WEB 3.0, DATAWEB E INTELIGENCIA ARTIFICIAL

CBD – Complemento de Base de Datos

Jesús Ortiz Calleja y Juan Carlos Utrilla

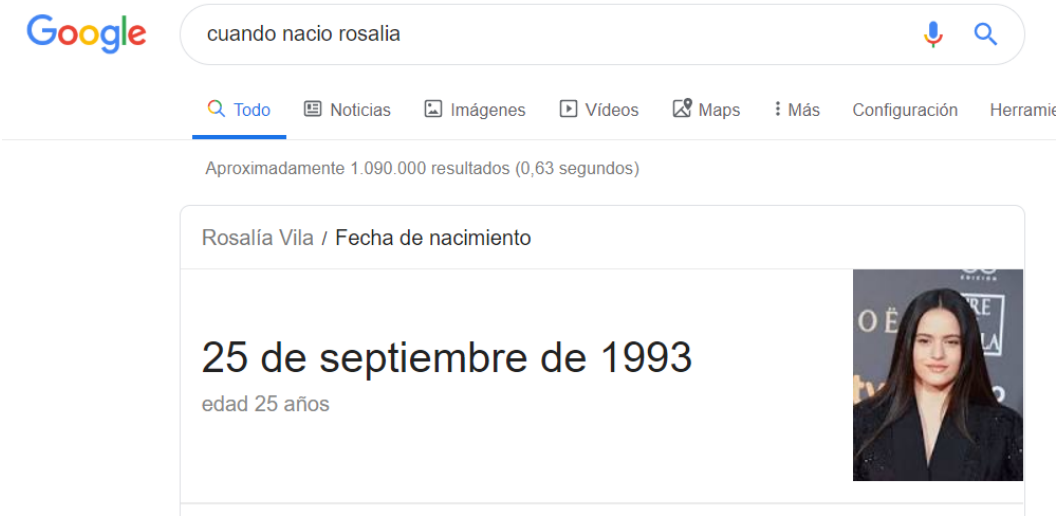
Contenido

1. Introducción 1
2. Conceptos básicos 2
   2.1. Dataweb, semántica y ontología 2
   2.2. Evolución de la web 2
      2.2.1. Web 1.0 (1989 - 2004) 3
      2.2.2. Web 2.0 o Web social (2004 - 2010) 3
      2.2.3. Web 3.0 o Web semántica (2010 - 2025) 4
      2.2.4. Web 4.0 - (2015 – ¿?) 4
   2.3. Inteligencia artificial 6
      2.3.1. Machine learning 6
      2.3.2. Deep learning 9
      2.3.3. Redes neuronales artificiales 10
      2.3.4. Underfitting y overfitting 11
3. Stack tecnológico 13
4. Trabajo 16
   4.1. Obtención y preprocesado de los datos 16
      4.1.1. Clase Query 17
      4.1.2. Clase SourceData 17
   4.2. Definición de la red neuronal, entrenamiento y testing 21
   4.3. Usando la red neuronal 23
5. Resultados de las pruebas 25
   5.1. Modificación la longitud de texto 25
   5.2. Modificando el batch_size 27
   5.3. Modificar el número de épocas 28
   5.4. Escogiendo los valores óptimos de entrenamiento 30
6. Conclusiones 30
7. Bibliografía 31

Dataweb e inteligencia artificial

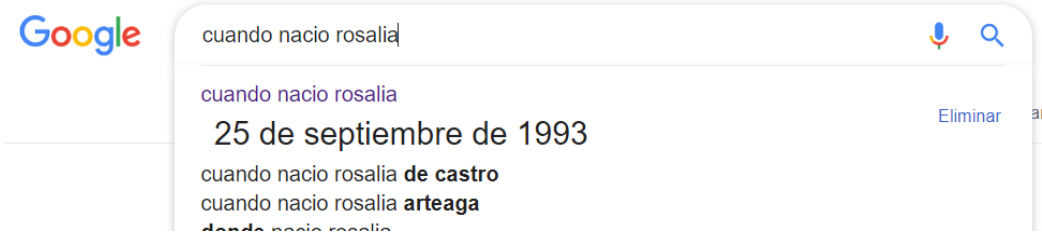
1. Introducción

Hace algunos años, las formas que tenía un usuario de interactuar con la web eran muy limitadas. Una parte fundamental de la Web tal como hoy la conocemos sigue siendo los buscadores. Con el tiempo hemos aprendido su funcionamiento y nos hemos adaptado a sus limitaciones. Su principal limitación es que no hablan el lenguaje del usuario. De hecho, hasta hace pocos años no eran capaces de responder preguntas del estilo ¿Cuándo nació Rosalía? Simplemente porque no eran capaces de entenderla. Pero hoy en día…



Todo esto ha sido posible gracias a la web semántica, también conocida como Web 3.0, utilizando técnicas de procesado del lenguaje natural, pero solo se limita a obtener resultados de búsqueda más precisos. No podría realizar peticiones del tipo: “Quiero un taxi a las 12:00 en la calle Miguel Fleta”.

El futuro (y presente) de la web es entender el lenguaje que hablan las personas, siendo capaz de anticiparse, predecir posibilidades y ofrecer soluciones concretas e instantáneas; una web que se fusione con el mundo real y que proporcione experiencias de usuario perfectas.



Para que esto sea posible, necesitamos añadir una capa de integración para poder explotar la Web 3.0 y sus enormes posibilidades haciendo uso de la inteligencia artificial, tal y como se define en la Web 4.0.

2. Conceptos básicos

2.1. Dataweb, semántica y ontología

Dataweb son aquellos datos que se encuentran alojados en la web los cuales pueden ser analizados permitiéndonos extraer información valiosa y más aún si podemos relacionar varias fuentes entre sí. Por ejemplo, podemos vincular los datos de un depósito de petróleo almacenado en una fuente de datos A con las predicciones guardadas en la fuente de datos B y con las mediciones en tiempo real de la extracción de petróleo guardada en la fuente de datos C.

El problema es que la mayoría de las veces estos datos no se encuentran estructurados o bien clasificados, lo que representa un gran problema para una máquina a la hora de procesarlos. Por lo que necesitamos añadir datos que sean legibles por estas, como metadatos semánticos y ontológicos.

La **semántica** permite a la máquina comprender el significado de los datos con los que está trabajando y la **ontología** define de manera formal tipos, propiedades y relaciones entre entidades existentes dentro de un dominio, estableciendo una clasificación de los datos que limita la complejidad y organiza los datos.

Ambas en conjunto son herramientas imprescindibles que ayudan a entender la información, almacenarla, relacionarla con más información, compartirla, buscar y encontrar nueva información.

No obstante, la tendencia que se persigue es que las máquinas no dependan de la definición de los metadatos para su comprensión de tal forma que sea capaz de inferirlos automáticamente por el contexto.

2.2. Evolución de la web

Aunque normalmente asociamos Internet y Web (World Wide Web), realmente no se trata de lo mismo. Podríamos decir que Internet es la infraestructura y la Web es una aplicación que se apoya en el primer concepto. Ni que decir tiene que Internet ha revolucionado nuestra vida en todos los aspectos: trabajo, relaciones, compras, estudios, etc. Por ello, haremos un repaso a la evolución de la web a lo largo de casi 30 años.

2.2.1. Web 1.0 (1989 - 2004)

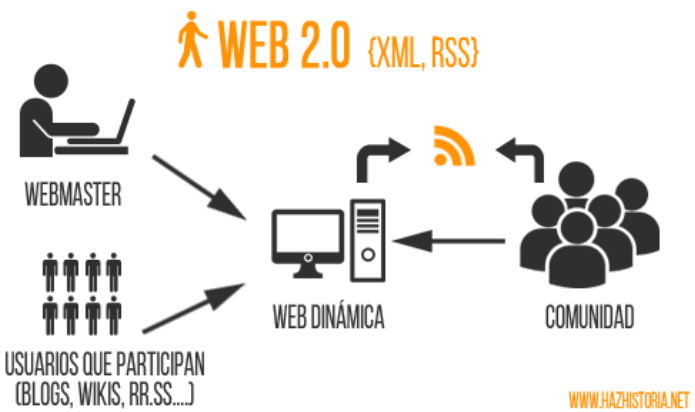
Considerada como la web original, el principio, el primer contacto con las páginas web. Básicamente funcionaba como una mera fuente de información, caracterizándose por ser unidireccional y con una interacción bastante restringida en la que sólo se puede buscar y leer solo aquello que el webmaster publique.

Tenía un carácter principalmente divulgativo y con información principalmente cultural. Poco a poco las empresas empezaron a tomar parte y las primeras webs de empresa surgieron, con diseños muy pobres y contenidos que rápidamente quedaban anticuados al ser complejo actualizarlos.

2.2.2. Web 2.0 o Web social (2004 - 2010)

Este término fue acuñado por Tom O'Reilly en el año 2004 para referirse a la segunda generación de modelos de páginas web. Surge como un fenómeno social a partir de auge de los blogs, los foros, las redes sociales y el e-commerce.

La Web 2.0 es puramente social. En ella, el usuario se convierte en el principal protagonista. Ya no sólo consumimos información, sino que empezamos a escribir y producir contenido. De esta forma, la web se transforma en un espacio bidireccional centrado en las personas. **En este punto es donde nos encontramos la mayor parte de los consumidores**.



2.2.3. Web 3.0 o Web semántica (2010 - 2025)

Nota: dependiendo de la bibliografía consultada y de la evolución del propio concepto en sí, algunos se refieren a la web 3.0 (o semántica) incluyendo inteligencia artificial mientras que otros prefieren hacer una distinción refiriéndose a la web 3.0 sin inteligencia artificial y la web 4.0 incluyéndola. **En el presente documento consideraremos esta última**.

La web 3.0 supone un salto tecnológico en donde la clave y principal factor diferencial reside en cómo accedemos a la información. Aquí, los buscadores son clave, pero no por sus mejores algoritmos, mayor indexación de información u otros extras. Lo son porque **permiten hacer uso de lenguaje más natural**, de forma que obtenemos una web (información) más personalizada y descartando información irrelevante para nosotros.

Para que esto sea posible, **se utilizan los datos de forma más eficiente mejorando su organización mediante la inclusión de datos semánticos y ontológicos** permitiendo que puedan ser rastreados por sistemas de procesamiento y, por consiguiente, garantizando búsquedas por significado y no por contenido textual.

El simple hecho de que una máquina pueda entender la semántica y ontología de los datos le permite utilizar información entre máquinas distribuidas en la Web y, a su vez, manejar la información con mayor eficiencia.

2.2.4. Web 4.0 - (2015 – ¿?)

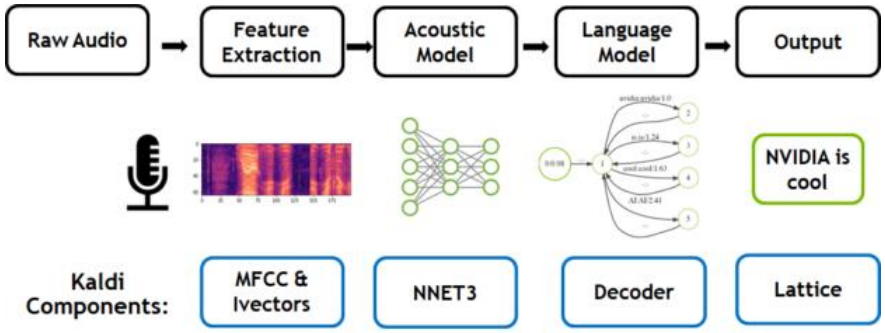
En la actualidad, la Web 3.0 (o semántica) permite conectar toda la información contenidas en redes sociales, aplicaciones, etcétera de una forma más evolucionada que en la Web 2.0. Aun así, no resulta del todo eficiente, pues hay limitaciones tecnológicas que le impiden ofrecer más debido a la escasa implantación que ha tenido.

