ESCUELA TÉCNICA SUPERIOR DE INGENIERÍA INFORMÁTICA

Grado en Matemáticas

Curso Académico 2018-2019

Trabajo Fin de Grado

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

Autor

Guillem García Subies

Tutores

Javier Martínez Moguerza

Álvaro Barbero Jiménez

RESUMEN

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

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

Dada la gran complejidad de este tema y sus dos grandes ramas; la teórica y la técnica, se ha optado por hacer dos Trabajos de Fin de Grado con la misma temática. Uno enfocado en la teoría, para el Grado en Matemáticas y otro enfocado en la realización de un proyecto de procesado de secuencias de texto con herramientas software.

En este trabajo se ha puesto el foco en estudiar las bases del aprendizaje automático, las redes neuronales y los métodos de procesado del lenguaje natural con el objetivo de tener el suficiente conocimiento teórico para poder aplicar estos métodos a datos reales. En el apartado de experimentos se muestran los resultados de las pruebas llevadas a cabo usando dichos métodos.

ÍNDICE GENERAL

ÍNDICE DE FIGURAS

ÍNDICE DE TABLAS

1. INTRODUCCIÓN

1.1. Motivación del trabajo

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

Dado el claro interés por parte de la industria en estos métodos y su proyección futura, es importante poder entenderlos para fomentar la transferencia de conocimiento y evitar las posibles implicaciones negativas que tendría no tener ese conocimiento en el dominio público (creación de las llamadas "fake news" de forma masiva y automática, creación masiva de críticas hacia personajes públicos en redes sociales con el fin de desprestigiarlos, robos de identidad, etc.).

1.2. Objetivos generales

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

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

Por lo tanto, el principal objetivo será explorar las técnicas del estado del arte en

generación automática de texto en base a secuencias y modelado de lenguajes.

2. MARCO TEÓRICO

2.1. Procesado del lenguaje natural

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

Desde los años 50, casi en paralelo a la aparición de las computadoras modernas, ya se especulaba con la posibilidad de tener máquinas parlantes que fueran indistinguibles de un humano [9]. Obviamente, poco o nada distaba esto de la filosofía por aquella época. Sin embargo, con el paso de los años y la mejora de capacidad de cálculo de los ordenadores, estamos presenciando una grandísima evolución de esta disciplina. Tal es dicha evolución que organizaciones como OpenAI aseguran que no hacen pública la totalidad de sus modelos porque están preocupados por los posibles usos maliciosos que se les dé [3] [10].

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

2.2. Aprendizaje automático

Goodfellow [11] explica que "un algoritmo de aprendizaje automático es aquél que puede aprender de los datos". Sin embargo, esta definición se queda corta pues hay que explicar qué significa aprender. Tanto Goodfellow como Mitchell [12] concuerdan en que para que un algoritmo aprenda se necesita una experiencia E, una tarea a desempeñar T, una métrica para cuantificar el rendimiento P que mejora con la experiencia E.

Los algoritmos de aprendizaje automático se pueden clasificar de dos formas; según la experiencia E o la forma de entrenar y según la tarea T que pretenden resolver.

2.2.1. La experiencia

La experiencia E es la forma en la que un algoritmo adquiere su conocimiento sobre el problema que ha de resolver. Se suele hablar de la experiencia como el entrenamiento de un algoritmo. A grandes rasgos se puede clasificar en dos tipos; aprendizaje supervisado y aprendizaje no supervisado. También existen otros tipos de aprendizaje automático menos comunes como el aprendizaje por refuerzo.

- Aprendizaje supervisado:

  El aprendizaje supervisado es aquél en el que cada muestra de los datos tiene una etiqueta u objetivo. De este modo, a medida que el algoritmo vaya aprendiendo a relacionar muestras y objetivos, y una vez acabe de entrenar, podrá diferenciar nuevas muestras que se le presenten y no tengan etiquetas.

- Aprendizaje no supervisado:

  Por otro lado, el aprendizaje no supervisado aprende sobre las características y distribuciones probabilísticas concretas de los datos. Sin embargo, la línea que separa el aprendizaje supervisado del no supervisado puede llegar a ser bastante confusa como veremos cuando planteemos nuestro problema en concreto.

2.2.2. La tarea

La tarea T o labor que desempeña un algoritmo de aprendizaje automático es qué tipo de problema revuelve. Nos vamos a centrar solo en los dos grandes grupos de problemas aunque hay innumerables subgrupos:

- Regresión:

  En este tipo de problemas, la labor principal del algoritmo es predecir un valor $v \in \mathbb{R}$ dado un *input*. Esto se generaliza haciendo que la función de *output* sea $f : \mathbb{R}^n \to \mathbb{R}$.

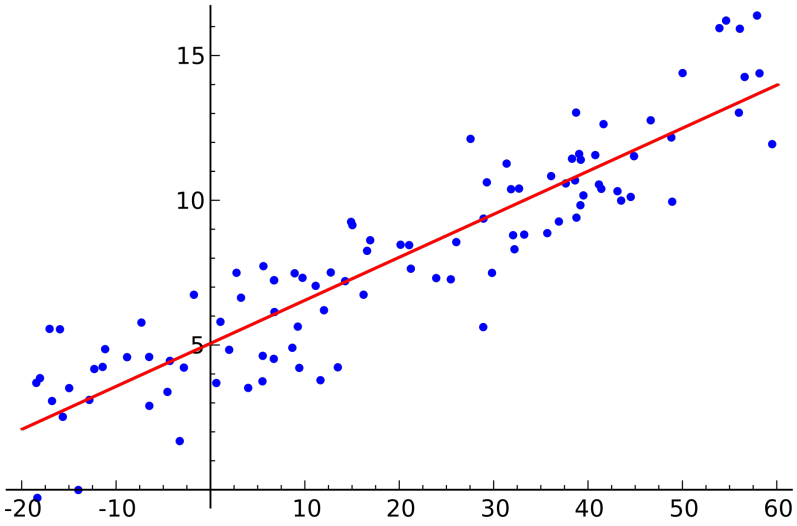


Fig. 2.1. Ejemplo de una regresión lineal [13]

- Clasificación

  La clasificación se diferencia de la regresión en que los valores que se predicen son discretos, esto es, todos los *outputs* de la función serán discretos y estarán acotados por el número n de clases que queramos tener en el problema. Generalmente

las funciones de clasificación son del tipo $f : \mathbb{R}^n \rightarrow \{0, .., n\}$, sin embargo hay algoritmos que predicen la probabilidad $p(x)/x \in \{i\} \forall i \in \{0, .., n\}$. Poder predecir la probabilidad de pertenecer a una clase, en vez de solo poder predecir a qué clase se pertenece es muy potente como se verá más adelante.

Hay muchas más variantes de la clasificación. Por ejemplo, el *clustering* que es básicamente lo mismo que la clasificación pero sin saber de antemano el tamaño de n. De este modo, las n clases se eligen de tal manera que los objetos pertenecientes al mismo grupo (o *cluster*) son mas similares entre ellas que de lo que lo son las de los demás grupos. Este tipo de aprendizaje, en contraposición a los explicados anteriormente, es no supervisado ya que no se asigna ninguna clase a los patrones de los datos.

