todas las épocas) en el caso de que empiece a tener malos resultados de aprendizaje. Además, junto con el parámetro *restore_best_weights*, se pueden guardar los pesos de la mejor época entrenamiento de la red para que se usen en el modelo final, de esta forma, se minimizan los posibles resultados negativos de entrenar la red demasiado tiempo.

```
            if earlystop:
                callbacks.append(
                    EarlyStopping(
                        monitor="val_perplexity",
                        min_delta=0,
                        patience=20,
                        verbose=verbose,
                        mode="min",
                        restore_best_weights=restore_best_weights,
                    )
                )
```

checkpoints tiene una funcionalidad similar a la de *restore_best_weights*. Sin embargo, en este caso, el *callback* se encarga de guardar el modelo en un archivo cada vez que la red encuentra una combinación de pesos con mejor puntuación que la anterior mejor.

Para guardar una red neuronal en un archivo, se suele usar el formato HDF5, de código abierto. Sin embargo en nuestro caso es necesario guardar también todos los parámetros extra que no están en Keras y todo el tokenizador, para ello se ha tenido que crear un método *save* y otro *load_model* que sobrescriban a los de Keras.

```
            if checkpoints:
                model_name = f"Best_model_{datetime.datetime.now().time
                    ()}"
                print(f"The model will be saved with the name: {
                    model_name}")
                callbacks.append(
                    ModelFullCheckpoint(
                        modelo=self,
                        filepath=model_name,
                        save_best_only=True,
                        monitor="val_perplexity",
                        mode="min",
```

```
                        verbose=verbose,
                    )
                )
```

El método *save* guarda la red neuronal en dos partes. Primero guarda todos los parámetros extra a Keras (incluido el tokenizador) en un archivo .json. Para ello se utiliza el paquete de Python jsonpickle, de código abierto. Dicho paquete se encarga de serializar y desserializar objetos complejos de Python en formato .json. Las librerías standard de Python, en general, solo pueden serializar objetos simples de Python, pero no objetos como el tokenizador, por ejemplo.

A continuación, simplemente se guarda la red neuronal con el método de Keras para guardarlas, en un archivo .h5.

```
    def save(self, path=None):
        if path is None:
            path = f"{self}"
        kwargs = dict()
        for key in self.__dict__:
            if key != "net":
                kwargs[key] = self.__dict__[key]
        try:
            self.net.save(f"{path}_network.h5")
            with open(f"{path}_attrs.json", "w") as outfile:
                json.dump(jsonpickle.encode(kwargs), outfile)
            return "Model saved successfully!"
        except:
            return "Something went wrong when saving the model..."
```

El método *load_model* es una factoría que crea un objeto de la clase *BaseNetwork* a partir de los archivos creados por el método *save*.

```
    @classmethod
    def load_model(cls, path):
        with open(f"{path}_attrs.json") as infile:
            kwargs = jsonpickle.decode(json.load(infile))
        kwargs["net"] = load_model(
            f"{path}_network.h5", custom_objects={"perplexity":
               perplexity}
        )
        return cls(**kwargs)
```

Antes de acabar, hay que entrenar la red neuronal. Aquí se plantea un problema de prestaciones hardware. Aunque se usen GPUs con mucha memoria, los textos vectorizados y las matrices de *embeddings* ocupan mucho, además los sets de datos de texto pueden llegar a ser muy grandes. Cuando se llama al *fit* de Keras, este vectoriza y vuelca en memoria todo el set de datos y esto suele llevar a agotar toda la memoria de la máquina.

Afortunadamente existe un método en Keras llamado *fit_generator* que aprovecha la gran flexibilidad y eficiencia en memoria de los generadores de Python para que en memoria solo se cargue la muestra que se esté usando en ese preciso momento para entrenar y las demás no se hayan generado.

```
        self.net.fit_generator(
            self.patterngenerator(
                corpus, batchsize=self.batchsize, infinite=True,
                    mask=self.mask
            ),
            steps_per_epoch=ceil(self.num_train_samples / self.
                batchsize),
            callbacks=callbacks,
            epochs=epochs,
            validation_data=self.patterngenerator(
                corpus, batchsize=self.batchsize, infinite=True,
                    mask=self.testmask
            ),
            validation_steps=ceil(self.num_test_samples / self.
                batchsize),
            verbose=verbose,
            **kwargs,
        )
```

Por lo general este método es muy útil y se puede aplicar directamente pasando un generador como set de datos. Sin embargo, nosotros necesitamos que nuestro generador vectorice los textos antes de pasárselos a la red y que los transforme en una secuencia (como hemos explicado en el apartado de la *ETL*), haciendo que ocupen aun más memoria. Idealmente también queremos que el generador pueda crear muestras de entrenamiento y de validación de forma que cada *n* muestras consecutivas, se guarde la siguiente para validación. Al estar usando modelos de secuencias, es muy importante que sea así ya que, de otro modo, se le pasarían secuencias aleatorias al modelo que nada tienen que ver con los datos de entrenamiento y no aprendería.

