Universitat Oberta de Catalunya (UOC)

Máster Universitario en Ciencia de Datos (*Data Science*)

TRABAJO FINAL DE MÁSTER

Área: PLN

Modelización de temas de llamadas en tiempo real

Borrador

Autor: Manuel E. Gómez Montero

Tutora UOC: Ana Valdivia Garcia

Tutor TE: Antonio Fernández Gallardo

Profesor: Jordi Casas

Madrid, 14 de octubre de 2019

Resumen

Un call-center es el área de una empresa el cuál se encarga de recibir y transmitir llamadas desde o hacia clientes, socios comerciales u otras compañías externas. Debido a la gran cantidad de información que se transfiere en estos centros, resulta una tarea esencial optimizar el tiempo de respuesta para así reaccionar en tiempo real a las peticiones de los clientes y mejorar la percepción que estos tienen sobre la compañía.

Una manera de mejorar el rendimiento es detectar el tema de las llamadas mediante técnicas de *machine learning* dando la posibilidad a la empresa de reaccionar en tiempo real, en función de la temática que se este tratando en cada momento.

El sistema que se presenta en el documento nos permite, a partir de la transcripción de las llamadas al *call-center* de Telefónica España, descubrir en tiempo real la temática de las mismas. Esta modelización de *topics* se ha realizado utilizando métodos de Procesamiento de Lenguaje Natural y aprendizaje profundo. El sistema realiza la clasificación de las nuevas llamadas en tiempo real, permitiendo a los usuarios visualizar la evolución en la temática de las mismas y generar alertas en base a anomalías.

TODO Es un borrador volver al resumen una vez acabado el proyecto.

Palabras clave: “natural language processing”, “sentiment analysis”, “real time”, “call center”, “topic modeling”, “deep learning”

Índice general

Capítulo 1

Introducción

Este primer capítulo del trabajo tiene como objetivo presentar, a grandes rasgos, la propuesta (sección 1.1), los objetivos que pretendemos lograr (sección 1.3), la motivación que nos ha llevado a abordar este proyecto (sección 1.2) y un repaso a las tareas que serán necesarias para la ejecución del mismo (sección 1.4).

Por último, dedicaremos una sección que describa brevemente los diferentes apartados de los que constará el documento y el objetivo de cada uno (sección 1.5).

1.1. Descripción general de la propuesta

En los últimos años, la explosión ingente en la generación de datos y el avance en las capacidades tecnológicas que nos permiten recolectar, almacenar y procesar los datos generados; han provocado que empecemos a abordar el estudio de otro tipo de datos no estructurados que antes no se podían analizar como imágenes, textos, audios, etc. Como resultado, diferentes áreas del conocimiento (Procesamiento del Lenguaje Natural, Análisis de Imágenes) han experimentado un creciente interés tanto en la comunidad científica como en el mundo de los negocios.

Dentro de los datos no estructurados, una de las fuentes de información con mayor potencial en todas las grandes empresas que prestan servicio al público general, son las llamadas que los clientes realizan a su *call-center*, ya que nos permiten obtener una idea de la percepción que los clientes tienen de nuestra empresa y de sus preocupaciones en cada momento.

La propuesta que pretendemos abordar en este trabajo consiste en extraer la temática de estas llamadas en el momento en el que son capturadas. Aunque actualmente esta captura se hace periódicamente pretendemos construir una solución que nos permita el tratamiento de las mismas en tiempo real o streaming, y de esta manera mejorar el rendimiento de estos centros.

Esta extracción en tiempo real nos permitirá conocer cómo evolucionan los temas que tratan nuestros clientes cuando llaman a nuestro *call-center* para así poder reaccionar inmediatamente

ante una preocupación concreta.

1.2. Motivación

La motivación que nos ha llevado a acometer un proyecto de esta naturaleza viene originada por diferentes factores que están ligados tanto al negocio como a las capacidades técnicas disponibles en la empresa.

Por un lado, la capacidad de obtener la temática de las llamadas en tiempo real se presenta como una oportunidad de mejorar la operatividad de un *call-center* y por ende la satisfacción de los clientes, permitiéndonos entenderlos mejor y así reaccionar de una manera ágil a sus necesidades reales.

Desde el punto de vista técnico, también es el momento ideal para emprender este proyecto debido tanto a la disponibilidad periódica de transcripciones de las llamadas, que nos permiten ahorrarnos el paso de realizar un *Speech 2 Text* para obtener nuestro conjunto de datos; como al aumento de capacidades técnicas en la empresa que nos permitirán tanto entrenar nuestros modelos, como poder tratar y explotar los datos en tiempo real.

1.3. Objetivos

En este apartado definiremos los objetivos que se pretenden conseguir con este proyecto. Estos objetivos deben ser *SMART*, es decir:

- ***S**pecific*: Deben plantearse de una forma detallada y concreta.
- ***M**easurable*: Deben poder medirse con facilidad.
- ***A**chievable*: Deben ser objetivos realistas.
- ***R**elevant*: Tienen que ser relevantes para la empresa y ofrecernos un beneficio claro.
- ***T**imely*: Estos objetivos tienen que tener un tiempo establecido.

El objetivo general es optimizar el proceso de atención de llamadas en el call-center mediante técnicas de Procesamiento del Lenguaje Natural y Aprendizaje Profundo. Concretamente, los objetivos específicos que se pretenden conseguir con este proyecto son:

- **Construir un modelo que nos permita extraer la temática de las llamadas** a partir de su transcripción a texto. Este objetivo debemos alcanzarlo en la fase de modelado y podremos medir su éxito atendiendo al porcentaje de llamadas que podamos clasificar correctamente en un proceso de test. Se trata del objetivo principal del proyecto.

- Desarrollar un mecanismo que nos permita **extraer esta temática para nuevas llamadas en tiempo real**. De este modo tendremos un sistema vigente cuando la frecuencia en la recepción de las llamadas aumente. Este objetivo se deberá alcanzar en la fase de productivización.

- Disponer de una **visualización en tiempo cuasi real** para que pueda visualizarse la evolución de las temáticas a lo largo del tiempo. Este objetivo se deberá alcanzar en la fase de productivización.

- Proporcionar un **sistema de alertado** que nos permita detectar anomalías en el número de llamadas que se reciben de un determinado tema. Este objetivo se deberá alcanzar en la fase de productivización.

En las conclusiones de este proyecto se evaluará el éxito o fracaso del mismo en función del grado de cumplimiento de estos objetivos.

1.4. Tareas y planificación

El proyecto se llevará a cabo desde el 16 de Septiembre hasta el 20 de Febrero. Para poder abordar la ejecución del mismo se han extraído las siguientes tareas principales:

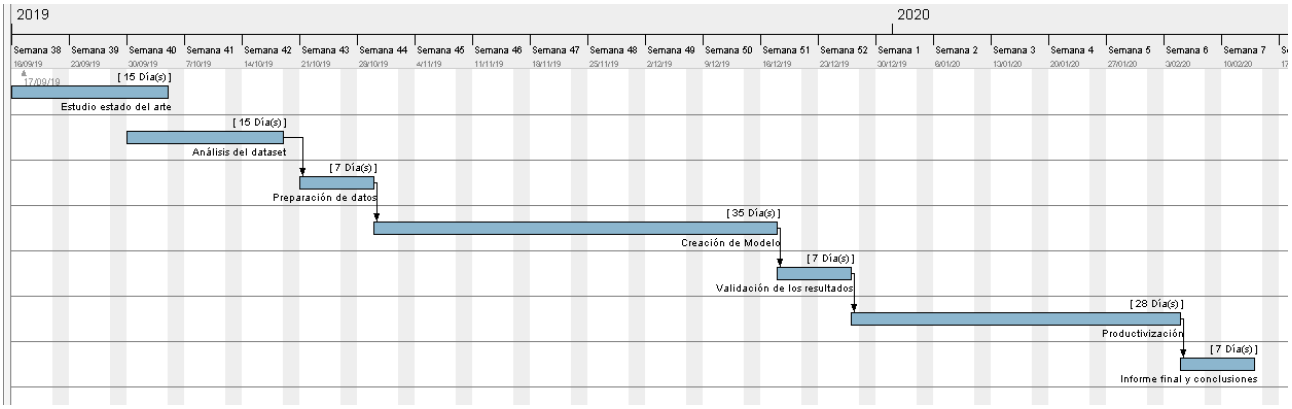

Figura 1.1: Diagrama de Gantt

- **Estudio estado del arte**: En esta fase se realizará una prospección para conocer el estado del arte en todos los puntos relacionados con el proyecto: Procesamiento del Lenguaje Natural, tecnologías de tratamiento de datos en tiempo real y *Big Data*.

- **Análisis del *dataset***: El propósito de esta tarea es entender el *dataset* y estudiar las posibilidades del mismo.

- **Preparación del *dataset***: Una vez realizado el estudio del *dataset* es necesario realizar labores de limpieza y transformación de los datos de modo que estos datos sean válidos para nuestro objetivo.
- **Creación del modelo**: En esta fase se procederá a la creación de un modelo capaz de obtener los temas de los que habla una determinada llamada. Este modelo será el *core* de nuestro proyecto.
- **Validación de los resultados**: Una vez entrenado el modelo será necesario validar los resultados obtenidos para poder evaluar la bondad de nuestro modelo.
- **Productivización**: El trabajo no acaba con la creación de un buen modelo que nos permita extraer los temas de nuestras llamadas. Este modelo tendrá que ser puesto en producción y permitir al usuario final extraer los temas de las llamadas en tiempo real y darle la opción de crear alarmas basadas en la variación del número de eventos (llamadas) de un determinado tema.
- **Informe final y conclusiones**: Por último, una vez llevado a a producción nuestro modelo, se realizará un informe final donde, entre otros puntos, se evaluarán los resultados obtenidos y se extraerán conclusiones y pasos futuros.