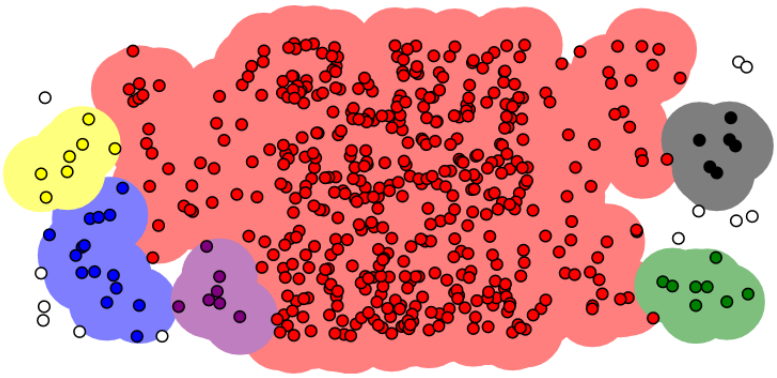


Fig. 2.2. Ejemplo de clustering en dos dimensiones [14]

2.2.3. La medida de rendimiendo

Para diseñar un algoritmo que resuelva problemas y *aprenda* necesitamos algún tipo de métrica cuantitativa que nos diga cuán bien lo está haciendo. Hay que tener en cuenta que la medida de rendimiento P es específica para un problema T.

Veamos, por ejemplo, la precisión para un problema de clasificación binaria (solo hay dos clases, 0 y 1). Se define como

$$P = \frac{VP + VN}{VP + VN + FP + FN} \tag{2.1}$$

Donde: VP (verdadero positivo) son las predicciones clasificadas como positivas correctamente; FP (falso positivo) predicciones clasificadas como positivas, pero que eran negativas; VN (verdadero negativo) son las predicciones clasificadas como negativas correctamente y FN (falso negativo) son las predicciones clasificadas como negativas, pero que realmente son positivas. De este modo, y según 2.1, si FP y FN son 0, implica que la precisión es 1.

De este manera, en problemas críticos como, por ejemplo, la predicción del fraude, esta métrica no sirve. Esto se debe a que en este tipo de problemas, la distribución de

las clases no está equilibrada, por ejemplo hay un 99,9 % de muestras que no son fraude y solo un 0,01 % de fraude. Si tuviéramos un modelo cuya función de decisión fuera: $f(x) = 0 \; \forall x \in \mathbb{R}^n$ obtendríamos una precisión de 0,999, casi perfecto.

Lo que se hace en este tipo de problemas es usar métricas como el ratio de falsos positivos: $RFP = FP/N$ donde N es el total de negativos que hay. Esto se debe a que lo que hace perder dinero por encima de todo es que algo que es fraude pase por desapercibido. De esta forma, probablemente se detecten más falsos negativos, sin embargo eso se compensa con la detección de muchos más casos fraudulentos.

En el aprendizaje automático, la elección de una métrica P adecuada al problema T es crucial y no debe pasarse por alto como veremos en la presentación del problema a resolver.

2.3. Redes neuronales artificiales

Mucha gente piensa que las redes neuronales artificiales realmente imitan el comportamiento del cerebro humano, sin embargo solo tienen una vaga inspiración en éste. Pero antes de estudiarlas, ¿por qué redes neuronales y no otros algoritmos de aprendizaje automático?

A medida que el tamaño de los datos aumenta y hay más variables explicativas para describirlos, más difícil es abordar los los problemas de aprendizaje automático con algoritmos tradicionales. Uno de los mayores problemas es la maldición de la dimensionalidad [15] [11]. Cuando el número de posibles combinaciones de los datos es mucho mayor al número de datos que hay para entrenar, los métodos tradicionales no suelen ser muy efectivos y hay que antes reducir la dimensionalidad o probar con otros métodos [16] [17]. Esto ocurre ya que al haber más dimensiones, aumenta el espacio vacío que hay entre las muestras de los datos, de manera que al modelo le cuesta más inferir cual es la forma ideal de predecir esos vacíos.

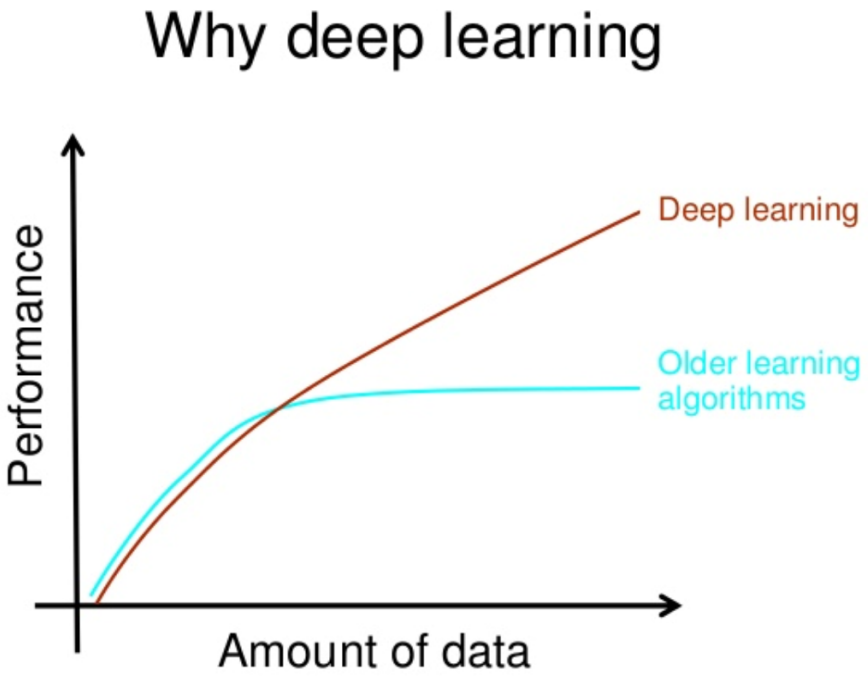


Fig. 2.3. Calidad de los modelos en función de la cantidad de datos [18]

Como podemos observar en 2.3, los métodos de redes neuronales artificiales (*deep learning*, aprendizaje profundo en inglés) son mucho mejor a los demás métodos a medida que aumenta la cantidad de datos a tratar. Esto se debe principalmente a su gran capacidad de adaptación ya que las redes neuronales, en contraposición a los modelos tradicionales, tienen muchísimos más parámetros, arquitecturas que se ajustan a el tipo de datos e, incluso, formas distintas de entrenarse. Además, como veremos más adelante, su ejecución se puede paralelizar y también se pueden usar GPUs para su entrenamiento, optimizando así el tiempo de entrenamiento que, de por sí, suele ser muy grande cuando hay muchos datos.

2.3.1. La neuronas

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

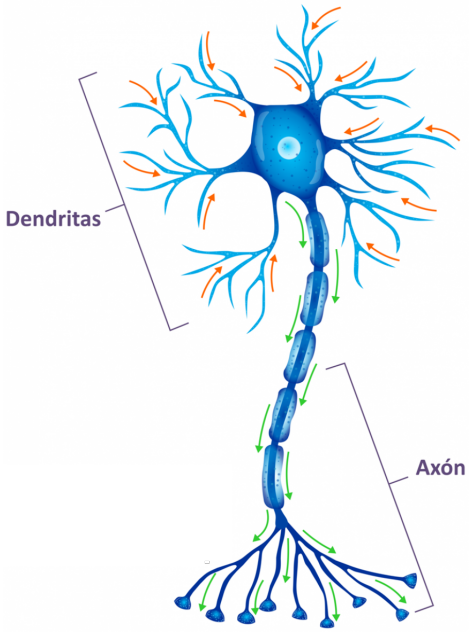


Fig. 2.4. La neurona biológica [19]

2.3.2. La neurona de McCulloch-Pitts

El primer intento de modelizar matemáticamente una neurona fue llevado a cabo por Warren. S McCulloch y Walter H. Pitts en 1943 [20].

