DOBLE GRADO EN
MATEMÁTICAS Y ESTADÍSTICA

TRABAJO FIN DE GRADO

DEEP LEARNING

Alba Centeno Franco

Sevilla, Julio 2019

Índice general

Resumen III
Abstract IV
Índice de Figuras V
Índice de Tablas VII

1. Introducción **1**
1.1. Inteligencia Artificial y Aprendizaje Automatico 1
1.2. ¿Qué es Deep Learning? 3

2. Redes Neuronales **5**
2.1. Introducción 5
2.2. Redes Neuronales Artificiales 6
2.3. Perceptrón simple 9
2.4. Perceptrón múltiple 11
2.4.1. Propagación de los patrones de entrada 13
2.4.2. Fase de aprendizaje 14
2.4.2.1. Regla de aprendizaje 14

3. Deep Learning **19**
3.1. Optimalidad global Deep Learning 20
3.1.1. El reto de las funciones no convexas en las redes neuronales . . . 20
3.1.2. Estabilidad geométrica en el Deep Learning 21
3.2. Error de generalización 22
3.3. Limitaciones del Deep Learning 22
3.3.1. Problemas derivados del sobreaprendizaje 22
3.3.2. Las redes neuronales como cajas negras 24

4. Redes Deep Learning **27**
4.1. Tipos de redes Deep Learning 27
4.2. Redes Neuronales Recurrentes (RNN) 34
4.2.0.1. Backpropagation a través del tiempo 36
4.2.1. Redes Recurrentes Bidireccionales 36
4.2.2. LSTM 37
4.2.3. El futuro de las RNNs 38

5. Deep Learning y Procesamiento del Lenguaje Natural **39**
5.1. ¿Qué es el Procesamiento del Lenguaje Natural? 39
5.2. Análisis de sentimientos 39
5.2.1. Aplicación 40

5.2.1.1. Conjunto de datos 40
5.2.1.2. Incrustaciones 41
5.2.1.2.1. Factorización de matriz global 41
5.2.1.2.2. Probabilidades de co-ocurrencia 42
5.2.1.3. Aplicación en Python 47
5.2.1.3.1. Introducción a Python 47
5.2.1.3.2. Librerías utilizadas 48
5.2.1.3.3. Aplicación 50

Bibliografía **59**

Resumen

Desde hace unos años es incuestionable que la Inteligencia Artificial avanza a pasos agigantados. Es especialmente destacable el rápido desarrollo que ha experimentado la rama de la Inteligencia Artificial denominada Aprendizaje Profundo (Deep Learning), que hace uso de una metodología de aprendizaje similar a la que emplean los seres humanos, permitiendo la existencia de estructuras especializadas en la detección de determinadas características ocultas en los datos. El análisis de este campo será objeto de este Trabajo Fin de Grado, partiendo de sus fundamentos hasta abarcar sus arquitecturas más complejas.

El trabajo se inicia con una introducción a las redes de neuronas artificiales, por ser la base sobre la que se asienta el concepto de Aprendizaje Profundo. Seguidamente se presentan los fundamentos de esta metodología, para pasar a continuación a analizar las principales arquitecturas de redes en este contexto. La parte final de este trabajo se centra en un caso particular de estas arquitecturas, como son las Redes Recurrentes y, más concretamente, LSTM, donde se presenta una aplicación en lenguaje Python sobre análisis de sentimientos, que permite asignar un Emoji a una frase.

Abstract

For some years it is no doubt that Artificial Intelligence is advancing by leaps and bounds. Particularly noteworthy is the rapid development of the Artificial Intelligence branch called Deep Learning, which utilizes a learning methodology similar to that used by human beings, allowing the existence of structures specialized in the detection of certain characteristics hidden in the data. The analysis of this field will be the subject of this Final Degree Project, starting from its foundations to include its most complex architectures.

This work begins with an introduction to the artificial neural networks, as the basis on which the concept of Deep Learning is based. Then, the foundations of this methodology are presented, to then the analysis of the main network architectures in this context is carried out. The final part of this work focuses on a specific case of those architectures, namely Recurrent Networks and, more specifically, LSTM, and an application in Python language on feelings analysis, which allows us to assign an Emoji to a phrase, is presented.

Índice de figuras

1.1. Deep Learning como subcampo de la Inteligencia Artificial 2

2.1. Estructura general de una neurona 5
2.2. Red Artificial 7
2.3. Red Neuronal Artificial 7
2.4. Funciones de Activación 8
2.5. Perceptrón simple 9
2.6. Clases linealmente separables 10
2.7. Perceptrón Múltiple 12

4.1. Evolución de las redes utilizadas en Deep Learning 27
4.2. Tipos de nodos 28
4.3. RNN 29
4.4. LSTM 29
4.5. GRU 30
4.6. CNN 31
4.7. Convolución CNN 32
4.8. Convolución DBN 33

5.1. Emoji y etiqueta 40
5.2. Matrices 42

Índice de tablas

4.1. Ejemplos de aplicaciones de redes Deep Learning 28

5.1. Ejemplo conjunto de datos 40
5.2. Probabilidades de co-ocurrencia de palabras objetivo de hielo y vapor con palabras de contexto 43

Capítulo 1

Introducción

1.1. Inteligencia Artificial y Aprendizaje Automatico

La inteligencia artificial (IA) nació a principio del siglo pasado, de la mano de algunos pioneros del reciente campo de las ciencias de la computación. Estos pioneros entonces comenzaron a preguntarse acerca de poder lograr que los ordenadores "pensaran". Por lo tanto, una definición concisa acerca del campo de la IA sería: el esfuerzo de automatizar tareas intelectuales normalmente realizadas por humanos. Como tal, IA es un campo general que contiene al Aprendizaje Automático (AA) y al Aprendizaje Profundo, pero que también incluye otro tipo de subcampos que no necesariamente involucran "Aprendizaje" como tal.

Durante mucho tiempo, muchos expertos creyeron que la IA con nivel humano podía alcanzarse al hacer que los programadores crearan a mano un conjunto de reglas lo suficientemente grande para manipular el conocimiento y así generar máquinas inteligentes. Este enfoque es conocido como IA simbólica, y fue el paradigma que dominó el campo de la IA desde 1950 hasta finales de la década de los 80 y alcanzó su máxima popularidad durante el boom de los Sistemas Expertos en 1980.

Aunque la IA simbólica demostró ser adecuada para resolver problemas lógicos y bien definidos, como jugar al ajedrez, resultó intratable el encontrar reglas explícitas para resolver problemas mucho más complejos, como la clasificación de imágenes, el reconocimiento de voz y la traducción entre lenguajes naturales (como español, inglés, etc., diferentes a los no naturales como los lenguajes de programación). Un nuevo enfoque surgió entonces para tomar el lugar de la IA simbólica: el Aprendizaje Automático (AA).

El Aprendizaje Automático (del inglés, *Machine Learning*) es la rama de la Inteligencia Artificial que tiene como objetivo desarrollar técnicas que permitan a los ordenadores aprender. De forma más concreta, se trata de crear algortimos capaces de generalizar comportamientos y reconocer patrones a partir de una información suministrada en forma de ejemplos. Es, por lo tanto, un proceso de inducción del conocimiento, es decir, un método que permite obtener por generalización un enunciado general a partir de enunciados que describen casos particulares.

En 1950, Alan Tuning, introdujo el test de Turing, y llegó a la conclusión de que ordenadores de propósito general podrían ser capaces de "aprender" y "ser originales". El Aprendizaje Automático surgió entonces de preguntas como:

¿Puede un ordenador ir mas allá de lo que le ordenamos hacer y aprender por si mismo cómo realizar una tarea específica? ¿Podría un ordenador sorprendernos? Y, en vez de programadores especificando regla por regla cómo procesar datos, ¿podría un ordenador automáticamente aprender esas reglas directamente de los datos que le pasamos?

Las preguntas anteriores abrieron la puerta a un nuevo paradigma de programación. A diferencia del clásico paradigma de la IA simbólica, donde humanos proporcionan reglas (un programa) y datos para ser procesados acorde con dichas reglas, para así obtener respuestas a la salida del programa, con el Aprendizaje Automático, los humanos suministramos los datos como entrada al igual que las respuestas esperadas de dichos datos con el fin de obtener a la salida las reglas que nos permiten hacer la correspondencia efectiva entre entradas y sus correspondientes salidas. Estas reglas pueden ser luego aplicadas a nuevos datos para producir respuestas originales, es decir, generadas automáticamente por las reglas que el sistema "aprendió" y no por reglas explícitamente codificadas por programadores.

Aunque el Aprendizaje Automático empezó a tenerse en cuenta desde los años 90 del siglo pasado, este se ha convertido en el más popular y exitoso de los subcampos de la IA, una tendencia guiada por la disponibilidad de mejor hardware y conjuntos de datos enormes.Dentro del AA nos encontramos con una técnica particular de reciente aparición llamada Aprendizaje Profundo o *Deep Learning*, sobre el que versa este Trabajo de Fin de Grado. Por otra parte, el AA, y en especial el Aprendizaje Profundo, muestra poca teoría matemática, en comparación con el campo de la Estadística. Es decir, el AA es una disciplina aplicada, en la cual las ideas son probadas habitualmente de forma empírica más que de forma teórica.

El AA está fuertemente relacionado con la Estadística Matemática. Sin embargo, difiere de la Estadística en varios aspectos. Por ejemplo, el AA trabaja habitualmente con grandes y complejos conjuntos de datos (los cuales pueden contener, por ejemplo, millones de imágenes, cada una de ellas con miles de pixeles) para lo cual el análisis estadístico clásico no podría ser utilizado.

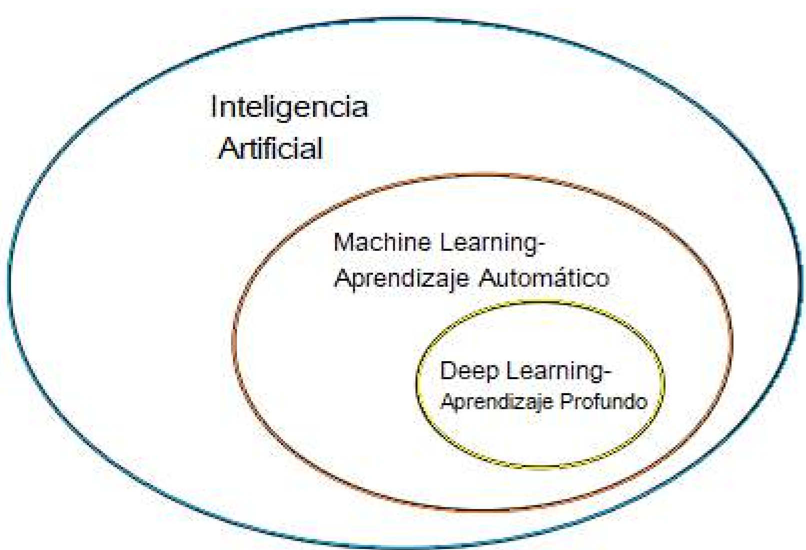


Figura 1.1: Deep Learning como subcampo de la Inteligencia Artificial

Los problemas que se pueden resolver a través del AA son básicamente de dos clases:

- Aprendizaje supervisado (*Supervised Learning*)
- Aprendizaje no supervisado (*Unsupervised Learning*)

Hablamos de aprendizaje supervisado cuando disponemos de un conjunto de datos y sabemos para esos datos cuál es la respuesta correcta. Lo que no sabemos es cómo llegar de los datos a la respuesta. En el caso de aprendizaje no supervisado lo que tenemos es un conjunto de datos, y lo que buscamos es extraer información de los mismos, pero sin que nadie le haya dicho al algoritmo qué es lo que esperamos encontrar. Podríamos decir que estos son los extremos, puede haber matices donde conocemos parte de los valores esperados, y en ese caso tendríamos aprendizaje supervisado parcial (*Partially Supervised Learning* o *Semisupervised learning*), pero nos vamos a concentrar en las categorías principales.

Dentro de la categoría de aprendizaje supervisado podemos distinguir dos tipos de problemas: regresión y clasificación. El objetivo de la Regresión es predecir un resultado que varía en un rango continuo a partir de la información proporcionada por una serie de variables predictoras. Por ejemplo, dada la fotografía de una persona determinar su edad. Otro ejemplo podría ser, dadas las características de una vivienda que está en venta, predecir cuál es el precio adecuado de venta. Por otra parte, la Clasificación se centra en predecir resultados que son discretos. Por ejemplo, dado un paciente con un tumor, determinar si el tumor es maligno o no. Otro problema que entra en esta categoría es, por ejemplo, dada la imagen de un número de un dígito escrito a mano, identificar de qué dígito se trata.

En el aprendizaje no supervisado buscamos identificar estructuras en los datos. No tenemos la respuesta conocida para cada caso, por lo que es necesario que el algoritmo busque las relaciones entre las variables involucradas. Un ejemplo de esto puede visualizarse así: dado un conjunto de genes, encontrar una forma de agruparlos de acuerdo a las relaciones entre ellos. Una de las técnicas más usadas de aprendizaje no supervisado es la de *Clustering*, donde el algoritmo agrupa los valores de entrada según determinado criterio. Esto se puede usar, por ejemplo, para sugerir productos relacionados en un sitio o aplicación de venta de productos. En este caso los clusters pueden ser los productos relacionados, que se usan para sugerirle al usuario que está viendo o compró un producto determinado. Otra técnica que se puede utilizar es la de *Anomaly Detection*, donde el algoritmo aprende a partir de los datos de entrenamiento cuáles son los valores "normales" de ciertos parámetros, y luego de entrenado puede indicar si algún elemento cae fuera de los rangos normales. Este tipo de algoritmos se puede usar por ejemplo para detectar posibles fraudes en el uso de tarjetas de crédito, o detectar un funcionamiento anómalo en determinada maquinaria.

1.2. ¿Qué es Deep Learning?

El Aprendizaje Profundo o Deep Learning, es un subcampo de Machine Learning como se comentó anteriormente que usa una estructura jerárquica de redes neuronales artificiales, que se construyen de una forma similar a la estructura neuronal del cerebro humano, con los nodos de neuronas conectadas como una tela de araña. Esta arquitectura permite abordar el análisis de datos de forma no lineal.

El aprendizaje profundo es una técnica que, al igual que otros algoritmos de aprendizaje, enseña a los ordenadores a hacer lo que es natural para los humanos: aprender con el ejemplo. El aprendizaje profundo es una tecnología clave detrás de los vehículos sin conductor, que les permite reconocer una señal de tráfico o distinguir un peatón de un farola. Es la clave para el control por voz en dispositivos de consumo como teléfonos, tabletas, televisores y altavoces inteligentes. El aprendizaje profundo está recibiendo mucha atención últimamente y es por una buena razón: está logrando resultados que antes no eran posibles.

El aprendizaje profundo puede aprender a realizar tareas de clasificación directamente desde imágenes, texto o sonido. Los modelos se entrenan utilizando un gran conjunto de datos etiquetados y arquitecturas de redes neuronales que contienen muchas capas.

Los modelos de aprendizaje profundo pueden lograr una precisión de vanguardia, que a veces supera el rendimiento a nivel humano. El aprendizaje profundo logra una precisión de reconocimiento en niveles más altos que nunca. Esto ayuda a que los productos electrónicos de consumo cumplan con las expectativas del usuario, y es crucial para las aplicaciones críticas para la seguridad como las presentes en los automóviles sin conductor.

Si bien el aprendizaje profundo se teorizó por primera vez en la década de 1980, hay dos razones principales por las que solo recientemente ha cobrado gran importancia.

- El aprendizaje profundo requiere grandes cantidades de datos etiquetados. Por ejemplo, el desarrollo de automóviles sin conductor requiere millones de imágenes y miles de horas de vídeo.

- El aprendizaje profundo requiere una gran potencia de cálculo. Las GPU (*Graphics Processing Unit*) de alto rendimiento tienen una arquitectura paralela que es eficiente para el aprendizaje profundo. Cuando se combina con clústeres o computación en la nube, esto permite a los equipos de desarrollo reducir el tiempo de entrenamiento para una red de aprendizaje profundo de semanas a horas o menos.

Capítulo 2

Redes Neuronales

2.1. Introducción

A finales del siglo XIX se logró un mayor conocimiento sobre el cerebro humano y su funcionamiento, gracias a los trabajos de Ramón y Cajal en España y Sherrington en Inglaterra. El primero trabajó en la anatomía de las neuronas y el segundo en los puntos de conexión de las mismas o sinapsis. El tejido nervioso es el más diferenciado del organismo y está constituido por células nerviosas, fibras nerviosas y la neuroglia, que está formada por varias clases de células.

La célula nerviosa se denomina **neurona**, que es la unidad funcional del sistema nervioso. Pueden ser neuronas sensoriales, motoras y de asociación. Se estima que en cada milímetro cúbico del cerebro hay cerca de 50.000 neuronas. La estructura de una neurona se muestra en la figura 2.1.

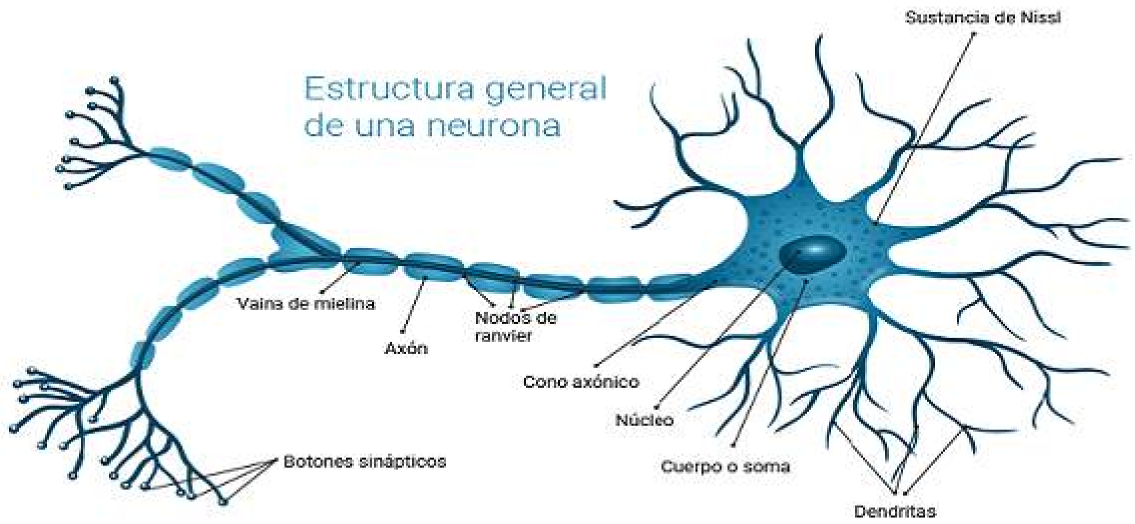


Figura 2.1: Estructura general de una neurona

El cuerpo de la neurona o soma, contiene el núcleo. Se encarga de todas las actividades metabólicas de la neurona y recibe la información de otras neuronas vecinas a través de las conexiones sinápticas. Las dendritas son las conexiones de entrada de la neurona. Por

su parte, el axón es la “salida” de la neurona y se utiliza para enviar impulsos o señales a otras células nerviosas. En las proximidades de sus células destino el axón presenta muchas ramificaciones que forman sinapsis con el soma o axones de otras células. La transmisión de una señal de una célula a otra por medio de la sinapsis es un proceso químico. En él se liberan substancias transmisoras en el lado del emisor de la unión que tienen como efecto elevar o disminuir el potencial eléctrico dentro del cuerpo de la célula receptora. Si su potencial alcanza un cierto umbral, se envía un pulso o potencial de acción por el axón. Este pulso alcanza otras neuronas a través de las distribuciones de los axones.