Lo que propone la Web 4.0 es mejorar esa experiencia mediante el uso de nuevas tecnologías que permitirán un nivel de interacción más completo y personalizado.

Mientras que en la Web 3.0 tenemos que filtrar los resultados obtenidos por un buscador hasta encontrar lo que se desea, en la Web 4.0 podemos mejorar esta experiencia de usuario haciendo que el nivel de interacción sea más completo y personalizado. Es decir, podríamos decirle "Reserva una mesa para cenar hoy" o "Pide un taxi a las 12:00 en mi casa" a tu dispositivo -que puede ser un smartphone, wearable o cualquier otro- y automáticamente ejecutaría dicha acción sin más intervención propia. Por tanto, **pasamos de una web que proporciona información a otra que proporciona soluciones**.

Esta es la idea de la Web 4.0: ofrecer soluciones a partir de toda la información que damos y existe en la Web. Para lograrlo, se fundamentará en 4 pilares:

- **La comprensión del lenguaje natural (NLU) y tecnologías Speech-to-Text**: con estas técnicas podemos convertir el lenguaje el texto y, mediante un análisis semántico y morfológico, crear representaciones sin ambigüedades. Así, la máquina puede responder como si prácticamente fuese humana.



  Podemos contemplar algunos de los avances que se están logrando: algunos **asistentes virtuales** como Siri, Google Now o Cortana cada vez entienden de forma más precisa y correcta lo que les decimos mediante *Speech-to-Text*. Es más, ya hay smartphones que siempre están "escuchando" para activarse en el momento que oigan "Oye, Siri" u "Ok Google".

  Otro ejemplo son los **bots**. Estos programas dependen de la introducción de texto, pero, con la evolución de estas tecnologías seremos capaces de conversar con ellos de igual modo que lo haríamos con un amigo en una cafetería.

- **Sistemas de comunicación máquina-máquina (M2M)**: la red estará compuesta por una serie de agentes inteligentes que se comunicarán entre sí de forma autónoma y delegando la respuesta al agente adecuado.

- **Uso útil de la información en un determinado contexto e historial del usuario**: mediante la inteligencia artificial y el Big Data podremos determinar, por ejemplo, la actitud de los usuarios a través de wearables que, por ejemplo, monitorizan el ritmo cardíaco o a través de técnicas de minería de datos realizar un análisis de sentimientos.

- **Definición de modelos de interacción con el usuario**: dispositivos móviles, los bots, el Internet de las Cosas (IoT) y otras tecnologías como el reconocimiento facial, sistemas

biométricos, chips subcutáneos, vehículos autónomos. Todo esto junto a aprendizajes automáticos con *Deep Learning* o *Machine Learning* suponen un importante impulso tecnológico en ese sentido.

Empresas como Google, Microsoft o Facebook, entre otras, están desarrollando nuevos sistemas que gracias al *Deep Learning* y *Machine Learning* serán capaces de procesar información de forma similar como lo haría el cerebro humano.

Conclusión: podemos considerar a la Web 4.0 como una web abierta, conectada, predictiva y más inteligente con el objetivo de ofrecer soluciones específicas al usuario, basándose tanto en información que este da o ha dado como en toda la que ya existe en la web gracias a las nuevas tecnologías que permiten una interacción más completa y personalizada.

2.3. Inteligencia artificial

La **inteligencia artificial (IA)** es **la inteligencia llevada a cabo por máquinas mediante la combinación de algoritmos que pretende emular las mismas capacidades del ser humano**.

La IA se divide en dos escuelas de pensamiento:

- **La inteligencia artificial convencional**: basada en el análisis formal y estadístico del comportamiento humano ante diferentes problemas.

- **La inteligencia computacional**: centrada en el estudio de mecanismos adaptativos para permitir el comportamiento inteligente de sistemas complejos y cambiantes. Dentro de ella podemos encontrar técnicas como la computación evolutiva, *swarm intelligence*, *machine learning*, entre otros. **Durante el desarrollo del trabajo nos centraremos en *machine learning***.

2.3.1. Machine learning

El ***machine learning* o aprendizaje automático** o aprendizaje de máquinas es el subcampo de ciencias de la computación y una rama de la inteligencia artificial cuyo objetivo es desarrollar técnicas que doten a las máquinas de la capacidad de aprender y realizar predicciones sin necesidad de programar nada.

2.3.1.1. Grupos de algoritmos del Machine Learning

Una vez entendido que es el Machine Learning, cabe conocer los tres tipos de que existen:

- **Aprendizaje supervisado**: técnica de aprendizaje automático que consiste en descubrir la relación existente (si existe) entre unas variables de entrada y otras de salida, es decir, enseñar a un algoritmo cuál es el resultado que quieres obtener a partir de una cantidad suficiente de datos de entrenamiento (ejemplos).

Tras esto, el algoritmo será capaz de dar un resultado correcto incluso cuando se le muestren valores que no haya visto antes permitiéndonos generalizar conocimiento.

Datos de entrenamiento	
Datos de entrada	**Datos de salida**
1	2
2	4
3	6
4	8
5	10

Supongamos que partimos de una serie de datos de entrada y salida para entrenar al algoritmo.

Una vez entrenada, si el dato de entrada es 10, ¿cuál será el dato de salida? Lógicamente, el resultado será 20.

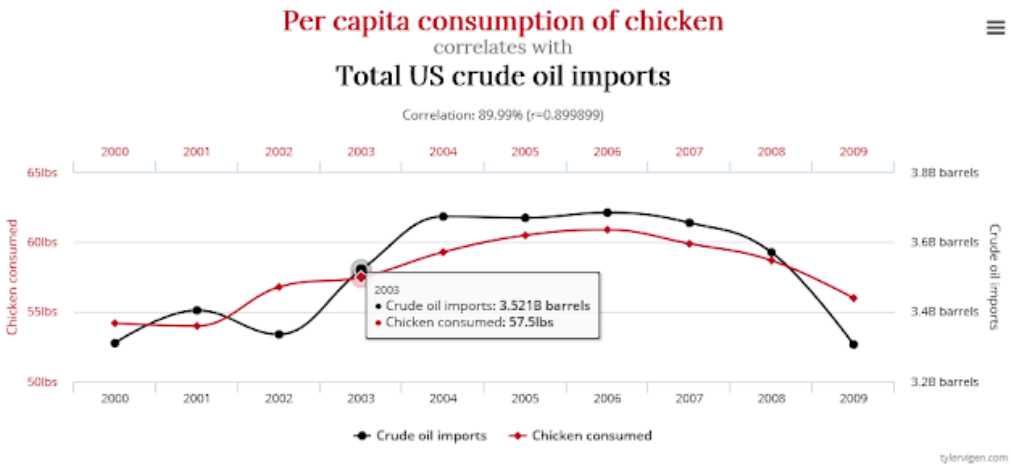
Con esto podemos concluir que el algoritmo de aprendizaje supervisado ha sido capaz de encontrar mediante observación la relación existente, que es multiplicar un dato de entrada por 2.

Otro ejemplo práctico puede ser la detección de que un correo electrónico sea considerado Spam o no. A simple vista, es sencillo, pero ¿seríamos capaces de explicar por qué patrones hacen que un correo sea o no Spam? Con un buen conjunto de datos de entrenamiento seríamos capaces de resolverlo.

- **Aprendizaje no supervisado**: técnica de aprendizaje automático que consiste en producir conocimiento sin disponer de datos etiquetados para el entrenamiento, es decir, sólo conocemos los datos de entrada, pero no los de salida.

El objetivo de este paradigma es encontrar patrones de similitud (o agrupamiento de similitudes) de los datos de entrada generando una estructura interna para encontrar algún tipo de organización que simplifique el análisis, pero sin garantizar que estas tengan algún significado o utilidad. Por ello, tienen un carácter **exploratorio**.

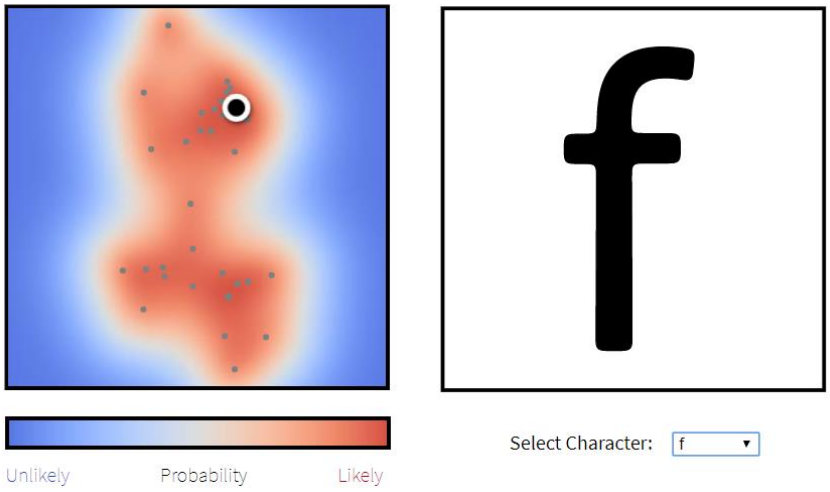
En ocasiones, al explorar los datos sin un objetivo definido, se pueden encontrar correlaciones espúreas curiosas, pero poco prácticas. Por ejemplo, en la gráfica que se presenta a continuación en la web Tylen Vigen Spurious Correlations, podemos apreciar una fuerte correlación entre el consumo per capita de pollo en Estados Unidos y sus importaciones de petróleo.

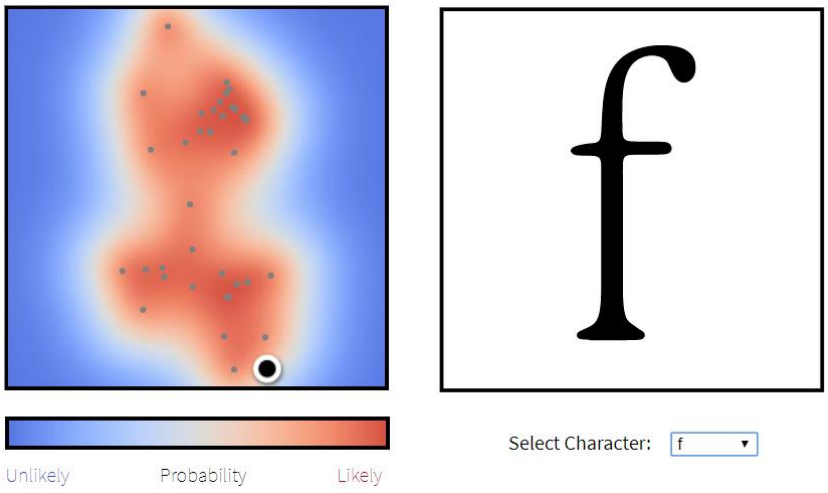


Como ventaja, generar los datos de entrenamiento implica menos recursos humanos. Por ejemplo, para entrenar un algoritmo que clasifica perros y gatos no solo necesitamos imágenes de entrada, sino también que alguien (un humano) vaya visualizando imagen por imagen etiquetando cada una de ellas. Además, estás bases de datos suelen superar 100.000 ejemplo. Tal y como podemos observar, no es una tarea fácil o barata de realizar.