Estas fases están basadas en el estándar ***CRISP-DM*** ([6]), añadiendo una última tarea para nuestro informe final, *CRISP-DM* nos proporciona una descripción del ciclo de vida de los proyectos de minería de datos de un modo bastante similar al que se aplica en los modelos de ciclo de vida de desarrollo *software*.

En la Figura 1.2 se observa el diseño de este modelo y cómo representa el ciclo de vida de un proyecto de minería de datos. En la imagen podemos ver en primer lugar un círculo exterior que refleja la naturaleza cíclica de los proyectos de minería de datos, además vemos cómo la secuencia de tareas no es rígida, pudiendo saltar hacia adelante o atrás entre tareas. En la gráfica se representan mediante flechas las dependencias más importantes y usuales entre tareas.

En nuestro desarrollo usaremos este modelo, aunque en el diagrama de la Figura 1.1 aparezca una secuencia de tareas más rígida, será usual, por ejemplo, el salto recíproco entre las fases de preparación de los datos y creación del modelo.

1.5. Estructura del documento

TODO Hacer repaso breve de los apartados del documento final.

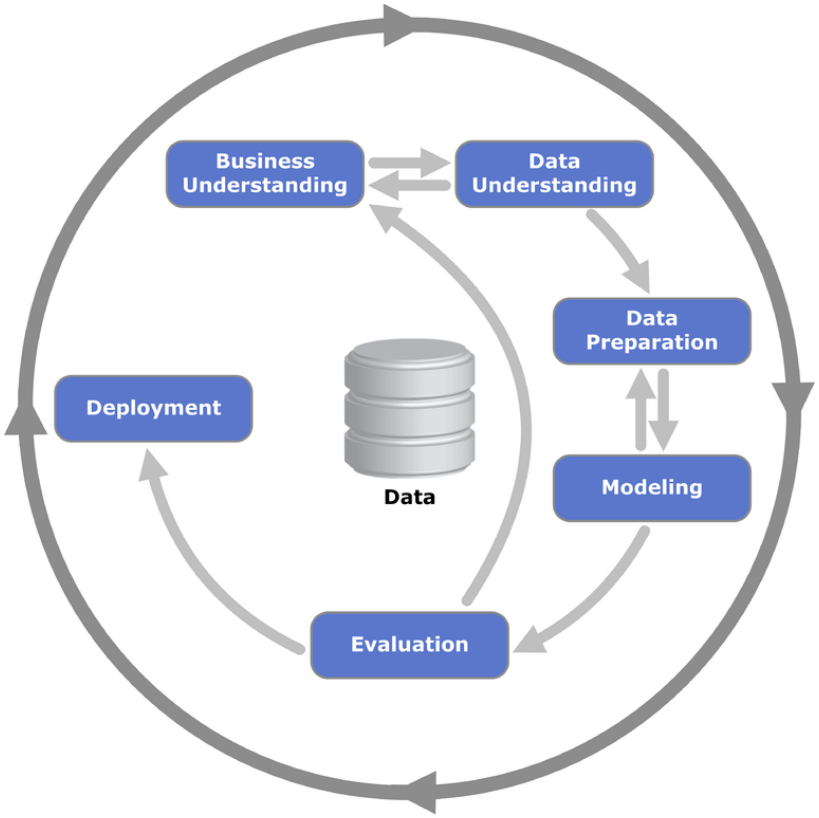

Figura 1.2: Fases del modelo CRISP-DM

Capítulo 2

Estado del Arte

El objetivo de este apartado es hacer un recorrido por el estado del arte relacionado con el proyecto, este recorrido lo enfocaremos desde tres puntos de vista diferentes:

- **Procesamiento del Lenguaje Natural**: En la sección 2.1 nos centraremos en el procesamiento del lenguaje natural y su evolución a lo largo del tiempo.
- ***Deep Learning* y aplicación al Procesamiento del Lenguaje Natural**: En la sección 2.2 pondremos foco en el *Deep Learning*, sus ventajas y cómo se están aplicando estos métodos al procesamiento de lenguaje natural.
- ***Big Data* y *Fast Data***: Por último, en la sección 2.3, haremos un repaso a la evolución del *Big Data* y cómo la tendencia actual es realizar el procesamiento en tiempo real mediante *Fast Data*.

2.1. Procesamiento de lenguaje natural

2.1.1. Historia

Para hablar de los orígenes del Procesamiento del Lenguaje Natural (a partir de ahora se usarán indistintamente las siglas PLN) tal y como lo conocemos, tendríamos que remontarnos a los años 50, concretamente al artículo "*Computing Machinery and Intelligence*" escrito por Alan Turing [24]. En este artículo aparece el PLN dentro del campo de la inteligencia artificial y se presenta por primera vez el conocido "Test de Turing". Este test convirtió la pregunta abstracta de "¿Son capaces de pensar las máquinas?" en un juego llamado: "*The Imitation Game*". El juego propuesto inicialmente, de forma muy resumida, consiste en ver si una persona (interrogador) interrogando a dos personas (un hombre y una mujer), era capaz de descubrir el sexo de cada una; la modificación del mismo sustituye las dos personas de distinto sexo por

una persona y una máquina y el interrogador debe ser capaz de descubrir si las preguntas están siendo respondidas por un humano o una máquina. En el caso de que no sepa discernir, la computadora gana la partida. Podemos encontrar más información al respecto en el libro [25].

A partir de los avances de Turing y hasta los años 80 el crecimiento en el campo del PLN se produjo principalmente con la creación de complejos sistemas basados en reglas escritas a mano. Fue en esta década cuando empezamos a vivir la incorporación de algoritmos de *machine learning* enfocados al procesamiento del lenguaje natural. Este hecho se vio motivado principalmente por el increíble avance en la capacidad de cómputo, ya predicho por la ley de Moore, y por la aplicación de teorías ya existentes como los trabajos de Chomsky.

Desde el comienzo de la aplicación de modelos de *machine learning*, y de nuevo motivados por el crecimiento de la capacidad computacional de los sistemas actuales, se ha pasado de utilizar árboles de decisión, que creaban de manera automática reglas similares a las que se venían creando manualmente, a los modelos de *deep learning* que están en auge en la última década.

2.1.2. Aplicaciones

En el apartado anterior hicimos referencia a "*The Imitation Game*" como inicio de lo que hoy conocemos como procesamiento de lenguaje natural, sin embargo, las aplicaciones en este campo han crecido de forma vertiginosa en estos 70 años, principalmente en las últimas décadas. Hoy en día, si tuviéramos que contestar a la pregunta: "¿son capaces de pensar las máquinas?", implicaría algo más que superar el test de Turing. Mirando a nuestro alrededor nos encontraríamos con asistentes de voz como Alexa o Siri que, no solo contestan a nuestras preguntas, si no que realizan un trabajo de pasar nuestra voz a texto (*Speech to Text*) y de nuevo el texto resultante a voz (*Text to Speech*). Nos encontraríamos también con sistemas capaces de realizar traducciones simultáneas, otros capaces de autocompletar textos, de identificar preguntas y respuestas, de clasificar textos de acuerdo a temas o autores, incluso de analizar sentimientos positivos o negativos teniendo como entrada un texto u opinión...

Según [12] todos estos problemas tan diversos podríamos clasificarlos según en el punto del análisis que nos centremos:

- **Análisis de palabras**: En este tipo de problemas se pone foco en las palabras, como pueden ser "perro", "hablar", "piedra" y necesitamos decir algo sobre ellas. Por ejemplo: "¿estamos hablando de un ser vivo?", "¿a qué lenguaje pertenece?", "¿cuáles son sus sinónimos o antónimos?". Actualmente este tipo de problemas son menos frecuentes, ya que normalmente no pretendemos analizar palabras aisladas sino que es preferible basarse en un contexto.

- **Análisis de textos**: En este tipo de problemas no trabajamos solo con palabras aisladas, sino que disponemos de una pieza de texto que puede ser una frase, un párrafo o un documento completo y tenemos que decir algo sobre él. Por ejemplo: "¿se trata de spam?", "¿qué tipo de texto es?", "¿el tono es positivo o negativo?", "¿quién es su autor?".Este tipo de problemas son muy comunes y nos vamos a referir a ellos como **problemas de clasificación de documentos**.

- **Análisis de textos pareados**: En esta clase de análisis disponemos de dos textos (también podrían ser palabras aisladas) y tenemos que decir algo sobre ellos. Por ejemplo, "¿los textos son del mismo autor?", "¿son pregunta y respuesta?", "¿son sinónimos?" (para el caso de palabras aisladas).

- **Análisis de palabras en contexto**: En estos casos de uso, a diferencia del primer análisis que trataba unicamente con palabras aisladas, tenemos que clasificar una palabra en particular en función del contexto en el que se encuentra.

- **Análisis de relación entre palabras**: Este último tipo de análisis tiene como objetivo deducir la relación entre dos palabras existentes en un documento.

Dependiendo del problema que queramos abordar usaremos un tipo de características del lenguaje u otro, por ejemplo, es usual que si estamos analizando palabras aisladas nos centremos en las letras de una palabra, sus prefijos o sufijos, su longitud, la información léxica extraída de diccionarios como *WordNet* [10], etc. En cambio, si estamos trabajando con texto, lo normal es que nos fijemos en otros conceptos estadísticos como el histograma de las palabras dentro del texto, ratio de palabras cortas vs largas, número de veces que aparece una palabra en un texto comparado con el resto de textos...

El proyecto que se presenta en este documento está centrado en el análisis de textos, concretamente en extraer los temas de un documento (o llamada). Este tipo de problemas se conoce como modelización de *topics*.

En el siguiente punto de este apartado nos centraremos en algunos modelos y avances en este área que puedan servirnos de apoyo para nuestro proyecto.

2.1.3. Modelización de temas

La modelización de topics hace referencia a un grupo de algoritmos de *machine learning* que infieren la estructura latente existente en un grupo de documentos.

Aunque la mayoría de los algoritmos de modelización son no supervisados, al igual que los algoritmos tradicionales de *clustering*, existen también algunas variantes supervisadas que necesitan disponer de documentos etiquetados.