El sistema de neuronas biológico está compuesto por neuronas de entrada (sensores) conectadas a una compleja red de neuronas “calculadoras”(neuronas ocultas), las cuales, a su vez, están conectadas a las neuronas de salida que controlan, por ejemplo, los músculos. Los sensores pueden ser señales de los oídos, ojos, etc. Las respuestas de las neuronas de salida activan los músculos correspondientes. En el cerebro hay una gigantesca red de neuronas “calculadoras” u ocultas que realizan la computación necesaria.

2.2. Redes Neuronales Artificiales

Una Red Neuronal Artificial (RNA) es un modelo matemático inspirado en el comportamiento biológico de las neuronas y en la estructura del cerebro, y que es utilizada para resolver una amplia variedad de problemas. Debido a su flexibidad, una misma red neuronal es capaz de realizar diversas tareas.

Al igual que sucede en la estructura de un sistema neuronal biológico, los elementos esenciales de proceso de un sistema neuronal artificial son las neuronas. Una neurona artificial es un dispositivo simple de cálculo que genera una única respuesta o salida a partir de un conjunto de datos de entrada.

Las neuronas se agrupan dentro de la red formando niveles o capas. Dependiendo de su situación dentro de la red, se distinguen tres tipos de capas:

- La capa de entrada, que recibe directamente la información procedente del exterior, incorporándola a la red.
- Las capas ocultas, internas a la red y encargadas del procesamiento de los datos de entrada.
- La capa de salida, que transfiere información de la red hacia el exterior.

Las RNAs pueden tener varias capas ocultas o no tener ninguna. Los enlaces sinápticos (las flechas llegando o saliendo de una neurona) indican el flujo de la señal a través de la red y tienen asociado un peso sináptico correspondiente. Si la salida de una neurona va dirigida hacia dos o más neuronas de la siguiente capa, cada una de estas últimas recibe la salida neta de la neurona anterior. El número de capas de una RNA es la suma de las capas ocultas más la capa de salida.

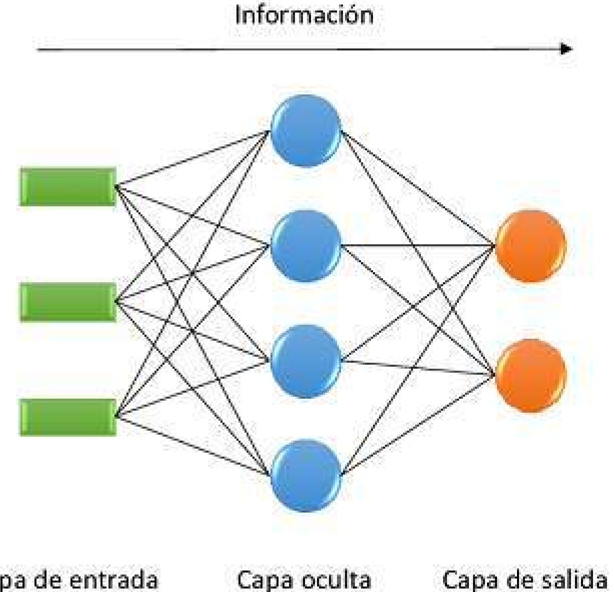


Figura 2.2: Red Artificial

Los elementos que intervienen en una neurona artificial son los siguientes:

- Conjunto de entradas x_i
- Pesos sinápticos w_i, $i = 1, .., m$.
- Regla de propagación s que combina las entradas y los pesos sinápticos. Usualmente: $s(x_1, ..., x_m, w_1, ..., w_m) = \sum_{i=1}^{m} w_i x_i$
- Función de activación g, que es función de s y de una constante u (denominado umbral o sesgo), que proporciona la salida y de la neurona.

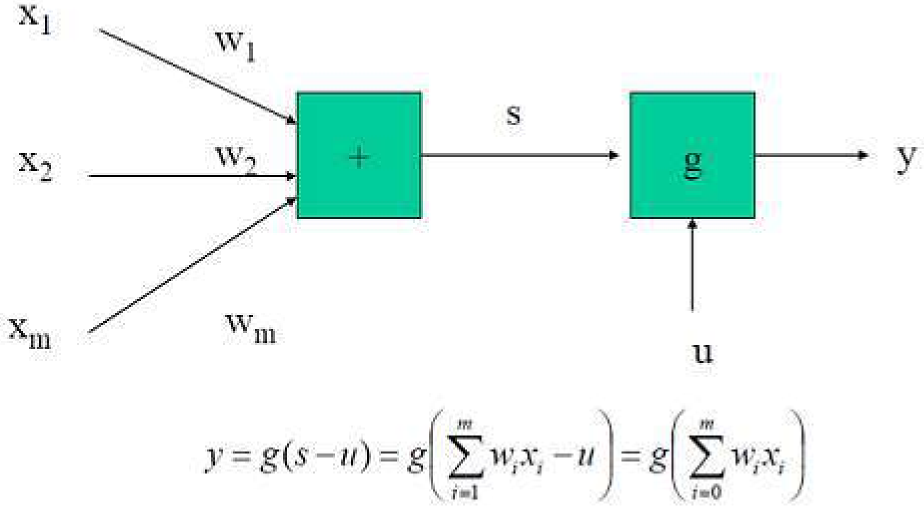


Figura 2.3: Red Neuronal Artificial

Las funciones de activación más usuales se muestran en la Figura 2.4:

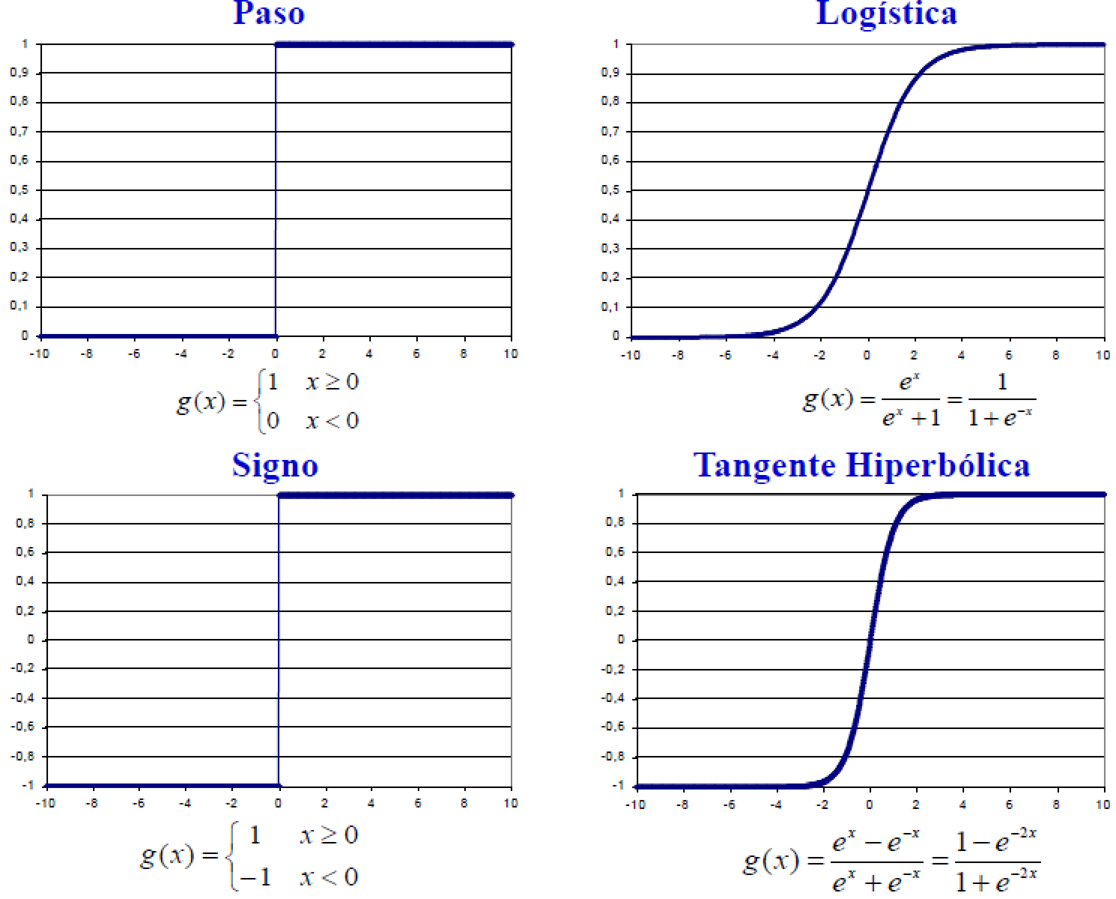


Figura 2.4: Funciones de Activación

Atendiendo al número de capas en que se organizan las neuronas, las redes neuronales se clasifican en dos grandes grupos:

- Redes monocapa, compuestas por un solo nivel de neuronas que se unen mediante conexiones laterales.
- Redes multicapa, donde las neuronas se disponen en dos o más capas.

Atendiendo al flujo de datos las redes neuronales se clasifican en dos grandes grupos:

- Redes unidireccionales (*feedforward*): sin ciclos.
- Redesrecurrentesorealimentadas (*feedback*): presentan ciclos

Dada una Red Neuronal Artificial, se llama algoritmo, método o regla de aprendizaje a cualquier procedimiento que permita obtener una asignación de valores para los coeficientes sinápticos (pesos sinápticos y umbral de activación). Pueden distinguirse distintos tipos de aprendizaje en Redes Neuronales Artificiales:

- **Aprendizaje supervisado**: Se dispone de un conjunto de ejemplos que contiene tanto los datos de entrada como la salida correcta.
- **Aprendizaje no supervisado**: Los ejemplos sólo contienen los datos de entrada, pero no la salida.

- **Aprendizaje híbrido**: Unas capas tienen aprendizaje supervisado y otras no supervisado.
- **Aprendizaje por refuerzo**: No se dispone de las salidas correctas, aunque se indica a la red en cada caso si ha acertado o no.

2.3. Perceptrón simple

El perceptrón simple es una Red Neuronal Artificial que posee las siguientes características:

- Está formada por dos capas de neuronas.
- Es una red unidireccional hacia adelante.
- Los nodos de la primera capa no realizan ningún cálculo, limitándose a proporcionar los valores de entrada a la red.
- La segunda capa está formada por un solo nodo, que aplica una función de activación paso o signo.

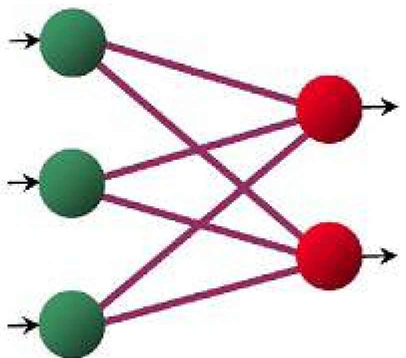


Figura 2.5: Perceptrón simple

A partir de un vector de entradas $x = (x_1, x_2, ..., x_m)'$, considerando la función de activación *signo* y definiendo x_0=1, la salida viene dada por:

$$y = \begin{cases} 1 & \text{si } \sum_{i=1}^{m} w_i x_i + u \geq 0 \\ -1 & \text{si } \sum_{i=1}^{m} w_i x_i + u < 0 \end{cases}$$

Una de las utilidades del perceptrón simple consiste en su utilización como clasificador binario. Se dispone de un conjunto de objetos donde cada uno de ellos viene descrito por un vector m-dimensional $(x_1, x_2, ..., x_m)'$ y pertenece a una de dos clases posibles, C_1 y C_2; se pretende obtener un procedimiento que permita pronosticar la clase a la que pertenece un nuevo objeto a partir del vector carasterístico m-dimensional. Esta utilización solo va a ser posible cuando las dos clases sean linealmente separables (Figura 2.6), es decir, cuando existan coeficientes $\omega_0, \omega_1, \omega_2, ..., \omega_m$, tales que

$$\begin{cases} \sum_{i=1}^{m} \omega_i x_i + u \geq 0 & \forall (x_1, x_2, ..., x_m)' \in C_1 \\ \sum_{i=1}^{m} \omega_i x_i + u < 0 & \forall (x_1, x_2, ..., x_m)' \in C_2 \end{cases}$$

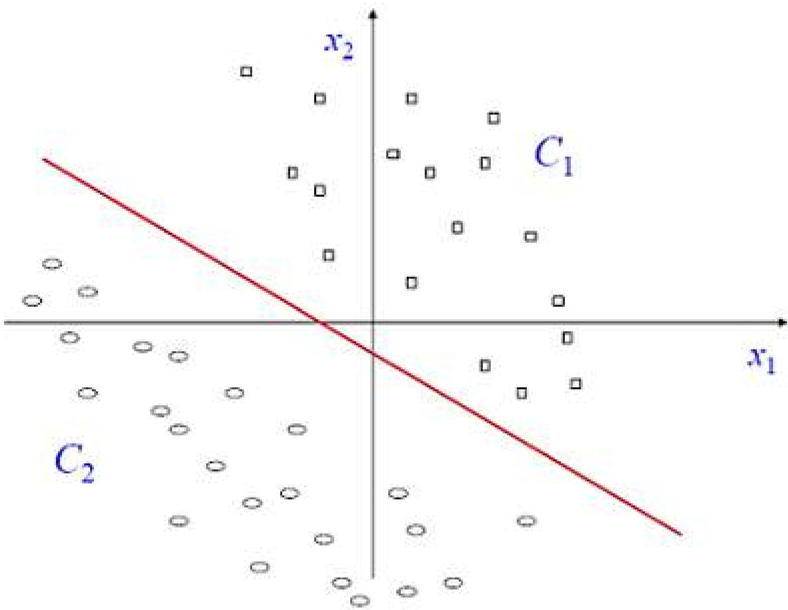


Figura 2.6: Clases linealmente separables

Sea $T = (x_r, z_r), r = 1, 2, ..., N$ el conjunto de patrones de entrenamiento, donde $x_r \in \mathbb{R}^m$, $z_r = 1$ si x^r pertenece a C_1 y $z_r = -1$ si x_r pertenece a C_2.

Sea $\omega = (w_1, ..., w_m)'$ el vector de pesos sinápticos, e $y(x, \omega)$ la salida que se obtiene a partir de un vector de entradas x. El error total de clasificación, que se denotará por $EC(T, \omega)$, viene dado por el total de patrones de entrenamiento clasificados incorrectamente:

$$EC(T, \omega) = \sum_{r|z_r=-1} I(y(x_r, \omega) = 1) + \sum_{r|z_r=1} I(y(x_r, \omega) = -1) \quad (2.1)$$

El algoritmo usado para ajustar los parámetros libres de esta red neuronal apareció por primera vez en un procedimiento de aprendizaje desarrollado por Rosenblatt (1958) para su modelo de perceptrón del cerebro. Este algortimo consiste en lo siguiente:

Sea y_r la salida de la red para $x_r, r = 1, ..., n$ tal que $x_{ri} \in \{-1, 1\}$, $y_{rj} \in \{-1, 1\} \quad i = 1, ..., m \quad j = 1, ..., n \quad r = 1, ..., N$

1. Asignar valores aleatorios a los pesos.
2. Seleccionar un vector de entrada x_r del conjunto de entrenamiento.
3. Determinar la salida de la red y_r correspondiente a x_r.
4. Si $z_r = y_r$, volver al paso 2.
5. Para cada neurona de entrada i y cada neurona de salida j:

- Calcular $\Delta_{rij} = \varepsilon(z_{rj} - y_{rj})x_{ri}$
- Actualizar los pesos: $\omega_{ij} := \omega_{ij} + \Delta_{rij}$

6. Volver al paso 2.

En este algoritmo las variaciones en los pesos solo pueden ser $-2\varepsilon,\quad 0 \quad \text{y} \quad 2\varepsilon$ donde ε, que se denomina coeficiente o tasa de aprendizaje, toma los valores $0 < \varepsilon < 1$. El algoritmo se detiene cuando todos los ejemplos son correctamente clasificados (si se puede) y el ajuste de los pesos después de procesar todos los ejemplos obedece a:

$$\omega_{ij} := \omega_{ij} + \sum_{r=1}^{N} \Delta_{rij}$$

Si las clases definidas por el conjunto de ejemplos son separables linealmente, entonces el algoritmo de Rosenblatt converge en un número finito de iteraciones, para cualquier elección inicial de los pesos, a un vector de coeficientes ω que verifica $EC(T, \omega) = 0$.

2.4. Perceptrón múltiple

Minsky y Papert (1969) [9] demostraron que el perceptrón simple no puede resolver problemas no lineales. La combinación de varios perceptrones podría resolver ciertos problemas no lineales, pero no existía un mecanismo automático para adaptar los pesos de las capas ocultas.

Rumelhart et al. [13] introdujeron en 1986 el perceptrón múltiple (PM), que es una red compuesta por varias capas de neuronas con conexiones hacia adelante (*feed-forward*) y en la que el aprendizaje es supervisado. En 1989, Cybenko [3] probó que el perceptrón multicapa puede aproximar a cualquier función continua, por lo que se considera un aproximador universal.

Un perceptrón múltiple puede aproximar relaciones no lineales entre variables de entrada y de salida y es una de las arquitecturas más utilizadas en la resolución de problemas reales por ser aproximador universal, por su fácil uso y aplicabilidad. El uso del PM es adecuado cuando:

- Se permiten largos tiempos de entrenamiento.
- Se necesitan respuestas muy rápidas ante nuevas instancias.

La arquitectura de un PM es la siguiente (Figura 2.7):

- **Capa de entrada**: sólo se encargan de recibir las señales de entrada y propagarlas a la siguiente capa.
- **Capa de salida**: proporciona al exterior la respuesta de la red para cada patrón de entrada.
- **Capas ocultas**: Realizan un procesamiento no lineal de los datos recibidos.

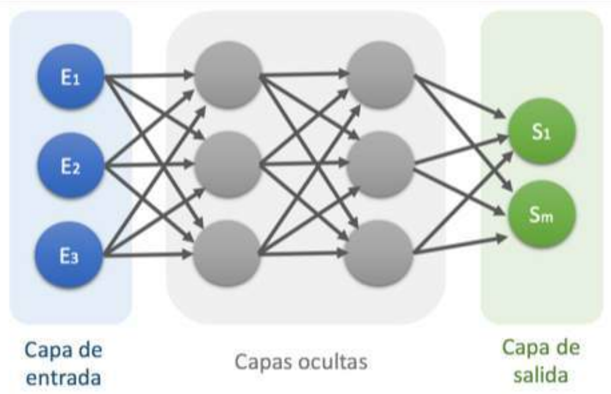


Figura 2.7: Perceptrón Múltiple

El algoritmo utilizado para la determinación de los parámetros de la red utiliza un ciclo de propagación-adaptación que consta de dos fases:

1. **Propagación** en la que se calcula el resultado de salida de la red desde los valores de entrada hacia delante.
2. **Aprendizaje** en la que los errores obtenidos a la salida del perceptrón se van propagando hacia atrás (*backpropagation* o Regla Delta Generalizada) con el objetivo de modificar los pesos de las conexiones para que el valor estimado de la red se asemeje cada vez más al real; esta aproximación se realiza mediante la función gradiente del error.