Un ejemplo puede ser detectar que es una silla o no. Todo el mundo comprende el concepto de silla. Sin embargo, cuando nos fijamos en la realidad, nos encontramos con que este concepto tiene muchas variantes diferentes. Entonces, ¿qué es realmente una silla? Esto es algo que nuestro cerebro consigue hacer de forma automática y algo que el aprendizaje no supervisado tiene que aprender a hacer.

Actualmente, los algoritmos ya son capaces de aprender estas estructuras conceptuales denominadas **espacios latentes**. Una vez construido este espacio, el algoritmo adquiere capacidades tan interesantes como saber si una cosa es similar a otra.





El aprendizaje no supervisado se suele usar en problemas de clustering, agrupamientos de co-ocurrencia y profiling. Sin embargo, los problemas que implican tareas de encontrar similitud, predicción de enlaces o reducción de datos, pueden ser supervisados o no.

2.3.2. Deep learning

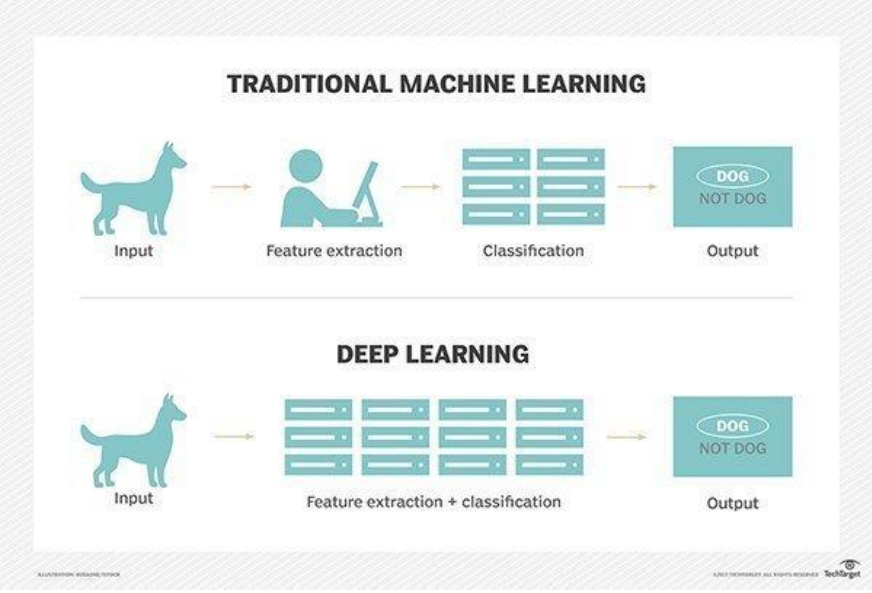
En el aprendizaje automático, los datos pasan principalmente a través de algoritmos que realizan transformaciones lineales para producir resultados, aunque también puede realizar transformaciones no lineales.

El ***deep learning* o aprendizaje profundo** es un subconjunto del aprendizaje automático en el que los datos pasan por varias transformaciones no lineales para obtener un resultado. Estos algoritmos se apilan en una jerarquía de creciente complejidad de abstracción.

"Profundo" se refiere a muchos pasos en este contexto. La salida de un paso es la entrada de otro paso, y esto se hace continuamente para obtener una salida final.

En los **sistemas de aprendizaje tradicionales**, este proceso es supervisado y el programador tiene que elaborar las reglas concretas para un problema específico y decidir, por ejemplo, si una imagen contiene un perro o no. Este es un proceso laborioso llamado extracción de características y la tasa de éxito de la computadora depende totalmente de la capacidad del programador para definir con precisión un conjunto de características para "perro".

La ventaja del **aprendizaje profundo** es que el programa construye el conjunto de características por sí mismo sin supervisión. Esto no es solo más rápido, sino que por lo general es más preciso.



Inicialmente, el programa podría ser provisto de datos de entrenamiento, un conjunto de imágenes para las cuales un humano ha etiquetado cada imagen "perro" o "no perro". El programa utiliza la información que recibe de los datos de entrenamiento para crear un conjunto de características para el perro y construir un modelo predictivo. En este caso, el modelo que la computadora crea por primera vez podría predecir que cualquier cosa en una imagen tenga

cuatro patas y una cola debería estar etiquetada como “perro”. Por supuesto, el programa no es consciente de las etiquetas “cuatro patas” o “cola”, simplemente buscará patrones de píxeles en datos digitales. Con cada iteración, el modelo predictivo que crea el equipo de cómputo se vuelve más complejo y preciso.

Los resultados son sorprendentes, pues no sólo es capaz de detectar en qué imágenes aparece un perro o un determinado color; en el caso de personas u objetos, es capaz de diferenciar qué es un perro real o una mancha cuya silueta se parece a un perro.

Por supuesto, para llevar a cabo todo esto, se ha realizado un trabajo previo de aprendizaje. Este le permite saber cómo es la forma de un perro, detectar objetos y mucho más.

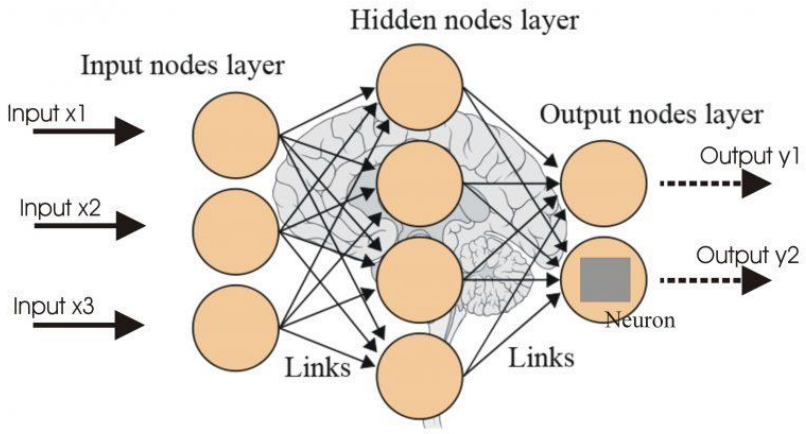
Debido a que este proceso imita el pensamiento humano, el aprendizaje profundo a veces se conoce como aprendizaje neuronal profundo o redes neuronales profundas (*Deep neural networks* – DNN). A diferencia del niño pequeño, que tardará semanas o incluso meses en comprender el concepto de “perro”, un programa informático que utiliza algoritmos de aprendizaje profundo puede mostrar un conjunto de entrenamiento y ordenarlo a través de millones de imágenes, identificando con precisión qué imágenes tienen perros en tan solo unos minutos.

Con el fin de conseguir un nivel de precisión aceptable, estos algoritmos requieren ingentes cantidades de datos de entrenamiento y capacidad de procesamiento, ninguno de los cuales estaba fácilmente disponible para los desarrolladores de software hasta hace bien poco.

2.3.3. Redes neuronales artificiales

Aunque existen varias maneras de implementar *deep learning*, una de las más comunes es utilizar **redes neuronales**.

Las **redes neuronales artificiales** son una herramienta matemática que, de forma muy simplificada, modela el comportamiento observado en su homologo biológico. Consiste en un conjunto de unidades, llamadas neuronas artificiales, conectadas entre sí para transmitirse señales. La información de entrada atraviesa la red neuronal (donde se somete a diversas operaciones) produciendo unos valores de salida.



Estos sistemas aprenden y se forman a sí mismos, en lugar de ser programados de forma explícita, y sobresalen en áreas donde la detección de soluciones o características es difícil de expresar con la programación convencional. Para realizar este aprendizaje automático, normalmente, se intenta minimizar una función de pérdida que evalúa la red en su totalidad. Los valores de los pesos de las neuronas se van actualizando, buscando reducir el valor de la función de pérdida. Este proceso se realiza mediante *backpropagation* (propagación hacia atrás).

Lo interesante de todo esto es que son capaces de aprender de forma jerarquizada, es decir, la información se aprende por niveles donde las primeras capas aprenden conceptos muy concretos (que es un tornillo, un espejo, una rueda, etcétera) mientras que en las capas posteriores se usa la información aprendida para aprender conceptos más abstractos, (qué es un coche, un camión, un avión, etcétera). Esto hace que **a medida que añadimos más capas, la información que se aprende es más abstracta e interesante** y **como no hay límites a la hora de añadir capas, la tendencia es que cada vez estos algoritmos añadan más y más capas convirtiéndose en algoritmos cada vez más complejos**.

2.3.4. Underfitting y overfitting

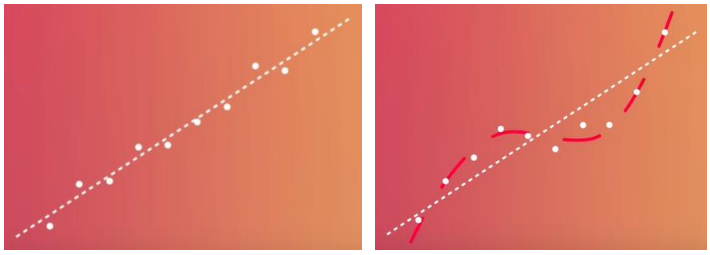
La **generalización** es la capacidad de obtener resultados correctos con datos nuevos ajenos a los datos de entrenamiento. Este concepto es muy importante en *Machine Learning*.

Que un modelo sea capaz de clasificar con acierto que una imagen contiene un perro o un gato con nuestros datos de entrenamiento no es relevante porque ya sabíamos de antemano la etiqueta (resultado esperado) de cada imagen.

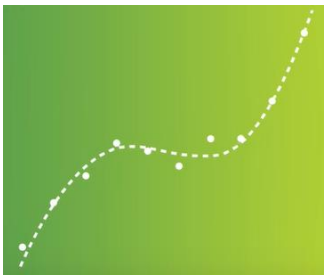
Lo que realmente importa en nuestro modelo es que sea capaz de generalizar el conocimiento permitiendo realizar predicciones certeras con datos nuevos y ajenos a los datos de entrenamiento.

Imaginemos que tenemos una nube de puntos del que se infiere un modelo a partir de una recta permitiéndonos hacer predicciones con datos que no tenemos.

Ahora supongamos que modificamos levemente la posición de la nube de puntos haciendo que se parezca más a una curva que a una recta. Con estos cambios nuestro modelo ya no podrá hacer predicciones y, por tanto, necesitaremos otra función que permita predecir futuros valores. En este estado, podemos decir que nuestro modelo está mal ajustado o que sufre de ***underfitting*** **(subajuste)**.