Quizás el algoritmo más conocido para la modelización de topics sea el *Latent Dirichlet Allocation* (normalmente conocido por su acrónimo, LDA). LDA fué presentado en 2003 en el artículo [5]. Este algoritmo no supervisado asume que cada documento es una distribución probabilística de *topics* y cada *topic*, a su vez, es una distribución de palabras del documento. LDA usa una aproximación llamada "*bag of words*", en la que cada documento es tratado como un vector con el conteo de las palabras que aparecen en el mismo. La principal característica de LDA es que la colección de documentos comparten los mismos topics, pero cada documento contiene esos topics en una proporción diferente.

A partir de LDA surgieron numerosas variantes que repasaremos de forma breve, por ejemplo, en el mismo año de la creación de LDA y también presentado por los mismos autores en [2], surgió una **variante jerárquica** que permitía representar los *topics* jerarquicamente. En 2006 en [4] se desarrolla un modelo LDA dinámico denominado DTM (*Dinamic Topic Model*), en el que se introduce la variable temporal y los *topics* pueden ir cambiando a lo largo del tiempo. En el artículo [3] nos encontramos con otra variante de LDA llamada CTM (*Correlated topic model*) que nos permite encontrar correlaciones entre *topics*, ya que algunos temas es probable que sean más similares entre sí. Por último, nos encontramos con una variante de LDA denominada ATM (*Author-Topic Model*) propuesta por Michal Rosen-Zvi en su artículo [23] y desarrollada por el mismo en 2010, en la que los documentos son una distribución probabilística tanto de autores como de *topics*.

Podemos encontrar un resumen más completo del estado del arte en cuanto a la modelización de *topics* en el artículo [16].

2.2. Deep Learning y aplicación al PLN

El objetivo de esta sesión es entender el concepto de *Deep Learning* y analizar el estado del arte del *Deep Learning* aplicado al Procesamiento del Lenguaje Natural. Para poder entender el *Deep Learning* es conveniente entender los modelos de aprendizaje supervisados y saber qué provoca su aparición y popularidad de los últimos años. Posteriormente nos centraremos en los fundamentos del *Deep Learning* para finalizar con las aplicaciones actuales en el ámbito del Procesamiento del Lenguaje Natural y que nos sirvan de apoyo para la ejecución del proyecto.

2.2.1. Aprendizaje supervisado

El aprendizaje supervisado consiste en aprender una función a través de un conjunto de datos, llamados de entrenamiento, mediante la cual podamos obtener una salida a partir de una determinada entrada. Se espera que esta función, una vez realizado el entrenamiento, sea

capaz de producir una salida correcta incluso para datos nunca vistos. Es muy habitual el uso de estos tipos de algoritmos para casos de clasificación y/o predicción.

Buscar entre todas las posibles infinitas funciones posibles para encontrar la función que mejor se adapte a nuestro conjunto de datos es un trabajo inviable, es por ello que normalmente se realiza la búsqueda entre un conjunto de funciones limitadas. En un primer lugar, y hasta hace aproximadamente una década, los modelos más populares de aprendizaje supervisado fueron los modelos lineales, provenientes del mundo de la estadística, estos modelos son fáciles de entrenar, fáciles de interpretar y muy efectivos en la práctica.

A partir de entonces, y motivado en parte por el aumento en las capacidades de cómputo, surgen otros modelos como las máquinas de vectores de soportes (*Support Vector Machines*, SVMs) o las redes neuronales, en las que nos centraremos en el siguiente apartado.

2.2.2. Deep Learning

Dentro del *Machine Learning* y usualmente relacionado con el aprendizaje supervisado, nos encontramos con un sub-campo denominado **Deep Learning** que utiliza las redes neuronales para la creación de modelos.

Como su nombre indica las redes neuronales consisten en unidades de cómputo llamadas neuronas que están interconectadas entre sí. Una neurona es una unidad de cómputo que posee múltiples entradas y una salida, esta neurona multiplica cada entrada por un peso para posteriormente realizar una suma y, por último, aplicar una función de salida no lineal. Si los pesos se establecen correctamente y tenemos un número suficiente de neuronas, una red neuronal puede aproximar a un conjunto muy amplio de funciones matemáticas.

En las redes neuronales, las neuronas suelen organizarse por capas que se encuentran conectadas entre sí. Mientras más capas tengamos, más características podremos extraer de nuestros datos de entrada y podremos aproximar un mayor número de funciones (sin perder de vista el sobrentrenamiento).

El primero y más simple de los tipos de redes neuronales es el denominado *Feed Forward Neural Network* (**FFNN**), este tipo de redes recibe este nombre porque no existen ciclos entre sus neuronas y las conexiones se realizan siempre desde las capas anteriores a las capas posteriores.

Una de las arquitectura más comunes de FFNN es el preceptrón multicapa (en inglés multilayer perceptron o **MLP**). Esta arquitectura contiene tres o más capas de neuronas totalmente conectadas, es decir, la salida de una neurona de una capa se encuentra conectada a la entrada de todas las neuronas de la siguiente capa. Las capas de una arquitectura MLP son:

- **Capa de entrada**: Se trata de la capa en la que introduciremos los datos en la red. Esta

capa carece de procesamiento.

- **Capas ocultas**: Son las capas intermedias, cuyo número puede variar, tienen como entrada la salida de las neuronas de la capa anterior y su salida alimenta a las neuronas de la capa posterior.
- **Capa de salida**: Los valores de salida de las neuronas de esta capa se corresponden con la salida de la red.

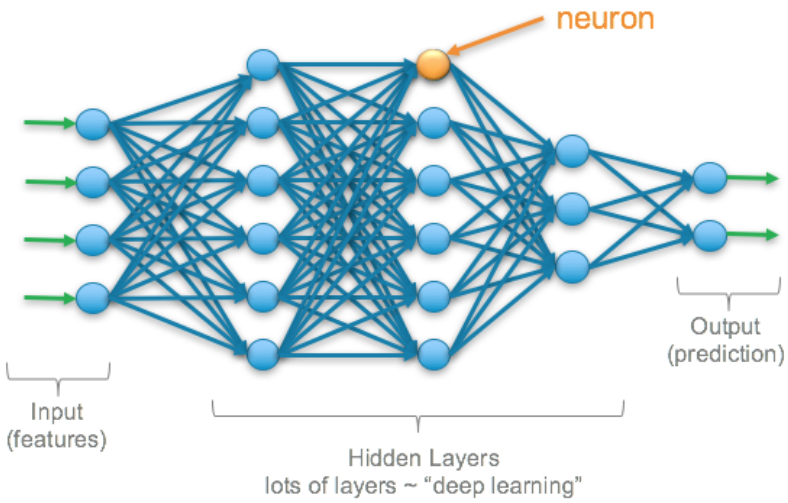

Figura 2.1: Ejemplo de arquitectura MLP. Fuente [22]

Hablamos que una red es profunda cuando contiene un gran número de capas, por ello el término de *Deep Learning*. En la figura 2.1 observamos un ejemplo de arquitectura MLP y como la denominamos *Deep Learning* al crecer el número de capas ocultas.

2.2.3. Representación de palabras en PLN

Es usual, en el ámbito del reconocimiento de imágenes, utilizar información acerca de la dimensionalidad de las mismas. Este tipo de información nos permite extraer características teniendo en cuenta los píxeles vecinos. Tradicionalmente, en el ámbito del Procesamiento del Lenguaje Natural, esto no se ha llevado a cabo debido a que cada palabra (o n-grama) se trataba como una entidad aislada utilizando una codificación de las palabra denominada **one-hot encoding**.

En cambio, existe otro método de representar las palabras en el lenguaje natural que sí es capaz de captar la "dimensionalidad" de una forma similar a como lo realizamos en las imágenes. Este modo, conocido como ***word embedding***, deja de tratar la palabra como un

ente aislado y es capaz de captar el significado de la misma, esta representación se denomina distribuida y consiste en convertir las palabras en vectores en los que cada dimensión capte características diferentes de las palabras. Este tipo de representaciones dará lugar a vectores similares para palabras semánticamente parecidas.

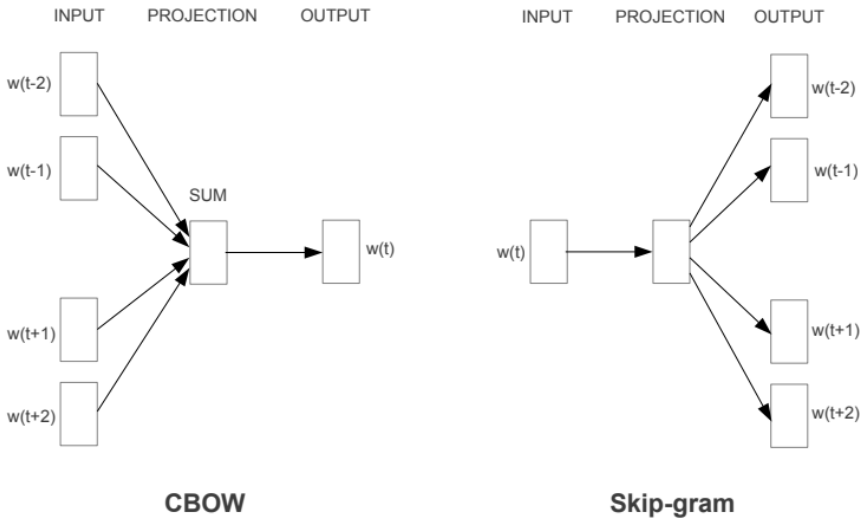

Figura 2.2: Arquitecturas CBOW y Skip-gram. Fuente [18]

Una de las soluciones más populares que nos permiten convertir una palabra a un vector (*word2vec*) que contenga información de la palabra en función del contexto se detallan en el artículo [18] . Aquí se presentaron dos modelos llamados **Skip-Gram** y **CBOW** cuya arquitectura podemos ver en la figura 2.2. Estos modelos utilizan redes neuronales para predecir una palabra en función de su contexto o el contexto en función de una palabra, el vector que se utiliza para representar la palabra es el vector de pesos de la capa oculta.