Para esto se ha creado el método *patterngenerator*, inspirado en que usa el proyecto de Neurowriter [23].

```
    def patterngenerator(self, corpus, **kwargs):

        # Pre-tokenized all corpus documents, for efficiency
        tokenizedcorpus = self.tokenizer.texts_to_sequences(corpus)
        self.max_sequence_len = min(
            len(max(tokenizedcorpus, key=len)), self.
                max_sequence_len
        )
        for pattern in self._tokenizedpatterngenerator(
            tokenizedcorpus, **kwargs):
            yield pattern
```

patterngenerator tokeniza el set de datos (no consume memoria y es rápido computacionalmente) y después calcula la longitud máxima de las muestras para poder generar las nuevas muestras en forma de secuencias tokenizadas. Y después se encarga de ir generando muestras a través de otro generador, *_tokenizedpatterngenerator*.

```
    @infinitegenerator
    @batchedpatternsgenerator
    @maskedgenerator
    def _tokenizedpatterngenerator(self, tokenizedcorpus, **kwargs):
```

_tokenizedpatterngenerator recibe las muestras perfectamente formateadas a través de sus tres decoradores del módulo *generators.py*. Cada uno de ellos se encarga de tratar los datos de una forma distinta. De esta manera se consigue una gran modularidad y reusabilidad.

```
def infinitegenerator(generatorfunction):

    def infgenerator(*args, **kwargs):
        if "infinite" in kwargs:
            infinite = kwargs["infinite"]
            del kwargs["infinite"]
        else:
            infinite = False
        if infinite is True:
            while True:
                for elem in generatorfunction(*args, **kwargs):
                    yield elem
        else:
            for elem in generatorfunction(*args, **kwargs):
                yield elem

    return infgenerator
```

Para usar el método *fit_generator* de Keras, el generador tiene que tener un bucle infinito. Sin embargo a veces puede ser interesante que solo se recorran una vez los datos, por ejemplo cuando queremos contarlos, en la *ETL*, solo recorremos los datos una vez. La función de *infinitegenerator* es un decorador para funciones generadoras y se encarga de añadirles un nuevo argumento que permite que estas funciones sean generadoras infinitas o no.

```
def batchedpatternsgenerator(generatorfunction):

    def modgenerator(*args, **kwargs):
        for batch in batchedgenerator(generatorfunction)(*args, **
            kwargs):
```

```
            Xb, yb = zip(*batch)
            yield np.stack(Xb), np.stack(yb)

    return modgenerator

def batchedgenerator(generatorfunction):

    def modgenerator(*args, **kwargs):
        if "batchsize" in kwargs:
            batchsize = kwargs["batchsize"]
            del kwargs["batchsize"]
        else:
            batchsize = 1
        for batch in _splitevery(generatorfunction(*args, **kwargs),
            batchsize):
            yield batch

    return modgenerator

def _splitevery(iterable, n):
    """Returns blocks of elements from an iterator"""
    i = iter(iterable)
    piece = list(islice(i, n))
    while piece:
        yield piece
        piece = list(islice(i, n))
```

batchedpatternsgenerator crea muestras de los datos dado un tamaño de *bacth*. Para eso se ayuda en la función *badgedgenerator* que devuelve iterables de longitud *batch*. Una vez *batchedpatternsgenerator* tiene los batches, devuelve la X y la y de forma que las pueda entender Keras.

```
def maskedgenerator(generatorfunction):

    def mskgenerator(*args, **kwargs):
        if "mask" in kwargs:
            mask = kwargs["mask"]
            del kwargs["mask"]
        else:
            mask = [True]
        for i, item in enumerate(generatorfunction(*args, **kwargs))
            :
            if mask[i % len(mask)]:
                yield item

    return mskgenerator
```

Por último, *maskedgenerator* aplica una máscara de booleanos a las muestras en forma de filtro. De esta forma, como se explicó en la parte de *ETL*, se consigue hacer una división

entre entrenamiento y validación. A la hora de decirle a Keras qué es para entrenar y qué para validar, se le pasan dos generadores, el primero tendrá una máscara y el segundo tendrá el contrario de esa máscara.

Una vez *_tokenizedpatterngenerator* recibe las muestras casi preparadas para entrenar, solo necesita crear las secuencias y tokenizar las palabras. En el caso del objetivo, la *y*, la transforma usando *one hot encoding*.

```
    def _tokenizedpatterngenerator(self, tokenizedcorpus, **kwargs):
        for token_list in tokenizedcorpus:
            for i in range(1, len(token_list)):
                sampl = np.array(
                    pad_sequences(
                        [token_list[: i + 1]],
                        maxlen=self.max_sequence_len,
                        padding="pre",
                        value=0,
                    )
                )
                X, y = sampl[:, :-1], sampl[:, -1]
                y = ku.to_categorical(y, num_classes=self.vocab_size
                    )
                if "count" in kwargs and kwargs["count"] is True:
                    yield 0, 0
                else:
                    yield X[0], y[0]
```

Para finalizar, se dibuja la evolución de la función de pérdida y de la perplejidad en función de las épocas para que el usuario pueda ver fácilmente si su red ha ido entrenando correctamente o no.