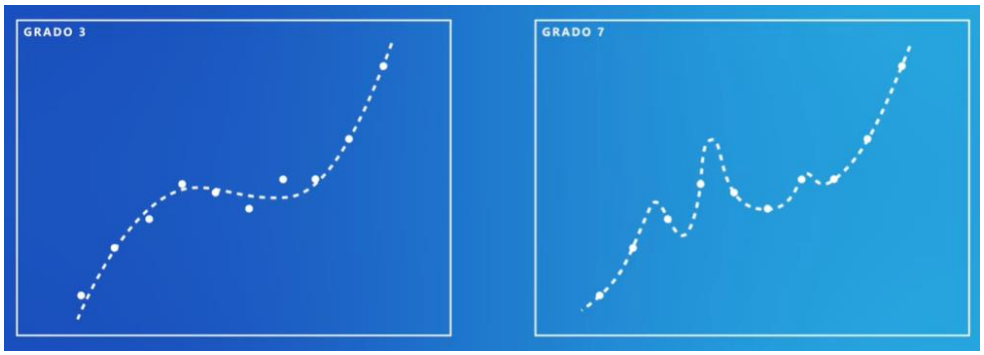


Al asumir que el modelo es una recta, el modelo es demasiado rígido y no tiene la flexibilidad suficiente como para adaptarse a los nuevos datos que se han añadido. Como podemos observar, este ejemplo se adapta mejor con la regresión polinomial.



Si al nuevo modelo se le añaden más puntos ajenos al entrenamiento y que están cercanos a nuestro modelo, podemos decir que el modelo se ajusta bien a la distribución de puntos original en comparación con el modelo que sufre *underfitting*.

Podemos afinar aún más nuestro modelo haciendo que se parezca lo máximo posible aumentando el grado del polinomio. El problema de realizar esta acción es que también modelamos el ruido de nuestros datos.



Si en el modelo anterior era capaz de predecir nuevos datos, ahora no es capaz porque las variables asociadas tienen un error mayor, es decir, el modelo generaliza peor. Por tanto, podemos decir que nuestro modelo está demasiado ajustado o que se sufre ***overfitting*** **(sobreajustado)**.

A nivel conceptual, nuestro problema consigue especializarse tan bien a los datos de entrenamiento que es casi como si memorizase las soluciones perdiendo la capacidad de generalizar y a su vez creando una falsa sensación de que el modelo está aprendiendo correctamente.

Por tanto, podemos concluir que un modelo bien ajustado no tiene porque generalizar bien si se produce *underfitting* o *overfitting*.

3. Stack tecnológico

Para desarrollar el proyecto se ha hecho uso de las siguientes tecnologías:

- **Python**: lenguaje de programación interpretado de tipado dinámico, multiparadigma y multiplataforma. Es el lenguaje base sobre el que desarrollaremos los scripts del proyecto.

- **Tensorflow**: es una biblioteca de código abierto desarrollada por Google, preparada para hacer cálculos matemáticos simbólicos, lo que incluye diferenciación automática y manejo de tensores. Todo esto está enfocado al desarrollo de modelos de aprendizaje

automático.

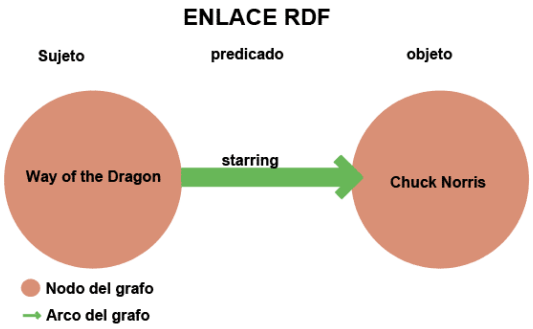
Está escrita en C++, contando con una API para Python. Algunas partes también están escritas en Cuda para poder aprovechar la potencia de las GPU compatibles (en general son GPU de Nvidia).

- **Keras**: es una biblioteca de código abierto escrita en Python que provee de una API de alto nivel para redes neuronales, estando centrada en la velocidad y la facilidad a la hora de definir los modelos y entrenarlos.

  Así mismo, Keras tiene que hacer uso de otra biblioteca de más bajo nivel para funcionar, sirviendo de interfaz con ésta. Se puede escoger entre: TensorFlow, CNTK y Theano. En concreto la versión de Keras que usamos es la que ha sido integrada por Google con TensorFlow.

TensorFlow realiza el proceso de diferenciación y entrenamiento de forma automática, mientras que todavía se necesita declarar una serie de operaciones a nivel matemático para definir cada capa de la red neuronal y Keras abstrae esto, pues tiene predefinidas una serie de capas de uso muy común con las que podemos resolver la mayoría de los problemas.

- **RDF (*Resource Description Framework*)**: es una especificación de la WC3 para establecer metadatos que representen la semántica de los recursos web. Está basado en tripletas: sujeto (recurso), predicado (propiedad) y objeto (valor). Es suficientemente flexible para modelar todo tipo de datos, más allá del ámbito de los recursos web.



- **Sujeto**: indica el recurso que se está describiendo (*Way of the Dragon*).
- **Predicado**: denota rasgos o aspectos del recurso y expresa la propiedad o relación entre el sujeto y el objeto, es decir, indica el significado de la relación (starring).
- **Objeto**: es el valor de la propiedad (*Chuck Norris*).

RDF tiene algunos detractores, como Google, que prefiere utilizar *microdata* en HTML5 y otros que abogan por Hypernotation.

- **RDFS (*Resource Description Framework Schema*)**: está construido sobre RDF, permite definir vocabularios específicos dentro de un dominio. Esto delimita la propiedades disponibles e identifica de manera única los recursos. El objetivo de RDFS es facilitar la generalización entre los diferentes recursos y propiedades; pues esto se usan repetidamente en todas las tripletas.

- **OWL (*Web Ontology Language*)**: es un lenguaje de la web semántica construido sobre RDFS que permite establecer relaciones entre distintas clases definidas dentro de RDFS, creando **ontologías**. Esto permite saber: que clases son subconjunto de otras, que clases son equivalentes, cardinalidad, etc.

- **SPARQL**: se trata de un lenguaje desarrollado por W3C para la consulta de grafos RDF. Permite el acceso a información disponible en la web a través de diversas fuentes de datos. Gracias a esto, podemos considerarla como una tecnología clave en el desarrollo de la web semántica.

4. Trabajo

Con la parte práctica de nuestro trabajo hemos pretendido hacer una pequeña demostración de cómo una IA es capaz de inferir la semántica de un texto de Wikipedia. Es decir, un primer paso para alcanzar la etapa en la que la máquina no necesita estructuras de datos web con metadatos semánticos.

Para ello aplicaremos *deep learning* con redes neuronales a la fuente de datos (o *endpoint*) DBPedia que contiene datos semánticos y ontológicos como fuente de datos de entrenamiento.

Aunque el código es bastante simple a nivel de complejidad nos sirve para satisfacer con los objetivos del proyecto.

4.1. Obtención y preprocesado de los datos

> **Nota**: pese a que nos hubiese interesado disponer de más datos de entrenamiento, el *endpoint* de dbpedia nos limita a 10.000 elementos.

Los datos de entramiento y testeo del proyecto se obtienen a partir de 2 queries para cada tipo de datos a obtener (Persona y trabajo intelectual) escritas en SPARQL por separado y que se encuentran definidas en el fichero **source_data.py**.

El resultado de cada query es cada uno de textos que más adelante la red neuronal utiliza tanto para entrenar como para testear. El tipo esperado se deduce a partir de cada query.

```
PREFIX dbr: <http://dbpedia.org/resource/>
PREFIX dbo: <http://dbpedia.org/ontology/>
PREFIX dbc: <http://dbpedia.org/property/>
PREFIX rdf: <http://www.w3.org/1999/02/22-rdf-syntax-ns#>
PREFIX umb: <http://umbel.org/umbel/rc/>

SELECT DISTINCT ?subject ?object WHERE {
   ?subject rdf:type umb:Work.
   ?subject dbo:abstract ?object.FILTER(lang(?object) = 'en')

}
LIMIT 15000
PREFIX dbr: <http://dbpedia.org/resource/>
PREFIX dbo: <http://dbpedia.org/ontology/>
PREFIX dbc: <http://dbpedia.org/property/>
PREFIX rdf: <http://www.w3.org/1999/02/22-rdf-syntax-ns#>
PREFIX umb: <http://umbel.org/umbel/rc/>

SELECT DISTINCT ?subject ?object WHERE {
   ?subject rdf:type umb:Person.
   ?subject dbo:abstract ?object.FILTER(lang(?object) = 'en')

}
LIMIT 15000
```

Para mejorar la legibilidad del código se han desarrollado las clases Query y SourceData en el script **source_data.py**.

4.1.1. Clase Query

Esta clase se encarga de ejecutar las queries para obtener los datos de entrenamiento. El código es el siguiente:

```
from SPARQLWrapper import SPARQLWrapper, JSON

class Query:
    _dbpedia = SPARQLWrapper("http://dbpedia.org/sparql") # Endpoint
    _dbpedia.setReturnFormat(JSON)

    # Construye una tupla de diccionarios con los datos de entrenamiento.
    # Dependiendo de la posición, contendrá un conjunto de textos de un
    # tipo determinado con una estructura sin procesar, siendo:
    #       result[0] --> de tipo Work
    #       result[1] --> de tipo Person

    @staticmethod
    def get_results(*args):
        results = ()

        for query in args:
            Query._dbpedia.setQuery(query)
            result = Query._dbpedia.query().convert()
            results = results + (result,)

        return results
```