2.2.4. Arquitecturas especializadas

Después de introducir las redes neuronales y el modo en el que podemos representar las palabras, frases o documentos para ser usados como entrada en nuestro modelo; vamos a centrarnos en comentar dos tipos de redes neuronales que se usan de manera tradicional en tareas de Procesamiento de Lenguaje Natural.

Los dos tipos de redes neuronales que comentaremos son las redes neuronales convolucionales y las redes neuronales recurrentes. Haremos una introducción a cada una de ellas, comentaremos sus aplicaciones al PLN y sus ventajas y convenientes con respecto a otro tipo de métodos.

2.2.4.1. Redes neuronales convolucionales

Las redes neuronales convolucionales (llamadas usualmente CNN, por su nombre en inglés ***C**onvolutional **N**eural **N**etworks*) son un tipo de redes neuronales que deben su nombre a la operación matemática de convolución que realizan. Esta operación consiste en aplicar a una matriz de entrada multidimensional, un filtro o kernel también multidimensional y obtener una salida, también denominada mapa de características. En la figura 2.3 podemos ver una representación gráfica de esta operación para un ejemplo de 2 dimensiones.

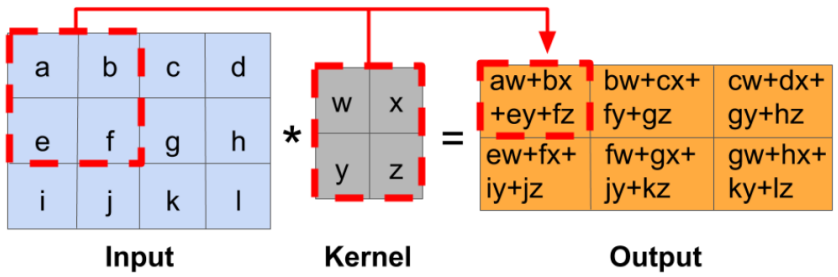

Figura 2.3: Ejemplo de una convolución de dos dimensiones. Fuente [1]

Es usual en las redes convolucionales utilizar diferentes kernels sobre una misma entrada, obteniendo diferentes salidas que permitan reconocer distintos patrones. Los pesos del kernel junto con el sesgo son los parámetros que serán necesarios calcular en el entrenamiento y en el caso de las redes convolucionales se denominan mapas de características.

Aunque la convolución simple que comentamos es la operación básica en las redes convolucionales es usual añadirle algunas variantes (o configurarla con algunos parámetros) que nos permitan variar la dimensión de salida una vez hemos aplicado la convolución, algunas de estas variaciones más comunes son el ***zero padding*** y la **convolución por pasos**.

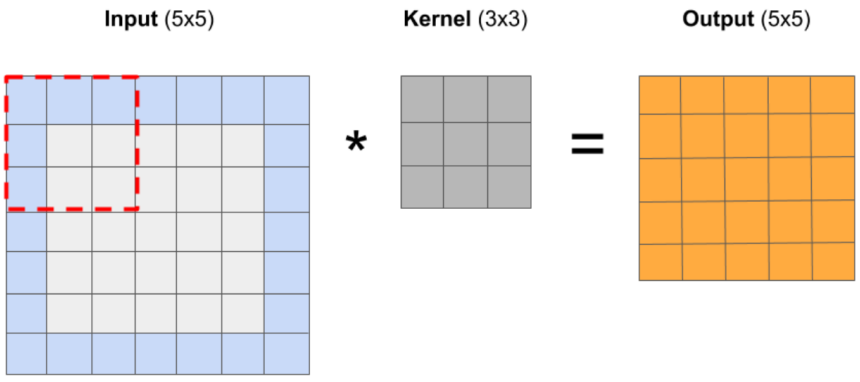

Figura 2.4: Ejemplo de aplicación de *zero padding* para mantener la dimensionalidad. Fuente [1]

Como observamos en la figura 2.3, al aplicar el kernel a los datos de entrada estamos reduciendo la dimensionalidad de la salida, a menudo es posible que esto no nos interese y queramos mantener la dimensionalidad en la salida; para ello recurrimos a un método denominado *zero padding* que consiste en añadir '0s' en los bordes de nuestra entrada con el objetivo de preservar la dimensionalidad en la salida. En la figura 2.4 podemos ver un ejemplo de aplicación de *zero padding*.

Por otro lado, es también posible que queramos reducir aún más la dimensión de salida, principalmente por un tema de eficiencia y reducción de los tiempos de ejecución, a costa de perder información de algunas características en la salida. Para ello podemos utilizar la convolución por pasos (o *strided* por su nombre en inglés), este método consiste en aplicar el kernel realizando saltos en lugar de hacerlo sobre celdas consecutivas, tal y como podemos ver en la figura 2.5.

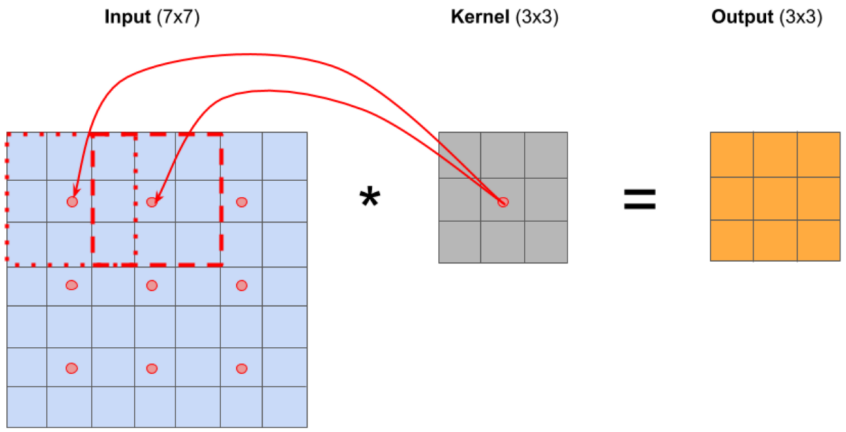

Figura 2.5: Ejemplo de aplicación de convolución por pasos para reducir la dimensionalidad. Fuente [1]

El proceso explicado anteriormente correspondería con una capa convolucional, que son el corazón de las redes neuronales convolucionales, sin embargo, en una red neuronal convolucional estas capas coexisten con otro tipo de capas que nos ayudaran a mejorar nuestros modelos. Las más usuales son:

- Capa de agrupamiento (*polling* en inglés): El objetivo de esta capa es agrupar un conjunto de salidas para obtener un único valor, al conjunto de valores de entrada (seleccionado de nuevo con un filtro) se le aplica una función para obtener un único valor. Aunque se pueden utilizar diferentes funciones, como puede ser la media, lo más usual es aplicar la función de máximo (*max-polling*). Es habitual, intuir que usando esta función nos estamos quedando con las características más relevantes de cada cuadrante (del tamaño del filtro)

de la entrada. En la figura 2.6 podemos ver un ejemplo de agrupamiento utilizando la función de máximo.

- Capa totalmente conectada: Hemos visto ejemplos de capas totalmente conectada al introducir las redes neuronales, esta capas usualmente se usan al final de nuestra red para tareas de clasificación, teniendo la última capa un número de neuronas igual al número de clases que pretendemos clasificar.

- Capa RELU: Si observamos la descripción de la operación de convolución nos damos cuenta de que se trata de una operación totalmente lineal, es por ello que después de cada capa de convolución es usual agregar una capa no lineal (también llamada capa de activación). Aunque se pueden utilizar otras funciones como la tangente o la función sigmoide, lo más usual es utilizar la función RELU.

- Capa de Dropout: Esta capa tiene como funcionalidad prevenir el sobreentrenamiento en las redes neuronales, desactivando un número aleatorio de entradas de la capa forzando a la red a ser redundante y permitir dar una clasificación correcta sin tener todas las entradas activas.

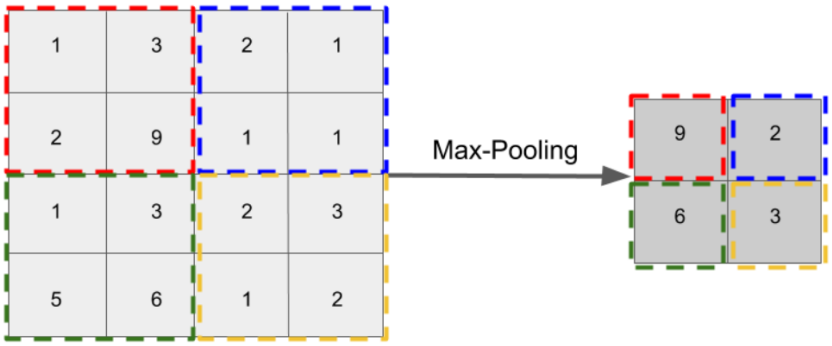

Figura 2.6: Ejemplo de aplicación de *max-pooling*. Fuente [1]

Tras esta visión general sobre las redes neuronales generales podemos enumerar las ventajas que conllevan:

- Por un lado, aunque no hemos entrado en detalles sobre el proceso de entrenamiento de las redes convolucionales, se puede intuir que el aplicar un mismo kernel sobre toda la entrada provoca que el número de parámetros a aprender (los valores del kernel) con respecto a una red totalmente conectada será mucho menor. Esto provoca una **reducción del tiempo de entrenamiento necesario**.

- Por otro lado, el hecho de compartir el kernel provoca que podamos **capturar una misma característica en la entrada a pesar de su traslación**. Por ejemplo, si estamos detectando un objeto en una imagen un modelo convolucional podrá detectar ese objeto a pesar de su movimiento por la imagen.