El primer algoritmo de entrenamiento para redes multicapa fue desarrollado por Paul Werbos en 1974; éste se desarrolló en un contexto general, para cualquier tipo de redes, siendo las redes neuronales una aplicación especial, razón por la cual el algoritmo no fue aceptado dentro de la comunidad de desarrolladores de redes neuronales. No fue hasta mediados de los años 80 cuando el algoritmo *Backpropagation* fue redescubierto al mismo tiempo por varios investigadores, David Rumelhart, Geoffrey Hinton y Ronal Williams, David Parker y Yann Le Cun. El algoritmo se popularizó cuando fue incluido en el libro *"Parallel Distributed Processing Group"* por los psicólogos David Rumelhart y James McClelland.

Uno de los grandes avances logrados con el algoritmo *Backpropagation* es que aprovecha la naturaleza paralela de las redes neuronales para reducir el tiempo requerido por un procesador secuencial para determinar la correspondencia entre unos patrones dados y sus salidas. Además, el tiempo de desarrollo de cualquier sistema que se esté tratando de analizar se puede reducir como consecuencia de que la red puede aprender el algoritmo correcto sin que alguien tenga que deducir por anticipado el algoritmo en cuestión.

La importancia de este proceso radica en que, a medida que se entrena la red, las neuronas de las capas intermedias se organizan a sí mismas de tal modo que las distintas neuronas aprenden a reconocer distintas características del espacio total de entrada. Después del entrenamiento, cuando se les presente un patrón arbitrario de entrada que contenga ruido o que esté incompleto, las neuronas de la capa oculta de la red responderán con una salida activa si la nueva entrada contiene un patrón que se asemeje a aquella característica que las neuronas individuales hayan aprendido a reconocer durante su entrenamiento. Y a la inversa, las unidades de las capas ocultas tienen tendencia a inhibir

su salida si el patrón de entrada no contiene la característica a reconocer, para la cual han sido entrenadas.

El entrenamiento de una red neuronal multicapa se realiza mediante un proceso de aprendizaje; para realizar este proceso se debe inicialmente tener definida la topología de la red, esto es: número de neuronas en la capa de entrada, el cual depende del número de componentes del vector de entrada, cantidad de capas ocultas y número de neuronas de cada una de ellas, número de neuronas en la capa de la salida, el cual depende del número de componentes del vector de salida o patrones objetivo y funciones de transferencia requeridas en cada capa. Una vez escogida la topología de la red se asignan valores iniciales a cada uno de los parámetros que conforman la red.

Es importante recalcar que no existe una regla para determinar el número de capas ocultas, ni el número de neuronas que debe contener cada una de ellas.

2.4.1. Propagación de los patrones de entrada

El perceptrón multicapa define una relación entre las variables de entrada y las variables de salida de la red. Esta relación se obtiene propagando hacia adelante los valores de las variables de entrada. Para ello, cada neurona de la red procesa la información recibida por sus entradas y produce una respuesta o activación que se propaga a través de las conexiones correspondientes hacia las neuronas de la siguiente capa. A continuación se muestran las expresiones para calcular las activaciones de las neuronas de la red. Durante la propagación hacia adelante, se fijan todos los pesos sinápticos de la red.

Sea un perceptrón multicapa con C capas ($C-2$ capas ocultas) y m_c el número de neuronas en la capa c, para $c = 1, 2, ..., C$. Sea $W^c = (w^c_{ij})$ la matriz de pesos donde w^c_{ij} representa el peso de la conexión j de la neurona i de la capa c para $c = 2, .., C$. Denotaremos a^c_i a la activación de la neurona i de la capa c. Estas activaciones se calculan del siguiente modo:

- Activación de las neuronas de la capa de entrada (a^1_i). Las neuronas de la capa de entrada se encargan de transmitir hacia la red las señales recibidas desde el exterior. Por tanto:

  $$a^1_i = x_i \text{ para } i = 1, 2, \ldots, m_1 \tag{2.2}$$

  donde $X = (x_1, x_2, ..., x_{m_1})$ representa el vector o patrón de entrada a la red.

- Activación de las neuronas de la capa oculta c (a^c_i). Las neurones ocultas de la red procesan la información recibida aplicando la función de activación g a la suma de los productos de las activaciones que reciben por sus correspondientes pesos, es decir:

  $$a^c_i = g\left(\sum_{j=1}^{m_{c-1}} w^{c-1}_{ij} a^{c-1}_j + u^c_i\right) \text{ para } i = 1, \ldots, m_c \text{ y } c = 2, 3, \ldots, C-1 \tag{2.3}$$

  donde a^{c-1}_j son las activaciones de las neuronas de la capa $c-1$.

- Activación de las neuronas de la capa de salida (a^C_i). Al igual que en el caso anterior, la activación de estas neuronas viene dada por la función de activación g aplicada a

la suma de los productos de las entradas que recibe por sus correspondientes pesos:

$$y_i = a_i^C = g(\sum_{j=1}^{m_{C-1}} w_{ij}^{C-1} a_j^{C-1} + u_i^C) \text{ para } i = 1, 2, \ldots, m_c \text{ y } c = 2, 3, \ldots, C-1 \quad (2.4)$$

donde $Y = (y_1, y_2, ..., y_{m_C})$ es el vector de salida de la red.

Para el perceptrón multicapa, las funciones de activación g más utilizadas son la función logística y la función tangente, ya vistas anteriormente (Figura 4.7).

2.4.2. Fase de aprendizaje

En esta fase se produce la retropropagación del error. Las salidas de error se propagan hacia atrás, partiendo de la capa de salida, hacia todas las neuronas de la capa oculta que contribuyen directamente a la salida. Sin embargo, las neuronas de la capa oculta solo reciben una fracción de la señal total del error, basándose aproximadamente en la contribución relativa que haya aportado cada neurona a la salida original.

Este proceso se repite, capa por capa, hasta que todas las neuronas de la red hayan recibido una señal de error que describa su contribución relativa al error total. Basándose en la señal de error percibida, se actualizan los pesos de conexión de cada neurona, para hacer que la red converja hacia un estado que permita clasificar correctamente todos los patrones de entrenamiento.

2.4.2.1. Regla de aprendizaje

Cada patrón de entrenamiento se propaga a través de la red y sus parámetros para producir una respuesta en la capa de salida, la cual se compara con los patrones objetivo o salidas deseadas para calcular el error en el aprendizaje, este error marca el camino mas adecuado para la actualización de los pesos y ganancias que al final del entrenamiento producirán una respuesta satisfactoria a todos los patrones de entrenamiento, esto se logra minimizando el error medio cuadrático en cada iteración del proceso de aprendizaje.

Sea y_r la salida de la red para la entrada $x_r, r = 1, ..., N$ tal que $x_{ri} \in \{-1, 1\}$, $y_{rj} \in \{-1, 1\} \quad i = 1, ..., m \quad j = 1, ..., n \quad r = 1, ..., N$ y sean los pesos de la red w_{ji}

Veamos primero un caso particular de obtener la regla cuando no hay capas ocultas, hay varias neuronas de salida y suponemos activación lineal. El error cuadrático cometido al presentar el ejemplo r 2.5, donde y indica el valor de salida de la RNA y z el valor esperado dado por la muestra de aprendizaje. El error total se obtiene sumando los errores parciales E_r correspondientes a los distintos ejemplos de entrenamiento.

$$E_r = \frac{1}{2} \sum_{j=1}^{n} (z_{rj} - y_{rj})^2 \quad (2.5)$$

Si derivamos en esta expresión respecto de los parámetros de la red (pesos), aplicando la regla de la cadena. La primera parte dice como cambia el error con la salida j y la segunda, como cambia esa salida, al cambiar el peso w_{ji}.

$$\frac{\partial E_r}{\partial w_{ji}} = \frac{\partial E_r}{\partial y_{rj}} \frac{\partial y_{rj}}{\partial w_{ji}} \quad \text{y} \quad \text{será} \quad \frac{\partial E_r}{\partial y_{rj}} = -(z_{rj} - y_{rj}) = -\delta_{rj}$$

La contribución de la neurona de salida j al error es proporcional a δ_{rj}. Además, por ser la activación lineal

$$y_{rj} = \sum w_{ji} x_{ri} + u \Rightarrow \frac{\partial y_{rj}}{\partial w_{ji}} = x_{ri}$$

y por tanto será

$$-\frac{\partial E_r}{\partial w_{ji}} = \delta_{rj} x_{ri} = \Delta_r w_{ij}$$

Podemos observar que la regla delta se realiza de acuerdo con el gradiente descendente (se mueve en el sentido del gradiente); realmente esto es cierto si los pesos no se modifican durante la iteración. Podemos añadir un factor de aprendizaje η que añade modificaciones son muy pequeñas para aceptar también la modificación de pesos después de procesar cada ejemplo. En este caso, en que no hay neuronas ocultas, está garantizado que se puede encontrar el mínimo del error (el mejor conjunto de pesos) ya que este es único.

En definitiva, el ajuste de pesos se hace con la siguiente expresión, conocida como **regla delta**

$$\Delta_r w_{ij} = \eta(z_{rj} - y_{rj}) x_{rj} = \eta \delta_{rj} x_{ri}$$
$$\delta_{rj} = z_{rj} - y_{rj}$$

Consideremos ahora el caso general en que existen capas ocultas y se utiliza una función de activación genérica. Para facilitar el tratamiento analítico vamos a considerar que la función de activación es una función no decreciente y diferenciable. En este caso, d va a representar la activación de la neurona y tendremos:

$$d_{rj} = \sum_{i=1}^{m} w_{ji} x_{ri} \Rightarrow y_{rj} = g_j(d_{rj})$$

Suponemos la misma función de error que antes, y aplicando de nuevo la regla de la cadena:

$$\frac{\partial E_r}{\partial w_{ji}} = \frac{\partial E_r}{\partial d_{rj}} \frac{\partial d_{rj}}{\partial w_{ji}}$$

Tenemos que

$$\frac{\partial d_{rj}}{\partial w_{ji}} = \frac{\partial}{\partial w_{ji}} \sum_{l=1}^{m} w_{jl} x_{rl} = x_{ri}$$

Para obtener las δ_{rj} llamamos

$$\delta_{rj} = -\frac{\partial E_r}{\partial d_{ji}}$$

y se obtiene

$$-\frac{\partial E_r}{\partial w_{ji}} = \delta_{rj} x_{ri}$$

por lo que los pesos deberán actualizarse conforme a $\partial_{rj} x_{ri}$ siendo

$$-\frac{\partial E_r}{\partial d_{rj}} = \delta_{rj}$$

Al igual que antes aquí también podemos observar que la regla delta se realiza de acuerdo con el gradiente descendente y podemos añadir el factor de aprendizaje η.

$$\Delta_r w_{ij} = \eta \delta_{rj} x_{ri}$$

Las δ_{rj} se pueden obtener de forma recursiva, a partir de los valores obtenidos en la capa de salida, para todas las capas ocultas, retropropagando el error en la salida por la red hacia las capas interiores. Para su cálculo, aplicamos de nuevo la regla de la cadena, poniéndolas como producto de dos derivadas parciales, la primera del error respecto de la salida en la neurona y, la segunda, de la salida de la neurona respecto de la activación de la misma. Este segundo factor es claramente la derivada de la función de activación usada en la unidad correspondiente.

$$\delta_{rj} = -\frac{\partial E_r}{\partial d_{rj}} = \frac{\partial E_r}{\partial y_{rj}} \frac{\partial y_{rj}}{\partial d_{rj}} = -\frac{\partial E_r}{\partial y_{rj}} g_j'(d_{rj})$$

Para calcular el primer factor, hay que distinguir que la neurona sea de la capa de salida o de una capa anterior.

Si es de la capa de salida, es inmediato su cálculo. Así, la exprexión para los δ_{pj} es inmediata:

$$\frac{\partial E_r}{\partial y_{rj}} = -(z_{rj} - y_{rj}) = -\delta_{rj} \Rightarrow \delta_{rj} = (z_{rj} - y_{rj}) g_j'(d_{rj})$$

Para las neuronas de las capas ocultas, como no tenemos una expresión para el error, el cálculo no es tan inmediato. Aplicaremos de nuevo la regla de la cadena en la siguiente forma

$$\sum_k \frac{\partial E_r}{\partial d_{rk}} \frac{\partial d_{rk}}{\partial y_{rj}} = \sum_k \frac{\partial E_r}{\partial d_{rk}} \frac{\partial}{\partial y_{rj}} \sum w_{ji} x_{ri} = \sum_k \frac{\partial E_r}{\partial d_{rk}} w_{kj} = -\sum_k \delta_{rk} w_{kj}$$

Por tanto, tenemos definitivamente para las neuronas en las capas ocultas:

$$\partial_{rj} = g_j'(d_{rj}) \sum_k \partial_{rk} w_{rj}$$

En definitiva, el proceso se resume en tres ecuaciones, que indican cómo se hace la adaptación de los pesos y los valores de las δrj distinguiendo que la neurona sea de la capa de salida o no:

$$\Delta_r w_{ij} = \eta \delta_{rj} x_{ri} \quad con \quad \delta_{rj} = (z_{rj} - y_{rj}) g'_j(d_{rj}) \quad \text{o} \quad \delta_{rj} = g'_j(d_{rj}) \sum_k \delta_{rk} w_{kj}$$

Si la función de activación es la sigmoide, tenemos las siguientes expresiones, que son las utilizadas en el algoritmo **Backpropagation**. La función sigmoide y su derivada son:

$$y_{rj} = \frac{1}{(1 + e^{d_{rj}})} \Rightarrow g'_j(d_{rj}) = y_{rj}(1 - y_{rj})$$

Entonces, los valores para las δ_{pj} son en este caso:

$$\delta_{rj} = (z_{rj} - y_{rj}) y_{rj}(1 - y_{rj})$$

si j es de salida y

$$\delta_{rj} = y_{rj}(1 - y_{rj}) \sum_k \delta_{rk} w_{kj}$$

si no lo es.

En la expresión para las neuronas ocultas las δ_{rk} que hay en el sumatorio se refieren a las calculadas en la iteración anterior, y corresponden a la capa siguiente (el cálculo se realiza de atrás hacia adelante, es decir empezando por la capa de salida).

Se suele usar también un término *momentun* (inercia) en el ajuste de pesos, que acelera la convergencia del algoritmo. Lo que se hace es tener en cuenta el ajuste realizado en la iteración n, para hacer el ajuste en la iteración $n + 1$, con un coeficiente del momentun. Pretende obviar salto bruscos en las direcciones de optimización que marca el gradiente descendente. El ajuste de los pesos se hace con

$$\Delta_r w_{ji}(n + 1) = \eta \delta_{rj} x_{ri} + \alpha \Delta_r w_{ji}(n)$$

Capítulo 3

Deep Learning

Las redes profundas son modelos paramétricos que realizan operaciones secuenciales en los datos de entrada. Cada una de esas operaciones consiste en una transformación lineal (por ejemplo, una convolución de su entrada), seguida de una "función de activación", como las presentadas en la Sección 2.2. Las redes profundas han llevado recientemente a mejoras dramáticas en el rendimiento de varias aplicaciones, tales como el procesamiento del habla y el lenguaje natural, y la visión por ordenador.

La propiedad crucial de las redes profundas que es la raíz de su rendimiento radica en que tienen una gran cantidad de capas en comparación con las redes neuronales clásicas. Además, uno de los factores importantes para el éxito del Deep Learning es la disponibilidad de conjuntos de datos masivos, ya que, a diferencia de lo que sucede con otros algoritmos de Machine Learning, el rendimiento del Deep Learning mejora al disponer de un mayor volumen de datos.

Hay tres factores fundamentales en este campo que son fundamentales para entrenar una red neuronal profunda de forma que tenga un buen rendimiento: la arquitectura, las técnicas de regularización y los algoritmos de optimización. Por tanto, entender la necesidad de estos factores es esencial para comprender su éxito; basado en los siguientes aspectos:

- **Aproximación**: Una propiedad esencial en el diseño de la arquitectura de una red neuronal es su habilidad para aproximar funciones arbitrarias sobre los datos de entrada. Aunque ya vimos anteriormente que una red neuronal con una capa oculta es un aproximador de funciones universal, se observa que arquitecturas profundas son capaces de mejorar la captación de propiedades invariantes de los datos en comparación con la redes neuronales convencionales.

- **Generalización y regularización**: Otra característica esencial de la arquitecturas de las redes neuronales es su habilidad para generalizar con un número pequeño de muestras de entrenamiento. Las redes neuronales profundas son entrenadas con muchos menos datos que número de parámetros y aun así se puede prevenir el sobreajuste utilizando técnicas de regularización muy simples, como es por ejemplo el *Dropout*, que básicamente consiste en elegir un subconjunto aleatorio de parámetros en cada iteración.

- **Facilidad para la detección de los patrones importantes**: Una de las propiedades más relevante que tienen estos métodos es la facilidad para detectar la

información importante y también los criterios que son invariantes entre casos similares, lo cual es esencial para predecir observaciones futuras.

- **Optimización**: El algoritmo *Backpropagation* ha evolucionado hasta dar lugar al algoritmo *Stochastic Gradient Descent* (SGD) para aproximar el gradiente sobre conjuntos de datos enormes. Hay que observar que, mientras el SGD ha sido muy estudiado para funciones de pérdida convexas, en Deep Learning la función de pérdida es en general no convexa respecto a los pesos de la red neuronal, por lo que no hay garantías de que SGD proporcione el mínimo global.

3.1. Optimalidad global Deep Learning

El funcionamiento básico de una red neuronal Deep Learning es el mismo que el de una red neuronal convencional; el cambio reside en la función de pérdida, esto es, la función a minimizar para encontrar los pesos óptimos. El problema de la obtención de pesos $W = \{W^k\}_{k=1}^K$ en una red profunda con N ejemplos de entrenamiento (X,Y), se formula como el siguiente problema de optimización:

$$\min_{\{W^k\}_{k=1}^K} \ell(Y, \phi(X, W^1, ..., W^K)) + \lambda\Theta(W^1, ..., W^K) \tag{3.1}$$

donde $\ell(Y, \phi)$ es la función pérdida que mide el ajuste entre la verdadera salida, Y, y la salida que ha predicho el modelo $\phi(X, W)$. Hay que observar que W^k denota el conjunto de pesos de la capa k-ésima, con $k \in \{1, , ..., K\}$. Por otro lado, tenemos que Θ es la función de regularización que previene el sobreajuste, y $\lambda \geq 0$ es un parámetro de equilibrio. Por ejemplo, $\Theta(W) = \sum_{k=1}^K \|W^k\|_F^2$, donde $\|.\|_F$ representa la norma de Frobenius, dada por:

$$\|A\|_F = \sqrt{\sum_i \sum_j |a_{ij}|^2}$$

3.1.1. El reto de las funciones no convexas en las redes neuronales

El principal reto en el problema de optimización de los pesos en la (3.1) es la no convexidad; incluso siendo la función de penalización o pérdida $\ell(Y, \phi)$ normalmente convexa respecto a ϕ, la función $\phi(X, W)$ no suele ser una función convexa respecto a los pesos W debido al producto entre las variables W^k y las funciones de activación. En los problemas no convexos, el conjunto de puntos críticos incluyen no sólamente mínimos globales sino también mínimos locales, máximos locales y puntos de silla, por lo que el algoritmos de optimización puede quedar atrapado en tales puntos si encontrar el óptimo global. Por tanto, la no convexidad hace que no sólo la formulación del modelo sea importante, sino también los detalles de implementación, como pueden ser, los valores iniciales de los pesos y ,sobre todo, el algoritmo de optimización empleado.