Para ello se llama al atributo *history* de la red neuronal de Keras, que será usado como parámetro de entrada en el método *plot_history* del módulo *plotting.py*

```
def plot_history(history):

    loss_list = history.history["loss"]
    val_loss_list = history.history["val_loss"]
    perplexity_list = history.history["perplexity"]
    val_perplexity_list = history.history["val_perplexity"]

    epochs = range(1, len(history.epoch) + 1)

    ## Loss
    plt.figure(1)
    plt.plot(epochs, loss_list, "b", label=f"Training loss ({round(
        loss_list[-1], 4)})")
    plt.plot(
        epochs,
```

```
            val_loss_list,
            "g",
            label=f"Validation loss ({round(val_loss_list[-1], 4)})",
        )

        plt.title("Loss")
        plt.xlabel("Epochs")
        plt.ylabel("Loss")
        plt.legend(loc="upper left")

        ## Perplexity
        plt.figure(2)
        plt.plot(
            epochs,
            perplexity_list,
            "b",
            label=f"Training perplexity ({round(perplexity_list[-1], 4)
                })",
        )
        plt.plot(
            epochs,
            val_perplexity_list,
            "g",
            label=f"Validation perplexity ({round(val_perplexity_list
                [-1], 4)})",
        )
        plt.title("Perplexity")
        plt.xlabel("Epochs")
        plt.ylabel("Perplexity")
        plt.legend(loc="upper left")
        plt.show()
```

El último método de la clase *BaseNetwork* es el que se encarga de generar nuevo texto. Esto lo hace en base a un texto que se introduce como *input* y la red neuronal intenta predecir la palabra siguiente.

Como el objetivo es generar texto, otro parámetro que se le pasa al modelo es cuantas palabras nuevas se quieren generar. Así el modelo generará una palabra por iteración y, en la siguiente iteración se añadirá la palabra generada a las demás palabras del texto introducido.

El único problema es que así el texto generado por la red sería predecible porque, la red neuronal predice la probabilidad que tiene cada palabra del vocabulario de ser la siguiente, dada una frase de *input* y elige la que más probabilidades tiene.

Para evitar esto, se introduce un factor de aleatoriedad que hace que la palabra más probable no domine la generación del esto. Esto se consigue haciendo un reescalado de las probabilidades para que sean más uniformes sin dejar de tener sentido con el contexto del texto que se ha pasado anteriormente. Más información sobre este proceso se puede

encontrar en el TFG de Matemáticas [9].

```
    def generate_text(self, seed_text, next_words):

        generated_text = seed_text
        for i in range(next_words):
            token_list = self.tokenizer.texts_to_sequences([
                generated_text])
            token_list = pad_sequences(
                token_list, maxlen=self.max_sequence_len - 1,
                    padding="pre"
            )
            predicted = self.net.predict(token_list, verbose=0)[0]
            sampled_predicted = sample(np.log(predicted), 0.5)
            try:
                generated_text += (
                    f" {self.tokenizer.sequences_to_texts([[
                        sampled_predicted]])[0]}"
                )
            except:
                # Predicted 0, pass this time
                pass

        return generated_text
```

5. EXPERIMENTOS

Para los experimentos nos hemos centrado en dos tipos de pruebas. La primera es generar texto coherente y creíble y la segunda es comprobar si se puede utilizar la perplejidad como medida de complejidad de un texto. Esto sería bastante interesante porque las medidas de complejidad actuales o son demasiado simples, o necesitan diccionarios específicos de palabras complicadas o son específicas para idiomas concretos.

5.1. Generación de texto

Para las primeras pruebas se ha usado el set de datos de WikiTex-2 [40] y Movie Lines [41]. El primero es una colección de más de 2 millones de tokens de una serie de 600 artículos marcados como buenos en la Wikipedia y con un tamaño de vocabulario de 33,278 palabras. Este set de datos se usa mucho en generación de texto ya que es bastante más grande que el Penn Treebank [42], otro set de datos muy usado en PLN.

El segundo set de datos se compone de casi 250,000 diálogos de películas y más de 40,000 palabras distintas. Se ha elegido este set de datos por su variedad y porque tiene diálogos continuados, por lo que nuestro modelo de secuencias puede aprovechar su estructura al máximo. [3]

Se han realizado muchos experimentos con distintas arquitecturas de redes, a continuación podemos ver una tabla resumen con los principales resultados de entrenar los modelos para los dos sets de datos:

Set de datos	WikiText-2	Movie Lines
MaxPooling (Baseline)	76.177	60.490
AveragePooling (Baseline)	76.269	68.012
LSTM-512	31.789	25.969
LSTM-1024	31.360	25.303
GRU-512	33.092	25.058
GRU-1024	24.715	23.221
GRU-Dropout 0.15	28.520	25.681
Grave et al. 2016 [40]	68.9	-
Gong et al. 2018 [43]	39.14	-
OpenAI 2019 [3]	18.34	-

TABLA 5.1. TABLA DE PERPLEJIDADES, CUANTO MENOR, MEJOR EL MODELO

Como *baselines* se han usado dos tipos de red neuronal simples como son la MaxPooling, que coge el máximo de la entrada, y la AveragePooling, que hace la media. En el Anexo A se podrá encontrar una tabla resumen con los principales parámetros utilizados para el entrenamiento de estas redes.

Como podemos ver, en general la diferencia que hay entre las LSTM y las GRU es muy pequeña, como se explica en el estudio teórico [9]. Asimismo, pese a usar dropout, la red no mejora.