4.1.2. Clase SourceData

Esta clase se encarga de ejecutar el preprocesamiento de los datos de entretamiento. El código es el siguiente:

```
class SourceData:

    # Obtenemos los datos de entrenamiento sin preprocesar

    _resultsWorks, _resultsPersons, _resultsAnimals =
            Query.get_results(workQuery, personQuery, animalQuery)
    _all_results = None
    _data_len = None
    _word_index = None
    _data_examples = None

    def __init__(self):

        # Preprocesa los datos de entrenamiento original en una lista
        # de elementos (textos).

        SourceData._all_results =
            SourceData.combine_results(SourceData._resultsWorks,
                                       SourceData._resultsPersons,
                                       SourceData._resultsAnimals)
        SourceData._data_len = len(SourceData._all_results)
        SourceData._word_index = SourceData.get_word_index()
        SourceData._data_examples = SourceData.data_examples()
```

```
    # Extrae cada elemento (texto) del diccionario en una única lista
    # de elementos, obteniendo un listado con todos los textos.

    @staticmethod
    def combine_results(*args):
        combination = []
        for result in args:
            combination.extend(result["results"]["bindings"])
        return combination

    # Devuelve un diccionario que indexa las palabras de todos los textos
    # por orden de llegada.

    @staticmethod
    def get_word_index():
        if SourceData._word_index is None:

            print("# Indexing vocabulary words #")

            word_index = {}
            index = 0
            count = 0.0                                  # Barra de progreso

            for result in SourceData._all_results:
                text = result["object"]["value"]
                text = SourceData._normalize_text(text)
                words = text.split(" ")
                progress(count, SourceData._data_len)    # Barra de progreso
                count = count+1                          # Barra de progreso

                for word in words:
                    if word not in word_index:
                        word_index[word] = index
                        index = index+1                  # Barra de progreso

            progress(count, SourceData._data_len)        # Barra de progreso
            sys.stdout.write("]\n")                      # Barra de progreso

            return word_index
        else:
            return SourceData._word_index

    # Método privado que normaliza el texto, es decir, transforma el texto
    # (cadena) en un vector (lista de índices).

    @staticmethod
    def _vectorize_text(normalize_text):
        vector = []
        word_index = SourceData.get_word_index()

        for word in normalize_text:
            if word_index.get(word):
                vector.append(word_index.get(word))

        return vector
```

```
    # Normaliza un texto en función de listado de índices.

    @staticmethod
    def vectorize_text(word_index, text):
        vector = []
        normalize_text = SourceData._normalize_text(text)

        for word in normalize_text:
            if word_index.get(word):
                vector.append(word_index.get(word))
            else:
                vector.append(1)

        return vector

    # Método que desordena los datos para garantizar que el procesa-
    # miento de los datos es el adecuado.

    @staticmethod
    def _randomize_lists_pair(list1, list2):
        list1_copy = list(list1)
        list2_copy = list(list2)
        index = 0
        index_used = []

        for _ in list1:
            random_index = random.randint(0, len(list1) - 1)

            if random_index not in index_used:
                list1_copy[index] = list1[random_index]
                list1_copy[random_index] = list1[index]
                list2_copy[index] = list2[random_index]
                list2_copy[random_index] = list2[index]
                index_used.append(index)
                index_used.append(random_index)

            index = index + 1

        return list1_copy, list2_copy

    # Método que devuelve una tupla de dos listas:
    #       - Lista vector: texto vectorizado
    #       - Lista label: resultado esperado

    @staticmethod
    def data_examples():_
        if SourceData._data_examples is None:
            print("# Vectorizing data examples #")
            examples = []
            labels = []
            count = 0                                       # Barra de progreso

            for result in SourceData._resultsWorks["results"]["bindings"]:
                examples.append(
                     SourceData._vectorize_text(
                     SourceData._normalize_text(result["object"]["value"])))
                labels.append(0)
                progress(count, SourceData._data_len)    # Barra de progreso
                count = count + 1                        # Barra de progreso

            for result in SourceData._resultsPersons["results"]["bindings"]:
                examples.append(
                     SourceData._vectorize_text(
                     SourceData._normalize_text(result["object"]["value"])))
                labels.append(1)

                progress(count, SourceData._data_len)    # Barra de progreso
```

```
                count = count + 1                       # Barra de progreso

            progress(count, SourceData._data_len)       # Barra de progreso
            sys.stdout.write("]\n")                     # Barra de progreso

            return SourceData._randomize_lists_pair(examples, labels)
        else:
            return SourceData._data_examples

# Método de calcula la barra de progreso

def progress(progress, total):
    ratio = progress/total
    if (ratio*100) % 2 == 0:
        update_progress(ratio)

# Método para actualizar la barra de progreso

def update_progress(progress_ratio):
    barLength = 50
    status = ""
    if isinstance(progress_ratio, int):
        progress_ratio = float(progress_ratio)
    if not isinstance(progress_ratio, float):
        progress_ratio = 0
        status = "error: progress var must be float\r\n"
    if progress_ratio < 0:
        progress_ratio = 0
        status = "Halt...\r\n"
    if progress_ratio >= 1:
        progress_ratio = 1
        status = "Done...\r\n"
    block = int(round(barLength * progress_ratio))
    text = "\rPercent: [{0}] {1}% {2}"
       .format("=" * block + "-" *
              (barLength - block), progress_ratio * 100, status)
    sys.stdout.write(text)
    sys.stdout.flush()
```

El código que invoca a estos métodos se encuentra en el script **classification.py**. La obtención y preprocesamiento de los datos quedaría de la siguiente manera:

```
from source_data import SourceData
import tensorflow as tf

print(tf.__version__)                               # Versión de Tensorflow

# Preprocesamiento de los datos

source = SourceData()
examples = source.data_examples()
n_examples = len(examples[0])

print(n_examples)

# Obtenemos los datos de entrenamiento

train_data = examples[0][:n_examples//2]
train_labels = examples[1][:n_examples//2]

# Obtenemos los datos de prueba

test_data = examples[0][n_examples//2:]
test_labels = examples[1][n_examples//2:]
```

```
# Indexamos todas las palabras disponibles, siendo las dos primeras reservadas

word_index = {k: (v + 2) for k, v in word_index.items()}
word_index["<PAD>"] = 0
word_index["<UNK>"] = 1     # Desconocido(unknown)
```

4.2. Definición de la red neuronal, entrenamiento y testing

Una vez que tenemos los datos con los que vamos a entrenar a la red neuronal procederemos a definirla. Para ello utilizaremos la librería Keras basada en el framework TensorFlow que nos proporciona un alto grado de abstracción a nivel de composición de capas.

El código quedaría de la siguiente manera:

```
# Como necesitamos que cada texto tenga la misma longitud, establecemos
# el valor "maxlen" en 256, donde posiblemente se pierdan algunos valores.

text_len = 256

# Obtenemos los datos de entrenamiento

train_data = keras.preprocessing.sequence.
                  pad_sequences(train_data,
                                value=word_index["<PAD>"],
                                padding='post',
                                maxlen=text_len)

# Obtenemos los datos de testing

test_data = keras.preprocessing.sequence.
                  pad_sequences(test_data,
                                value=word_index["<PAD>"],
                                padding='post',
                                maxlen=text_len)

# Calculamos el tamaño del vocabulario (todas las palabras disponibles)

vocab_size = len(word_index) + 1

# A partir de la vectorización obtenida de:
#
#               http://nlp.stanford.edu/data/glove.6B.zip
#
# El método create_embedding_matrix permite vectorizar las palabras de tal
# forma que representan relaciones con el resto de las palabras ayudándonos
# a inferir la semántica y relaciones de las palabras entre sí.
#
# Esto ahorra parte del trabajo que realiza la capa de embedding de nuestra
# red neuronal que utilizaremos a continuación. Esta capa, de por sí, se
# encarga de generar esta vectorización pero funcionará mejor si le pasamos
# una vectorización ya precalculada.

def create_embedding_matrix(filepath, word_index, embedding_dim):
    vocab_len = len(word_index) + 1
    embedding_matrix = np.zeros((vocab_len, embedding_dim))

    with open(filepath) as f:
        for line in f:
            word, *vector = line.split()
            if word in word_index:
                idx = word_index[word]
```

```
                embedding_matrix[idx] = np.array(
                    vector, dtype=np.float32)[:embedding_dim]

    return embedding_matrix

# DEFINICIÓN DE LA RED NEURONAL

model = keras.Sequential([
          keras.layers.Embedding(input_dim=vocab_size,
                                 output_dim=50,
                                 weights=[create_embedding_matrix(
                                                'glove.6B.50d.txt',
                                                 word_index,
                                                 50)],
                                 input_length=text_len,
                                 trainable=True),
          keras.layers.Conv1D(50, 6, activation='relu'),
          keras.layers.GlobalAveragePooling1D(),
          keras.layers.Dense(6),
          keras.layers.Activation('relu'),
          keras.layers.Dense(3, activation=tf.nn.softmax)
])

# Imprime una descripción de la estructura definida de la red neuronal

model.summary()

# Establece una serie de parámetros necesarios para la red neuronal
#    · Optimizer: indica la estrategia a seguir para ajustar los parámetros
#                 de la red neuronal.
#    · Loss: función de pérdida que se debe minimizar lo máximo posible.
#    · Metrics: función de precisión

model.compile(optimizer='adam',
              loss='sparse_categorical_crossentropy',
              metrics=['acc'])

# ENTRENAMIENTO Y TESTING

# Dividimos los datos de entrenamiento y en dos listas: una para entrenar a
# la red neuronal y otra para testear su correcto funcionamiento

x_val           = train_data[ :len(train_data)//2]
partial_x_train = train_data[len(train_data)//2: ]

# Dividimos los resultados (valor esperado) en dos listas: una para entrenar
# a la red neuronal y otra para testear su correcto funcionamiento

y_val           = train_labels[ :len(train_labels)//2]
partial_y_train = train_labels[len(train_labels)//2: ]

# ENTRENAMIENTO DEL MODELO DE RED NEURONAL [TODO]
#   · Epochs (épocas): número de iteraciones sobre todos los datos que permite
#                ajustar la red neuronal.
#   · Batch size: cada cuantos elementos se realiza un ajuste sobre la red
#                neuronal.
#   · Verbose: si es 1, muestra el progreso de la red neuronal.

history = model.fit(partial_x_train,
                    partial_y_train,
                    epochs=125,
                    batch_size=200,
                    validation_data=(x_val, y_val),
                    verbose=1)

results = model.evaluate(test_data, test_labels)

print(results)
```

```
history_dict = history.history

# REPRESENTACIÓN GRÁFICA DE LOS DATOS OBTENIDOS

acc = history_dict['acc']             # Grado de acierto con datos de entrenamiento
val_acc = history_dict['val_acc']     # Grado de precisión con datos de validación
loss = history_dict['loss']           # Grado de pérdida con datos de entrenamiento
val_loss = history_dict['val_loss']   # Grado de pérdida con datos de validación

epochs = range(1, len(acc) + 1)

# Imprimiendo gráfica de pérdida

plt.plot(epochs, loss, 'bo', label='Training loss')
plt.plot(epochs, val_loss, 'b', label='Validation loss')
plt.title('Training and validation loss')
plt.title('Training and validation loss')
plt.xlabel('Epochs')
plt.ylabel('Loss')
plt.legend()

plt.show()

# Imprimiendo gráfica de aciertos (precisión)

plt.plot(epochs, acc, 'bo', label='Training acc')
plt.plot(epochs, val_acc, 'b', label='Validation acc')
plt.title('Training and validation accuracy')
plt.xlabel('Epochs')
plt.ylabel('Accuracy')
plt.legend()

plt.show()

# Salvamos el modelo

model.save("model.h5")

# Salvamos el indexado de palabras para su posterior uso.

with open('word_index.pkl', 'wb') as f:
    pickle.dump(word_index, f, pickle.HIGHEST_PROTOCOL)

print("Saved model to disk")
```

4.3. Usando la red neuronal

Tras guardar la estructura y parámetros de la red neuronal creada y entrenada anteriormente, se podrá cargar de nuevo dichos datos recreando el modelo neuronal. Esto se realiza en el script **classificator.py**. El código quedaría de la siguiente forma:

```
from tensorflow import keras
from source_data import SourceData
import pickle

# Cargamos la red neuronal ya entrenada y la usamos para evaluar y inferir
# cuál es la clasificación ontológica de un texto

model = keras.models.load_model("model.h5")
print("Loaded model from disk")
model.compile(optimizer='adam', loss='sparse_categorical_crossentropy',
metrics=['acc'])
```

```
# Cargamos el indexado de palabras

word_index = []
with open('word_index.pkl', 'rb') as f:
    word_index = pickle.load(f)

# Textos a clasificar. Deben estar en inglés porque es el lenguaje con el
# que la red neuronal ha sido entrenado.

data_to_evaluate = [

    ...
    ["Texto a evaluar", "Salida esperada"],
    ...
]

for i in range(len(data_to_evaluate) - 1):
    text = data_to_evaluate[i][0]
    expected_result = data_to_evaluate[i][1]

    vectorized_text = SourceData.vectorize_text(word_index, text)

    data_preprocessed =
             keras.preprocessing.sequence
                  .pad_sequences([vectorized_text, ],
                                 value=word_index["<PAD>"],
                                 padding='post',
                                 maxlen=256)

    final_result = model.predict_classes(data_preprocessed)

    # 0 is a work and 1 is a person

    print("Text" + str(i) + " - Value expected: " + expected_result +
          "; Value returned: " + str(final_result[0]) + "\n")
```

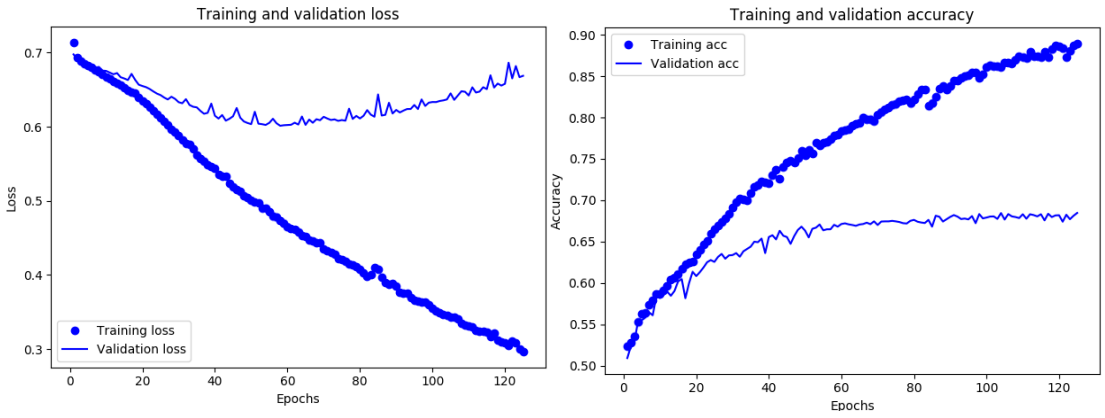
5. Resultados de las pruebas

Para la realización de las pruebas hemos considerado modificar una serie de variables en función de la tecnología con la que estamos trabajamos.