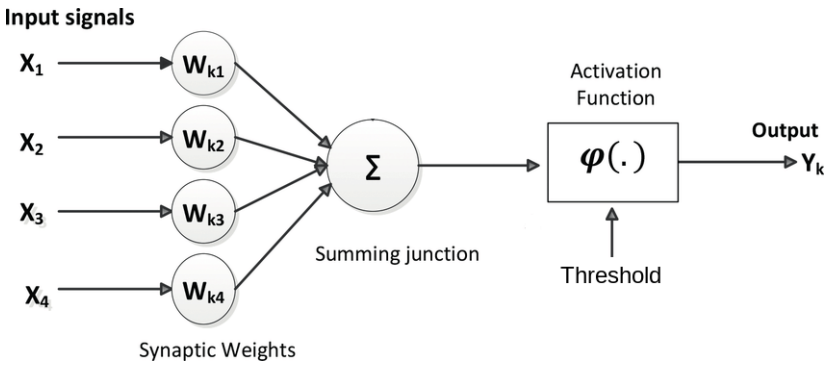


Fig. 2.5. La neurona de McCulloch-Pitts

Los *inputs* **x** representarían las dendritas que recaban información de otras neuronas. Los pesos **w** representan cuán fuertes son las conexiones de las otras neuronas x_i (esto intenta ser el análogo a las sinapsis). Después se hace una suma ponderada, lo cual vendría a asimilarse a cómo en el soma se mezclan todos los impulsos eléctricos entrantes. La función de activación φ viene a simular el umbral T necesario para transmitir la señal a otras neuronas. La salida y puede transmitirse a otras neuronas, como pasaría en el axón.

$$\varphi(v) = \begin{cases} 0, & \text{si } v < T \\ 1, & \text{si } v \geq T \end{cases} \tag{2.2}$$

El *output* es la función de activación 2.2 aplicada sobre una combinación lineal de los *inputs*. De esta forma, el *output* y_j para una neurona j sería:

$$y_j = \varphi\left(\sum_i w_i x_i\right) \tag{2.3}$$

La función de activación φ no es lineal, pasa directamente de 0 a 1 cuando se alcanza el umbral U. Como el *output* es binario, los *inputs*, en principio, deben serlo también (vienen de los *outputs* de otras neuronas).

De una forma equivalente, y combinando 2.2 con 2.3, podemos escribir:

$$y_j = \begin{cases} 0, & \text{si } \sum_i w_i x_i < T \\ 1, & \text{si } \sum_i w_i x_i \geq T \end{cases} \tag{2.4}$$

Una vez tenemos este modelo inicial, hay que ver qué cosas puede o no hacer. Por ejemplo puede hacer funciones lógicas simples como AND u OR. En el caso de AND, nos bastaría con que el umbral T sea 2 y los pesos **w** sean todos 1. De este modo, cuando ambos *inputs* sean 1, su suma ponderada sería $2 \geq T$ y el *output* sería 1. En cualquier otro caso, sería 0. De hecho, esto se podría generalizar trivialmente a una puerta lógica AND con *n inputs*.

En el caso de la función OR, el razonamiento es similar; si el umbral T es 1, se cumplirían las condiciones para que la neurona actúe como una función lógica OR.

La razón por la que estas dos funciones (y muchas otras más) funcionen es que son linealmente separables. En otras palabras, podemos encontrar un vector de pesos **w** y un umbral T que separe linealmente las dos clases de este problema (0 y 1).

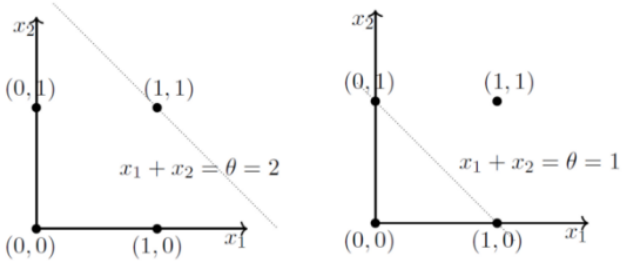


Fig. 2.6. Función AND y función OR respectivamente

Sin embargo, los problemas que no son separables linealmente, no se pueden resolver con este tipo de neuronas. Es el caso de la función lógica XOR:

$$XOR(x,y) = \begin{cases} 0, & \text{si } x = y \\ 1, & \text{si } x \neq y \end{cases} \quad \forall\{x,y\} \in \{0,1\} \tag{2.5}$$

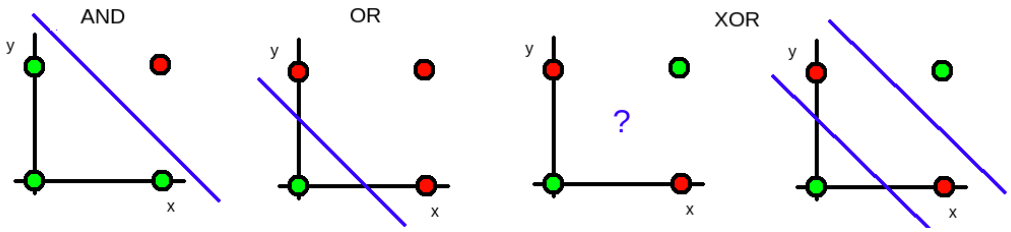


Fig. 2.7. Separación lineal de las funciones AND, OR y XOR respectivamente

Como podemos ver en 2.7, no hay forma de separar linealmente las dos clases que salen de una función lógica XOR usando solo un hiperplano.

Éste (las funciones no separables linealmente) no es el único problema de las neuronas de McCulloch-Pitts. Para que pudieran llegar a tener alguna aplicación habría que tener en cuenta los siguientes aspectos:

- ¿Qué hacer con las funciones que no son linealmente separables?
- Si hay que elegir manualmente siempre el umbral y los pesos, estas neuronas son de poca utilidad.
- Las neuronas de McCulloch-Pitts están restringidas a *inputs* binarios, ¿pero se podrían generalizar para usar *inputs* $\mathbf{x} \in \mathbb{R}^n$?

Una forma de simplificar la expresión matemática de este modelo es cambiar el uso de un umbral por el uso de un sesgo (*bias* en inglés):

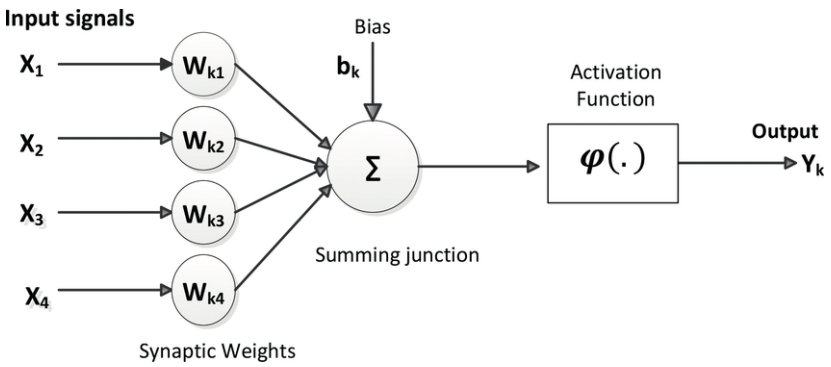


Fig. 2.8. La neurona de McCulloch-Pitts con sesgo [21]

Básicamente el sesgo es un *input* más que se suma con los demás pesos. Como ya no hay umbral, podemos expresar φ como:

$$\varphi(v) = \begin{cases} 0, & \text{si } v < 0 \\ 1, & \text{si } v \geq 0 \end{cases} \tag{2.6}$$

De la misma manera y_j sería:

$$y_j = \begin{cases} 0, & \text{si } \sum_i w_i x_i + b < 0 \\ 1, & \text{si } \sum_i w_i x_i + b \geq 0 \end{cases} \tag{2.7}$$

Como podemos ver, ahora siempre comparamos con 0 ya que hemos *movido* el umbral T a la derecha de la desigualdad.