Las peculiaridades de ka implementación son cruciales en el rendimiento del modnlo. Para solventar el problema de la no convexidad, una estrategia ampliamente utilizada en los modelos Deep Learning es

1. Inicializar los pesos de manera aleatoria.
2. Actualizar dichos pesos con el método del gradiente local.
3. Comprobar que el error decrece lo suficientemente rápido, y si no, tomar otros pesos para la inicialización.

3.1.2. Estabilidad geométrica en el Deep Learning

Una cuestión importante a tener en cuenta son las hipótesis matemáticas en las que se basan los métodos de Deep Learning y que hacen que en determinados casos funcione correctamente, o, al menos, mejor que los métodos clásicos.

Sea $\Omega = [0,1]^d$ un dominio Euclídeo compacto d-dimensional donde están definidas funciones $X \in L^2(\Omega)$ (recordemos que, dados un espacio de medida Ψ y μ una medida positiva, se define el espacio de funciones de potencia p-ésima integrable como:
$L^p(\mu) = \{f : \Psi \longmapsto \mathbb{C} : f \quad \text{medible} \quad y \quad \int_\Psi |f|^p d\mu < \infty\})$.

En aprendizaje supervisado, una función desconocida $f : L^2(\Omega) \longmapsto \mathcal{Y}$ es observada en el conjunto de entrenamiento

$$\{X_i \in L^2(\Omega), Y_i = f(X_i)\}_{i\in I} \tag{3.2}$$

donde el espacio de llegada $\mathcal{Y}$ puede ser considerado discreto con $C = |\mathcal{Y}|$ el número de clases de las que dispone la variable de salida, o $\mathcal{Y} = \mathbb{R}^{\mathcal{C}}$ en un problema de regresión.

En la mayoría de los casos, se considera que la función f satisface las siguientes condiciones:

1. **Estacionariedad**: Se considera un operador de traslación

   $$\tau_v X(u) = X(u-v) \quad \text{con} \quad u, v \in \Omega$$

   que actúa sobre funciones $X \in L^2(\Omega)$. Dependiendo del problema, se asume que la función f es invariante respecto a traslaciones. Formalmente, se tiene que:

   $$f(\tau_v X) = f(X) \quad \text{para cualquier} \;\; X \in L^2(\Omega) \quad \text{y} \quad v \in \Omega$$

   .

2. **Deformaciones locales y separaciones a escala**: De forma similar, una deformación L_τ , donde $\tau : \Omega \longmapsto \Omega$ es un vector de funciones de clase C^∞ que actúa sobre $L^2(\Omega)$,como $L_\tau X(u) = X(u - \tau(u))$. Las deformaciones pueden modelar traslaciones locales y rotaciones. La mayoría de los problemas que se estudian no son solamente invariantes respecto a traslaciones, además son estables respecto a deformaciones locales. En problemas que son invariantes respecto a traslaciones se tiene:

   $$|f(L_\tau X) - f(X)| \approx \|\nabla\tau\| \quad \forall X, \tau \tag{3.3}$$

   donde $\|\nabla\tau\|$ mide la derivabilidad de una deformación dada; es decir, cuantifica si las predicciones no cambian mucho si la entrada es deformada localmente. Esta propiedad es mucho más fuerte que la estacionariedad.

3.2. Error de generalización

El error de generalización es la diferencia entre el error empírico y el error esperado, y es un concepto fundamental en la teoría del aprendizaje estadístico. Los límites de error de generalización ofrecen una idea de por qué es posible aprender de ejemplos de entrenamiento.

Consideramos un problema de clasificación donde cada dato $X \in \mathcal{X} \subseteq \mathbb{R}^D$ tiene una etiqueta de clase correspondiente $Y \in \mathcal{Y}$, siendo C es el número de clases. Un conjunto de entrenamiento de N muestras extraídas de una distribución P se denota por $\gamma_N = \{X_i, Y_i\}_{i=1}^N$ y la función de pérdida se denota por $\ell(Y, \Phi(X, W))$, que mide la discrepancia entre la etiqueta verdadera Y y la etiqueta estimada $\Phi(X, W)$ proporcionadas por el clasificador. La pérdida empírica de una red $\Phi(\mathring{u}, W)$ asociada con el conjunto de entrenamiento γ_N se define como:

$$\ell_{emp}(\Phi) = \frac{1}{N} \sum_{X_t \in \gamma_N} \ell(Y_i, \Phi(X_i, W)) \tag{3.4}$$

y la pérdida esperada como

$$\ell_{exp}(\Phi) = \mathbb{E}_{(X,Y)\sim P}[\ell(Y, \Phi(X, W))] \tag{3.5}$$

El **error de generalización** se define entonces como:

$$GE(\Phi) = |\ell_{exp}(\Phi) - \ell_{emp}(\Phi)| \tag{3.6}$$

3.3. Limitaciones del Deep Learning

Las limitaciones que impidieron que las técnicas de Deep Learning se desarrollaran antes eran más de tipo práctico que teórico. Ni los ordenadores eran lo suficientemente potentes, ni los conjuntos de datos lo suficientemente grandes. Una vez solventadas esas dificultades prácticas, los fundamentos formales que ya se conocían y empleaban en entornos limitados desde los años 80 permitieron el despegue del Deep Learning.

No obstante, eso no quiere decir que el uso de redes neuronales artificiales esté por completo exento de problemas. Sus dos aspectos más problemáticos son los derivados del sobreaprendizaje y su carácter de cajas negras. Para resolver el primero, que es un problema común a otros muchos modelos de aprendizaje automático, se han propuesto multitud de técnicas que, en mayor o menor medida, permiten soslayarlo. En cuanto al segundo, es algo sobre lo que todavía queda mucho por hacer y que puede tener implicaciones con respecto a la seguridad de los sistemas que emplean redes neuronales internamente.

3.3.1. Problemas derivados del sobreaprendizaje

El sobreaprendizaje se produce siempre que un modelo se ajusta tan bien a su conjunto de entrenamiento que deja de generalizar correctamente cuando lo utilizamos sobre un

conjunto de prueba diferente al conjunto de datos de entrenamiento que utilizamos para construirlo. Es un problema habitual en muchas técnicas de aprendizaje automático.

Cuando utilizamos redes neuronales, las redes neuronales incluyen multitud de parámetros ajustables: los pesos que modelan las conexiones entre neuronas. Ese número elevado de parámetros las hace propensas a sufrir problemas de sobreaprendizaje [*overfitting*]. De hecho, hay quien piensa que es su principal inconveniente.

El conjunto de entrenamiento contiene las regularidades que nos permiten construir un modelo de aprendizaje automático. Sin embargo, también incluye dos tipos de ruido: errores en los datos y errores de muestreo.

Los errores en los datos son inevitables y provienen del proceso de adquisición de datos, especialmente si éste incluye algún tipo de procesamiento manual. Las etiquetas asociadas a los casos de entrenamiento pueden ser erróneas, por ejemplo, en un porcentaje nada desdeñable de éstos. Por otro lado, el conjunto de datos, por grande que sea, no deja de ser una muestra de una población mayor. En él existirán regularidades accidentales que no se deben al problema en sí que deseamos modelar, sino a los casos particulares que acaban formando parte de nuestro conjunto de datos. Cuando ajustamos un modelo a esos datos, nunca podemos saber si estamos aprovechando las regularidades "reales" de los datos o si nuestro modelo está identificando las regularidades accidentales debidas al error de muestreo. Cualquier modelo mezclará ambas. Si, como sucede con las redes neuronales, nuestro modelo es extremadamente flexible entonces será capaz de modelar todos los matices del conjunto de datos de entrenamiento, tanto esenciales como accidentales. Cuando el modelo ajusta los rasgos accidentales, pierde capacidad de generalización y, en la práctica, esto puede resultar desastroso.

En términos generales, un modelo de aprendizaje con muchos parámetros, como es el caso de las redes neuronales, será capaz de ajustarse mejor a los datos que le proporcionemos, aunque no resulte demasiado económico. En términos de la descomposición del error en sesgo y varianza, diríamos que el modelo exhibe una varianza elevada (si tuviese muy pocos parámetros hablaríamos de sesgo). No debería sorprendernos, por tanto, que un modelo con muchos parámetros se ajuste demasiado bien a los datos de entrenamiento, hasta el punto de sobreaprender.

Afortunadamente, además de las estrategias generales que podemos utilizar para cualquier modelo de aprendizaje automático, se han propuesto multitud de técnicas específicas que se pueden emplear para prevenir y atenuar los efectos del sobreaprendizaje en redes neuronales artificiales. Éstas incluyen sencillas heurísticas, como el decaimiento de pesos (*weight decay*), que penaliza la presencia de pesos elevados en la red, o el uso de pesos compartidos (*weight sharing*), que reduce el número de parámetros de una red neuronal, con lo que se reduce su capacidad y, por tanto, la posibilidad de sufrir sobreaprendizaje. Otras técnicas de prevención del sobreaprendizaje se incorporan en el propio algoritmo de entrenamiento de la red neuronal, como la parada temprana (*early stopping*), que detiene el proceso iterativo de aprendizaje en el momento en el que se detecta un aumento de la tasa de error en un conjunto de validación independiente del conjunto de entrenamiento.

Otra posibilidad consiste en la combinación de múltiples modelos, ya sea de forma explícita mediante la creación de ensembles (v.g. *model averaging*) o de forma implícita mediante la incorporación de técnicas que son, desde un punto de vista formal, equivalentes al uso de múltiples modelos (v.g. *dropout*). Hay incluso quien aboga por un proceso de preentrenamiento previo, realizado de forma no supervisada, para pre-ajustar los parámetros

de la red antes de someterla a un entrenamiento de tipo supervisado, lo que puede servir para prevenir el sobreaprendizaje a la vez que resulta útil en situaciones en los que disponemos de muchos más datos no etiquetados (para el pre-entrenamiento no supervisado) que etiquetados (para el entrenamiento supervisado).

Como vemos, existe una amplia gama de técnicas y herramientas a nuestra disposición para prevenir el sobreaprendizaje al que son propensas las redes neuronales. Dada esa propensión, la estrategia habitual en el entrenamiento de redes neuronales artificiales es permitir que la red sobreaprenda y luego regularizar. En Aprendizaje Automático, se denomina regularización a cualquier proceso mediante el cual se incorpora información adicional para prevenir el sobreaprendizaje, ya sea un término adicional en la función de coste que se pretenda minimizar (como en *weight decay*), un mecanismo de control durante el proceso de entrenamiento (*early stopping*) o la anulación selectiva de partes de una red neuronal para prevenir su coadaptación (*dropout*).

3.3.2. Las redes neuronales como cajas negras

En determinados dominios de aplicación, el problema más acuciante de las redes neuronales artificiales es nuestra incapacidad para determinar cómo llegan las redes neuronales a una conclusión. En una red neuronal, podemos observar la entrada de la red y ver cuál es su salida, pero su funcionamiento interno es algo que no se puede describir de forma simbólica. Para nosotros, una red neuronal es, en gran medida, una caja negra.

En situaciones en la que no sólo sea importante la validez del modelo, sino también su credibilidad, puede que el carácter opaco de una red neuronal juegue en nuestra contra. La validez de un modelo la podemos evaluar de múltiples formas, por ejemplo, con validación cruzada. Las pruebas que realicemos nos pueden indicar que, en efecto, una red neuronal ofrece mejores resultados que otras técnicas de aprendizaje automático. Sin embargo, su credibilidad es otra historia. Al tratarse de una faceta subjetiva, vinculada a la percepción que otros pueden tener de su fiabilidad, es algo que no podemos avalar con argumentos meramente cuantitativos. Por desgracia, dado que no podemos explicar su funcionamiento en detalle, tampoco podemos recurrir a los argumentos que utilizaríamos para defender un modelo simbólico, en el que su razonamiento es fácilmente interpretable. Esto puede hacer que muchos posibles usuarios se muestren reacios a adoptar el uso de redes neuronales artificiales.

Algunas de las técnicas de aprendizaje automático más versátiles funcionan de forma algo misteriosa. Las decisiones automatizadas con ayuda de redes neuronales o máquinas de vectores soporte son, en gran parte, completamente inescrutables. Aunque funcionen realmente bien, si son incapaces de ofrecer una explicación, puede que no nos dejen usarlas. Tal vez porque alguien se sienta inseguro al no poder justificar una decisión en caso de que dicha decisión se demuestre errónea en el futuro.

Dado que los sistemas de Deep Learning no toman decisiones de la misma forma que nosotros, esto ocasiona la aparición de problemas de seguridad en todos aquellos ámbitos donde se recurre a ellos:

- Dada la importancia que tienen los sistemas de recomendación en muchos aspectos de nuestra vida online, esos sistemas han sido objeto de numerosos ataques que pretenden manipular las recomendaciones que ofrecen. En ocasiones, para promover

un artículo determinado, haciendo que se recomiende más a menudo. En otras ocasiones, para tumbar su reputación [*nuke*] de forma que nunca se recomiende. Denominados ataques por complicidad [*shilling attacks*], los incentivos económicos para los atacantes convierten el diseño de sistemas de recomendación en una guerra de desgaste sin fin (como la de los fabricantes de antivirus con los desarrolladores de nuevos tipos de malware). Estamos en una situación en la que los sistemas automáticos basados en redes neuronales nos pueden ayudar en la construcción de mejores sistemas de recomendación, a la vez que dificultan la interpretación de los resultados que proporcionan e interfieren en nuestra capacidad de detectar manipulaciones.

- Las redes neuronales artificiales han demostrado ser especialmente útiles en el reconocimiento de patrones complejos, como puede ser la identificación de objetos en imágenes. Se ha demostrado que se puede engañar a una red neuronal para que crea percibir cosas que no están realmente en la imagen. Es más, se ha conseguido la creación de imágenes que, siendo indistinguibles de las originales para nosotros, confunden por completo a una red neuronal. Si dotamos a un vehículo autónomo de una cámara con la que interpretar señales de tráfico y semáforos, pensemos en las vulnerabilidades de seguridad de un sistema así cuando la red neuronal es la encargada de determinar el significado de una señal de tráfico y el coche autónomo actúa de acuerdo a la predicción realizada por la red neuronal.

Capítulo 4

Redes Deep Learning

4.1. Tipos de redes Deep Learning

El número de arquitecturas y algoritmos que se utilizan en el aprendizaje profundo es amplio y variado. En este apartado se explorarán algunas de las arquitecturas de aprendizaje profundo que abarcan los últimos 20 años.

En particular, RNN (Recurrent Neural Network) y LSTM (Long/Short Term Memory) son dos de los enfoques más antiguos pero también dos de los más utilizados y se explicarán con más profundidad, ya que se usarán en la aplicación de Python del apartado 5.2.1

En la siguiente imagen podemos ver la evolución de estas arquitecturas:

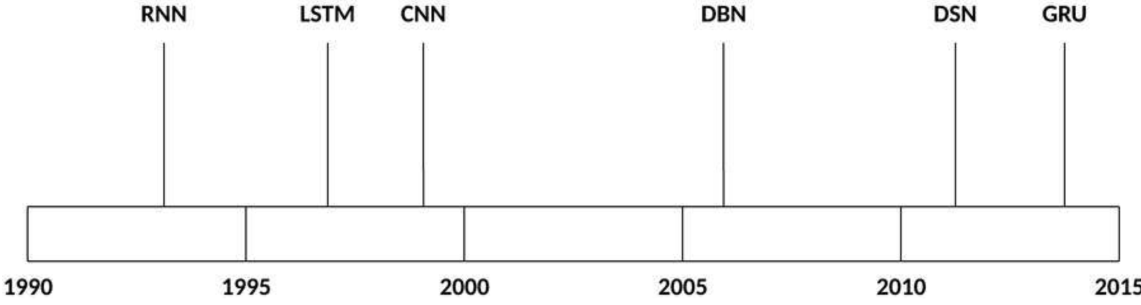


Figura 4.1: Evolución de las redes utilizadas en Deep Learning

Estas arquitecturas se aplican en una amplia gama de escenarios; la siguiente tabla enumera algunas de sus aplicaciones típicas.

Ahora exploremos estas arquitecturas de redes y los métodos que se utilizan para entrenarlas. En las descripciones que se darán a continuación vamos a utilizar la siguiente nomenclatura para los nodos:

Arquitectura	Aplicación
RNN	Reconocimiento de voz, reconocimiento de escritura a mano.
LSTM/GRU	Compresión de texto en lenguaje natural, reconocimiento de escritura a mano, reconocimiento de voz, reconocimiento de gestos, subtítulos de imágenes.
CNN	Reconocimiento de imágenes, análisis de video, procesamiento de lenguaje natural.
DBN	Reconocimiento de imágenes, recuperación de información, comprensión del lenguaje natural, predicción de fallas, etc.
DSN	Recuperación de información, reconocimiento continuo de voz.

Cuadro 4.1: Ejemplos de aplicaciones de redes Deep Learning

Figura 4.2: Tipos de nodos

- **Redes Neuronales Recurrentes (RNN):** Las RNN son una de las arquitecturas de red fundamentales a partir de las cuales se construyen otras arquitecturas de aprendizaje profundo. La diferencia principal entre una red de múltiples capas típica y una red recurrente es que, en lugar de conexiones completamente avanzadas, una red recurrente puede tener conexiones que se realicen en capas anteriores (o en la misma capa). Esta retroalimentación permite a las RNN mantener la memoria de entradas pasadas y modelar problemas a tiempo.

  Las RNN consisten en un rico conjunto de arquitecturas (veremos una topología popular llamada LSTM a continuación). El elemento diferenciador clave es la retroalimentación dentro de la red, que podría manifestarse desde una capa oculta, la capa de salida o alguna combinación de las mismas.

  Las RNN se pueden desplegar en el tiempo y entrenarse utilizando el algoritmo de Backpropagation estándar o utilizando una variante del mismo que se denomina propagación hacia atrás en el tiempo (BPTT).