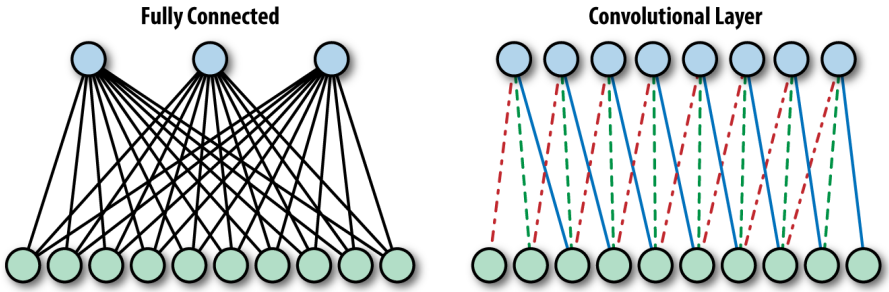

Figura 2.7: Comparación capa tradicional totalmente conectada con capa convolucional. Fuente [13]

Sin embargo, las redes neuronales convolucionales deben usarse en datos que contengan coherencia local ya que esa es su fortaleza. **En datos sin coherencia local las redes neuronales convolucionales no lograrían obtener un buen rendimiento** como las redes neuronales tradicionales vistas anteriormente. Si observamos la figura 2.7 podemos ver la diferencia entre las capas de ambas tipos de redes y como la capa convolucional se centra más en las estructuras locales.

Encontramos una explicación más profunda sobre las redes convolucionales y su uso general en [1]. Aunque, como podemos imaginar, el uso más extendido de este tipo de redes es para el tratamiento de imágenes, nosotros nos centraremos en tener una breve visión de su aplicación al Procesamiento del Lenguaje Natural, que también puede ampliarse en [12].

Al aplicar las redes tradicionales al PLN solemos ignorar el orden en el que las palabras aparecen en las frases, o las frases en el documento, siguiendo una aproximación CBOW; esto suele ser problemático a la hora de realizar, por ejemplo, un análisis de sentimientos ya que no es lo mismo encontrar la palabra “malo” aislada que el bigrama “no malo”. Aunque el uso de bi-gramas y N-gramas de mayor orden puede mejorar esta situación el coste puede volverse inasumible.

Es en este ámbito dónde las redes neuronales convolucionales pueden ser de gran ayuda, ya que gracias a la capacidad comentada para detectar estructuras locales serían capaces de identificar estos N-gramas de forma automática para ser usados posteriormente en tareas predictivas.

2.2.4.2. Redes neuronales recurrentes

Otro de los modelos usualmente usados en tareas de Procesamiento del Lenguaje Natural son las redes neuronales recurrentes, (llamadas usualmente RNN, por su nombre en inglés ***R**ecurrent **N**eural **N**etworks*). Hasta ahora todos los modelos de redes neuronales que hemos citado funcionaban siempre en una dirección, las neuronas de una capa anterior producían una salida que era la encargada de activar las neuronas de la capa posterior; en las redes recurrentes veremos que las salidas de una neurona en una capa posterior pueden tener una conexión con una neurona de una capa anterior. Esto crea una especie de **memoria** que nos permite modificar la respuesta de la red en función de los datos que se hayan procesado anteriormente (incluso reaccionar con datos que lleguen posteriormente).

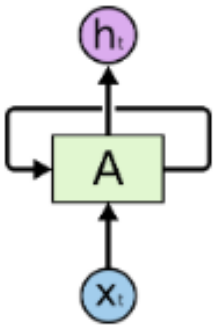

Figura 2.8: Ejemplo RNN. Fuente [21]

Las conexiones en una red neuronal recurrente pueden tener muchas variaciones por lo que es usual hablar del concepto de **celda**. Una celda suele tener como entrada los valores de la secuencia y el estado de la red neuronal en el paso anterior; y como salida la respuesta de la red neuronal a dicha entrada y el estado de la red neuronal en el paso actual.

En la figura 2.8 observamos un ejemplo de red neuronal recurrente en el que tenemos como entrada el valor de la secuencia x_t y la el estado de la red en el paso anterior (conexión en bucle); y producimos una respuesta h_t y un estado (conexión en bucle). Sin embargo posiblemente esta representación sea algo más confusa para comprender su funcionamiento que si procedemos a "desenrollar" la red neuronal haciéndola más similar a los modelos vistos hasta ahora, en la figura 2.9 podemos ver el resultado de "desenrollar" la red; hay que tener en cuenta con esta representación que los parámetros usados por cada celda son exactamente los mismos.

Aunque no entraremos en detalles sobre el entrenamiento de redes neuronales es importante saber que existen dos problemas diferentes provocados ambos por usar los mismos parámetros en todas las celdas que provocan la inestabilidad durante el proceso de entrenamiento. Estos problemas son la **desaparición del gradiente**, que ocurre al multiplicar el gradiente consigo mismo múltiples veces cuando este es menor que 1, y la **explosión del gradiente**, que ocurre

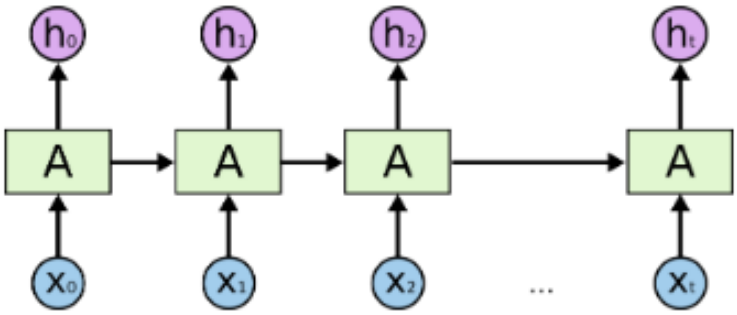

Figura 2.9: Ejemplo RNN "desenrollada". Fuente [21]

por el mismo motivo cuando este es mayor que 1.

Para mitigar estos problemas es importante el diseño de las celdas, a continuación veremos de manera resumida los dos tipos de celdas más usados en las redes neuronales recurrentes. Las celdas que comentaremos están compuestas por diferentes mecanismos internos, denominados puertas, que gestionan el flujo de información a lo largo de la misma.

El primer tipo de celdas son las celdas Long Short Term Memory (**LSTM**). Este tipo de celdas se comportan bien en situaciones que queremos encontrar patrones entre registros que se encuentran separados en la secuencia, esto es algo muy usual, por ejemplo, en el caso de PLN cuando en una misma frase una palabra hace referencia a otra que apareció a una distancia de varias palabras.

Para conseguir este objetivo, parece evidente que es importante controlar la memoria en cada una de las celdas. Una celda LSTM realiza esta tarea con las siguientes puertas:

- Puerta de entrada: Controla que información se añade a la memoria de la red.
- Puerta de olvido: Controla, a partir de la memoria del paso anterior y de la entrada, que información debe conservarse en la memoria.
- Puerta de salida: Es la encargada de calcular la salida de la red, h_t, en el paso actual.

Debido al éxito de las celdas LSTM, y al constante esfuerzo de optimización de las mismas, han surgido diferentes variantes. La más conocida de todas son las Gated Recurrent Unit (**GRU**) introducidas en 2014. La celda GRU es una simplificación de la celda LSTM y produce unos rendimientos bastante similares con un menor coste. De manera muy resumida una celda GRU se compone de:

- Puerta de reset: Permite seleccionar que información de la memoria va a ser utilizada en un paso concreto.

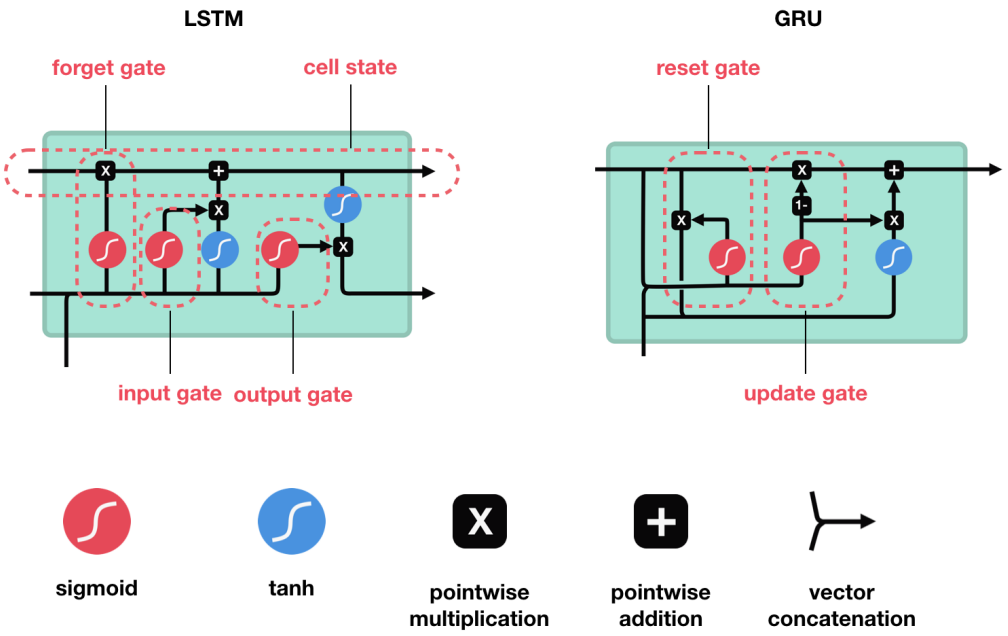

Figura 2.10: Arquitectura celdas LSTM y GRU. Fuente [20]

- Puerta de actualización: Realiza la función de las puertas de olvido y de entrada que hemos visto en las celdas LSTM.

Aunque no hemos entrado en el detalle del funcionamiento de cada una de las puertas, en la figura 2.10 podemos ver la arquitectura completa de ambos tipos, y observar la mayor simplicidad de las celdas GRU frente a las LSTM.