5.1. Modificación la longitud de texto

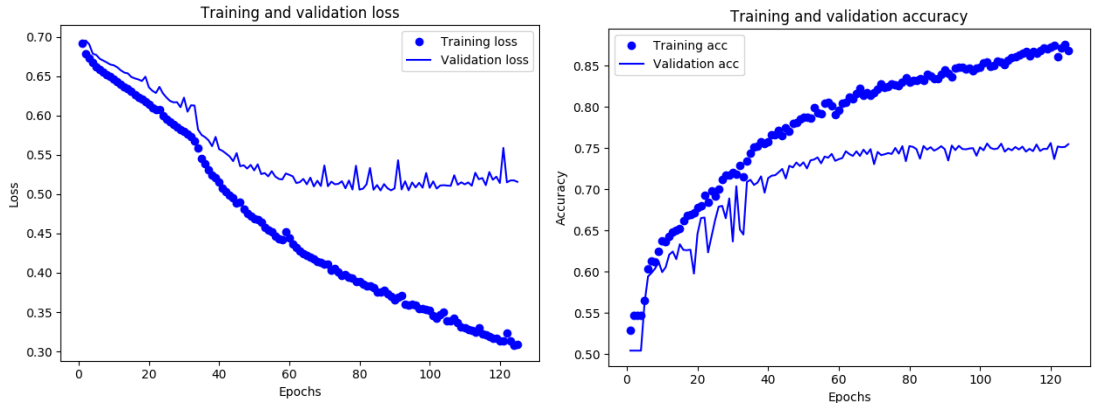
Cuando modificamos en el script **classification.py** la variable text_len, estamos alterando la cantidad de palabras con las que la red neuronal va a aprender de cada caso de ejemplo (texto).

Para text_len = 25, epoch = 125 y batch_size = 500



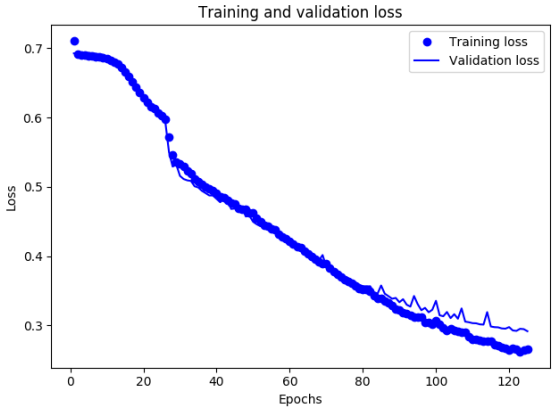

```
Epoch 125/125
5000/5000 [==============================] - 1s 135us/sample - loss: 0.2963 - acc: 0.8896 - val_loss: 0.6688 - val_acc: 0.6846
10000/10000 [==============================] - 0s 37us/sample - loss: 0.6754 - acc: 0.6804
```

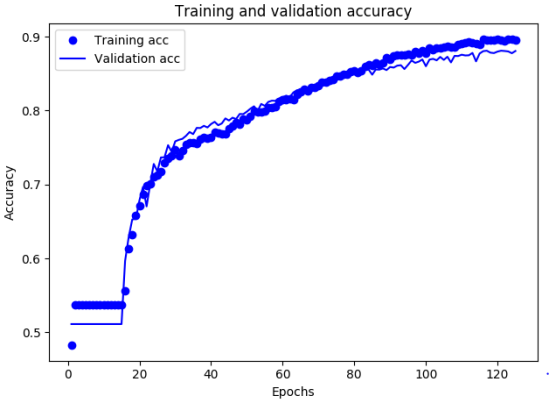
Para text_len = 50, epoch = 125 y batch_size = 500




```
Epoch 125/125
5000/5000 [==============================] - 1s 159us/sample - loss: 0.3094 - acc: 0.8680 - val_loss: 0.5156 - val_acc: 0.7546
10000/10000 [==============================] - 1s 92us/sample - loss: 0.5368 - acc: 0.7438
```

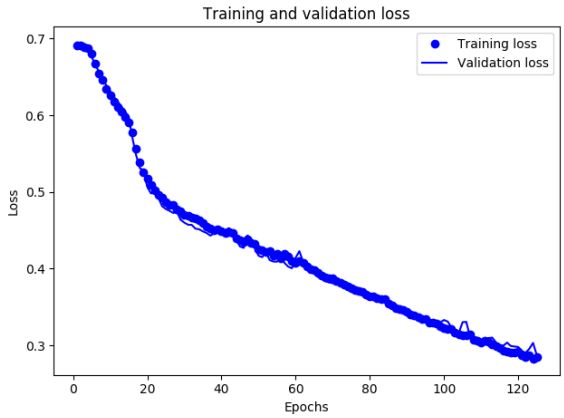
Para text_len = 256, epoch = 125 y batch_size = 500

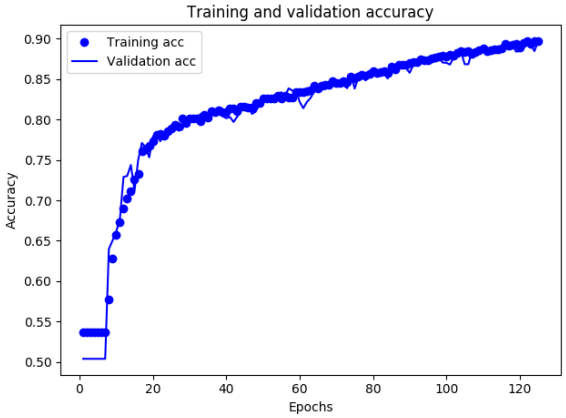





```
Epoch 125/125
5000/5000 [==============================] - 2s 448us/sample - loss: 0.2745 - acc: 0.8906 - val_loss: 0.3060 - val_acc: 0.8766
10000/10000 [==============================] - 1s 137us/sample - loss: 0.3328 - acc: 0.8596
```

Para text_len = 512, epoch = 125 y batch size = 500



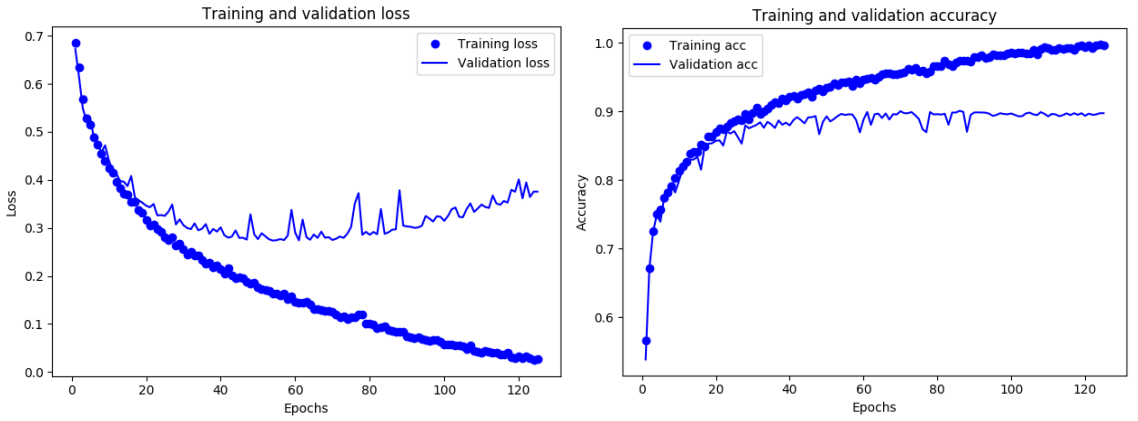



```
Epoch 125/125
5000/5000 [==============================] - 4s 823us/sample - loss: 0.2841 - acc: 0.8972 - val_loss: 0.2873 - val_acc: 0.8964
10000/10000 [==============================] - 1s 144us/sample - loss: 0.3164 - acc: 0.8728
```

5.2. Modificando el batch_size

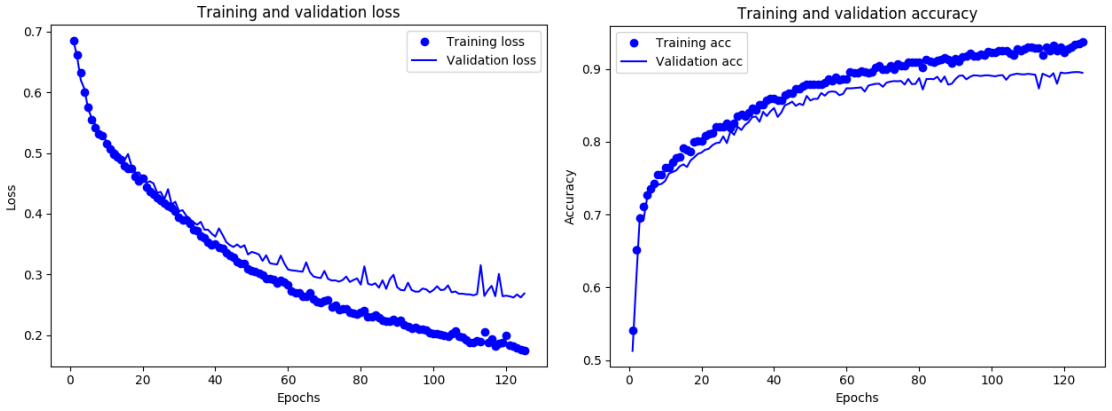
Cuando modificamos en el script **classification.py** la variable batch_size, modificamos cada cuantos elementos se realiza un ajuste sobre la red neuronal.