Partiendo de 2.7, el hiperplano que crea el perceptrón para separar dos clases sería el siguiente:

$$\sum_{i}^{n} w_i x_i + b = 0 \tag{2.8}$$

Este modelo, obviamente estaba muy lejos de parecerse remotamente a la mente humana, sin embargo McCulloch y Pitts abrieron la puerta de lo que en el futuro se convertiría en uno de los campos más prometedores del aprendizaje automático actualmente, las redes neuronales.

2.3.3. El perceptrón

Frank Rosenblatt en 1958, diseñó el perceptrón [22], lo que acabaría siendo la base de todas las redes neuronales modernas. Anteriormente, aparecieron otros tipos de aprendizaje como el propuesto por Hebb en 1949 [23], aunque, sin embargo, no acabó teniendo tanta versatilidad como el perceptrón.

La mayor novedad de este tipo de neurona es que el perceptrón aprende los pesos de una neurona de McCulloch-Pitts para solucionar un problema. Esta fue una de las primeras expresiones del aprendizaje supervisado, en el que para aprender había que proporcionar al algoritmo como *inputs* un conjunto de datos que contenían tanto información sobre las muestras como la predicción "correcta" para cada muestra, de forma que así se podía calcular el error entre lo que predecía el algoritmo y el resultado real.

Para aprender los pesos $\mathbf{w}$, se usa el llamado algoritmo de convergencia del perceptrón.

Teorema de convergencia del perceptrón

Vamos a seguir la prueba de Haykin [24]. Básicamente, este teorema demuestra que en un número finito de iteraciones, el perceptrón puede aprender cuales son los valores ideales del vector $\mathbf{w}$ para resolver un problema en el que las dos clases C_1 y C_2 son linealmente separables sin cometer ningún error de clasificación. El teorema no dice nada sobre los problema con clases no separables linealmente.

Una forma de simplificar la expresión del sesgo es considerarlo como un peso $w_0 = b$ que multiplica a un x_0 que es igual a 1 para no introducir ningún otro cambio que no sea

el sesgo. Así pues, el vector $\mathbf{x}(n)$ (donde n es el paso temporal en el que se está aplicando el algoritmo) lo definimos como

$$\mathbf{x}(n) = [1, x_1(n), ..., x_m(n)]$$

Análogamente, el vector de pesos $\mathbf{w}(n)$ será

$$\mathbf{w}(n) = [b, w_1(n), ..., w_m(n)]$$

Ahora podemos simplificar la función de la combinación lineal del perceptrón quedando en:

$$\begin{aligned} v(n) &= \sum_{i=0}^{m} w_i(n)x_i(n) \\ &= \mathbf{w}(n)\mathbf{x}(n) \end{aligned} \tag{2.9}$$

De esta forma, dado un n, el hiperplano que actúa de superficie de decisión para elegir entre las dos posibles clases del problema es $\mathbf{w} \cdot \mathbf{x} = 0$. En este caso, las posibles salidas del perceptrón serán $+1$ para C_1 y -1 para C_2.

Suponiendo el conjunto de datos $\mathbf{D} = \{(\mathbf{x_1}, y_1), ..., (\mathbf{x_n}, y_n)\}$ donde los *inputs* $\mathbf{x_i}$ son vectores y los objetivos de los que aprender $y_i \in \{-1, +1\}$. $\mathbf{D}$ se podría dividir en dos conjuntos disjuntos, D_1 y D_2 donde al primero pertenecen todos los $\mathbf{x_i}$ que tengan una $y_i = +1$ y al segundo, los que tengan una $y_i = -1$. Dado $\mathbf{D}$, el proceso de entrenamiento del perceptrón está finalizado cuando se tiene un $\mathbf{w}$ que hace que:

$$\begin{aligned} \mathbf{w} \cdot \mathbf{x} > 0 \ \ \forall \mathbf{x} \in C_1 \\ \mathbf{w} \cdot \mathbf{x} \leq 0 \ \ \forall \mathbf{x} \in C_2 \end{aligned} \tag{2.10}$$

En otras palabras, que clasifica perfectamente cualquier punto del espacio. Nótese que la desigualdad estricta se ha elegido de forma arbitraria, podría haber estado en cualquiera de las dos ecuaciones de 2.10.

El algoritmo para adaptar los pesos es el que sigue:

1. Si el elemento *n-ésimo* de $\mathbf{D}$, $\mathbf{x}(n)$, se clasifica correctamente con el vector de pesos $\mathbf{w}(n)$, calculado en la *n-ésima* iteración del algoritmo, no se hace ninguna corrección a el vector de pesos del perceptrón.

2. De lo contrario, el vector de pesos se actualiza siguiendo la siguiente fórmula:

$$\begin{aligned} \mathbf{w}(n+1) &= \mathbf{w}(n) - \eta\mathbf{x}(n) \ \ si \ \ \mathbf{x}(n) \in C_2 \\ \mathbf{w}(n+1) &= \mathbf{w}(n) + \eta\mathbf{x}(n) \ \ si \ \ \mathbf{x}(n) \in C_1 \end{aligned} \tag{2.11}$$

Donde el parámetro η es el ritmo del aprendizaje (o *learning-rate* en inglés) y controla cuán grande es el ajuste que se aplica a los pesos. Aquí lo ponemos como un único parámetro constante aunque también se podría usar un η para cada n.

La prueba de que el algoritmo converge en un número finito de iteraciones es independiente del valor de η siempre y cuando $\eta > 0$ ya que solo determina como cambian los pesos, pero no afecta a la separabilidad lineal de las clases. De este modo haremos la demostración con $\eta = 1$ por simplicidad.

Para demostrar la convergencia del algoritmo, tomamos como condición inicial que $\mathbf{w}(0) = \mathbf{0}$ (en la practica se puede inicializar a cualquier valor. Normalmente se suele inicializar a $\mathbf{0}$ o de forma aleatoria). Supongamos que $\mathbf{w}(n)\mathbf{x}(n) < 0$ para $n = 1, 2, ...$ y el vector de entrada $\mathbf{x}(n)$ pertenece a D_1, lo que significa que está clasificado de forma incorrecta (2.10) los vectores. Así pues, y con $\eta = 1$, si desarrollamos la segunda ecuación de 2.11 tenemos:

$$\mathbf{w}(n+1) = \mathbf{w}(n) + \mathbf{x}(n) \;\; para \;\; \mathbf{x}(n) \in C_1 \tag{2.12}$$

Dado que $\mathbf{w}(0) = \mathbf{0}$, desarrollamos la ecuación a:

$$\mathbf{w}(n+1) = \mathbf{x}(1) + \mathbf{x}(2) + ... + \mathbf{x}(n) \tag{2.13}$$

Como las clases C_1 y C_2 son linealmente separables (por hipótesis), existe una solución $\mathbf{w}_0$ para la que $\mathbf{w} \cdot \mathbf{x}(n) > 0$ para los vectores $\mathbf{x}(1), ..., \mathbf{x}(n)$ (pertenecientes a D_1). Para la solución fijada $\mathbf{w}_0$, definimos un número positivo α de la siguiente manera:

$$\alpha = \min_{\mathbf{x}(i) \in D_1} \mathbf{w}_0 \mathbf{x}(i) \tag{2.14}$$

Por lo tanto, multiplicando a ambos lados de 2.13 por $\mathbf{w}_0$ tenemos

$$\mathbf{w}_0 \mathbf{w}(n+1) = \mathbf{w}_0 \mathbf{x}(1) + \mathbf{w}_0 \mathbf{x}(2) + ... + \mathbf{w}_0 \mathbf{x}(n) \tag{2.15}$$