Como podemos imaginar las redes neuronales recurrentes son ampliamente usadas en el mundo del Procesamiento del Lenguaje Natural, debido a que una palabra no es otra cosa que una secuencia de letras, una frase a su vez se trata de una secuencia de palabras y un documento una secuencia de frases. Alguno de los usos de las RNN en este ámbito son:

- **Análisis de sentimientos**: Detectar el sentimiento positivo o negativo de un texto utilizando únicamente la última salida de la red.
- **Generador de texto**: Predecir la siguiente palabra de una secuencia, utilizando la salida de cada una de las celdas.
- **Traductores**: Traducir textos entre idiomas, en este caso es usual utilizar redes recurrentes bidirecionales.

Una vez analizadas, de manera global, las redes neuronales recurrentes podemos ver que nos ofrecen **numerosas ventajas al trabajar con secuencias** como hacemos en el ámbito

del PLN. Entre ellas podemos ver, que vamos incluso más haya que con las CNN analizando la relación entre las diferentes palabras, fundamentalmente con las celdas LSTM y GRU, pudiendo detectarlas entre palabras que estén alejadas entre sí y no ceñirnos al tamaño del kernel. Sin embargo, como hemos visto, y aunque sean mitigados por el uso de celdas LSTM y GRU, las redes neuronales recurrentes presentan **problemas durante el entrenamiento** tanto de desvanecimiento como de explosión del gradiente.

2.3. BigData y Fast Data

A lo largo de esta sección intentaremos tener una visión general del *Big Data* y su evolución a lo largo del tiempo hasta llegar al *Fast Data*. Posteriormente veremos las arquitecturas más usadas en el mundo del *Big Data*.

2.3.1. Evolución: del Big Data al Fast Data

El primer uso del término *Big Data* se da en un artículo de Michael Cox y David Ellsworth de la NASA publicado en 1997 [7], donde hacen referencia a la dificultad de procesar grandes volúmenes de datos con los métodos de la época. Sin embargo, fue en 2001 cuando encontramos la definición más conocida y aceptada de *Big Data* hecha por el analista Laney Douglas en su artículo "3D Data Management: Controlling Data Volume, Velocity y Variety" [15] en el que se hacía referencia a las ya "famosas" tres *V*s:

- **V**olumen: Cada vez los volúmenes de datos son mayores.
- **V**elocidad: Es cada vez mayor la velocidad con la que se generan los datos.
- **V**ariedad: Dejamos de tener únicamente datos completamente estructurados para trabajar con datos no estructurados y/o semi-estructurados.

Google, como es obvio, también se enfrentó a un importante problema a la hora de procesar la ingente cantidad de datos que generaba día a día y que no podían ser procesados de manera eficiente con el *software* existente, es por ello que en el año 2003 presenta en [11] su *"Google File System"* (GFS) y un año después *Map Reduce* [8], estas dos capas de almacenamiento y procesamiento distribuido dieron lugar al nacimiento de lo que hoy conocemos como ***Big Data***.

Sin embargo, estas aportaciones no empezaron a tomar una repercusión relevante fuera de Google hasta el nacimiento del *framework* Hadoop en 2006, un ecosistema con una gran cantidad de servicios pero cuya base fue Map Reduce y HDFS (basado en GFS). La complejidad del ecosistema *Hadoop* hizo que éste no empezara a aparecer en las mayoría de las empresas hasta la

creación de la compañía *Cloudera* en 2009, que empezó a empaquetar los diferentes componentes del ecosistema *Hadoop*, ofreciendo distribuciones estables y soporte para sus clientes.

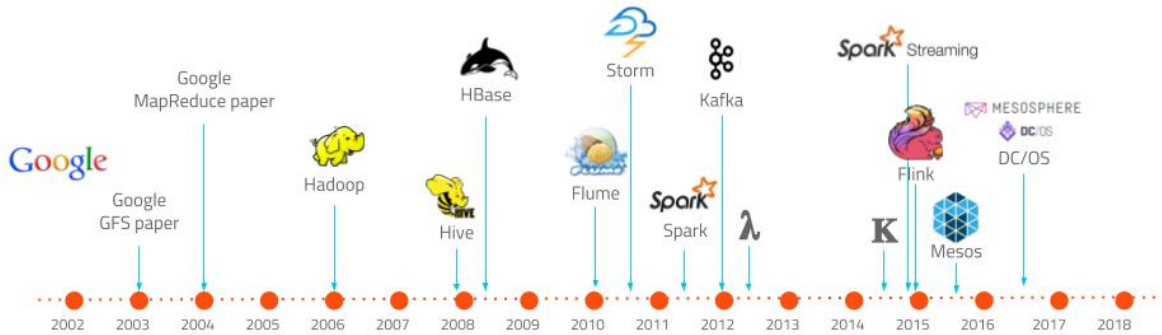

Figura 2.11: Evolución del *Big Data*. Fuente [9]

Durante estos 10 años la popularidad de *Hadoop* ha crecido exponencialmente y junto con las BBDD NoSQL, nacidas también a partir de Google con su BigTable, forman lo que hoy conocemos como Big Data. En la figura 2.11 observamos esta evolución en el mundo del *Big Data* con los hitos de aparición de algunas tecnologías representativas.

El auge del **Big Data** ha llevado a algunas empresas a tener verdadera obsesión por el almacenamiento de todos los datos de sus clientes y las operaciones realizadas, creando inmensos *datalakes* donde tener enormes históricos de todos sus datos. Este "síndrome de Diógenes digital" creado por falsas expectativas, por la imposibilidad de extraer valor de los datos o por la dimensión cambiante de las empresas actuales (en la que los datos de años atrás pueden no ser relevantes en el presente) es uno de los posibles motivos por lo que el tratamiento de los datos esta cambiando. Otro de los motivos para el cambio de rumbo del *Big Data* está relacionado con la *V* de Velocidad, hoy en día no solo es importante la capacidad de ingestar rápidamente los datos, sino la capacidad de poder procesar y obtener decisiones o actuar en tiempo real a partir de los datos, aportando valor al negocio. En este escenario se vuelve más importante la velocidad que el volumen de datos, esto es lo que se denomina *Fast Data*.

Dentro del *Fast Data* es habitual el uso de BBDD *in-memory*, de buses de eventos y de tecnologías de procesamiento capaces de procesar los eventos en tiempo real. Como veremos posteriormente al desarrollar nuestra arquitectura, el *Fast Data* será una parte fundamental en nuestro proyecto en el que tendremos que clasificar las llamadas en tiempo real y tomar decisiones (o alarmar) en función de las mismas.

Observando de nuevo la figura 2.11 podemos ver este cambio en la tendencia hacia el *Fast Data* a partir del 2012 cuando aparece la tecnología Kafka y en los años posteriores con la incorporación de diferentes herramientas para el procesamiento de eventos como son *Spark Streaming* o *Flink*.

2.3.2. Arquitecturas *RealTime*

La evolución que hemos visto en el apartado anterior, con la explosión del *Big Data* y la irrupción del *Fast Data*, hace necesaria la incorporación en las empresas de arquitecturas de procesamiento de datos en tiempo real que, como cualquier otra arquitectura de datos, sean capaces de ingestar, procesar y permitir la explotación y análisis de los datos. La diferencia fundamental en las arquitecturas *RealTime* y las arquitecturas de datos tradicionales son el **volumen de los datos a tratar** y la **capacidad para hacerlo en tiempo real**.

Como veremos, no existe una arquitectura que se adapte a todos los casos de uso (*one-size-fits-all*) y, según la necesidad, será necesario aplicar una u otra.

2.3.2.1. Arquitectura Lambda λ

La arquitectura lambda, representada por la letra griega λ, fue presentada en 2011 por Nathan Marz en un artículo publicado en su blog titulado "*How to beat the CAP theorem*" [17].

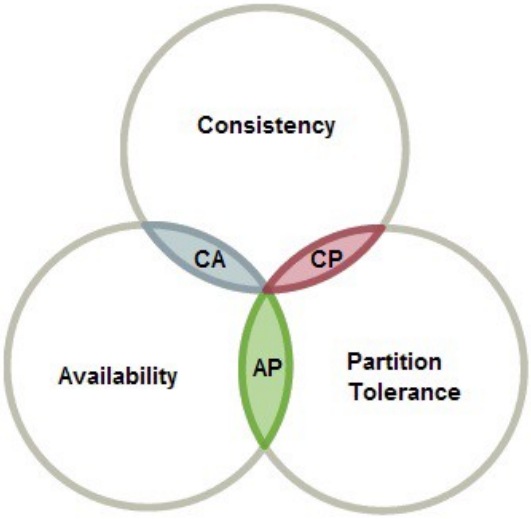

Figura 2.12: Teorema CAP. Fuente [19]

El propósito de Nathan Marz cuando crea su arquitectura, como indica el título del artículo, es batir el teorema CAP popularizado por la irrupción de las bases de datos NoSQL. Este teorema, ilustrado en la figura 2.12, viene a decir que si queremos tener tolerancia a particiones (imprescindible para bases de datos distribuidas necesarias para el *Big Data*) tenemos que optar entre consistencia, asegurar que el dato que leemos es el último que hemos escrito, o disponibilidad, que la base de datos se encuentra siempre lista.

El método propuesto para conseguir este objetivos con grandes cantidades de datos se basa en los siguientes principios:

- Una capa *batch* eventualmente consistente de una manera extrema, en la que las escrituras tardan siempre unas pocas horas en estar disponibles. Eliminando algunos problemas complejos con los que tratar como la concurrencia o las reparaciones de lectura.

- Reducir las operaciones CRUD (***C****reated,* ***R****ead,* ***U****pdate,* ***D****elete*) en la capa *batch* por únicamente CR, tratando los datos como objetos inmutables. Esto nos soluciona el problema de la consistencia, ya que de este modo un dato existe o no existe, pero no puede tener varias versiones.

- Una capa *realtime* que se encarga de los datos de las últimas horas (los que no están disponibles en la capa *batch*)

- Las querys atacan a ambas capas de forma simultanea realizando un *merge* de los datos.