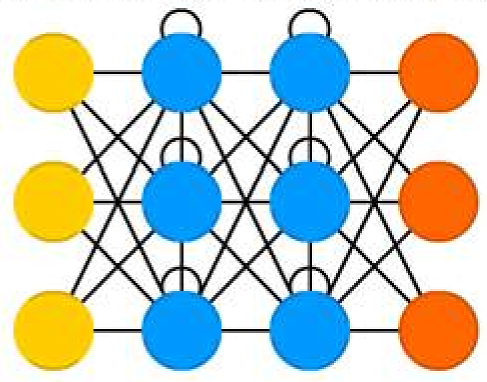
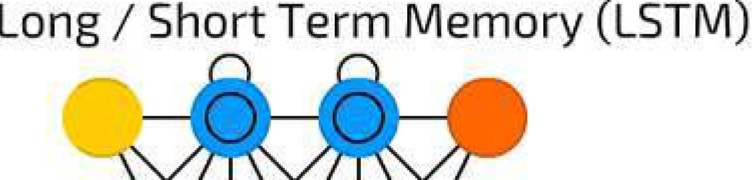


Figura 4.3: RNN

- **Redes LSTM/GRU:** Las redes LSTM (Long/Short Term Memory) fueron creadas en 1997 por Hochreiter y Schimdhuber, pero han crecido en popularidad en los últimos años como una arquitectura RNN para varias aplicaciones. Se encuentran en algunos productos que usamos diariamente como los teléfonos inteligentes.

  La LSTM se apartó de las típicas arquitecturas de redes neuronales basadas en neuronas y, en cambio, introdujo el concepto de una célula de memoria. La celda de memoria puede retener su valor por un tiempo corto o largo en función de sus entradas, lo que permite a la celda recordar lo que es importante y no solo el último valor procesado.

  La celda de memoria LSTM contiene tres puertas que controlan cómo la información fluye dentro o fuera de la celda. La puerta de entrada controla cuándo puede ingresar nueva información a la memoria. La puerta de olvido controla cuando se olvida una parte de la información existente, lo que permite a la celda recordar datos nuevos. Finalmente, la puerta de salida controla cuando la información que está contenida en la celda se utiliza en la salida de la celda. La celda también contiene pesos, que controlan cada puerta.



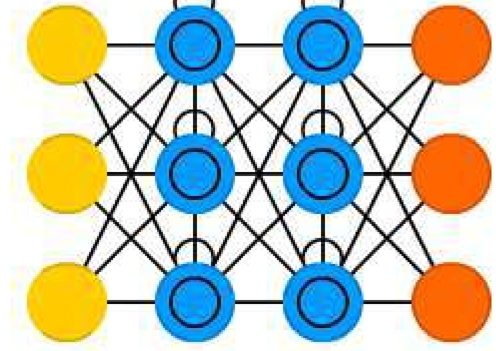


Figura 4.4: LSTM

En 2014, se introdujo una simplificación de la LSTM llamada unidad recurrente cerrada (Gatod Recurrent Unit o abreviadamente GRU) . Este modelo tiene solo dos puertas, eliminando la puerta de salida presente en el modelo LSTM. Para muchas aplicaciones, el GRU tiene un rendimiento similar al LSTM, pero, al ser más simple, involucra menos pesos y permite una ejecución más rápida.

La GRU incluye dos puertas: una puerta de actualización y una puerta de restablecimiento. La puerta de actualización indica la cantidad de contenido de la celda anterior a mantener. La puerta de reinicio define cómo incorporar la nueva entrada al contenido de la celda anterior. Una GRU puede modelar un RNN estándar simplemente configurando la puerta de restablecimiento en 1 y la puerta de actualización en 0.

La GRU es más simple que el LSTM, puede entrenarse más rápidamente y puede ser más eficiente en su ejecución. Sin embargo, el LSTM puede ser más expresivo y, con más datos, puede conducir a mejores resultados.

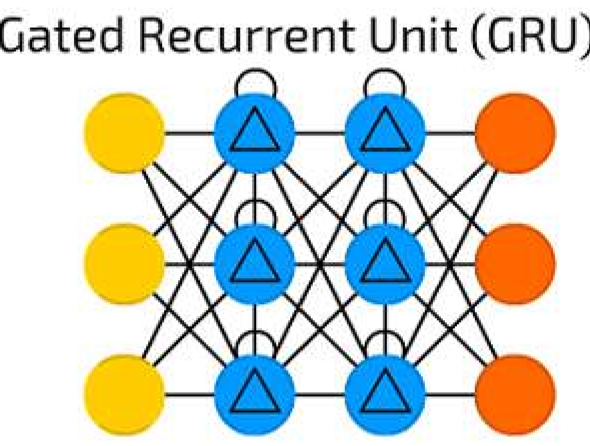


Figura 4.5: GRU

- **Redes Neuronales Convolucionales (CNN):** Una CNN es una red neuronal multicapa inspirada biológicamente en la corteza visual animal. La arquitectura es particularmente útil en aplicaciones de procesamiento de imágenes. La primera CNN fue creada por Yann LeCun; en ese momento, la arquitectura se centraba en el reconocimiento de caracteres manuscritos, como la interpretación de códigos postales. Como una red profunda, las primeras capas reconocen características (como los bordes), y las capas posteriores recombinan estas características en atributos de entrada de nivel superior.

  La principal ventaja de este tipo de redes es que cada parte de la red es entrenada para realizar una tarea; esto reduce significativamente el número de capas ocultas, por lo que el entrenamiento es más rápido. Además, es invariante por traslación de los patrones a identificar.

  Las redes neuronales convolucionales son muy potentes para todo lo que tiene que ver con el análisis de imágenes, debido a que son capaces de detectar las características simples como ,por ejemplo, bordes, lineas, etc y componer en estructuras más complejas hasta detectar lo que se busca.

Una red neuronal convolucional es una red multicapa que consta de capas convolucionales y de reducción alternadas, y, finalmente, tiene capas de conexión total como una red perceptrón multicapa.

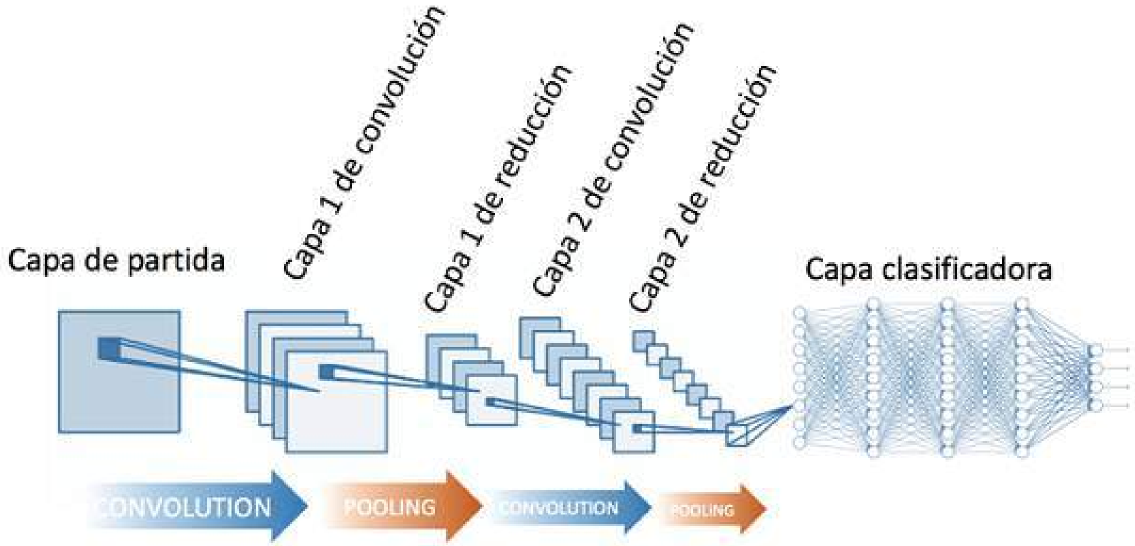


Figura 4.6: CNN

En la convolución se realizan operaciones de productos y sumas entre la capa de partida y los n filtros (o núcleos) que genera un mapa de características. Las características extraídas corresponden a cada posible ubicación del filtro en la imagen original.

La ventaja reside en que la misma neurona sirve para extraer la misma característica en cualquier parte de la entrada; con esto se consigue reducir el número de conexiones y el número de parámetros a entrenar en comparación con una red multicapa de conexión total.

Figura 4.7: Convolución CNN

Después de la convolución se aplica a los mapas de características una función de activación. La función de activación recomendada es signoide ReLU (Rectified Linear Unit), seleccionando una tasa de aprendizaje adecuada monitorizando la fracción de neuronas muertas, también se puede podría probar con *Leaky ReLu* o *Maxout*, pero nunca utilizar la función sigmoide logística.

En la reducción (*pooling*) se disminuye la cantidad de parámetros al quedarse con las características más comunes. La forma de reducir parámetros se realiza mediante la extracción de medidas estadísticas como el promedio o el máximo de una región fija del mapa de características; al reducir características el método pierde precisión aunque mejora su compatibilidad.

Al final de las capas convolucional y de reducción, se suelen utilizar capas completamente conectadas en la que cada pixel se considera como una neurona separada al igual que en un perceptrón multicapa. La última capa de esta red es una capa clasificadora que tendrá tantas neuronas como número de clases a predecir haya.

Además del procesamiento de imágenes, la CNN se ha aplicado con éxito al reconocimiento de video y varias tareas dentro del procesamiento de lenguaje natural. Las aplicaciones recientes de CNN y LSTM han dado lugar a sistemas de subtítulos de imagen y video en los que una imagen o video se resume en lenguaje natural. La CNN implementa el procesamiento de imágenes o video, y la LSTM está capacitada para convertir la salida de la CNN a un lenguaje natural.

- **Redes de creencias profundas (DBN):** La DBN es una arquitectura de red típica pero incluye un algoritmo de entrenamiento novedoso. La DBN es una red multicapa (generalmente profunda, que incluye muchas capas ocultas) en la que cada par de capas conectadas es una máquina de Boltzmann restringida (RBM) que es un tipo de red neuronal recurrente estocástica. Las máquinas de Boltzmann pueden considerarse

como la contrapartida estocástica y generativa de las redes de Hopfield. Fueron de los primeros tipos de redes neuronales capaces de aprender mediante representaciones internas, son capaces de representar y (con tiempo suficiente) resolver complicados problemas combinatorios. Sin embargo, las máquinas de Boltzmann sin restricciones de conectividad no han demostrado ser útiles para resolver los problemas que se dan en la práctica en el aprendizaje o inferencia de las máquinas. Aunque el aprendizaje es por lo general poco práctico en las máquinas de Boltzmann, puede llegar a ser muy eficiente en una arquitectura llamada Máquina de Boltzmann restringida. Esta arquitectura no permite las conexiones entre las unidades de las capas ocultas. Después de entrenar a una RBM las actividades de sus unidades ocultas pueden ser tratadas como datos para el entrenamiento de una RBM de nivel superior. Este método de apilamiento RBM hace que sea posible entrenar muchas capas de unidades ocultas de manera eficiente y que cada nueva capa sea añadida para mejorar el modelo generativo principal. De esta manera, un DBN se representa como una pila de RBM.

Las RBM se asemejan, en la estructura, a las máquinas Boltzmann (BM), pero, debido a su restricción, permiten ser entrenados usando la propagación hacia atrás solo como *feed forward*. Las máquinas Boltzmann son muy similares a las HN (Hopfield), donde algunas celdas están marcadas como entrada y permanecen ocultas. Las celdas de entrada se convierten en salida tan pronto como cada celda oculta actualiza su estado (durante el entrenamiento, las BM / HN actualizan las celdas una por una, y no en paralelo).

En las DBN, la capa de entrada representa las entradas sensoriales en bruto, y cada capa oculta aprende representaciones abstractas de esta entrada. La capa de salida, que se trata de manera algo diferente a las otras capas, implementa la clasificación de red. El aprendizaje se realiza en dos pasos: entrenamiento previo sin supervisión y ajuste fino supervisado.

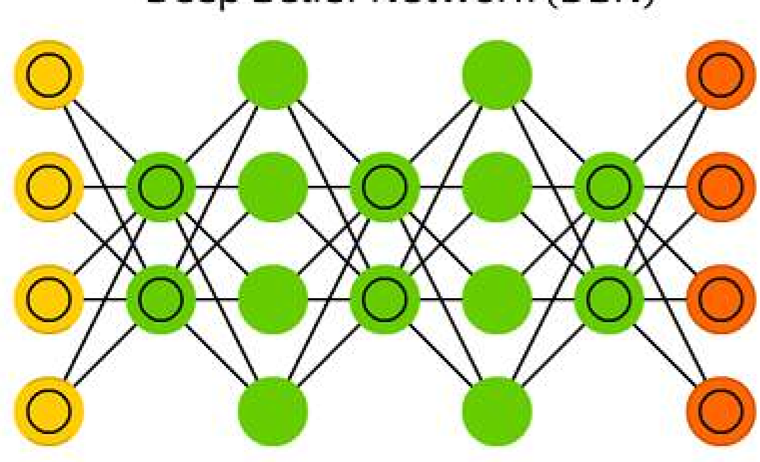


Figura 4.8: Convolución DBN

En el entrenamiento previo sin supervisión, cada RBM está entrenada para reconstruir su entrada (por ejemplo, la primera RBM reconstruye la capa de entrada a la primera capa oculta). La siguiente RBM se entrena de manera similar, pero la

primera capa oculta se trata como la capa de entrada (o visible), y la RBM se entrena utilizando las salidas de la primera capa oculta como las entradas. Este proceso continúa hasta que cada capa está pre-entrenada. Cuando se completa el entrenamiento previo, comienza el ajuste fino. En esta fase, los nodos de salida son etiquetas aplicadas para darles un significado (lo que representan en el contexto de la red). Luego, se aplica el entrenamiento de la red completa mediante el aprendizaje de descenso de gradiente (SGD) o la propagación hacia atrás para completar el proceso de aprendizaje.

- **Redes de apilamiento profundo (DSN):** Una DSN es diferente de los marcos tradicionales de aprendizaje profundo en que, aunque consta de una red profunda, en realidad es un conjunto profundo de redes individuales, cada una con sus propias capas ocultas. Esta arquitectura es una respuesta a uno de los problemas del aprendizaje profundo: la complejidad del entrenamiento. Cada capa en una arquitectura de aprendizaje profundo aumenta exponencialmente la complejidad del entrenamiento, por lo que el DSN considera el entrenamiento no como un problema único, sino como un conjunto de problemas de capacitación individuales.

  La DSN consiste en un conjunto de módulos, cada uno de los cuales es una subred en la jerarquía general del DSN. En una instancia de esta arquitectura, se crean tres módulos para el DSN. Cada módulo consta de una capa de entrada, una sola capa oculta y una capa de salida. Los módulos se apilan uno encima de otro, donde las entradas de un módulo constan de las salidas de la capa anterior y el vector de entrada original. Esta estratificación permite que la red general aprenda una clasificación más compleja de lo que sería posible dado un solo módulo.

  La DSN permite el entrenamiento de módulos individuales de forma aislada, lo que lo hace eficiente dada la capacidad de entrenar en paralelo. El aprendizaje supervisada se implementa como propagación hacia atrás para cada módulo en lugar de propagación hacia atrás en toda la red. Para muchos problemas, las DSN pueden funcionar mejor que las DBN típicas, lo que las convierte en una arquitectura de red popular y eficiente.

4.2. Redes Neuronales Recurrentes (RNN)

Como se comentó al principio de este capítulo, se van a abordar con más profundidad las RNN y, en concreto, las LSTM, ya que son las utilizadas en la aplicación posterior de análisis de sentimientos en Python.

Por su topología, las redes de tipo *feed-forward* carecen por completo de memoria. Cuando las conexiones entre neuronas de la misma capa se generalizan (o se admiten conexiones de salida de una capa como entradas de una capa anterior), obtenemos redes neuronales con memoria: las redes recurrentes (RNNs: *Recurrent Neural Networks*).

Las conexiones de una red recurrente pueden ser dirigidas, como sucede en las redes neuronales multicapa habituales, aunque también pueden ser bidireccionales. En este caso, obtenemos una red que es capaz de completar patrones incompletos, recibidos con ruido u ocultos parcialmente.

Un ejemplo clásico de este tipo de redes recurrentes con conexiones bidireccionales (esto es, enlaces no dirigidos) son las redes de Hopfield (redes neuronales recurrentes con un comportamiento dinámico estable).

Igual que otros tipos de memorias asociativas, las redes de *Hopfield* son capaces de almacenar datos y de recuperarlos incluso cuando se les proporciona una entrada con ruido. Esta señal de entrada se puede interpretar como una versión corrompida de los datos almacenados en la red. La red se encarga de restaurar esos datos y generar en su salida el patrón original completo correspondiente a una entrada incompleta.

Este tipo de redes recurrentes se caracteriza por su comportamiento dinámico complejo. Ante una señal de entrada ambigua, que se pueda interpretar de distintas formas, una red recurrente puede oscilar entre varios estados semiestables. En cierto modo, la red es capaz de dudar. El fenómeno es similar al que experimentamos cuando vemos una imagen ambigua, con varias interpretaciones posibles incompatibles entre sí. Desde el punto de vista de la dinámica de una red neuronal recurrente, cada interpretación corresponde a un atractor estable, con cierto grado de solapamiento entre los estados correspondientes a ambas interpretaciones, de ahí que pueda alternar.

Centrándonos en las redes neuronales recurrentes (RNN) que contienen bucles dirigidos, estos bucles representan la propagación de activaciones a entradas futuras en una secuencia. Una vez entrenado, en lugar de aceptar una única entrada vectorial x como ejemplo de prueba, un RNN puede aceptar una secuencia de entradas vectoriales $(x_1, ..., x_T)$ para valores de T arbitrarios y variables, donde cada x_t tiene la misma dimensión.

Podemos imaginar "desenrollar" un RNN para que cada una de estas entradas x_t tenga una copia de una red que comparta los mismos pesos que las otras copias. Para cada borde de bucle en la red, conectamos el borde al nodo correspondiente en la red de $x_t + 1$, creando así una cadena, que rompe cualquier bucle y nos permite utilizar las técnicas estándar de propagación hacia atrás de las redes neuronales avanzadas.

La definición moderna de "recurrente" fue introducida inicialmente por Jordan (1986) como:
Si una red tiene uno o más ciclos, es decir, si es posible seguir una ruta desde una unidad a sí misma, entonces se hace referencia a la red como recurrente. Una red no recurrente no tiene ciclos. Su modelo en (Jordan, 1986) se conoce más tarde como Jordan Network. Para una red neuronal simple con una capa oculta, con la entrada denotada como X, los pesos de la capa oculta denotados como w_h y los pesos de la capa de salida denotados como w_y, los pesos de cómputo recurrente denotados como w_r, la representación oculta denotada como h_y la salida denotada como y, Jordan Network se puede formular como

$$h^t = \sigma(W_h X - W_r y^{t-1})$$
$$y = \sigma(W_y h^t)$$

Unos años más tarde, Elman (1990) introdujo otro RNN cuando formalizó la estructura recurrente de manera ligeramente diferente. Más tarde, su red se denominó Elman Network.

La red de Elman se formaliza como sigue:

$$h^t = \sigma(W_h X - W_r h^{t-1})$$

$$y = \sigma(W_y h^t)$$

La única diferencia reside en si la información del paso de tiempo anterior se proporciona mediante la salida anterior o la capa oculta anterior.

Se puede observar que no hay una diferencia fundamental entre estas dos estructuras ya que $y_t = W_y h_t$; por lo tanto, la única diferencia radica en la elección de W_r (originalmente, Elman solo introduce su red con $W_r = I$, pero a partir de ahí podrían derivarse casos más generales).