Respecto al estado del arte, este modelo mejora los resultados con la excepción del modelo de OpenAI. Sin embargo, este trabajo se empezó antes de la publicación de los resultados de dicho modelo y no se han podido aplicar todas las mejoras que introduce en la generación de texto. De hecho, la complejidad del trabajo de OpenAI es tal que, para replicar sus resultados y sus métodos, se requeriría un trabajo mucho más amplio que simplemente dos Trabajos de Fin de Grado.

Aun así, podemos ver como obtiene una perplejidad mejor a la de Grave (el creador de WikiText) y mejor que la última mejor perplejidad reportada antes de la de OpenAI. Sin embargo es bastante difícil hacer una comparación más aproximada ya que cada autor calcula la perplejidad de una forma ligeramente distinta. Por ejemplo aquí se hace una partición en entrenamiento y validación, sin embargo, en otros *papers* no hacen validación pero calculan la perplejidad usando otros datos de test. También hay que tener en cuenta que ellos calculan la perplejidad por caracteres (más fácil) y nosotros por palabras.

En este caso se usa la perplejidad de una distribución discreta de probabilidades. La explicación completa se puede encontrar en el TFG de Matemáticas [9].

Por otro lado se observa claramente una gran mejoría respecto a los *baselines*, lo que demuestra claramente que las redes neuronales recurrentes, en concreto las de la familia de las LSTM, son muy útiles para la generación de texto.

Para finalizar, veremos algunos extractos del texto generado. Primero podemos ver algunos extractos del set de datos de Chupitos que, dado a su simplicidad, son todos bastante creibles e indistinguibles de los que podría haber creado un humano:

"Tequila y licor de mora."

"Martini, licor de naranja y granadina."

"Ginebra, licor y piña y kiwi."

"Ron, licor de melocotón y kiwi."

A continuación, se usa un set de datos más complejo:

"Wait a minute, you're not exactly how I am. I don't know what I am for who is no more than I."

Como podemos ver en este extracto de una red que ha aprendido con el set de datos de Movie Lines, la red intenta replicar complejas construcciones sintácticas, sin embargo a medida que va generando más palabras, va perdiendo el hilo. Aun así, esto tiene mucho sentido dado que en las películas las intervenciones de los personajes suelen ser cortas y necesitan de la respuesta de otro personaje para poder seguir. También se puede apreciar como al final de la frase dice "who is no more than **I**." cuando la construcción gramaticalmente correcta sería "who is no more than **me**." Esto podría debersee, en parte a que en las películas muchas veces se suele usar un vocabulario vulgar y poco correcto y la red aprende de lo que ve y no del lenguaje perfecto.

A continuación podemos ver como la red también puede generar conversaciones entre personajes:

"Hey, you know I'm just a good girl."

"And I think so."

Muchas veces, cuando genera frases largas (porque se le ha pedido que genere muchas palabras) en vez de crear cosas sin sentido, genera la que sería la respuesta del siguiente personaje a intervenir en una posible conversación.

En el siguiente caso, incluso usa el nombre de alguien para hablarle. Podemos ver que, pese a generar una oración completa y con sentido, no es formalmente correcta. Aunque informalmente sea común decir "I think you didn't", la forma correcta sería "I don't think you did":

"Brother Albert I think you didn't do that, you were in the hospital with your father."

Para finalizar, podemos ver también que, a veces, cuando el *input* de la generación de la red no está en su vocabulario, la red llega a un mínimo local y no se le puede sacar de ahí:

“Hasta la vista la la la la la la la la la la la la la la la la la la la la la la”

En este caso se usó el fragmento “Hasta la vista” para generar el texto, pero como la red estaba entrenada con textos en inglés, no supo como reaccionar.

Esto demuestra que, pese a los esperanzadores resultados, el comportamiento no es siempre el deseado y, a veces se podría mejorar la generación.

5.2. Complejidad de un texto

La mayoría de medidas de complejidad y legibilidad de textos tienen varios puntos débiles, por eso se intenta usar el aprendizaje automático para crear otras medidas más robustas y generalizables [44] [45].

Para este experimento, se ha entrenado una red neuronal con varios sets de datos abiertos y se ha buscado correlación entre el número de palabras distintas en el set de datos y la perplejidad obtenida durante el entrenamiento y una métrica llamada Type-Token Ratio que calcula el ratio entre el número de palabras distintas y el número total de palabras:

$$ttr = tamaño\_del\_vocabulario / número\_de\_tokens \tag{5.1}$$

Antes de pasar a los resultados, se explicará brevemente los nuevos sets de datos usados:

- Apocalipsis: El libro del Apocalipsis del Nuevo Testamento de la Biblia
- Biblia: La biblia entera
- Chupitos: Pequeña serie de nombres de chupitos.
- Superhéroes: Extractos cortos de películas de superhéroes.
- Nietzsche: Una serie de ensayos del filósofo alemán Friedrich Nietzsche.
- Lovecraft: Varias novelas del escritor H.P. Lovecraft.
- Titles: Lista con títulos de películas, obtenida en IMDB.
- Taglines: Lista con las frases que acompañan a los títulos de las películas en los carteles, obtenida en IMDB.
- Plots: Lista de resúmenes de películas, obtenida en IMDB.
- HarryPotter: Toda la saga de Harry Potter de J.K. Rowling.
- QuijAvellaneda: El libro de Don Quijote escrito por Alonso Fernández de Avellaneda.