Para text_len = 256, epoch = 125 y batch size = 50



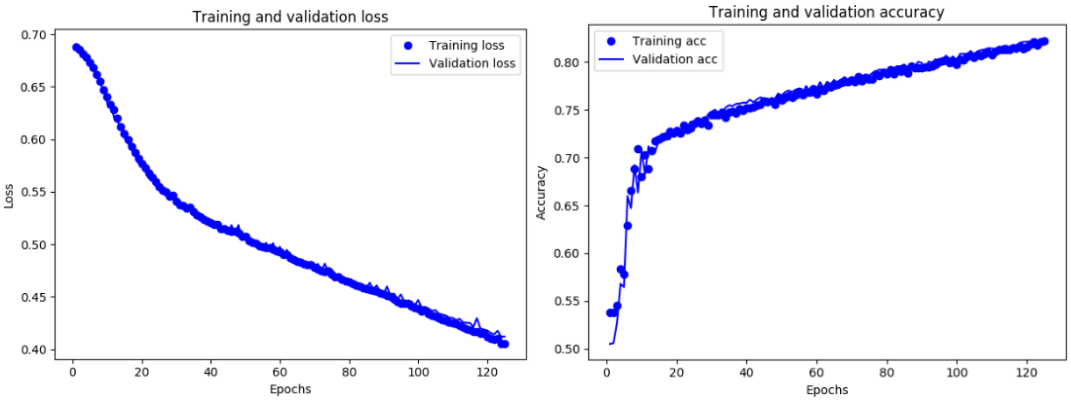

```
Epoch 125/125
5000/5000 [==============================] - 7s 1ms/sample - loss: 0.0261 - acc: 0.9966 - val_loss: 0.3755 - val_acc: 0.8974
10000/10000 [==============================] - 1s 95us/sample - loss: 0.3772 - acc: 0.8983
```

Para text_len = 256, epoch = 125 y batch size = 200



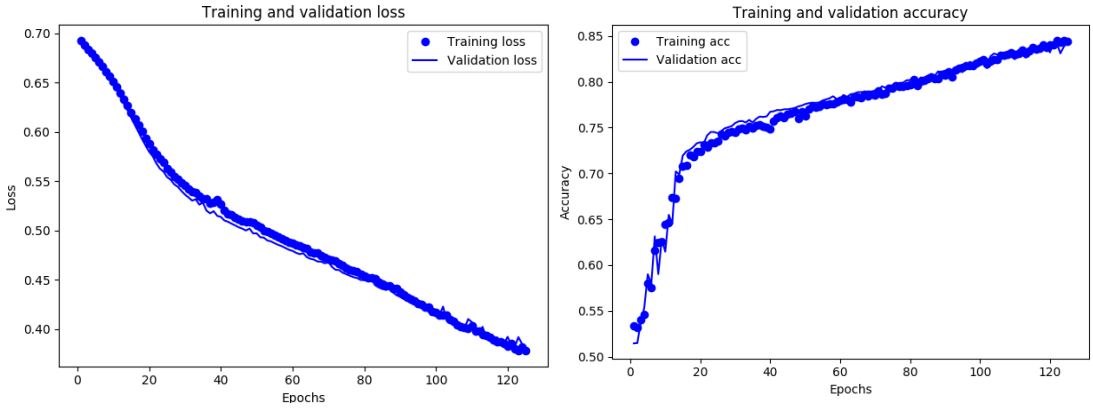

```
Epoch 125/125
5000/5000 [==============================] - 3s 594us/sample - loss: 0.1744 - acc: 0.9376 - val_loss: 0.2687 - val_acc: 0.8948
10000/10000 [==============================] - 1s 136us/sample - loss: 0.2611 - acc: 0.9009
```

Para text_len = 256, epoch = 125 y batch_size = 1000



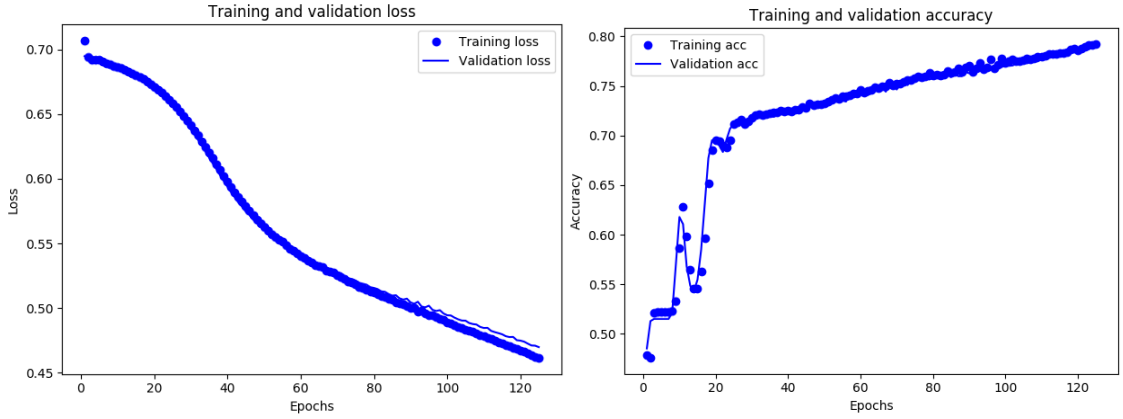

```
Epoch 125/125
5000/5000 [==============================] - 2s 403us/sample - loss: 0.4052 - acc: 0.8220 - val_loss: 0.4122 - val_acc: 0.8234
10000/10000 [==============================] - 1s 138us/sample - loss: 0.4447 - acc: 0.7968
```

Para text_len = 256, epoch = 125 y batch size = 1500




```
Epoch 125/125
5000/5000 [==============================] - 2s 389us/sample - loss: 0.3781 - acc: 0.8436 - val_loss: 0.3799 - val_acc: 0.8420
10000/10000 [==============================] - 1s 90us/sample - loss: 0.4139 - acc: 0.8216
```

Para text_len = 256, epoch = 125 y batch size = 2500

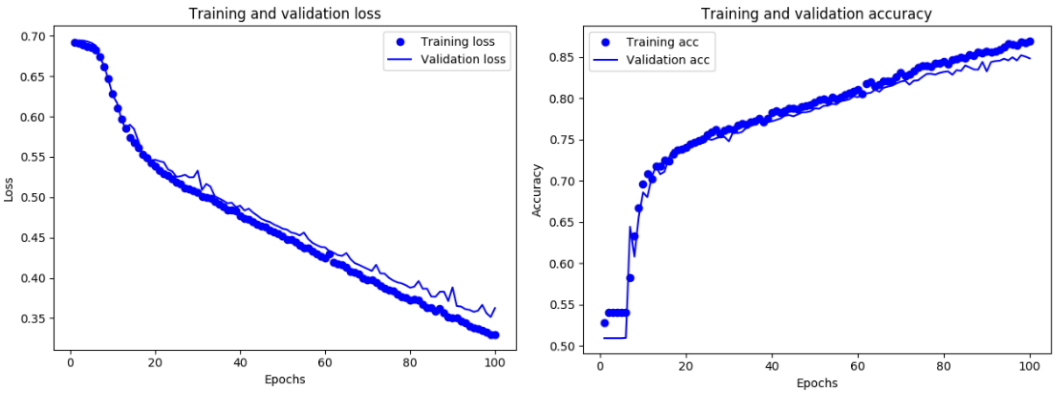



```
Epoch 125/125
5000/5000 [==============================] - 2s 382us/sample - loss: 0.4612 - acc: 0.7924 - val_loss: 0.4698 - val_acc: 0.7902
10000/10000 [==============================] - 1s 89us/sample - loss: 0.4855 - acc: 0.7727
```

5.3. Modificar el número de épocas

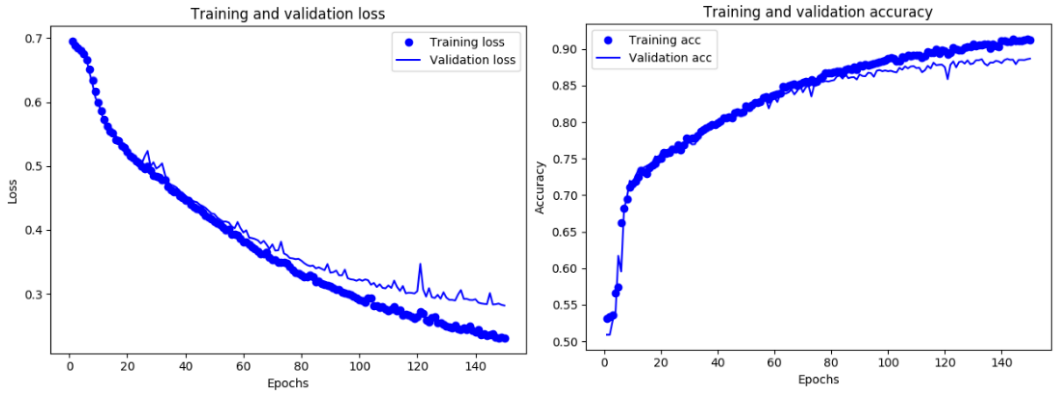
Cuando modificamos en el script **classification.py** la variable epoch, modificamos el número de veces que se itera sobre el conjunto completo de datos de entrenamiento y validación.

Para text_len = 256, epoch = 100 y batch size = 500



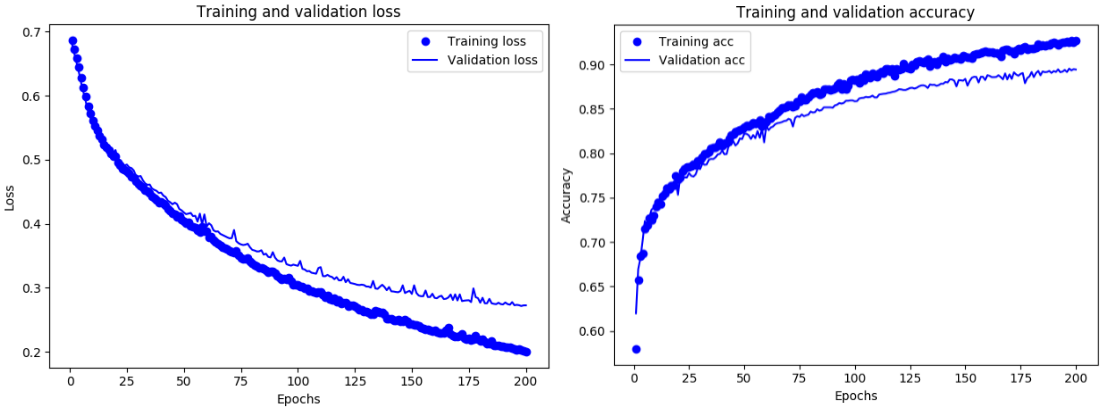

```
Epoch 100/100
5000/5000 [==============================] - 2s 441us/sample - loss: 0.3292 - acc: 0.8696 - val_loss: 0.3626 - val_acc: 0.8484
10000/10000 [==============================] - 1s 82us/sample - loss: 0.3987 - acc: 0.8205
```