Sin embargo, el paso de Jordan Network a Elman Network aún es notable, ya que presenta la posibilidad de pasar información desde capas ocultas, lo que mejora significativamente la flexibilidad del diseño de la estructura en trabajos posteriores [15].

4.2.0.1. Backpropagation a través del tiempo

La estructura recurrente hace inviable la propagación hacia atrás tradicional debido a que con la estructura recurrente, no hay un punto final en el que la propagación hacia atrás pueda detenerse.

Intuitivamente, una solución es desplegar la estructura recurrente y expandirla como una red neuronal avanzada con ciertos pasos de tiempo y luego aplicar la propagación hacia atrás tradicional en esta red neuronal desplegada. Esta solución se conoce como Backpropagation a través del tiempo (BPTT), propuesta de manera independiente por varios investigadores, entre ellos (Robinson y Fallside, 1987; Werbos, 1988; Mozer, 1989).

Uno de los problemas más fundamentales de la formación de RNN es el problema del desvanecimiento del gradiente, que se presenta en detalle en (Bengio et al., 1994). El problema básicamente establece que para las funciones de activación tradicionales, el gradiente está limitado. Cuando los gradientes se calculan mediante backpropagation siguiendo la regla de la cadena, la señal de error disminuye exponencialmente en los pasos de tiempo que el BPTT puede rastrear, por lo que la dependencia a largo plazo se pierde.

4.2.1. Redes Recurrentes Bidireccionales

A lo largo del desarrollo de las redes neuronales,las redes bidireccionales han estado desempeñando roles importantes (como la Red de campo Hop, RBM, DBM).

La Red Neural Recurrente Bidireccional (BRNN) fue propuesta por Schuster y Paliwal (1997) con el objetivo de introducir una estructura que se desplegó para ser una red neuronal bidireccional. Por lo tanto, cuando se aplica a datos de series de tiempo, no solo se puede pasar la información siguiendo las secuencias temporales naturales, sino que la información adicional también puede proporcionar conocimiento inverso a los pasos de tiempo anteriores.

BRNN está formulado de la siguiente manera:

$$h_1^t = \sigma(W_{h1} X + W_{r1} h_1^{t-1})$$
$$h_2^t = \sigma(W_{h2} X + W_{r2} h_2^{t+1})$$

$$y = \sigma(W_{y1}h_1^t + W_{y2}h_2^t)$$

donde los subíndices 1 y 2 indican las variables asociadas con las capas 1 y 2 ocultas, respectivamente.

Con la introducción de conexiones "recurrentes" del futuro, la propagación hacia atrás a través del tiempo ya no es directamente factible. La solución es tratar este modelo como una combinación de dos RNN: uno estándar y otro inverso, luego aplicar BPTT en cada uno de ellos. Los pesos se actualizan simultáneamente una vez que se calculan dos gradientes.

4.2.2. LSTM

Una de las inestabilidades conocidas de la RNN estándar es el problema del desvanecimiento del gradiente (Pascanu et al., 2012). El problema implica la disminución rápida en la cantidad de activación que se pasa a los pasos de tiempo subsiguientes. Esto limita hasta qué punto en el pasado la red puede recordar.

Para solucionar este problema, se crearon unidades de memoria a corto plazo (LSTM) para reemplazar los nodos recurrentes normales (Hochreiter et al., 1997). Estas unidades introducen una variedad de puertas que regulan la propagación de activaciones a lo largo de la red. Esto, a su vez, permite que una red aprenda cuándo ignorar una nueva entrada, cuándo recordar un estado oculto pasado y cuándo emitir una salida distinta de cero.

Mientras que una RNN puede sobrescribir su memoria en cada paso de tiempo de una manera bastante incontrolada, una LSTM transforma su memoria de una manera muy precisa: mediante el uso de mecanismos de aprendizaje específicos para recordar qué datos recordar, qué actualizar y a cuáles prestar atención. Esto ayuda a mantener un registro de la información durante largos períodos de tiempo.

En sus orígenes, el término "LSTM" se usó para referirse al algoritmo que está diseñado para superar el problema del gradiente de fuga, con la ayuda de una celda de memoria diseñada especialmente. Hoy en día, "LSTM" se usa ampliamente para denotar cualquier red recurrente con esa celda de memoria, que actualmente se conoce como celda LSTM.

LSTM se introdujo para superar el problema de que las RNN no pueden tener dependencias a largo plazo (Bengio et al., 1994). Para superar este problema, se requiere una celda de memoria especialmente diseñada.

LSTM consta de varios componentes:

- **Estados:** valores que se utilizan para ofrecer la información de salida.
  - Datos de entrada: se denota como x.
  - Estado oculto: valores de la capa oculta anterior. Esto es igual que en RNN tradicional. Se denota como h.
  - Estado de entrada: valores que son una combinación (lineal) de estado oculto y entrada del paso de tiempo actual. Se denota como i, y tenemos: $i^t = \sigma(W_{ix}x^t + W_{ih}h^{t-1})$
  - Estado interno: valores que sirven de "memoria". Se denota como m.

- **Puertas:** valores que se utilizan para decidir el flujo de información de los estados.
  - Puerta de entrada: decide si el estado de entrada entra en estado interno. Se denota como g, y tenemos: $g^t = \sigma(W_{gi}i^t)$
  - Puerta de olvido: decide si el estado interno se olvida del estado interno anterior. Se denota como f, y tenemos: $f^t = \sigma(W_{fi}i^t)$
  - Puerta de salida: decide si el estado interno pasa su valor a la salida y al estado oculto del siguiente paso de tiempo. Se denota como o y tenemos: $o^t = \sigma(W_{oi}i^t)$

Finalmente, considerando cómo las puertas deciden el flujo de información de los estados, tenemos las dos últimas ecuaciones para completar la formulación de LSTM:

$$m^t = g^t \circ i^t + f^t m^{t-1} \qquad \text{y} \qquad h^t = o^t \circ m^t$$

donde $\circ$ denota el producto elemento a elemento.

Todos los pesos son parámetros que deben aprenderse durante el entrenamiento. Por lo tanto, en teoría, LSTM puede aprender a memorizar la dependencia durante mucho tiempo si es necesario y puede aprender a olvidar el pasado cuando sea necesario, convirtiéndose en un modelo poderoso. Con esta importante garantía teórica, se han realizado muchos trabajos para intentar mejorar la LSTM. Por ejemplo, Gers y Schmidhuber (2000) agregaron una conexión de mirilla que permite que la puerta use información del estado interno. Cho et al. (2014) introdujo la Unidad Recurrente Bloqueada, conocida como GRU, que simplificó LSTM fusionando el estado interno y el estado oculto en un estado, y fusionando la puerta olvidada y la puerta de entrada en una puerta de actualización simple.

Curiosamente, a pesar de las nuevas variantes de LSTM propuestas, un experimento a gran escala para investigar el rendimiento de las LSTM llegó a la conclusión de que ninguna de las variantes puede mejorar significativamente la arquitectura estándar de LSTM. Probablemente, la mejora de LSTM es en otra dirección, en lugar de actualizar la estructura dentro de una celda.

4.2.3. El futuro de las RNNs

Las RNN se han mejorado de varias maneras diferentes, como ensamblar las piezas junto con el Campo aleatorio condicional (Yang et al., 2016) y junto con los componentes de la CNN (Ma y Hovy, 2016). Además, la operación convolucional se puede integrar directamente en LSTM, resultando en ConvLSTM (Xingjian et al., 2015), y luego este ConvLSTM también se puede conectar con una variedad de componentes diferentes (De Brabandere et al., 2016; Kalchbrenner et al. , 2016).

Se sabe que LSTM y ReLU son buenas soluciones para el problema del desvanecimiento del gradientes. Sin embargo, estas soluciones introducen formas de evitar este problema con un diseño inteligente, en lugar de resolverlo de manera fundamental. Si bien estos métodos funcionan bien en la práctica, el problema fundamental para una RNN general aún no se ha resuelto. Pascanu et al. (2013b) intentó algunas soluciones, pero todavía hay más por hacer.

Capítulo 5

Deep Learning y Procesamiento del Lenguaje Natural

5.1. ¿Qué es el Procesamiento del Lenguaje Natural?

El NLP (Neuro Linguistic Programming) es un subcampo de la informática dirigido a conseguir que los ordenadores entiendan el lenguaje de una manera “natural”, como lo hacen los humanos. Por lo general, esto se refiere a tareas como comprender el sentimiento del texto, el reconocimiento de voz y generar respuestas a las preguntas.

El NLP se ha convertido en un campo de rápida evolución, y cuyas aplicaciones han representado una gran parte de los avances en inteligencia artificial. Algunos ejemplos de implementaciones que utilizan el aprendizaje profundo son los chatbots que manejan las solicitudes de servicio al cliente, la verificación automática de la ortografía en los teléfonos celulares y los asistentes de inteligencia artificial, como Cortana y Siri, en los teléfonos inteligentes. Para aquellos que tienen experiencia en aprendizaje automático y aprendizaje profundo, el procesamiento del lenguaje natural es una de las áreas más emocionantes para que las personas apliquen sus habilidades.

El procesamiento del lenguaje natural (NPL) es una tarea extremadamente difícil en ciencias de la computación. Los idiomas presentan una gran variedad de problemas que varían de un idioma a otro. La estructuración o extracción de información significativa a partir de texto libre representa una gran solución, si se hace de la manera correcta. Anteriormente, los científicos informáticos rompían un lenguaje en sus formas gramaticales, como partes del habla, frases, etc., utilizando algoritmos complejos. Hoy en día, el aprendizaje profundo es clave para realizar los mismos ejercicios.

5.2. Análisis de sentimientos

El análisis de sentimientos (también conocido como minería de opinión) es un área de investigación activa en el procesamiento del lenguaje natural. El objetivo de la tarea es identificar, extraer y organizar los sentimientos de los textos generados por los usuarios en redes sociales, blogs o reseñas de productos.

Durante las dos últimas décadas, muchos estudios en la literatura explotan los enfoques de aprendizaje automático para resolver tareas de análisis de sentimientos desde diferentes perspectivas. Dado que el rendimiento de un aprendiz de máquina depende en gran medida de las opciones de representación de datos, muchos estudios se dedican a crear un potente extractor de características con experiencia en dominios e ingeniería cuidadosa. Recientemente, los enfoques de aprendizaje profundo emergen como poderosos modelos computacionales que descubren intrincadas representaciones semánticas de textos automáticamente a partir de datos sin ingeniería de características. Estos enfoques han mejorado el estado de la técnica en muchas tareas de análisis de sentimientos, incluida la clasificación de sentimientos, la extracción de opiniones, el análisis de sentimientos detallado, etc.

5.2.1. Aplicación

Con objeto de comprobar la potencialidad del Aprendizaje Profundo en el Análisis de Sentimientos, se realizará una aplicación en Python en la cual se implementará un modelo que ingresa una oración y encuentra el emoji más apropiado para usar con esta oración. El código está adaptado del Curso de Andrew Ng 'Modelos secuenciales' [2].

5.2.1.1. Conjunto de datos

Se dispoe de un pequeño conjunto de datos (X, Y) donde:

- X contiene 127 frases.
- Y contiene una etiqueta entera entre 0 y 4 correspondiente a un emoji para cada oración.

CÓDIGO	EMOJI	ETIQUETA
:heart:	❤️	0
:baseball:	⚾	1
:smile:	😀	2
:disappointed:	😞	3
:fork_and_knife:	🍴	4

Figura 5.1: Emoji y etiqueta

Ejemplo del conjunto de datos:

Cuadro 5.1: Ejemplo conjunto de datos

X(frase)	Y(etiqueta)
I love you	0

X(frase)	Y(etiqueta)
Congrats on the new job	2
Happy new year	2
I am frustrated	3
I want to have sushi for dinner	4
He can pitch really well	1

5.2.1.2. Incrustaciones

El algoritmo Vectores Globales para la Representación de Palabras, o GloVe, es una extensión del método Word2Vec para el aprendizaje eficiente de vectores de palabras, desarrollado por Pennington [6], en Stanford. Las representaciones clásicas de modelos vectoriales espaciales de palabras fueron desarrolladas usando técnicas de factorización matricial tales como el Análisis Semántico Latente (LSA, por sus siglas en inglés) que hacen un buen trabajo en el uso de estadísticas globales de texto, pero que no son tan buenas como los métodos aprendidos como Word2Vec para capturar el significado y demostrarlo en tareas como el cálculo de analogías.

GloVe es un enfoque que combina las estadísticas globales de técnicas de factorización matricial como LSA con el aprendizaje local basado en el contexto en Word2Vec. Es uno de los métodos más nuevos para crear modelos de espacio vectorial de semántica de palabras, más comúnmente conocidos como incrustaciones de palabras. En lugar de usar una ventana para definir el contexto local, GloVe construye una matriz explícita de palabra-contexto o co-ocurrencia de palabras usando estadísticas a través de todo el corpus del texto. El resultado es un modelo de aprendizaje que puede dar como resultado mejores incrustaciones de palabras.

5.2.1.2.1. Factorización de matriz global

En el procesamiento del lenguaje natural, la factorización de matriz global es el proceso de utilizar métodos de factorización de matrices del álgebra lineal para realizar la reducción de rango en una matriz de frecuencia de término grande. Estas matrices generalmente representan las frecuencias de términos en los documentos , en las que las filas son palabras y las columnas son documentos (o, a veces, párrafos), o frecuencias de términos, que tienen palabras en ambos ejes y miden la coexistencia. La factorización de matriz global aplicada a matrices de frecuencia de documento de término es comúnmente conocida como análisis semántico latente (LSA) . En el análisis semántico latente, la matriz de alta dimensión se reduce a través de la descomposición de valores singulares (SVD). Es esencialmente una factorización de una matriz general M de dimensión $m \times n$ en un producto $U\Sigma V*$, donde U tiene dimensión $m \times m$, Σ es una matriz mxn diagonal rectangular (cuyas entradas no nulas son conocidas como los valores singulares de m), y V tiene dimensión $n \times n$ y U y V ortogonales.

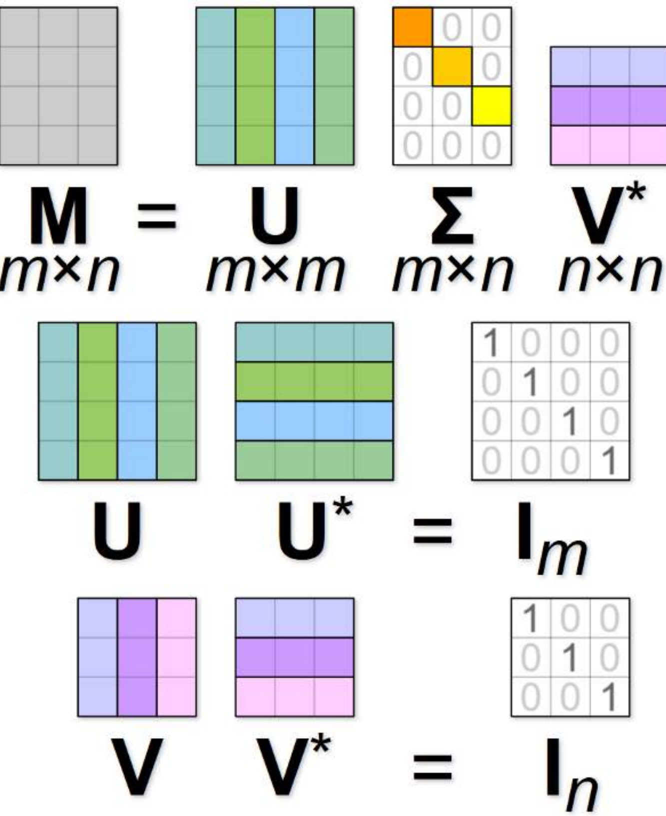


Figura 5.2: Matrices

Recordemos que la transpuesta conjugada $A*$ de una matriz A es la matriz dada tomando el conjugado complejo de cada entrada en la transpuesta (la reflexión sobre el diagonal) de A . Una matriz unitaria es cualquier matriz cuadrada cuya transposición conjugada es su inversa, es decir, una matriz A tal que $AA* = A*A = I$. Esta factorización se usa para encontrar una aproximación de bajo rango a M, seleccionando primero r, el rango deseado de nuestra matriz de aproximación M' , y luego computando Σ' , que es solo Σ pero con solo los r mayores valores singulares. Entonces la aproximación viene dada por $M' = U\Sigma'V*$.

Estas aproximaciones de bajo rango nos proporcionan incrustaciones en el espacio vectorial de tamaño razonable de las estadísticas globales del corpus.

5.2.1.2.2. Probabilidades de co-ocurrencia

Las matrices de término a término codifican la frecuencia con la que aparecen los términos en el contexto de los demás al enumerar cada muestra única en el corpus a lo largo de los dos ejes de una gran matriz bidimensional. La realización de la factorización de matriz nos da una aproximación de bajo rango de la totalidad de los datos contenidos en la matriz original. Sin embargo, como explicaremos a continuación, los autores de GloVe descubrieron a través de métodos empíricos que, en lugar de conocer las probabilidades brutas de co-ocurrencia, puede tener más sentido aprender las proporciones de estas probabilidades de co-ocurrencia, que parecen discriminar mejor.

Supongamos que deseamos estudiar la relación entre dos palabras, i= hielo y j=vapor. Haremos esto examinando las probabilidades de co-ocurrencia de estas palabras con varias

palabras de "prueba". Definimos la probabilidad de co-ocurrencia de una palabra arbitraria i con una palabra arbitraria j como la probabilidad de que la palabra j aparezca en el contexto de la palabra i. Esto está representado por la ecuación y las definiciones que se presentan a continuación.

Tenemos las siguientes probabilidades de co-ocurrencia:

$$P_{ij} = P(j|i) = \frac{X_{ij}}{X_i} = \frac{X_{ij}}{\sum_k X_{ik}}$$

donde X_{ij} es el número de veces que la palabra j aparece en el contexto de la palabra i y X_i se define como el número de veces que aparece una palabra en el contexto de la palabra i, esto es, la suma sobre todas las palabras k del número de veces que la palabra k aparece en el contexto de la palabra i.

Si elegimos una palabra de sonda $k = sólido$ que está estrechamente relacionado con $i = hielo$ pero no con $j = vapor$, esperamos que la proporción $\frac{P_{ik}}{P_{jk}}$ de probabilidades de co-ocurrencia sea grande, ya que *sólido* debería aparecer en el contexto del *hielo* más a menudo de lo que aparece en el contexto del *vapor*, ya que el *hielo* es un *sólido* y el *vapor* no lo es. Por el contrario, para una elección de $k = gas$, esperaríamos que la misma proporción fuera pequeña, ya que el vapor está más relacionado con el *gas* que con el *hielo*. Luego también tenemos palabras como *agua*, que están estrechamente relacionadas con el *hielo* y el *vapor*, pero no más que la otra. Y también tenemos palabras como la *moda* que no están estrechamente relacionadas con ninguna de las palabras en cuestión. Tanto para el *agua* como para la *moda*, esperamos que nuestra proporción sea cercana a 1, ya que no debería haber ningún sesgo respecto a *hielo* o *vapor*.