- Quijote: Los libros de Don Quijote escritos por Miguel de Cervantes.

Como podemos ver en la siguiente gráfica, hay cierta correlación entre el tamaño del vocabulario y la perplejidad. Sin embargo hay algunos resultados que no acaban de parecer coherentes, como por ejemplo textos muy grandes y muy fáciles o viceversa. La razón de esto es que el tamaño del texto no es significativo de su complejidad o legibilidad, ni siquiera el Type-Token Ratio porque es una métrica muy simple.

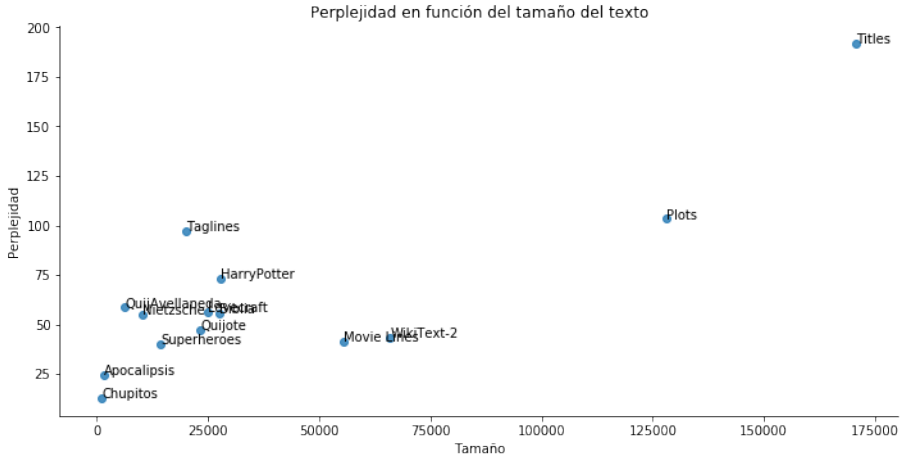

Fig. 5.1. Perplejidad en función del tamaño de los datos

A continuación vemos la comparativa de las tres métricas juntas:

Set de datos	Type-Token Ratio	Perplejidad	Tamaño de vocabulario
Apocalipsis	0.1482	24.439	1728
Biblia	0.03683	55.497	27675
Chupitos	0.2537	12.779	1031
Superheroes	0.1333	39.948	14276
Nietzsche	0.11	54.915	10262
Lovecraft	0.05743	56.181	24904
Titles	0.1008	192.018	170818
Taglines	0.07545	97.157	20077
Plots	0.02074	103.362	127997
HarryPotter	0.02151	73.098	27723
QuijAvellaneda	0.1511	58.695	6304
Quijote	0.06128	47.397	23270
Movie Lines	0.01483	41.431	55384
WikiText-2	0.03748	43.145	65937

TABLA 5.2. TYPE-TOKEN RATIOS, PERPLEJIDADES Y TAMAÑOS DE VOCABULARIOS

Podemos observar que, a mayor tamaño del vocabulario, mayor es la perplejidad del modelo. Este fenómeno se puede observar sobre todo en los dos extremos; textos con poca riqueza de vocabulario como Apocalipsis o Chupitos tienen las menores perplejidades de todos los sets de datos usados, en cambio, textos como Titles o Plots, cuyo vocabulario es enorme, tienen perplejidades que se disparan.

También podemos observar como, en efecto, el Type-Token Ratio no es una buena métrica en todos los casos ya que la medición más alta es en set de datos de Chupitos que, curiosamente es el más corto y simple, sin embargo se da la paradoja que en comparación a su tamaño tiene un vocabulario muy amplio.

Un fenómeno similar se puede apreciar al comparar los resultados de QuijAvellaneda y Quijote, el primero es mucho más corto que el segundo y tiene un vocabulario menos variado. Sin embargo, en proporción sí es más complejo que el Quijote original porque a la red no le da tiempo a aprender todas las estructuras del texto con tan pocas muestras.

Una posible desventaja de este tipo de métricas es que una red neuronal tarda en entrenarse más de lo que podría ser razonable para una métrica que, en principio, debería ser fácil. Aun así, compensaría por la robustez y calidad de la nueva métrica.

Por lo tanto, a priori, calcular la perplejidad con una red neuronal puede ser un muy buen indicador para saber cuán difícil es de entender un texto por un ser humano, evitando caer en la simplicidad de algunas métricas tradicionales que se quedan cortas en algunos textos. Aun así habría que hacer más experimentos con más textos y en los que se comparase también con la dificultad de lectura reportada por una muestra estadísticamente relevante de personas.

6. CONCLUSIONES

6.1. Cronograma

En el siguiente cronograma se puede apreciar el esfuerzo dedicado a cada tarea principal de este trabajo. Cabe destacar la gran cantidad de tiempo dedicado a la creación de la solución al problema usando código abierto. Esto se debe al uso de tecnologías y librerías no vistas en la carrera y el alto nivel de complejidad que ha adquirido el código.