Ahora, según lo definido en 2.14 obtenemos la siguiente desigualdad:

$$\mathbf{w}_0 \mathbf{w}(n+1) \geq n\alpha \tag{2.16}$$

A continuación usamos la desigualdad de Cauchy-Schwarz: Dados dos vectores $\mathbf{w}_0$ y $\mathbf{w}(n+1)$

$$||\mathbf{w}_0||^2 ||\mathbf{w}(n+1)||^2 \geq [\mathbf{w}_0 \mathbf{w}(n+1)]^2 \tag{2.17}$$

donde $\|\cdot\|$ es la norma Euclídea de un vector. Ahora podemos ver que por 2.16, $[\mathbf{w}_0\mathbf{w}(n+1)]^2 \geq n^2\alpha^2$ y usando 2.17, tenemos que:

$$\|\mathbf{w}_0\|^2\|\mathbf{w}(n+1)\|^2 \geq n^2\alpha^2 \tag{2.18}$$

o, equivalentemente

$$\|\mathbf{w}(n+1)\|^2 \geq \frac{n^2\alpha^2}{\|\mathbf{w}_0\|^2} \tag{2.19}$$

Ahora, reescribimos 2.12 de la forma:

$$\mathbf{w}(k+1) = \mathbf{w}(k) + \mathbf{x}(k) \;\; para \;\; k = 1, ..., n \;\; y \;\; \mathbf{x}(k) \in D_1 \tag{2.20}$$

Aplicamos la norma Euclídea al cuadrado a ambos lados de 2.20 y tenemos

$$\|\mathbf{w}(k+1)\|^2 = \|\mathbf{w}(k)\|^2 + \|\mathbf{x}(k)\|^2 + 2\mathbf{w}(k)\mathbf{x}(k) \tag{2.21}$$

Pero $\mathbf{w}(k)\mathbf{x}(k) \leq 0$ (como hemos supuesto más arriba). Así, deducimos que

$$\|\mathbf{w}(k+1)\|^2 \leq \|\mathbf{w}(k)\|^2 + \|\mathbf{x}(k)\|^2 \tag{2.22}$$

o, reescribiendo

$$\|\mathbf{w}(k+1)\|^2 - \|\mathbf{w}(k)\|^2 \leq \|\mathbf{x}(k)\|^2, \;\; con \;\; k = 1, ..., n \tag{2.23}$$

Sumando estas desigualdades para $k = 1, ..., n$ y recordando la condición inicial de que $\mathbf{w}(0) = \mathbf{0}$ obtenemos la siguiente desigualdad

$$\|\mathbf{w}(n+1)\|^2 \leq \sum_{k=1}^{n} \|\mathbf{x}(k)\|^2 \leq n\beta \tag{2.24}$$

donde β se define como

$$\beta = \max_{\mathbf{x}(k) \in D_1} \|\mathbf{x}(k)\|^2 \tag{2.25}$$

ahora, cogiendo un n_{max} suficientemente grande, conseguimos que 2.24 y 2.19 se cumplan justo en sus extremos, en la igualdad, de manera que tenemos

$$\frac{n_{max}^2\alpha^2}{\|\mathbf{w}_0\|^2} = n_{max}\beta \tag{2.26}$$

desarrollando y despejando n_{max} llegamos a

$$n_{max} = \frac{\|\mathbf{w}_0\|^2 \beta}{\alpha^2} \tag{2.27}$$

De esta forma, hemos probado que para $\eta = 1$, para $\mathbf{w}(0) = \mathbf{0}$ y $\forall n$ y dadas dos clases C_1 y C_2 separables linealmente, la regla para adaptar los pesos del perceptrón consigue terminar en, como mucho, n_{max} iteraciones. Dadas las dos cotas 2.19 y 2.24 para el vector de pesos $\mathbf{w}$; la primera, cota inferior que escala al cuadrado con las iteraciones y la segunda, cota superior que crece de forma no cuadrática. Se consigue que en un punto (n_{max}) se corten ambas cotas. A partir de este punto, el módulo del vector de pesos $\mathbf{w}$ ya no se actualiza ya que se ha conseguido separar las dos clases. □

La prueba se ha hecho asumiendo que se los vectores pertenecen a D_1, pero se puede demostrar análogamente para D_2. También se podría probar fácilmente lo anterior para un $\eta(n)$ variable (entre 0 y 1) o para un $\mathbf{w}(0)$ genérico.

2.3.4. El perceptrón multicapa

Gracias al perceptrón se solucionaron algunos de los problemas de la neurona de McCulloch-Pitts, sin embargo si los problemas no eran linealmente separables, no había certeza sobre su convergencia y, como hemos visto, algunos directamente no se podían resolver.

El perceptrón multicapa surge para intentar solventar las limitaciones que había hasta entonces en los perceptrones. Como podemos ver en 2.9 (donde cada punto gris representaría un perceptrón), ahora hay una estructura en forma de red. Dicha estructura consta de una capa de *input* y otra de *output* y una o varias capas intermedias u ocultas.

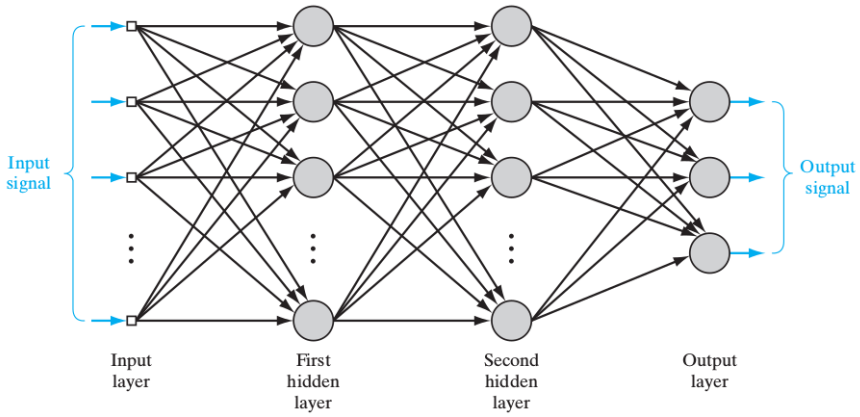


Fig. 2.9. La arquitectura de un perceptrón multicapa [25]

Nótese que la capa de *input* no tiene función de activación (su función de activación es la identidad realmente). La capa de output tiene tantas neuronas como clases haya en el

problema (también conocido como *one hot encoding*). Sin embargo y, según qué funciones de activación se usen, se puede conseguir que los *outputs* de las neuronas sean muy distintos del simple -1 y 1 que había antes, de forma que se pueden resolver problemas de regresión y otros problemas no linealmente separables.

Los problemas anteriormente explicados con respecto a los perceptrones se pueden enmendar gracias a estas estructuras con varias capas y activaciones no lineales para sus neuronas ocultas. A esto también ha ayudado en gran medida el el algoritmo del la propagación hacia atrás o *backpropagation* en inglés que sirve para ajustar automáticamente los parámetros de estas redes multicapa en base a datos etiquetados.

Caracterización del perceptrón multicapa

Antes de explicar con profundidad el algoritmo de *backpropagation*, es importante entender qué es un perceptrón multicapa (o una red neuronal moderna).

Como hemos visto, las redes neuronales no son más que funciones parametrizables que asignan *outputs* a sus *inputs*. Por lo tanto, "entrenar" una red neuronal consiste en minimizar una función de error que compara el *output* de la red contra la solución real. Visto así, el problema de aprendizaje se puede ver como un problema de optimización en el que hay que encontrar los mejores parámetros que minimicen el error.

De este modo, se pueden aplicar técnicas clásicas de optimización numérica como el descenso por gradiente, etc.