Ahora es importante tener en cuenta que, dado que estamos tratando de determinar la información sobre la relación entre las palabras *hielo* y *vapor*, el agua no nos da mucha información útil. Para fines discriminatorios, no nos da una buena idea de cómo el vapor está "muy alejado" del *hielo*, y la información de que el *vapor*, el *hielo* y el *agua* están relacionados está ya capturada en la información discriminatoria entre *hielo* y *agua*, y *vapor* y *agua*. Palabras que no nos ayudan a distinguir entre i y j se conocen como puntos de ruido, y el uso de la relación entre las probabilidades de co-ocurrencia ayuda a filtrar estos puntos de ruido. Esto está bien ilustrado por los datos reales de estos puntos de ejemplo, que se muestra en la tabla 5.2, donde se presentan las probabilidades de co-ocurrencia de palabras objetivo de *hielo* y *vapor* con palabras de contexto seleccionadas de un corpus de 6 mil millones de fichas. Solo en la proporción el ruido de las palabras no discriminatorias como el *agua* y la *moda* se cancela, de modo que los valores grandes (mucho mayores que 1) se correlacionan bien con las propiedades específicas del *hielo*, y los valores pequeños (mucho menores que 1) se correlacionan bien con las propiedades específicas de *vapor*.

Cuadro 5.2: Probabilidades de co-ocurrencia de palabras objetivo de hielo y vapor con palabras de contexto

Probabilidad y proporción	k=sólido	k=gas	k=agua	k=moda
P(k/hielo)	0.00019	6.6e-05	0.003	1.7e-05
P(k/vapor)	2.2e-05	0.00078	0.0022	1.8e-05
P(k/hielo)/P(k/vapor)	8.9	0.085	1.36	0.96

Queremos tomar datos del corpus en forma de estadísticas globales y aprender una función que nos brinde información sobre la relación entre dos palabras en dicho corpus, dadas solo las palabras en sí. Jeffrey Pennington, Richard Socher, Christopher D. Manningas [6] han descubieto que las proporciones de probabilidades de co-ocurrencia son una buena fuente de esta información, por lo que sería bueno si nuestra función se asignara desde el espacio de dos palabras para comparar, así como una palabra de contexto al espacio de cocientes de probabilidad de co-ocurrencia. Así que deja que la función que nuestro modelo está aprendiendo sea dada por F. Estos autores dan una interpretación "ingenua" del modelo deseado como:

$$\text{Modelo Naive } F(w_i, w_j, \tilde{w_k}) = \frac{P_{ik}}{P_{jk}}$$

Las w son vectores de palabras con valores reales. Ahora, como queremos codificar información sobre las relaciones entre dos palabras, los autores sugieren usar diferencias vectoriales como entradas para nuestra función. Luego tenemos lo siguiente:

$$\text{Modelo Vector Diferencia } F(w_i - w_j, \tilde{w_k}) = \frac{P_{ik}}{P_{jk}}$$

La función final del modelo GloVe será considerablemente más compleja para reflejar con precisión ciertas simetrías deseables, ya que la distinción entre las palabras i y j debe ser invariante bajo la conmutación de las entradas. Los autores también diseñan un esquema de ponderación de co-ocurrencias para reflejar la relación entre la frecuencia y la relevancia semántica.

Tenemos dos vectores de palabras con los que nos gustaría discriminar, y un vector de palabras de contexto que se utiliza para este efecto. Nuestro modelo simplemente recorre directamente las probabilidades de los vectores. Desafortunadamente, hay una gran cantidad de funciones diferentes que satisfarán estas restricciones y, por lo tanto, debemos encontrar una que refleje mejor la relación que estamos tratando de modelar, es decir, la similitud de significado.

Entonces, los autores decidieron usar la diferencia vectorial de las dos palabras i y j que estamos comparando como una entrada en lugar de estas dos palabras individualmente, ya que nuestra salida es una relación entre sus probabilidades de co-ocurrencia con la palabra de contexto. Así que ahora tenemos dos argumentos, el vector de palabras de contexto y la diferencia vectorial de las dos palabras que estamos comparando. Dado que los autores desean llevar los valores a valores escalares (debe tenerse en cuenta que la proporción de probabilidades es un escalar), se toma el producto puntual de estos dos argumentos, por lo que la siguiente iteración de nuestro modelo se ve así:

$$\text{Modelo Entrada Escalar } F((w_i - w_j)\tau_{\tilde{w_k}}) = \frac{P_{ik}}{P_{jk}}$$

Tratamos a continuación de resolver el problema del etiquetado de ciertas palabras como "palabras de contexto". El problema con esto es que la distinción entre vectores de palabras ordinarios y vectores de palabras de contexto es en realidad arbitraria: no hay distinción. Deberíamos poder intercambiarlos sin causar problemas. La forma de

solucionar esto es exigiendo que F sea un homomorfismo del grupo aditivo de números reales al grupo multiplicativo de números reales positivos.

Recordemos de la teoría de grupos elementales que un homomorfismo es una correspondencia bien definida que preserva la operación de grupo:

$$\text{Definición de Homomorfismo:}$$
$$F(a+b) = F(a)F(b) \quad \forall a, b \in \mathbb{R}$$

Debe tenerse en cuenta la adición dentro del dominio de la función y la multiplicación en el espacio de destino. El dominio de nuestra función ahora es escalar, específicamente todos los números reales. Eso significa que cualquier entrada debe ser el producto puntual de dos vectores de palabras, en lugar de ser un vector de una sola palabra. Esto se debe a que si solo tomamos un vector de una sola palabra como entrada, no sería escalar. Así que podemos pensar en a y b en la condición anterior como productos de puntos de dos vectores de palabras arbitrarios w_a y v_a y w_b y v_b. Si V es el espacio vectorial donde residen todos nuestros vectores de palabras, podemos reescribir la condición:

$$\text{Definición de Homomorfismo (Vectorial):}$$
$$F(w_a^T v_a + w_b^T v_b) = F(w_a^T v_a)F(w_b^T v_b)) \quad \forall w_a^T, v_a, w_b^T, v_b \in \mathbb{R}$$

Queremos definir todo en términos de diferencias de vectores. Entonces, en lugar de agregar el dominio, agregaremos el inverso aditivo, es decir, restar. Y ya que queremos que esto sea un homomorfismo, esto corresponderá a la multiplicación por el inverso multiplicativo en el espacio objetivo (el espacio objetivo es el grupo de números reales positivos bajo multiplicación). Y esto es solo división. Entonces tenemos:

$$F(w_a^T v_a - w_b^T v_b) = \frac{F(w_a^T v_a)}{F(w_b^T v_b))} \quad \forall w_a^T, v_a, w_b^T, v_b \in \mathbb{R}$$

Renombramos para reflejar los vectores de palabras de contexto v_a y v_b siendo iguales, y haciendo uso de la distributividad en el espacio euclidiano, llegamos a la condición que dan los autores:

$$\text{Condición Homomorfismo Glove: } F((w_i - w_j)\tau_{\tilde{w}_k}) = \frac{F(w_i^T \tilde{w}_k)}{F(w_j^T \tilde{w}_k)}$$

Ahora, al establecer esta ecuación igual al modelo de entrada escalar que derivamos anteriormente, tenemos:

$$\frac{P_{ik}}{P_{jk}} = \frac{F(w_i^T \tilde{w}_k)}{F(w_j^T \tilde{w}_k)}$$

Tenemos la siguiente definición natural para las cantidades que estamos dividiendo a la izquierda:

$$F(w_i \tau_{\tilde{w}_k}) = P_{ik} = \frac{X_{ik}}{X_i}$$

Como ya se ha indicado anteriormente, X_{ik} representa el número de veces que la palabra k aparece en el contexto de la palabra i, y X_i es el número de veces que cualquier palabra aparece en el contexto de la palabra i.

Ahora, lo que queda es encontrar una función F que se comporte como la arbitraria que hemos descrito anteriormente. Comenzamos con algo que nos dé un homomorfismo natural entre los números reales aditivos y multiplicativos, es decir, una función que convierta la suma en multiplicación, o viceversa, siempre que tengamos una inversa donde la necesitemos. La opción clara es una función del tipo a^x para alguna constante a. Para simplificar aquí, usaremos e , por lo tanto, sea F la función exponencial. Luego, tomando el logaritmo natural de ambos lados en la ecuación anterior, obtenemos:

$$F(w_i \tau_{\tilde{w_k}}) = log(P_{ik}) = log\left(\frac{X_{ik}}{X_i}\right) = log(X_{ik}) - log(X_i)$$

Queremos "intercambiar" la simetría de nuestra función. Esto significa que queremos poder cambiar los lugares de w_i y w_k sin alterar el lado derecho. Al recordar la definición de X_{ik}, el número de veces que la palabra k aparece en el contexto de la palabra i, vemos que $X_{ik} = X_{ki}$, ya que la palabra k aparece en el contexto de la palabra i si y solo si la palabra i aparece en el contexto de la palabra k. Entonces tendríamos esta simetría en la ecuación anterior si no fuera por el término $logX_i$. Pero como este término es independiente de la elección de la palabra k, podemos reemplazarlo con un término de sesgo para la palabra i. Introducir un término similar para la palabra k hace que nuestra función sea simétrica:

$$w_i \tau_{\tilde{w_k}} + b_i + \tilde{b_k} = log(X_{ik})$$

Se puede pensar en estos términos de sesgo como funciones de la palabra i y la palabra k. Ya que los estamos agregando a nuestra expresión, la conmutación de la suma guarda nuestra simetría. Esto se puede verificar simplemente cambiando los lugares de las palabras i y k a ambos lados de la ecuación anterior y observando que la igualdad aún se mantiene. Dado que los vectores de palabras son la información que nos interesa aprender, los sesgos también deben aprenderse.

Sin embargo, la función anterior (tratada como una función de X_{ik}) no está bien definida cuando el número de co-ocurrencias es cero. Este problema se puede solucionar cambiando el argumento, es decir, tomando el registro $(1 + X_{ik})$. Además, que todo este enfoque sigue siendo defectuoso porque la función pone un peso igual en todas las entradas de la matriz de co-ocurrencia del término, incluso aquellas X_{ik} que son muy pequeñas o cero (recuérdese que son enteros). Se podría argumentar que la entrada del logaritmo constituye algún tipo de ponderación implícita porque este término se ve afectado por el tamaño de X_{ik}, pero el propio registro reduce drásticamente el efecto de la variación en la magnitud de la entrada. En cambio, los autores sugieren ponderar cada ejemplo de entrenamiento aproximadamente proporcional al valor de X_{ik}. De esa manera, cuando estamos aprendiendo los vectores de palabras y los sesgos, ya sea por algún tipo de regresión u otro algoritmo, las co-ocurrencias de cero no son enfatizadas, ya que los valores bajos son probablemente ruidosos, y los valores de co-ocurrencia altos se enfatizan Los autores incorporan esto en un problema de regresión de mínimos cuadrados donde se intenta minimizar la expresión

$$J = \sum_{i,j=1}^{V} f(X_{ij})(w_i \tau_{\tilde{w_k}} + b_i + \tilde{b_k})^2$$

donde f es nuestra función de ponderación, debe tener las siguientes propiedades:

1. $f(0) = 0$. Si se supone que f es contínua, queremos que vaya a cero lo suficientemente rápido como para que $f(x)log^2(x)$ tenga un límite finito, ya que, de lo contrario, para los recuentos de co-ocurrencia de cero, nuestra función puede querer "explotar” Las magnitudes del producto de puntos y los sesgos para compensar el registro.

2. $f(x)$ no decreciente. Esto trata el problema que describimos en el párrafo anterior. Asegura que las entradas de la matriz pequeña no estén sobreponderadas y que las cuentas más grandes sean más altas. Los autores señalan que no es irrazonable tener el 85

3. $f(x)$ debe ser relativamente pequeño para valores grandes de x , es decir, no explota demasiado rápido. Un crecimiento más rápido que el de x^2 probablemente no funcionaría tan bien.

Por supuesto, un gran número de funciones satisfacen estas propiedades, pero una clase de funciones que puede funcionar bien se puede parametrizar como,

$$y = \begin{cases} \left(\dfrac{x}{x_{max}}\right)^{\alpha} & \text{si} \quad x < x_{max} \\ 1 & \text{en caso contrario} \end{cases}$$

5.2.1.3. Aplicación en Python

5.2.1.3.1. Introducción a Python

Python es un lenguaje de programación interpretado cuya filosofía hace hincapié en una sintaxis que favorezca un código legible. Se trata de un lenguaje de programación multiparadigma, ya que soporta orientación a objetos, programación imperativa y, en menor medida, programación funcional.

Por otro lado, Python es un lenguaje de programación que se usa en muchos casos en el mundo, el cual se identifica como un lenguaje de programación de alto nivel, razón por la que es más sencillo de aprender y con la ventaja de ser de código abierto.

Uno de los objetivos que se persigue con este lenguaje de programación es que se automaticen procesos para que así se consiga ahorrar tiempo y evitar complicaciones. Por esa razón son varias las soluciones que se logran con pocas líneas de código en este programa. Por otro lado, Python es muy pertinente para el trabajo con volúmenes de datos grandes al favorecer su extracción y procesamiento.

Destaca por la versatilidad del lenguaje, sus plantillas, módulos, paquetes, frameworks, bibliotecas, sistemas de gestión y más. Se lo puede utilizar para:

- Cálculos científicos o de ingeniería.
- Desarrollo web.
- Videojuegos o similares.

- Programas gráficos.
- Distintas aplicaciones.

Python es definido como un lenguaje de programación multiparadigma. Lo anterior implica que no se trata de forzar a quienes programas a que interioricen un estilo de programación en especial, sino que son varios los estilos que se pueden implementar:

- Programación imperativa.
- Programación orientada a objetos.
- Programación funcional.
- Otros paradigmas al incluir extensiones.

Por otro lado, Python se caracteriza porque usa tipado dinámico y conteo de referencias al administrar la memoria. Además, se destaca su resolución dinámica de nombres, por ende, enlaza un método y un nombre de variable al ejecutar el programa.

Otro de los objetivos al diseñar este lenguaje de programación es facilitar su extensión. Por esa razón es relativamente sencillo que se escriban módulos nuevos en C o C++. Python puede ser incluido en ciertas aplicaciones que requieran de una interfaz que se pueda programar.

El creador de Python fue Guido van Rossum y lo hizo durante los últimos años de la década de los ochenta. El nombre del lenguaje de programación se debe a que este hombre es fan de la serie Monty Python's Flying Circus y un diciembre de 1990 tomó la decisión de bautizar de este modo a su proyecto.

A grandes rasgos la historia de Python nos dice que nació como una búsqueda de un sucesor para el lenguaje de programación ABC, que era capaz de manejar excepciones o de poder interactuar con Amoeba, un sistema operativo. Fue en 1991 cuando se publicó el código de Python en su versión 0.9.0 y a partir de allí se fue complejizando.

Con el paso de los años se ha dado la evolución de Python y se han ido liberando nuevas versiones que mejoran las características ya conocidas y que a su vez se plantean con base en la filosofía de Python ya descrita. En la actualidad es uno de los lenguajes de programación que más se utiliza para toda clase de desarrollos y hay grandes empresas detrás de su uso y mejora.

5.2.1.3.2. Librerías utilizadas

A continuación se describen las principales librerías de Python que se han utilizado en el desarrollo de la aplicación:

- Csv: implementa clases para leer y escribir datos tabulares en formato CSV. Permite a los programadores decir, "escriba estos datos en el formato preferido por Excel" o "lea los datos de este archivo generado por Excel", sin conocer los detalles precisos del formato CSV utilizado por Excel. Los programadores también pueden describir los formatos CSV entendidos por otras aplicaciones o definir sus propios formatos CSV para propósitos especiales.
- NumPy: NumPy es el paquete fundamental para la computación científica con Python. Contiene entre otros recursos:

  - Un poderoso objeto de matriz N-dimensional.
  - Funciones sofisticadas (difusión).
  - Herramientas para la integración de código C / C ++ y Fortran.
  - Álgebra lineal útil, transformada de Fourier y capacidades de números aleatorios.

- Emoji: permite implementar emoticonos en el código Python.

- Pandas: es una librería para el análisis de datos que cuenta con las estructuras de datos que necesitamos para limpiar los datos en bruto y que sean aptos para el análisis (por ejemplo, tablas). Es importante señalar aquí que, dado que *pandas* lleva a cabo tareas importantes, como alinear datos para su comparación, fusionar conjuntos de datos, gestión de datos perdidos, etc., se ha convertido en una librería muy importante para procesar datos a alto nivel en Python (es decir, estadísticas). Pandas fue diseñada originalmente para gestionar datos financieros, y como alternativo al uso de hojas de cálculo (es decir, Microsoft Excel).

  La estructura de datos básica de pandas se denomina DataFrame, que es una colección ordenada de columnas con nombres y tipos, parecido a una tabla de base de datos, donde una sola fila representa un único caso (ejemplo) y las columnas representan atributos particulares. Cabe señalar aquí que elementos en distintas columnas pueden ser de diferentes tipos.

  Por lo tanto, como resumen decir que *pandas* nos proporciona las estructuras de datos y funciones necesarias para el análisis de datos.

- Matplotlib: es una librería de gráficos 2D en Python que produce imágenes de calidad para publicaciones en una variedad de formatos de papel y entornos interactivos en todas las plataformas. Se puede usar en scripts de Python, el shell de python e ipython, servidores de aplicaciones web y seis kits de herramientas de interfaz gráfica de usuario. Puede generar gráficos, histogramas, espectros de potencia, gráficos de barras, gráficos de error, diagramas de dispersión, etc., con solo unas pocas líneas de código.

- Scikit-learn: es una librería de python para Machine Learning y Análisis de Datos. Está basada en NumPy, SciPy y Matplotlib. La ventajas principales de scikit-learn son su facilidad de uso y la gran cantidad de técnicas de aprendizaje automático que implementa. Con scikit-learn podemos realizar aprendizaje supervisado y no supervisado. Podemos usarlo para resolver problemas tanto de clasificación como de regresión.

  Es muy fácil de usar porque tiene una interfaz simple y muy consistente. El interfaz es muy fácil de aprender. Permite cambiar de técnica de machine learning combiando solo una línea de código. Otro punto a favor de scikit-learn es que los valores de los hiper-parámetros tienen unos valores por defecto adecuados para la mayoría de los casos.

  Estas son algunas de las técnicas de aprendizaje automático que podemos usar con scikit-learn:

  - regresión lineal y polinómica
  - regresión logística

- máquinas de vectores de soporte
- árboles de decisión
- bosques aleatorios (random forests)
- agrupamiento (clustering)
- modelos basados en instancias
- clasificadores bayesianos
- reducción de dimensionalidad
- detección de anomalías
- etc.

- Keras: Keras es una API de redes neuronales de alto nivel, escrita en Python y capaz de ejecutarse sobre TensorFlow, CNTK o Theano. Fue desarrollado con el propósito de permitir la experimentación rápida. Poder pasar de la idea al resultado con el menor retraso posible es clave para hacer una buena investigación. Keras se usa como una librería de aprendizaje profundo que:
  - Permite la creación de prototipos fácil y rápida (a través de la facilidad de uso, la modularidad y la extensibilidad).
  - Admite redes convolucionales y redes recurrentes, así como combinaciones de las dos.
  - Funciona a la perfección en CPU y GPU.
- Tensorflow: es una librería de python, desarrollada por Google, para realizar cálculos numéricos mediante diagramas de flujo de datos. En vez de codificar un programa, codificamos un grafo cuyos nodos con operaciones matemáticas y cuyas aristas representan los tensores (matrices de datos multidimensionales). Con esta computación basada en grafos, TensorFlow puede usarse para deep learning y otras aplicaciones de cálculo científico.