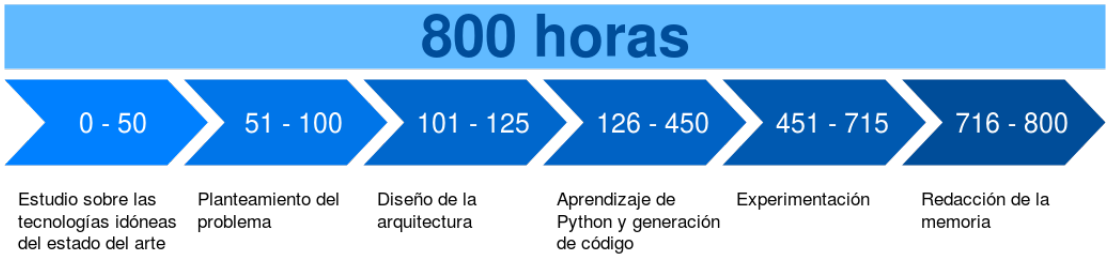

Fig. 6.1. Cronograma del desarrollo del trabajo

Por otra parte, la experimentación ha sido muy costosa debido a que había un límite de hardware que hacía que los experimentos no pudieran ir máś rápido. Asimismo, se han llevado a cabo una gran cantidad de experimentos para asegurar la mejor solución, esto también ha dilatado el proceso.

La mayor ventaja en la realización de este trabajo ha sido tener los conocimientos teóricos obtenidos en el Trabajo de Fin de Grado de Matemáticas [9], sin los cuales no hubiera habido tanto tiempo para conseguir una solución tan refinada y una cantidad de experimentos tan abundante.

6.2. Logros

Durante el desarrollo de este trabajo, se han logrado todos los objetivos propuestos. Hemos conseguido demostrar que con software libre y con unas especificaciones de hardware modestas se puede conseguir entrenar modelos de secuencias de generación de texto que se acercan a los usados por grandes entidades y grupos de investigación.

Además se han descubierto estrategias de software para optimizar mucho el uso de memoria de estas redes neuronales. Los generadores y, en concreto, los generadores como decoradores encadenados tienen un gran potencial hacer de forma relativamente simple tareas que son muy complejas.

6.3. Trabajo futuro

Se han demostrado los buenos resultados de las redes recurrentes para generar texto, sin embargo también se ha demostrado que hay margen de mejora, tanto para estas redes como para el estado del arte. De esta manera, es importante seguir con la investigación en este campo, sobre todo si esas investigaciones desembocan en modelos y código abiertos para contribuir a la prosperidad de la humanidad.

Como objetivos de investigación futuros, el principal es replicar los modelos de OpenAI para conseguir sus mismos resultados. Asimismo, otro objetivo importante es probar los nuevos paradigmas de *embeddings* que han crecido mucho en paralelo al desarrollo de este trabajo. Un ejemplo de estos nuevos paradigmas es BERT [46] que combina los *embeddings* con un modelo estudio de la atención para obtener más información sobre las relaciones contextuales no obvias inherentes al texto.

Por otro lado, una vez terminados dichos experimentos, la mejor forma real de comprobarlos sería enfrentándolos al criterio de las personas. Primero de la forma que se explica en los experimentos, pero, más ambiciosamente realizando el Test de Turing [10], pues cada vez los modelos del lenguaje son mejores y, a medida que el tamaño de los datos y de las redes neuronales incremente, los resultados van a mejorar.

BIBLIOGRAFÍA

[1] A. Kannan et al., "Smart Reply: Automated Response Suggestion for Email", en *Proceedings of the ACM SIGKDD Conference on Knowledge Discovery and Data Mining (KDD) (2016).*, 2016. eprint: arXiv:1606.04870.

[2] T. Young, D. Hazarika, S. Poria y E. Cambria, *Recent Trends in Deep Learning Based Natural Language Processing*, 2017. eprint: arXiv:1708.02709.

[3] OpenAI, *Better Language Models and Their Implications*, 2019. eprint: arXiv:1502.02367. [En línea]. Última consulta: Junio de 2019: https://openai.com/blog/better-language-models/.

[4] O. Vinyals y Q. Le, *A Neural Conversational Model*, 2015. eprint: arXiv:1506.05869.

[5] L. Leppänen, M. Munezero, M. Granroth-Wilding y H. Toivonen, "Data-Driven News Generation for Automated Journalism", ene. de 2017, pp. 188-197. DOI: 10.18653/v1/W17-3528.

[6] S. Hochreiter y J. Schmidhuber, "Long Short-term Memory", *Neural computation*, vol. 9, pp. 1735-80, dic. de 1997. DOI: 10.1162/neco.1997.9.8.1735.

[7] K. Cho et al., *Learning Phrase Representations using RNN Encoder-Decoder for Statistical Machine Translation*, 2014. eprint: arXiv:1406.1078.

[8] J. Chung, C. Gulcehre, K. Cho e Y. Bengio, *Gated Feedback Recurrent Neural Networks*, 2015. eprint: arXiv:1502.02367.

[9] G. G. Subies, *Estudio teórico sobre modelos de secuencias con redes neuronales recurrentes para la generación de texto*, 2019.