Para text_len = 256, epoch = 150 y batch_size = 500



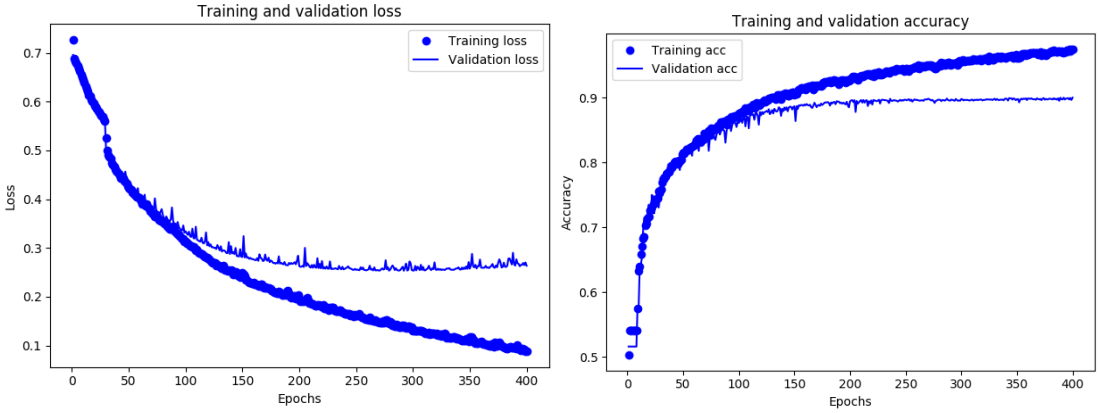

```
Epoch 150/150
5000/5000 [==============================] - 3s 543us/sample - loss: 0.2307 - acc: 0.9128 - val_loss: 0.2819 - val_acc: 0.8868
10000/10000 [==============================] - 1s 98us/sample - loss: 0.2914 - acc: 0.8797
```

Para text_len = 256, epoch = 200 y batch_size = 500




```
Epoch 200/200
5000/5000 [==============================] - 3s 514us/sample - loss: 0.1997 - acc: 0.9272 - val_loss: 0.2725 - val_acc: 0.8944
10000/10000 [==============================] - 1s 101us/sample - loss: 0.2774 - acc: 0.8899
```

Para text_len = 256, epoch = 400, batch_size = 500



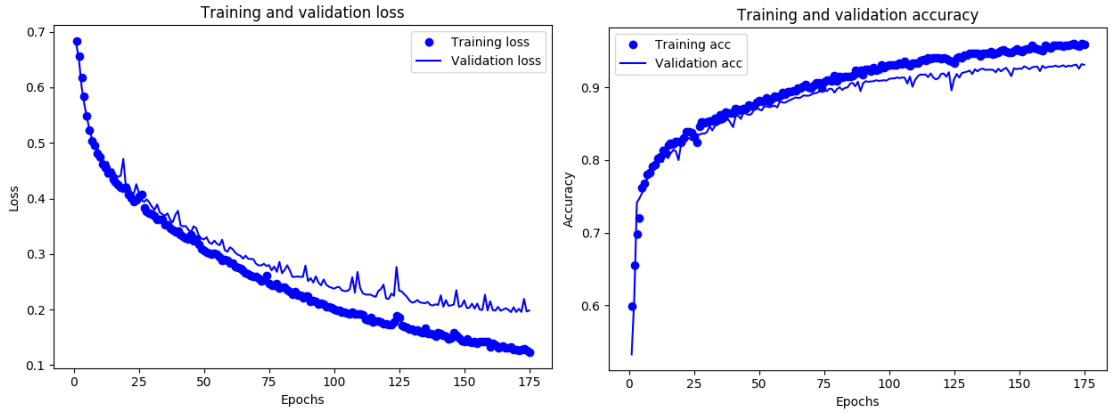

```
Epoch 400/400
5000/5000 [==============================] - 2s 477us/sample - loss: 0.0877 - acc: 0.9744 - val_loss: 0.2635 - val_acc: 0.9002
10000/10000 [==============================] - 1s 99us/sample - loss: 0.2632 - acc: 0.9016
```

5.4. Escogiendo los valores óptimos de entrenamiento

Una vez realizadas las pruebas de aprendizaje de la red neuronal y mediante la observación de las gráficas obtenidas al modificar los valores de longitud del texto, épocas y tamaño de lote de ejemplos para el ajuste de la red neuronal, podemos determinar que los valores óptimos para nuestro problema son:

Valores óptimos	
Longitud del texto	512
Épocas	175
Tamaño de lote de ejemplos	200

Para text_len = 512, epoch = 175, batch_size = 200




```
Epoch 175/175
5000/5000 [==============================] - 5s 1ms/sample - loss: 0.1227 - acc: 0.9590 - val_loss: 0.1983 - val_acc: 0.9308
10000/10000 [==============================] - 1s 149us/sample - loss: 0.2188 - acc: 0.9183
```

Viendo los resultados obtenidos, podemos concluir que **el algoritmo presenta unos resultados bastante aceptables acercándose a un nivel de acierto superior al 90 % sin cometer *overfitting***.

6. Conclusiones

En lugar de reescribir la web para que las computadoras puedan entender, estamos haciendo que los computadores sean inteligentes y les estamos dando poder para obtener **conocimiento**. Y esa es una gran diferencia porque estamos agregando información de todo el mundo y aprendiendo lo que significa. Y de eso se trata la Web 4.0.

Y esto cambia la web semántica tal y como la conocemos. Hoy en día, no tenemos claro cuál será el futuro de la web semántica. Lo más probable es que la web semántica sea definida por una inteligencia artificial o por un conjunto de inteligencias artificiales de forma automática en lugar de realizarlo de forma manual tal y como se hace actualmente.

Varios grandes jugadores han decidido unir esfuerzos y crear redes neuronales, Word2Vecs, redes bayesianas y muchas otras herramientas para que los computadores aprendan.

Google se está posicionando como una compañía de inteligencia artificial, Nvidia está enfocando sus GPU para la inteligencia artificial, al igual que Microsoft y Facebook van en esta dirección.

En resumen, la web semántica fue una gran idea y aún lo es. Pero no le vemos un futuro en su estado actual. Necesita evolucionar e integrar sus ideas con inteligencia artificial vinculando datos sin requerir una completa reescritura de lo que ya tenemos.

7. Bibliografía

- **Introduction to the Semantic Web**: https://slideplayer.com/slide/5215612/
- **Historia de la web – Universidad Marcelino Champahnat**: http://umch.edu.pe/arch/hnomarino/74_Historia%20de%20la%20Web.pdf
- **El impredecible futuro de la web semántica:** https://www.humanlevel.com/articulos/desarrollo-web/el-futuro-de-la-web-semantica.html
- **La ontología y la Web Semántica – Bibliopos**: http://www.bibliopos.es/Biblion-A2-Bibliografia-Documentacion/18ontologia-Web-Semantica.pdf
- **¿Qué es el RDF? Introducción al RDF:** https://www.seofreelance.es/que-es-rdf-introduccion-a-rdf/
- **Web 1.0, Web 2.0 y Web 3.0 – Estudio Seijo**: http://www.estudioseijo.com/noticias/web-10-web-20-y-web-30.htm
- **¿Qué es la web 4.0? – Nobbot**: https://www.nobbot.com/general/que-es-la-web-4-0/
- **Web 4.0 – cuando no haga falta preguntar al buscador:** https://www.paradigmadigital.com/portfolio/web-4-0/
- **Web 4.0: el próximo desafío ya está aquí – Profile Software Services**: https://profile.es/blog/web-4-0-el-proximo-desafio-ya-esta-aqui/
- **Evolución cronológica de la Web 1.0, 2.0, 3.0 y 4.0**: https://www.timetoast.com/timelines/evolucion-web-1-0-2-0-3-0-4-0
- **Web 4.0 – cuando no haga falta preguntar al buscador**: https://www.paradigmadigital.com/portfolio/web-4-0/
- **¿Qué es un buscador semántico? – Natzir Turrado**: https://www.analistaseo.es/posicionamiento-buscadores/buscador-semantico/
- **Descripción de la web semántica – Universidad de la Rioja**: https://dialnet.unirioja.es/descarga/articulo/4786666.pdf
- **¿Qué es el marketing digital o marketing online?**: http://www.mailclick.com.mx/que-es-el-marketing-digital-o-marketing-online/#web-3-semantica
- **Sentiment Analysis: nearly everything you need to know – MonkeyLearn**: https://monkeylearn.com/sentiment-analysis/
- **Imagen web 1.0**: https://www.hazhistoria.net/sites/default/files/web1_0.png
- **Imagen web 2.0**: https://www.hazhistoria.net/sites/default/files/web2_0.png
- **Imagen web 3.0**: https://www.hazhistoria.net/sites/default/files/web3_0.png
- **NVIDIA Accelerates Real Time Speech to Text Transcription 3500x with Kaldi**: https://devblogs.nvidia.com/nvidia-accelerates-speech-text-transcription-3500x-kaldi/
- Imagen Speech to Text:
- **Machine Learning & Open Source Speech-to-text Engine Development Project**: https://research.mozilla.org/machine-learning/

- **Los 2 tipos de aprendizaje en Machine Learning - supervisado y no supervisado**: https://empresas.blogthinkbig.com/que-algoritmo-elegir-en-ml-aprendizaje/
- **Learning a Manifold of Fonts – UCL – Department of Computer Science:** http://vecg.cs.ucl.ac.uk/Projects/projects_fonts/projects_fonts.html
- **¿Qué es el Aprendizaje Supervisado y no Supervisado? |DotCSV:** https://www.youtube.com/watch?v=oT3arRRB2Cw
- **¿Aprendizaje supervisado o no supervisado? – Juan Zambrano – Medium:** https://medium.com/@juanzambrano/aprendizaje-supervisado-o-no-supervisado-39ccf1fd6e7b
- **Red neuronal artificial – Wikipedia**: https://es.wikipedia.org/wiki/Red_neuronal_artificial
- **¿Qué es y cómo funciona el deep learning? – Rubén López:** https://rubenlopezg.wordpress.com/2014/05/07/que-es-y-como-funciona-deep-learning/
- **The evolution and Core Concepts of Deep Learning & Neural Networks – Analytics Vidhya**: https://www.analyticsvidhya.com/blog/2016/08/evolution-core-concepts-deep-learning-neural-networks/
- **¿Qué es el aprendizaje profundo (deep learning)**? – Definición en WhatIs.com: https://searchdatacenter.techtarget.com/es/definicion/Aprendizaje-profundo-deep-learning
- **¿Qué es el Machine Learning? ¿Y el Deep Learning? – DotCSV:** https://www.youtube.com/watch?v=KytW151dpqU
- **Semantic Web is Dead, Long live the AI!!! – José Cabeda, Hackernoon – Medium**: https://hackernoon.com/semantic-web-is-dead-long-live-the-ai-2a5ea0cf6423
- **Learning SPARQL – Bob DuCharme – O'Reilly**: https://the-eye.eu/public/Books/IT%20Various/learning_sparql.pdf
- **Las Redes Neuronales… ¿Aprenden o memorizan? – Overfitting y underfitting – Dot CSV**: https://www.youtube.com/watch?v=7-6X3DTt3R8&t=3s
- **Cómo identificar el Overfitting en tu Red neuronal – Dot CSV:** https://www.youtube.com/watch?v=7-6X3DTt3R8&t=3s
- **Riesgos del sobreajuste – Curso intensivo de aprendizaje automático – Google**: https://developers.google.com/machine-learning/crash-course/generalization/peril-of-overfitting?hl=es-419

Código

- **Keras Official Site**: https://keras.io/
- **Tensorflow Official Site:** https://www.tensorflow.org/
- **Practical Text Classification with Python and Keras – Real Python**: https://realpython.com/python-keras-text-classification/
- **How to Save and Load Your Keras Deep Learning Model:** https://machinelearningmastery.com/save-load-keras-deep-learning-models/
- **Train your first neural network -** basic classification | TensorFlow Core: https://www.tensorflow.org/tutorials/keras/basic_classification