  TensorFlow permite, por ejemplo, que el grafo que representa la red neuronal profunda y sus datos, se pueda ejecutar en una o varias CPU o GPU en un PC, en un servidor o en un móvil.

5.2.1.3.3. Aplicación

Se muestra a continuación el código fuente de la aplicación desarrollada.

Importamos todas las librerías necesarias:

```
import csv
import numpy as np
import emoji
import pandas as pd
import matplotlib.pyplot as plt
from sklearn.metrics import confusion_matrix
import numpy as np
from keras.models import Model
```

```
from keras.layers import Dense, Input, Dropout, LSTM, Activation
from keras.layers.embeddings import Embedding
from keras.preprocessing import sequence
from keras.initializers import glorot_uniform
```

```
Using TensorFlow backend.
```

Creamos una funcion que lea el vector Glove:

```
def read_glove_vecs(glove_file):
    with open(glove_file, 'r', encoding="utf8") as f:
    #abrimos el archivo glove ('r'==read)
        words = set()
        # creamos un conjunto
        word_to_vec_map = {}
        # creamos un diccionario
        for line in f:
        # leemos cada linea del fichero
            line = line.strip().split()
            # le quitamos el espacio del principio
            # y del final y separamos cada linea por espacio
            # y creamos una lista
            curr_word = line[0]
            # primer elemento de cada fila
            words.add(curr_word)
            # le añadimos a words el curr_word
            word_to_vec_map[curr_word] = np.array(line[1:],
            dtype=np.float64)
            # le asignas a cada palabra la matriz
            # de los numeros correspondientes

        i = 1
        words_to_index = {}
        index_to_words = {}
        for w in sorted(words):
            words_to_index[w] = i
            index_to_words[i] = w
            i = i + 1
    return words_to_index, index_to_words, word_to_vec_map
```

Calculamos valores de softmax para cada conjunto de puntuaciones en x:

```
def softmax(x):
    e_x = np.exp(x - np.max(x))
    return e_x / e_x.sum()
```

Leemos nuestro csv y nos devuelve por un lado la frase y por otro el emoji que le hemos asignado:

```
def read_csv(filename = 'emojify_data.csv'):
    phrase = []
```

```
    emoji = []

    with open (filename) as csvDataFile:
        csvReader = csv.reader(csvDataFile)

        for row in csvReader:
            phrase.append(row[0])
            emoji.append(row[1])

    X = np.asarray(phrase)
    Y = np.asarray(emoji, dtype=int)

    return X, Y
```

```
def convert_to_one_hot(Y, C):
# C = numero de casos
# Y = lo que queremos convertir
    Y = np.eye(C)[Y.reshape(-1)]
    # np.eye = matriz id de C
    # reshape -1 = cambio fila/columnas
    # para realizar el producto
    return Y
```

Cargamos los emojis que vamos a utilizar:

```
emoji_dictionary = {"0": "\u2764\uFE0F",
                    "1": ":baseball:",
                    "2": ":smile:",
                    "3": ":disappointed:",
                    "4": ":fork_and_knife:"}
```

Convertimos una etiqueta en el código emoji correspondiente listo para salir:

```
def label_to_emoji(label):
    return emoji.emojize(emoji_dictionary[str(label)], use_aliases=True)
    # convierte el numero a emoji
```

Convertimos una matriz de frases en una matriz de índices que corresponden a las palabras en las frases. La forma de salida debe ser tal que se pueda dar a “Embedding()”

Argumentos:

- X: matriz de frases de forma (m, 1)
- word_to_index: un diccionario que contiene cada palabra asignada a su índice.
- max_len: número máximo de palabras en una frase. Puede asumir que cada frase en X no es más larga que esta.

Devuelve:

- Índices X: matriz de índices correspondientes a palabras en las frases de X, de forma (m, max len)

```
import numpy as np
np.random.seed(1)
def sentences_to_indices(X, word_to_index, max_len):

    m = X.shape[0]  # número de registros del
    # conjunto de entrenamiento

    # Inicializamos X_indices como una matriz numpy de ceros
    # de tamaño (m,max_len)
    X_indices = np.zeros((m, max_len))

    for i in range(m):  # bucle sobre ejemplos de entrenamiento
        # Convertimos la frase de entrenamiento en minúsculas
        # y la dividimos en palabras.
        # Obtenemos una lista de palabras
        sentence_words = (X[i].lower()).split()
        # Inicializamos en j=0
        j = 0
        # Recorremos las palabras de sentence_words
        for w in sentence_words:
            # Establecemos la entrada (i, j) de X_indices en el índice
            # de la palabra correcta.
            X_indices[i, j] = word_to_index[w]
            # Incremento de j a j + 1
            j = j + 1
    return X_indices
```

Creamos una capa **Keras Embedding()** y se cargan los vectores de GloVe 50-dimensionales pre-entrenados.

Argmentos:

- word_to_vec_map: diccionario que recorre palabras a su representación vectorial de GloVe.
- word_to_index: diccionario de palabras a sus índices en el vocabulario (400001 palabras).

Devuelve:

- embedding_layer: instancia de la capa Keras pre-entrenada.

```
def pretrained_embedding_layer(word_to_vec_map, word_to_index):

    vocab_len = len(word_to_index) + 1
    # agregando 1 para encajar incrustaciones Keras (requisito)

    emb_dim = word_to_vec_map["the"].shape[0]
    # Definimos la dimensionalidad de sus vectores de
```

```
    # palabras GloVe (= 50)
    # Se podría coger cualquier palabra, es para que nos de la
    # dimension del vector de indices que es 50.

    # Inicializamos la matriz de "embedding"" como una matriz
    # numpy de ceros de forma (vocab_len, dimensiones de los
    # vectores de palabras = emb_dim)
    emb_matrix = np.zeros((vocab_len, emb_dim))

    # Creamos una matriz donde cada fila tiene los 50 numeros
    # de cada palabra
    # .items muestra como biblioteca los pares de
    # palabra e indice
    for word, index in word_to_index.items():
        emb_matrix[index, :] = word_to_vec_map[word]

    # Definimos la capa de incrustación de Keras (Keras embedding)
    # con los tamaños de salida / entrada correctos.
    # Con trainable=False
    embedding_layer = Embedding(vocab_len, emb_dim)

    # Creamos la capa de incrustación, es necesario antes de configurar
    # los pesos.
    embedding_layer.build((None,))

    # Establecemos los pesos de la capa de incrustación a traves de la
    # matriz de numeros construida antes.
    # La capa ahora está pre-entrenada.
    embedding_layer.set_weights([emb_matrix])

    return embedding_layer
```

Creamos nuestro modelo:

Argumentos:

- input_shape: forma de la entrada, generalmente (max len,)
- word_to_vec_map: diccionario que recorre cada palabra en un vocabulario en su representación vectorial de 50 dimensiones.
- word_to_index: recorre el diccionario de palabras a sus índices en el vocabulario (400001 palabras)

Devuelve:

- modelo: una instancia de modelo en Keras.

```
def SentimentAnalysis(input_shape, word_to_vec_map, word_to_index):

    # Definimos sentence_indices como la entrada del grafo,
    # debe tener la forma input_shape y dtype 'int32'
```

```
    # (ya que contiene índices).
    sentence_indices = Input(shape=input_shape, dtype=np.int32)

    # Creamos la capa de incrustación pre-entrenada con vectores GloVe
    embedding_layer = pretrained_embedding_layer(word_to_vec_map,
    word_to_index)

    # Propagamos la instrucción de las frases a través de la capa de
    # incrustación, recupera las incrustaciones.???
    embeddings = embedding_layer(sentence_indices)

    # Propagamos las incrustaciones a través de una capa LSTM con un
    # estado oculto de 128 dimensiones
    # La salida devuelta debe ser un conjunto de secuencias
    X = LSTM(128, return_sequences=True)(embeddings)
    # Añadimos abandono con una probabilidad de 0.5
    X = Dropout(0.5)(X)
    # Propagamos X a través de otra capa LSTM con un estado oculto de
    # 128 dimensiones.La salida devuelta debe ser un
    # solo estado oculto,no un conjunto de secuencias.
    X = LSTM(128)(X)
    # Añadimos abandono con una probabilidad de 0.5
    X = Dropout(0.5)(X)
    # Propagamos X a través de una capa densa con la activación de
    # softmax para recuperar un conjunto de vectores de 5 dimensiones.
    X = Dense(5, activation='softmax')(X)
    # Añadir una activación softmax
    X = Activation('softmax')(X)

    # Creamos una instancia de modelo que convierte
    # sentences_indices en X.
    model = Model(sentence_indices, X)

    return model
```

```
# Leemos datos de entrenamiento y test:
X_train, Y_train = read_csv('train_emoji.csv')
X_test, Y_test = read_csv('test_emoji.csv')
maxLen = len(max(X_train, key=len).split())
# coge el maximo de palabras de una frase
```

Visualizamos parte de nuestro conjunto train:

```
[1] "never talk to me again"
[2] "I am proud of your achievements"
[3] "It is the worst day in my life"
[4] "Miss you so much"
[5] "food is life"
[6] "I love you mum"
```

```
 [7] "Stop saying bullshit"
 [8] "congratulations on your acceptance"
 [9] "The assignment is too long "
[10] "I want to go play"
```

```
 [1] 3 2 3 0 4 0 3 2 3 1
```

```
# Convertimos un tipo de codificación one-hot-encoding en
# =5, [1,0,0,0,0] ya que tenemos {0,1,2,3,4}
Y_oh_train = convert_to_one_hot(Y_train, C=5)
Y_oh_test = convert_to_one_hot(Y_test, C=5)

# Leemos el archivo de glove
word_to_index, index_to_word,
word_to_vec_map = read_glove_vecs('glove.6B.50d.txt')

# Modelo y resumen del modelo:
model = SentimentAnalysis((maxLen,), word_to_vec_map, word_to_index)
model.summary()
model.compile(loss='categorical_crossentropy',
optimizer='adam', metrics=['accuracy'])

X_train_indices = sentences_to_indices(X_train, word_to_index, maxLen)
Y_train_oh = convert_to_one_hot(Y_train, C=5)

# Entrenamos el modelo:
model.fit(X_train_indices, Y_train_oh, epochs=100,
batch_size=32, shuffle=True)

X_test_indices = sentences_to_indices(X_test, word_to_index,
max_len=maxLen)
Y_test_oh = convert_to_one_hot(Y_test, C=5)
```

Layer (type)	Output Shape	Param #
input_3 (InputLayer)	(None, 10)	0
embedding_2 (Embedding)	(None, 10, 50)	20000050
lstm_3 (LSTM)	(None, 10, 128)	91648
dropout_3 (Dropout)	(None, 10, 128)	0
lstm_4 (LSTM)	(None, 128)	131584
dropout_4 (Dropout)	(None, 128)	0
dense_2 (Dense)	(None, 5)	645
activation_2 (Activation)	(None, 5)	0

Total params: 20,223,927
Trainable params: 20,223,927
Non-trainable params: 0

```
# Evaluamos el modelo, loss y accuracy
loss, acc = model.evaluate(X_test_indices, Y_test_oh)
print()
print("Test accuracy = ", acc)

# Comparamos la predicción y el emoji esperado
C = 5
y_test_oh = np.eye(C)[Y_test.reshape(-1)]
X_test_indices = sentences_to_indices(X_test, word_to_index,
maxLen)
pred = model.predict(X_test_indices)
for i in range(len(X_test)):
    x = X_test_indices
    num = np.argmax(pred[i])
    if (num == Y_test[i]):
        print('Expected emoji:' + label_to_emoji(Y_test[i]) +
        ' prediction: ' + X_test[i] + label_to_emoji(num).strip())

# Pruebamos nuestra frase:
x_test = np.array(['very happy'])
X_test_indices = sentences_to_indices(x_test, word_to_index, maxLen)
print(x_test[0] + ' ' +
label_to_emoji(np.argmax(model.predict(X_test_indices))))
```

```
Epoch 1/100
132/132 [==============================] - 22s 166ms/step - loss: 1.6061 - acc: 0.2197
Epoch 2/100
132/132 [==============================] - 5s 41ms/step - loss: 1.5829 - acc: 0.3485
Epoch 3/100
132/132 [==============================] - 4s 34ms/step - loss: 1.5613 - acc: 0.2727
Epoch 4/100
132/132 [==============================] - 5s 34ms/step - loss: 1.5554 - acc: 0.2727
Epoch 5/100
132/132 [==============================] - 4s 34ms/step - loss: 1.5335 - acc: 0.3561
Epoch 6/100
132/132 [==============================] - 5s 35ms/step - loss: 1.5461 - acc: 0.3333
Epoch 7/100
132/132 [==============================] - 4s 34ms/step - loss: 1.5146 - acc: 0.4242
Epoch 8/100
132/132 [==============================] - 5s 36ms/step - loss: 1.4834 - acc: 0.4545
Epoch 9/100
132/132 [==============================] - 5s 35ms/step - loss: 1.4347 - acc: 0.5227
Epoch 10/100
132/132 [==============================] - 4s 31ms/step - loss: 1.3734 - acc: 0.5833
..............................
Epoch 90/100
132/132 [==============================] - 2s 19ms/step - loss: 0.9050 - acc: 1.0000
Epoch 91/100
132/132 [==============================] - 2s 18ms/step - loss: 0.9051 - acc: 1.0000
Epoch 92/100
132/132 [==============================] - 2s 18ms/step - loss: 0.9052 - acc: 1.0000
Epoch 93/100
132/132 [==============================] - 2s 18ms/step - loss: 0.9050 - acc: 1.0000
Epoch 94/100
132/132 [==============================] - 2s 19ms/step - loss: 0.9050 - acc: 1.0000
Epoch 95/100
132/132 [==============================] - 3s 20ms/step - loss: 0.9051 - acc: 1.0000
Epoch 96/100
132/132 [==============================] - 3s 20ms/step - loss: 0.9051 - acc: 1.0000
Epoch 97/100
132/132 [==============================] - 2s 18ms/step - loss: 0.9050 - acc: 1.0000
Epoch 98/100
132/132 [==============================] - 3s 19ms/step - loss: 0.9051 - acc: 1.0000
Epoch 99/100
132/132 [==============================] - 2s 19ms/step - loss: 0.9051 - acc: 1.0000
Epoch 100/100
132/132 [==============================] - 2s 18ms/step - loss: 0.9050 - acc: 1.0000
56/56 [==============================] - 1s 20ms/step
```

```
Test accuracy =  0.8571428571428571
Expected emoji:🍴 prediction: I want to eat     🍴
Expected emoji:😞 prediction: he did not answer 😞
Expected emoji:😄 prediction: he got a very nice raise  😄
Expected emoji:😄 prediction: she got me a nice present 😄
Expected emoji:😄 prediction: ha ha ha it was so funny  😄
Expected emoji:😄 prediction: he is a good friend       😄
Expected emoji:😞 prediction: I am upset        😞
Expected emoji:😄 prediction: We had such a lovely dinner tonight       😄
Expected emoji:🍴 prediction: where is the food 🍴
Expected emoji:😄 prediction: Stop making this joke ha ha ha    😄
Expected emoji:⚾ prediction: where is the ball ⚾
Expected emoji:😞 prediction: are you serious😞
Expected emoji:🍴 prediction: I am hungry🍴
Expected emoji:🍴 prediction: See you at the restaurant 🍴
Expected emoji:😄 prediction: I like to laugh   😄
Expected emoji:⚾ prediction: I will  run⚾
Expected emoji:❤️ prediction: I like your jacket        ❤️
Expected emoji:❤️ prediction: i miss her        ❤️
Expected emoji:⚾ prediction: what is your favorite baseball game       ⚾
Expected emoji:😄 prediction: Good job  😄
Expected emoji:❤️ prediction: I love you to the stars and back ❤️
Expected emoji:😄 prediction: What you did was awesome 😄
Expected emoji:😄 prediction: ha ha ha lol      😄
Expected emoji:😞 prediction: I do not want to joke     😞
Expected emoji:😞 prediction: yesterday we lost again   😞
Expected emoji:❤️ prediction: family is all I have      ❤️
Expected emoji:😞 prediction: you are failing this exercise     😞
Expected emoji:😄 prediction: Good joke 😄
Expected emoji:😄 prediction: You deserve this nice prize       😄
very happy 😄
```

Como podemos observar en la frase introducida *Very Happy* predice correctamente el emoticono, además tenemos un accuracy de 0.8571 por lo que es un buen modelo.

Bibliografía

[1] Allaire, J., Cheng, J., Xie, Y., McPherson, J., Chang, W., Allen, J., Wickham, H., Atkins, A., Hyndman, R. and Arslan, R. 2017. *Rmarkdown: Dynamic documents for r.*

[2] Andrew Ng "Sequence models". Disponible en https://es.coursera.org/learn/nlp-sequence-modelsi.

[3] Cybenko, G. 1989. Approximation by superpositions of a sigmoidal function. *Mathematics of control, signals and systems.* 2, (1989), 303–314.

[4] Gravitar, I. sin límites 2013. *¿Qué es phyton?* Disponible en http://gravitar.biz/pentaho/, version 1.6.0.

[5] Ian Goodfellow, Yoshua Bengio, Aaron Courville 2016. *Deep Learning [draft of March 30, 2015].* MIT Press.

[6] Jeffrey Pennington, C.D.M., Richard Socher 2014. GloVe: Global vectors forWord representationg. *Computer Science Department, Stanford University.* (2014), 1–12.

[7] Li Deng & Yang Liu 2016. *Deep Learning in Natural Language Processing.* Springer.

[8] Luque-Calvo, P.L. 2017. *Escribir un trabajo fin de estudios con r markdown.* Disponible en http://destio.us.es/calvo.

[9] Minsky, M., & Papert, S. 1969. *Perceptron: an introduction to computational geometry.* The MIT Press, Cambridge, MA.

[10] Palash Goyal,Sumit Pandey,Karan Jain 2018. *Deep Learning for Natural Language Processing.* Apress.

[11] René Vidal, R.G.&.S.S., Joan Bruna 2017. Mathematics of deep learning. *Journal of Examples.* 1, 1712.04741 (2017), 1–10.

[12] RStudio Team 2015. *RStudio: Integrated development environment for r.* RStudio, Inc.

[13] Rumelhart, D. E., Hinton, G. E., & Williams, R. J 1986. *Parallel Distributed Processing: Explorations in the Microstructure of Cognition.* The MIT Press, Cambridge, MA.

[14] Trevor Hastie,Robert Tibshirani,Jerome Friedman 2018. *The Elements of Statistical Learning.* Springer.

[15] Xiaolin Hu and P. Balasubramaniam 2008. *Recurrent Neural Networks.* I-Tech.

[16] Xie, Y. 2017. *Knitr: A general-purpose package for dynamic report generation in r.*