Sin embargo, como para ejecutar estas técnicas se requiere tanto poder de cómputo y se tienen que ejecutar tantas veces seguidas, se precisa de una forma rápida de calcular los gradientes. Es aquí donde aparece el algoritmo de *backpropagation*,

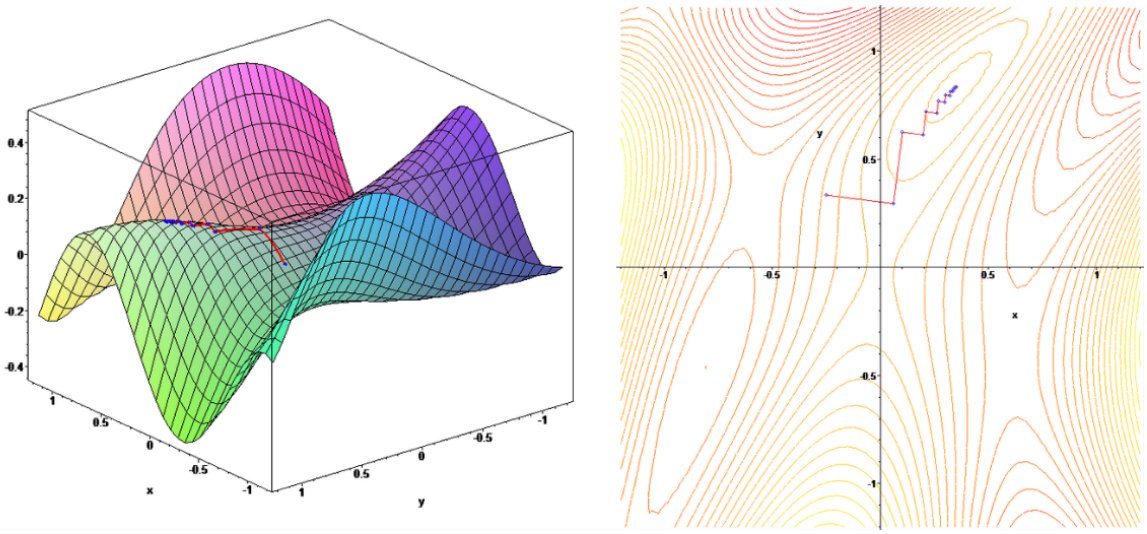


Fig. 2.10. Ilustración del descenso por gradiente [26]

Backpropagation

Empezando por la notación:

- a^l_j: resultado de la función de activación de la *j-ésima* neurona en la *l-ésima* capa.
- b^l_j sesgo de la *j-ésima* neurona en la *l-ésima* capa
- w^l_{jk} peso desde la *k-ésima* neurona en la *(l-1)-ésima* capa a la *j-ésima* neurona de la *l-ésima* capa

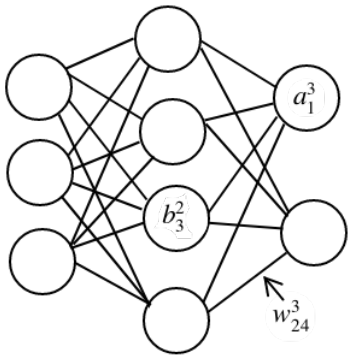


Fig. 2.11. Parámetros de una red

En este caso usaremos también los sesgos de forma explicita por completitud de la explicación.

Veamos como se calculan las activaciones, que dependen de las activaciones y los pesos de la capa anterior:

$$a^l_j = \varphi\left(\sum_k w^l_{jk} a^{l-1}_k + b^l_j\right) = \varphi\left(z^l_j\right) \tag{2.28}$$

Donde k va por todas las neuronas en la capa $l-1$, z^l_j son los valores de salida de las capas antes de aplicar la función de activación y $\varphi\left(z^l_j\right)$ es función de activación para la *j-ésima* neurona en la *l-ésima* (más abajo se explica. Dicha función tiene que ser continua, no lineal y diferenciable (o diferenciable en todos los puntos del dominio salvo uno).

Generalizando y calculando todas las neuronas obtendríamos:

$$\mathbf{a^l} = \varphi\left(\mathbf{w^l a^{l-1}} + \mathbf{b^l}\right) = \varphi\left(\mathbf{z^l}\right)$$

$$\mathbf{w^l} = \begin{pmatrix} w^1_{11} & \cdots & w^l_{1n^{l-1}} \\ \vdots & \ddots & \vdots \\ w^1_{n^l 1} & \cdots & w^l_{n^l n^{l-1}} \end{pmatrix}, \quad \mathbf{a^{l-1}} = \begin{pmatrix} a^{l-1}_1 \\ \vdots \\ a^{l-1}_{n^{l-1}} \end{pmatrix}, \quad \mathbf{b^l} = \begin{pmatrix} b^l_1 \\ \vdots \\ b^l_{n^l} \end{pmatrix}, \quad \varphi\begin{pmatrix} z^l_1 \\ \vdots \\ z^l_{n^l} \end{pmatrix} = \begin{pmatrix} \varphi\left(z^l_1\right) \\ \vdots \\ \varphi\left(z^l_{n^l}\right) \end{pmatrix} \tag{2.29}$$

Ahora solo habría que repetir el proceso capa a capa hasta llegar al final de la red y obtener la salida. Una vez obtenida, hay que comparar el *output* de la red con el *output* real. De esta forma se puede calcula el error. Definamos la función del del Error Cuadrático Medio:

$$E(\mathbf{w}, \mathbf{b}) = \frac{1}{n} \sum_{i=1}^{n} \|y_i - a_i\|^2 \tag{2.30}$$

Donde y_i y a_i son el valor real de los datos y el valor que predice la red para los datos en la muestra i respectivamente. Para cada muestra i, obtendríamos:

$$E_i = \frac{1}{2} \sum_{j} (y_j - a_j^L)^2 \tag{2.31}$$

Donde j es el índice de las neuronas de *output* y L denota que estamos en la capa de *output*.

Para poder recalcular los pesos otra vez, primero necesitamos saber qué error ha cometido cada neurona de las capas internas individualmente. Sin embargo esto no es posible. Para ello, la idea del algoritmo de *backpropagation* es propagar el error de toda la red hacia atrás para asignar un error individual a cada neurona.

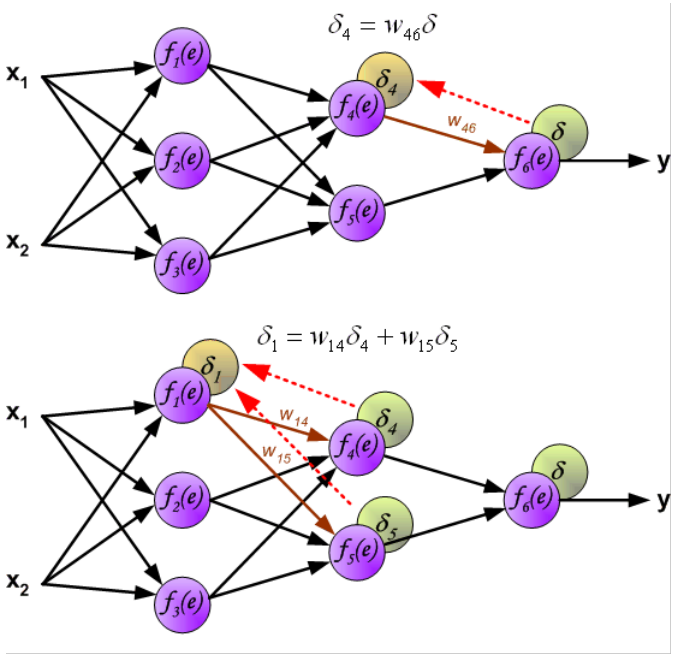


Fig. 2.12. Esquema de la *backpropagation* [27]

Donde δ es la derivada del error, conocida como deltas de error:

$$\delta_j^l = \frac{\partial E}{\partial z_j^l} \tag{2.32}$$

De esta forma si derivamos 2.31 obtenemos:

$$\frac{\partial E_i}{\partial a_j^L} = y_i - a_l^L \tag{2.33}$$

Así, las deltas de error para la última capa (aplicando la regla de la cadena) son:

$$\delta_j^L = \frac{\partial E_i}{\partial z_j^L} = \frac{\partial E_i}{\partial a_j^L}\frac{\partial a_j^L}{\partial z_j^L} \tag{2.34}$$

Ahora, usando 2.28 y 2.33 sustituimos en la ecuación anterior para obtener:

$$\delta_j^L = (y_j - a_j^L)\,\varphi'(z_j^L) \tag{2.35}$$

Hay que notar que, aquí necesitaremos tener una función de activación que sea derivable ya que, de otro modo, estos cálculos no podrían llevarse a cabo.