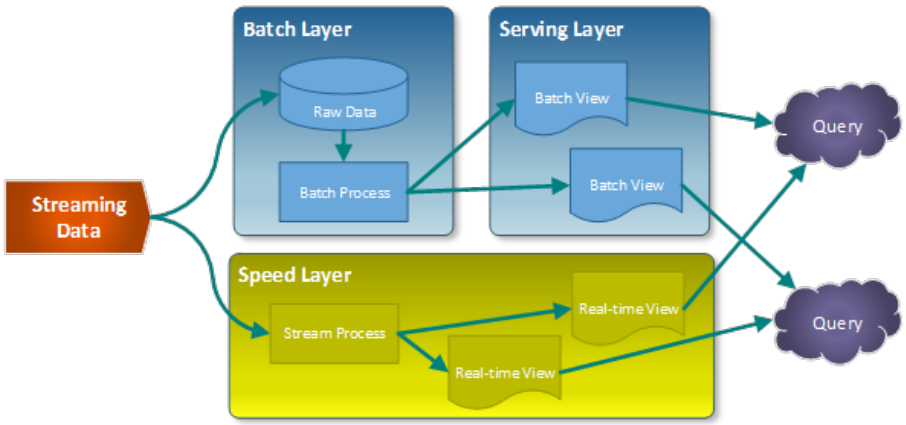

Figura 2.13: Arquitectura Lambda definida por Nathan Marz. Fuente [9]

Podemos ver un esquema de la arquitectura en la figura 3.1. Esta arquitectura, aparte de resolver los problemas de consistencia y disponibilidad, tiene algunas ventajas:

- Disponer de todos los datos en un único punto (capa *batch*) pudiendo realizar cualquier tipo de consulta sobre los mismos.

- Al utilizar los datos como un ente inmutable facilita las auditorias.

- Según Marz, evita el error humano en la capa *batch* (en parte también por usar datos inmutables), y cualquier error en la capa *realtime* sería subsanado en pocas horas en la capa *batch*.

2.3.2.2. Arquitectura Kappa κ

En su artículo "Questioning the Lambda Architecture"[14], Jay Kreps cuestiona la arquitectura Lambda propuesta por Nathan Marz y propone una simplificación de la misma, basada en su experiencia en LinkedIn trabajando con Kafka y Samza. Esta simplificación se denomina arquitectura Kappa y viene representada por la letra griega κ.

Kreps describe la complejidad que supone en una arquitectura Lambda mantener idénticos procesos en *realtime* y *batch*. También expone que se menosprecia la capacidad de la capa *realtime* (probablemente por la madurez del procesamiento *realtime* con respecto al *bacth*) y opina que es posible realizar el mismo procesamiento, incluso reprocesar el histórico de datos, en la capa *realtime*.

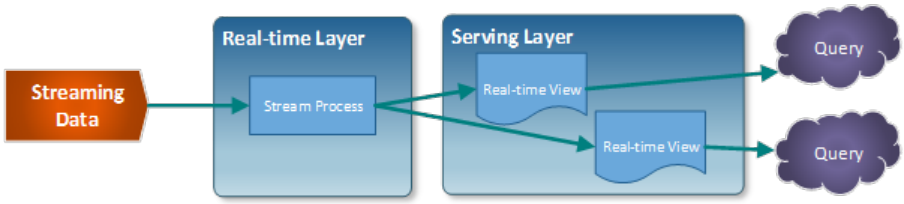

Figura 2.14: Arquitectura Kappa definida por Jay Kreps. Fuente [9]

Con esta premisa, en el artículo se presenta la arquitectura que observamos en la figura 2.14, en la que existe un único flujo de procesamiento *realtime* para todo el modelo. La simplicidad de Kappa con respecto a Lambda es tal que el propio Kreps afirma que puede ser una idea demasiado simple para merecer una letra griega.

Capítulo 3

Arquitectura y tecnologías

El objetivo de este capítulo es tener un diseño esquemático de la solución planteada, este diseño será abordado en primer lugar desde un punto de vista lógico, de acuerdo a las necesidades del proyecto, posteriormente aterrizaremos esta arquitectura con tecnologías concretas que nos ayuden a conseguir el objetivo deseado.

3.1. Arquitectura

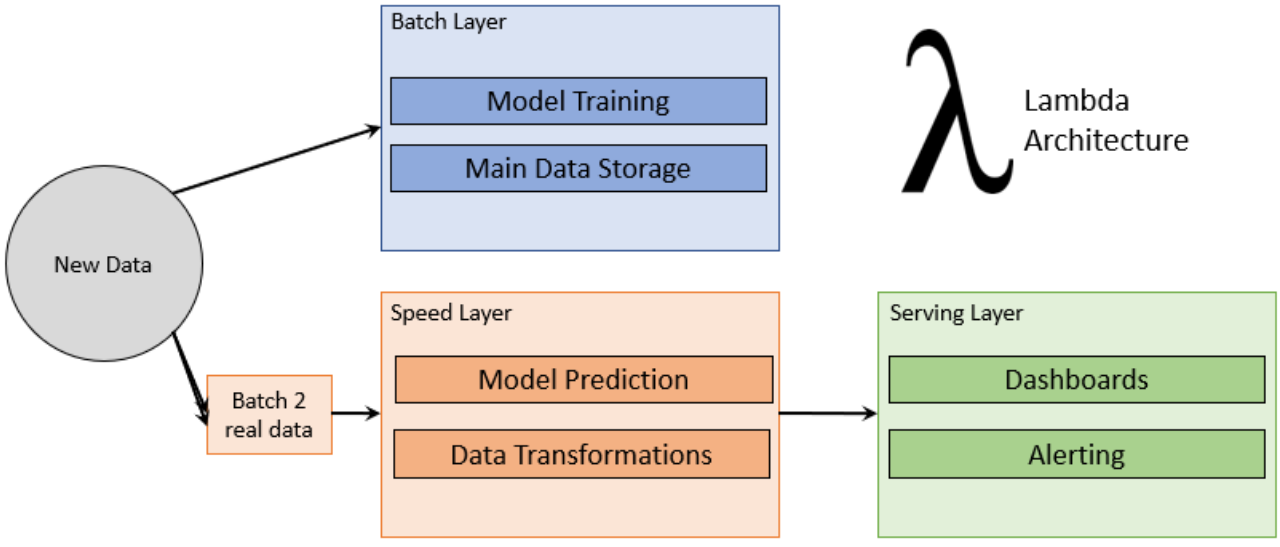

Figura 3.1: Arquitectura Lambda propuesta

En este apartado definiremos la arquitectura desde un punto de vista lógico, esta arquitectura debe responder a los objetivos propuestos para cumplir con las necesidades de negocio

existentes.

En líneas generales, podemos ver la arquitectura de nuestro sistema como una arquitectura *Lambda* en la que disponemos de una capa *batch*, una capa rápida o *real-time* y una última capa de servicio.

La capa batch será la encargada de entrenar el modelo a través de los datos de las llamadas, la capa rápida obtendrá los *topics* de las llamadas en tiempo real usando el modelo previamente entrenado y por último la capa de servicio será la encargada de mostrar estos datos al usuario mediante cuadros de mando.

En la Figura 3.1 podemos ver un esquema general de nuestra arquitectura. A continuación definimos con más detalle cada una de las capas del modelo.

3.1.1. Capa Batch

El *core* del proyecto que abordamos es el modelo, encargado de extraer los *topics* de las transcripciones de llamadas al servicio de atención al cliente. Este modelo debe entrenarse usando un histórico suficientemente amplio.

El modelo es un elemento vivo en nuestra arquitectura y, además de por posibles mejoras en los hiperparámetros o por la tecnología, debe re-entrenarse conforme se vayan recibiendo datos nuevos en el histórico, ya que es lógico pensar que la temática de las consultas variarán a lo largo del tiempo debido por ejemplo al lanzamiento de nuevos productos.

3.1.2. Capa Real-Time

La *speed layer* de nuestro proyecto será la encargada de recibir los datos en tiempo real, las llamadas serán publicadas en un bus de eventos y estos eventos serán consumidos por una capa de procesamiento que será la encargada de aplicar el modelo entrenado en la capa *batch* a los nuevos datos. Los *topics* resultantes de cada llamada serán publicados de nuevo en este bus para poder ser ingestados posteriormente a una BBDD NO-SQL, que será la encargada de proporcionar la información a la capa de servicio.

Aprovecharemos también las características de la BBDD NoSQL para, mediante un módulo de *Machine Learning*, detectar anomalías en series temporales y poder alarmar en el caso de que un *topic* concreto se dispare en algún momento.

Debido a la situación actual, las llamadas no se ingestan en *real-time* si no que se procesan en mini *batches* cada cierto tiempo. Es muy probable que este escenario cambie en el futuro por lo que se construirá un elemento de entrada a la capa rápida que transformará el mini-batch en eventos conforme vayan ejecutándose (este elemento puede observarse en la figura 3.1 como *Batch 2 real data*). Esta pieza será suprimida una vez las llamadas sean recibidas en tiempo

real.

3.1.3. Capa de Servicio

Una vez almacenados los datos en la BBDD debemos construir un frontal que muestre al usuario el número de llamadas analizadas, el modelo utilizado y principalmente la evolución de los *topics* a lo largo del tiempo.

Lo ideal es que esta capa de servicio se vaya actualizando en tiempo cuasi-real y permita a los diferentes usuarios realizar cuadros de mando personalizados según sus necesidades.

Otro requisito esencial en este tipo de proyectos es que la información este disponible vía API-REST para poder ser consumida por terceros.

3.2. Integración y Despliegue Continuos

Como ya hemos visto al hablar del entrenamiento del modelo, el desarrollo de este tipo de proyectos no tiene un principio y un final, si no que se trata de un proceso cíclico en el que por necesidades del negocio, por cambio en los datos o por cambio en la tecnologías, será necesario añadir mejoras o modificaciones en nuestro desarrollo.