[10] A. M. Turing, "Computing Machinery and Intelligence", *Mind*, vol. 49, pp. 433-460, 1950.

[11] A. Radford et al., "Language Models are Unsupervised Multitask Learners",

[12] I. Goodfellow, Y. Bengio y A. Courville, *Deep Learning*. MIT Press, 2016. [En línea]. Última consulta: Junio de 2019: http://www.deeplearningbook.org.

[13] T. M. Mitchell, *Machine Learning*, 1.ª ed. New York, NY, USA: McGraw-Hill, Inc., 1997.

[14] Sewaqu, *Regresión lineal*. [En línea]. Última consulta: Junio de 2019: https://commons.wikimedia.org/w/index.php?curid=11967659.

[15] . [En línea]. Última consulta: Junio de 2019: https://www.naftaliharris.com/blog/visualizing-dbscan-clustering/.

[16] . [En línea]. Última consulta: Junio de 2019: https://asociacioneducar.com/neurona-multipolar.

[17] F. Rosenblatt, “The Perceptron: A Probabilistic Model for Information Storage and Organization in The Brain”, *Psychological Review*, pp. 65-386, 1958.

[18] S. S. Haykin, *Neural networks and learning machines*. Pearson Education, 2009, p. 124.

[19] J. Gillis. [En línea]. Última consulta: Junio de 2019: https://en.wikipedia.org/wiki/Gradient_descent.

[20] D. E. Rumelhart, G. E. Hinton y R. J. Williams, “Learning representations by back-propagating errors”, *Nature*, vol. 323, pp. 533-, oct. de 1986. [En línea]. Última consulta: Junio de 2019: http://dx.doi.org/10.1038/323533a0.

[21] J. J. Hopfield, “Neural networks and physical systems with emergent collective computational abilities”, *Proceedings of the National Academy of Sciences of the United States of America*, vol. 79, n.º 8, pp. 2554-2558, 1982. [En línea]. Última consulta: Junio de 2019: http://view.ncbi.nlm.nih.gov/pubmed/6953413.

[22] C. Olah, *Understanding LSTM Networks*. [En línea]. Última consulta: Junio de 2019: http://colah.github.io/posts/2015-08-Understanding-LSTMs/.

[23] Á. B. Jiménez, *Neurowriter*. [En línea]. Última consulta: Junio de 2019: https://github.com/albarji/neurowriter.

[24] A. Karpathy, *The Unreasonable Effectiveness of Recurrent Neural Networks*. [En línea]. Última consulta: Junio de 2019: http://karpathy.github.io/2015/05/21/rnn-effectiveness/.

[25] M. Schuster y K. Paliwal, “Bidirectional Recurrent Neural Networks”, *Trans. Sig. Proc.*, vol. 45, n.º 11, pp. 2673-2681, nov. de 1997. DOI: 10.1109/78.650093.

[26] P. Chapman et al., *CRISP-DM 1.0*. [En línea]. Última consulta: Junio de 2019: ftp://ftp.software.ibm.com/software/analytics/spss/support/Modeler/Documentation/14/UserManual/CRISP-DM.pdf.

[27] G. Piatetsky-Shapiro, *KDnuggets Methodology Poll*, 2002. [En línea]. Última consulta: Junio de 2019: https://www.kdnuggets.com/polls/2002/methodology.htm.

[28] ——, *KDnuggets Methodology Poll*, 2004. [En línea]. Última consulta: Junio de 2019: https://www.kdnuggets.com/polls/2004/data_mining_methodology.htm.

[29] ——, *KDnuggets Methodology Poll*, 2007. [En línea]. Última consulta: Junio de 2019: https://www.kdnuggets.com/polls/2007/data_mining_methodology.htm.

[30] ——, *KDnuggets Methodology Poll*, 2014. [En línea]. Última consulta: Junio de 2019: https://www.kdnuggets.com/polls/2014/analytics-data-mining-data-science-methodology.html.

[31] K. Jense, *Process diagram showing the relationship between the different phases of CRISP-DM*. [En línea]. Última consulta: Junio de 2019: https://commons.wikimedia.org/w/index.php?curid=24930610.

[32] H. or Tails, *What We Do in the Kernels - A Kaggle Survey Story*, 2018. [En línea]. Última consulta: Junio de 2019: https://www.kaggle.com/headsortails/what-we-do-in-the-kernels-a-kaggle-survey-story.

[33] Kaggle, *2018 Kaggle ML & DS Survey*, 2018. [En línea]. Última consulta: Junio de 2019: https://www.kaggle.com/kaggle/kaggle-survey-2018.

[34] S. Overflow, *Developer Survey Results*, 2019. [En línea]. Última consulta: Junio de 2019: https://insights.stackoverflow.com/survey/2019#technology.

[35] F. Chollet. [En línea]. Última consulta: Junio de 2019: https://keras.io/why-use-keras/.

[36] M. Abadi et al., *TensorFlow: Large-Scale Machine Learning on Heterogeneous Distributed Systems*, 2016. eprint: arXiv:1603.04467.

[37] *Keras en tensorflow*. [En línea]. Última consulta: Junio de 2019: https://www.tensorflow.org/api_docs/python/tf/keras.