Por otra parte, hay que tener en cuenta que por definición (2.28) z_j^l influencia a todos los z de la capa $l + 1$:

$$z_j^l = \sum_k w_{jk}^l a_k^{l-1} + b_j^l = \sum_k w_{jk}^l \varphi\left(z_k^{l-1}\right) + b_j^l \tag{2.36}$$

Ahora, usando la ecuación de arriba, podemos propagar hacia atrás los deltas:

$$\delta_j^l = \frac{\partial E_i}{\partial z_j^l} = \sum_k \frac{\partial E_i}{\partial z_k^{l+1}}\frac{\partial z_k^{l+1}}{\partial z_j^l} = \sum_k \delta_k^{l+1}\frac{\partial z_k^{l+1}}{\partial z_j^l} = \sum_k \delta_k^{l+1} w_{jk}^{l+1}\ \varphi'(z_j^l) \tag{2.37}$$

Una vez hecho esto para toda la red, podemos calcular todas las derivadas del error:

$$\frac{\partial E_i}{\partial w_{jk}^l} = \frac{\partial E_i}{\partial z_j^l}\frac{\partial z_j^l}{\partial w_{jk}^l} = \delta_j^l\, a_k^{l-1}, \qquad \frac{\partial E_i}{\partial b_j^l} = \frac{\partial E_i}{\partial z_j^l}\frac{\partial z_j^l}{\partial b_j^l} = \delta_j^l \tag{2.38}$$

Si repetimos la fase hacia adelante (cuando se calcula la salida con los nuevos pesos) y la fase hacia atrás (calculando los los errores) para todas las muestras del conjunto de datos, obtenemos el gradiente del error:

$$\frac{\partial E}{\partial w_{jk}^l} = \frac{1}{n}\sum_{i=1}^{n}\frac{\partial E_i}{\partial w_{jk}^l}, \qquad \frac{\partial E}{\partial b_j^l} = \frac{1}{n}\sum_{i=1}^{n}\frac{\partial E_i}{\partial b_j^l} \tag{2.39}$$

De esta forma y, resumiendo, el algoritmo de *backpropagation* sería el siguiente:

Algorithm 1: Backpropagation

for $i \leftarrow 1$ **to** n **do**
  Coger muestra $\mathbf{x_i}$ y calcular las activaciones $\mathbf{a_j^l}$ hasta la capa de salida ;
  Calcular δ_j^L para la capa de salida ;
  Propagar hacia atrás para obtener el resto de δ_j^l ;
  Calcular el gradiente $\frac{\partial E_i}{\partial w_{jk}^l}$ y $\frac{\partial E_i}{\partial b_j^l}$;
  Guardar estos resultados para la salida del bucle ;
end
Hacer la media del gradiente para obtener $\frac{\partial E}{\partial w_{jk}^l}$ y $\frac{\partial E}{\partial b_j^l}$;

Ahora solo nos faltaría ver como se entrena un perceptrón multicapa (como actualiza sus pesos) utilizando este algoritmo:

Algorithm 2: Entrenamiento de un perceptrón multicapa

Inicializamos los pesos w_{jk}^l y los sesgos b_j^l aleatoriamente (o a 0);
repeat
  Calcula el gradiente del error $\nabla E = (\frac{\partial E}{\partial w_{jk}^l}, \frac{\partial E}{\partial b_j^l})$ usando *backpropagation* ;
  Actualiza los pesos $w_{jk}^l \leftarrow w_{jk}^l - \eta \frac{\partial E}{\partial w_{jk}^l}$;
  Actualiza los sesgos $b_j^l \leftarrow b_j^l - \eta \frac{\partial E}{\partial b_j^l}$;
until $\nabla E \approx 0$;

Donde η tiene la misma función que tenía en el algoritmo del perceptrón.

Asimismo, el criterio de parada $\nabla E \approx 0$ no suele aplicarse ya que eso implicaría tiempos de ejecución altísimos. En la práctica se para después de un cierto número de iteraciones o de cuando el error es suficientemente bajo para la tarea que se está intentando llevar a cabo.

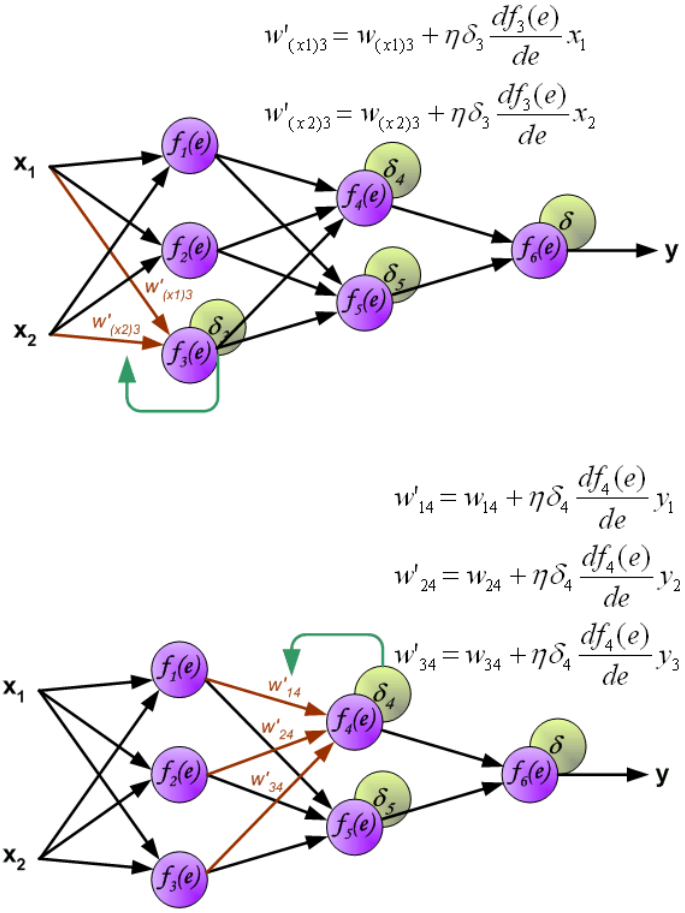


Fig. 2.13. Esquema de la actualización de pesos del perceptrón multicapa [27]

Ahora hemos conseguido superar todos los problemas que se planteaban anteriormente con el perceptrón. De hecho, el teorema de aproximación universal, demuestra que un perceptrón multicapa puede aproximar cualquier función de $\mathbb{R}^n$.