Por estos motivos es conveniente definir mecanismos que nos permitan, tras cada cambio efectuado, poder realizar las pruebas necesarias y desplegar estos cambios de una manera totalmente automatizada y sin intervención del usuario.

Aunque este apartado queda fuera de la implementación inicial del proyecto es imprescindible llevarlo a cabo para que este sea sostenible a lo largo del tiempo.

TODO (Posible ampliar con apuntes sobre el ciclo de vida de los datos y contar beneficios de un despliegue continuo)

3.3. Tecnologías

Al igual que la arquitectura descrita anteriormente era la encargada de responder a las necesidades de negocio, las tecnologías descritas en este apartado nos darán las piezas necesarias para poder construir esa arquitectura y dar respuesta a nuestro caso de uso.

En el proceso de selección de las tecnologías, no solo se ha tenido en cuenta la idoneidad de las mismas para el caso de uso, si no que se ha valorado también la experiencia en la misma y la disponibilidad dentro del entorno de trabajo. Esto puede provocar que en algunos casos aunque la tecnología se adapte al caso de uso, puedan existir otras soluciones más óptimas cuyo uso era menos viable dado los plazos de ejecución del proyecto.

A continuación enumeraremos las tecnologías agrupadas en las diferentes capas que hemos comentado en el apartado de arquitectura, además añadiremos las tecnologías que se usarán para la integración y despliegue continuo.

3.3.1. Capa batch

- **Spark**: Es un framework de computación en clúster. Este *framework* se encuentra desplegado sobre un clúster Hadoop, específicamente una distribución HortonWorks, y posee librerías específicas para Machine Learning como MLLIB.

- **Tensorflow** sobre GPUs: Para el entrenamiento de los modelos también disponemos de un cluster con GPUs sobre el que podremos correr código usando la librería de Google Tensorflow para entrenar nuestros modelos.

3.3.2. Capa Real-Time

- **Kafka**: Es el *core* de la capa rápida, se trata de un bus de evento distribuido a través del cual se realizara la ingesta o publicación de los eventos (llamadas). Las diferentes capas de procesamiento que requieran estos eventos se suscribirán a este Bus.

- **Microservicios**: En nuestra capa Real Time construiremos diferentes microservicios en la capa de procesamiento, estos microservicios serán por ejemplo los encargados de devolver los *topics* de cada llamada a partir de una llamada API REST. Estos microservicios correran sobre contenedores en una plataforma Openshift.

- **Kafka Stream y KSQL**: A la hora de procesar la información en eventos ingestada en nuestro Bus Kafka disponemos de dos herramientas muy potentes que son Kafka Stream y KSQL. El primero consiste en una serie de librerías que nos permiten construir aplicaciones y microservicios cuyo origen y destino sean un Bus Kafka. KSQL en cambio es un motor SQL aplicable a eventos que nos llegan mediante *streaming*.

- **Logstash**: Una vez hayamos extraído los *topics* correspondientes a cada llamada o evento, tendremos que ingestar esta información en nuestra BBDD, que en este caso será Elasticsearch. Logstash nos permitirá leer de Kafka, realizar las transformaciones necesarias y volcar la información resultante en Elasticsearch.

- **Elasticsearch**: Aunque no se trata en el sentido más estricto de una BBDD No-SQL, si no de un motor de búsqueda, Elasticsearch nos permite almacenar la información en forma de documentos json y realizar consultas y agregaciones sobre cualquier campo.

Entre las características que podemos aprovechar de Elasticsearch para nuestro objetivo están:

- Ingesta en tiempo casi real.
- Consulta en tiempo casi real.
- Disponibilidad de mecanismos de ingesta (Logstash) y consulta (kibana).
- Posibilidad de crear alarmas en base a consultas.
- Posibilidad de crear *jobs* de Machine Learning que detecten anomalías en series temporales.

3.3.3. Capa Servicio

- **Elasticsearch API REST**: Toda la información almacenada previamente en Elasticsearch puede ser accedida a través de su API REST por lo que será nuestro método de publicación de la información.
- **Kibana**: Será el frontal donde los usuarios podrán consultar sus diferentes cuadros de mando y construir nuevos de acuerdo a sus necesidades. También, gracias al modulo de *machine learning* de Elasticsearch, los usuarios podrán crear *jobs* de *machine learning* para detectar anomalías en los temas tratados y generar las alertas necesarias.

3.3.4. Integración y Despliegue Continuo

Aunque la implementación de esta parte se escapa de los plazos del proyecto, las tecnologías que se usarán para llevarla a cabo serán.

- **BitBucket**: Será el repositorio usado para almacenar las nuevas versiones de nuestro *software* de manera que podamos tener un control de versiones.
- **Jenkins**: Es un servidor de integración continua *open source* que mediante la creación de tareas nos ayudará a realizar el *build* de nuestro software realizando de manera automática las pruebas necesarias.

Capítulo 4

Conjunto de datos

Bibliografía

[1] Toni Lozano Bagén Anna Bosch Rué, Jordi Casas Roma. *Deep Learning Principios y Fundamentos*. 2018.

[2] David M. Blei, Michael I. Jordan, Thomas L. Griffiths, and Joshua B. Tenenbaum. Hierarchical topic models and the nested chinese restaurant process. In *Proceedings of the 16th International Conference on Neural Information Processing Systems*, NIPS'03, pages 17–24, Cambridge, MA, USA, 2003. MIT Press.

[3] David M. Blei and John D. Lafferty. Correlated topic models. In *Proceedings of the 18th International Conference on Neural Information Processing Systems*, NIPS'05, pages 147–154, Cambridge, MA, USA, 2005. MIT Press.

[4] David M. Blei and John D. Lafferty. Dynamic topic models. In *Proceedings of the 23rd International Conference on Machine Learning*, ICML '06, pages 113–120, New York, NY, USA, 2006. ACM.

[5] David M. Blei, Andrew Y. Ng, and Michael I. Jordan. Latent dirichlet allocation. *J. Mach. Learn. Res.*, 3:993–1022, 2003.

[6] Pete Chapman, Julian Clinton, Randy Kerber, Thomas Khabaza, Thomas Reinartz, Colin Shearer, and Rudiger Wirth. Crisp-dm 1.0 step-by-step data mining guide. Technical report, The CRISP-DM consortium, Agosto 2000.

[7] Michael Cox and David Ellsworth. Application-controlled demand paging for out-of-core visualization. In *Proceedings of the 8th Conference on Visualization '97*, VIS '97, pages 235–ff., Los Alamitos, CA, USA, 1997. IEEE Computer Society Press.

[8] Jeffrey Dean and Sanjay Ghemawat. Mapreduce: Simplified data processing on large clusters. In *Proceedings of the 6th Conference on Symposium on Operating Systems Design & Implementation - Volume 6*, OSDI'04, pages 10–10, Berkeley, CA, USA, 2004. USENIX Association.

[9] Jesús Domínguez. De lambda a kappa: evolución de las arquitecturas big data. https://www.paradigmadigital.com/techbiz/de-lambda-a-kappa-evolucion-de-las-arquitecturas-big-data/, Abril 2018. Último acceso 2019-10-13.

[10] Christine Fellbaum. *WordNet: an electronic lexical database*. MIT Press, 1998.

[11] Sanjay Ghemawat, Howard Gobioff, and Shun-Tak Leung. The google file system. *SIGOPS Oper. Syst. Rev.*, 37(5):29–43, October 2003.

[12] Yoav Goldberg. *Neural network methods in natural language processing*. Morgan & Claypool publishers, 2017.

[13] Tom Hope, Itay Lieder, and Yehezkel S. Resheff. *Learning TensorFlow: a guide to building deep learning systems*. OReilly Media, 2017.

[14] Jay Kreps. Questioning the lambda architecture. https://towardsdatascience.com/cap-theorem-and-distributed-database-management-systems-5c2be977950e, Julio 2014. Último acceso 2019-10-14.

[15] Douglas Laney. 3D data management: Controlling data volume, velocity, and variety. Technical report, META Group, February 2001.

[16] A. S. M. Ashique Mahmood. *Literature Survey on Topic Modeling*. 2013.

[17] Nathan Marz. How to beat the cap theorem. http://nathanmarz.com/blog/how-to-beat-the-cap-theorem.html, Octubre 2011. Último acceso 2019-10-14.

[18] Tomas Mikolov, Kai Chen, Greg S. Corrado, and Jeffrey Dean. Efficient estimation of word representations in vector space, 2013.

[19] Syed Sadat Nazrul. Cap theorem and distributed database management systems. https://towardsdatascience.com/cap-theorem-and-distributed-database-management-systems-5c2be977950e, Abril 2018. Último acceso 2019-10-14.

[20] Michael Nguyen. Illustrated guide to lstm's and gru's: A step by step explanation. https://towardsdatascience.com/illustrated-guide-to-lstms-and-gru-s-a-step-by-step-explanation-44e9eb85bf21, Septiembre 2018. Último acceso 2019-10-13.

[21] Christopher Olah. Understanding lstm networks – colah's blog. https://colah.github.io/posts/2015-08-Understanding-LSTMs/, Agosto 2015. Último acceso 2019-10-13.

[22] Stacey Ronaghan. Deep learning: Common architectures. https://medium.com/@srnghn/deep-learning-common-architectures-6071d47cb383, Agosto 2018. Último acceso 2019-10-13.

[23] Michal Rosen-Zvi, Thomas Griffiths, Mark Steyvers, and Padhraic Smyth. The author-topic model for authors and documents. In *Proceedings of the 20th Conference on Uncertainty in Artificial Intelligence*, UAI '04, pages 487–494, Arlington, Virginia, United States, 2004. AUAI Press.

[24] A. M. Turing. *Computing machinery and intelligence*. Blackwell for the Mind Association, 1950.

[25] Kevin Warwick and Huma Shah. *Turings imitation game: conversations with the unknown*. Cambridge Univeristy Press, 2016.