[38] G. G. Subies, *Modelos de secuencias con redes neuronales recurrentes para la generación de texto*, 2019. [En línea]. Última consulta: Junio de 2019: https://github.com/GuillemGSubies/TFG.

[39] L. Buitinck et al., "API design for machine learning software: experiences from the scikit-learn project", en *ECML PKDD Workshop: Languages for Data Mining and Machine Learning*, 2013, pp. 108-122. [En línea]. Última consulta: Junio de 2019: https://scikit-learn.org/stable/modules/generated/sklearn.base.BaseEstimator.html.

[40] S. Merity, C. Xiong, J. Bradbury y R. Socher, "Pointer Sentinel Mixture Models", *CoRR*, vol. abs/1609.07843, 2016. arXiv: 1609.07843.

[41] C. Danescu-Niculescu-Mizil y L. Lee, "Chameleons in imagined conversations: A new approach to understanding coordination of linguistic style in dialogs.", en *Proceedings of the Workshop on Cognitive Modeling and Computational Linguistics, ACL 2011*, 2011.

[42] M. Marcus et al., "The Penn Treebank: Annotating Predicate Argument Structure", en *Proceedings of the Workshop on Human Language Technology*, ép. HLT '94, Plainsboro, NJ: Association for Computational Linguistics, 1994, pp. 114-119. DOI: 10.3115/1075812.1075835.

[43] C. Gong et al., "FRAGE: Frequency-Agnostic Word Representation", *CoRR*, vol. abs/1809.06858, 2018. arXiv: 1809.06858.

[44] K. Collins-Thompson y J. P. Callan, "A Language Modeling Approach to Predicting Reading Difficulty.", ene. de 2004, pp. 193-200.

[45] S. E. Schwarm y M. Ostendorf, “Reading Level Assessment Using Support Vector Machines and Statistical Language Models”, en *Proceedings of the 43rd Annual Meeting on Association for Computational Linguistics*, ép. ACL ’05, Ann Arbor, Michigan: Association for Computational Linguistics, 2005, pp. 523-530. DOI: 10.3115/1219840.1219905.

[46] J. Devlin, M. Chang, K. Lee y K. Toutanova, “BERT: Pre-training of Deep Bidirectional Transformers for Language Understanding”, *CoRR*, vol. abs/1810.04805, 2018. arXiv: 1810.04805.

ANEXO A

En la siguiente tabla se pueden encontrar todos los parámetros de las redes neuronales usadas en los experimentos, sin embargo, se ha experimentado con una gran cantidad de combinaciones de parámetros.

Tipo de red	LSTM-512	LSTM-1024	GRU-512	GRU-1024	GRU-Dropout 0.15	Red para la complejidad
Optimizador	ADAM	ADAM	ADAM	ADAM	ADAM	ADAM
Dropout	0	0	0	0	0.15	0
Capas ocultas	1	1	1	1	1	1
Neuronas	512	1024	512	1024	1024	1024
Tipo de Neuronas	LSTM Bidireccional	LSTM Bidireccional	GRU Bidireccional	GRU Bidireccional	GRU Bidireccional	GRU Bidireccional
Épocas*	300	300	300	300	300	300
Tamaño de batch	64	64	64	64	64	64
Longitud de secuencia	300	300	300	300	300	300
Embeddings	FastText	FastText	FastText	FastText	FastText	FastText
Dimensión de salida de la capa de embeddings	64	64	64	64	64	64
Ratio entrenamiento/validación	3:1	3:1	3:1	3:1	3:1	3:1
Función de activación	softmax	softmax	softmax	softmax	softmax	softmax
Función de pérdida	categorical_crossentropy	categorical_crossentropy	categorical_crossentropy	categorical_crossentropy	categorical_crossentropy	categorical_crossentropy
Métrica	Perplejidad discreta	Perplejidad discreta	Perplejidad discreta	Perplejidad discreta	Perplejidad discreta	Perplejidad discreta
Ritmo de aprendizaje	Dinámico**	Dinámico**	Dinámico**	Dinámico**	Dinámico**	Dinámico**

TABLA 6.1. PARÁMETROS DE LAS REDES NEURONALES USADAS

* 300 épocas por defecto, pero al tener una parada temprana, en ocasiones eran menos.

** El ritmo por defecto de Keras y descenso de un factor de 0.8 cada 8 épocas sin mejorar los resultados.

Usando esos parámetros, la estructura de las redes ha sido la siguiente: Capa de *Embeddings* → Capas ocultas → Capa dropout (opcional) → Capa densa final.

- Capa de Embeddings: Puede ser un simple *one-hot encoding* o una capa con embeddings preentrenados, como los de FastText y su dimensión de salida es de 64.
- Capas ocultas: Principalmente GRU o LSTM aunque también se podían elegir capas más simples que hacen la media de los vectores de *embeddings*, etc.
- Capa dropout (opcional): Añade dropout a la red neuronal para reducir el sobreajuste. No aparece en la tabla dado que no se usa en esas redes.
- Capa densa: Intenta predecir el índice de la palabra siguiente en el tokenizador mediante la función de activación *softmax*. El tamaño de la salida es el tamaño del vocabulario ya que se predice la probabilidad para cada palabra.