Teorema de aproximación universal

Formalmente [28] [29]:

Sea $\varphi : \mathbb{R} \to \mathbb{R}$ una función no constante, acotada y continua y dado I_m un hipercubo m-dimensional $[0, 1]^m$. Denotamos el espacio de funciones continuas reales en I_m como $C(I_m)$. Ahora, para toda función $f \in C(I_m)$ y para todo $\epsilon > 0$, existe un entero N, constantes reales $v_i, b_i \in \mathbb{R}$ y vectores reales $\mathbf{w}_i \in \mathbb{R}^m$ para $i = 1, ..., N$ de forma que podemos definir:

$$F(\mathbf{x}) = \sum_{i=1}^{N} v_i \varphi \left(\mathbf{w}_i \mathbf{x} + b_i \right) \tag{2.40}$$

como una aproximación de la función f de forma que

$$|F(\mathbf{x}) - f(\mathbf{x})| < \epsilon \tag{2.41}$$

para todo $\mathbf{x} \in I_m$. Lo que significa que las funciones de la forma de $F(\mathbf{x})$ son densas en $C(I_m)$

Este teorema lo podemos aplicar directamente a un perceptrón multicapa porque las funciones de activación que se suelen usar en las redes neuronales son no constantes, acotadas y continuas, por lo que satisfacen las condiciones necesarias de φ. Asimismo, 2.40 se puede ver como el *output* de un perceptrón multicapa done

1. La red tiene $x_1, ..., x_m$ como *inputs* y una sola capa oculta con N neuronas.
2. Cada neurona oculta i tiene pesos $\mathbf{w}_i$.
3. El *output* de la red es una combinación lineal de las salidas de las neuronas ocultas, siendo los v_i los pesos de la capa de salida.

De esta forma, con una sola capa oculta podemos representar cualquier función. Es más, con la función de activación ReLU

$$f(x) = \text{máx}(0, x) \tag{2.42}$$

se puede probar que se necesitan como mucho $n + 4$ neuronas en la capa oculta donde n es la dimensión del *input* [30]. Sin embargo, el teorema no dice nada de la dificultad en el entrenamiento o si una red neuronal puede llegar a aprender cualquier función, solo dice que se pueden representar.

El problema de los gradientes desvanecientes

Este problema [31] proviene de derivar repetidamente las funciones de activación en la fase de *backpropagation* [32]. Cuando una red tiene muchas capas, las sucesivas derivadas del gradiente de la función de pérdida hacen que este acabe siendo muy cercano a 0, de forma que la red deja de aprender.

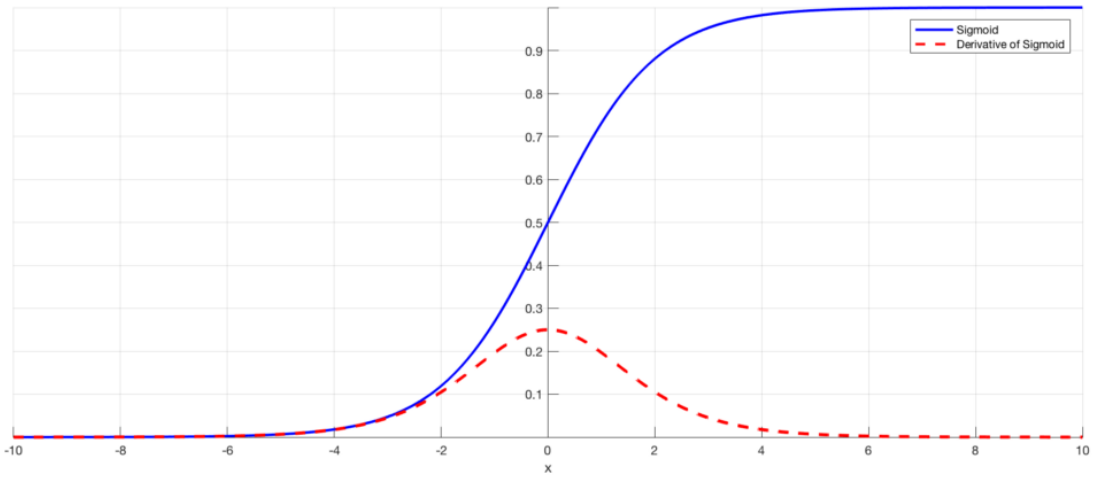


Fig. 2.14. La función sigmoidal y su derivada [33]

Como podemos ver en 2.14, cuando los *inputs* de la función son muy grandes en valor absoluto, su derivada se acerca mucho al 0. De esta forma, cuando hay muchas capas en una red usando activaciones parecidas a la sigmoidal, todas sus derivadas (que acaban siendo muy pequeñas) se multiplican entre sí al hacer la *backpropagation*. De esta forma, el gradiente desciende de forma exponencial a medida que se propaga hacia las primeras capas de la red.

Esto, sin embargo, sería menos problemático para algunas funciones de activación como la ReLU (que supuso un gran avance en el campo de las redes neuronales), aun así tampoco consiguen eliminar este problema al completo.

2.3.5. Redes neuronales recurrentes

Las redes neuronales recurrentes [34] [35] se centran en intentar procesar datos con forma de secuencias como, por ejemplo, listas de palabras o series temporales.

La novedad que introducen este tipo de redes es el uso compartido de parámetros a través de diferentes partes del modelo. Este parámetro actúa como si fuera una "memoria" de la red en puntos clave que, de otra forma, probablemente no conseguiría retener.

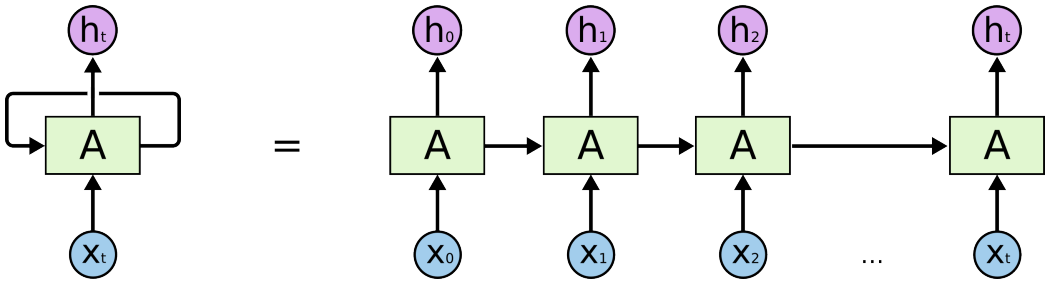


Fig. 2.15. Red neuronal recurrente y su desenrollado [36]

En la primera parte de la imagen de arriba podemos ver como las redes recurrentes

comparten la información, aunque parece que la compartan consigo mismas. Sin embargo, esos datos persisten durante las reiteradas iteraciones del entrenamiento, como podemos ver en la parte derecha de la imagen, cuando desenrollamos una red neuronal recurrente.

De esta forma, una red recurrente va calculando un estado interno en base a las "memorias" que va recopilando en cada iteración.

Uno de los mayores problemas de las redes neuronales recurrentes sigue siento el problema de los gradientes desvanecientes [37].

2.3.6. LSTM

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

Abajo podemos ver una imagen de que hay dentro de una red recurrente; solo una activación (en este caso la tangente hiperbólica), sin embargo, las LSTM tienen un comportamiento más complejo.

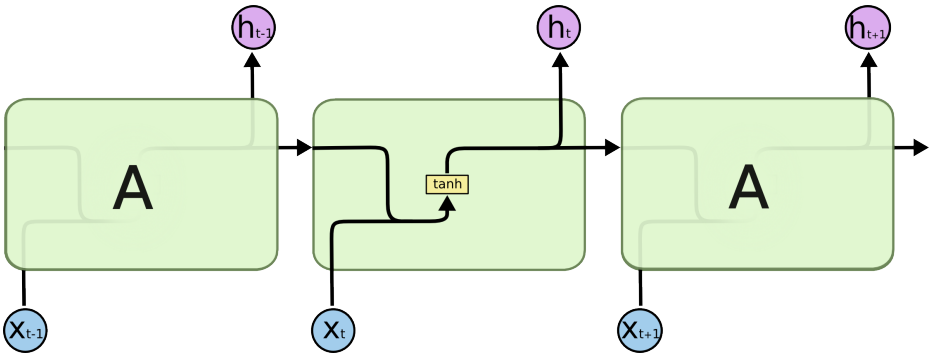


Fig. 2.16. Red neuronal recurrente por dentro [36]

En la figura 2.17 podemos ver que, en vez de tener una simple capa de activación, hay cuatro capas distintas.

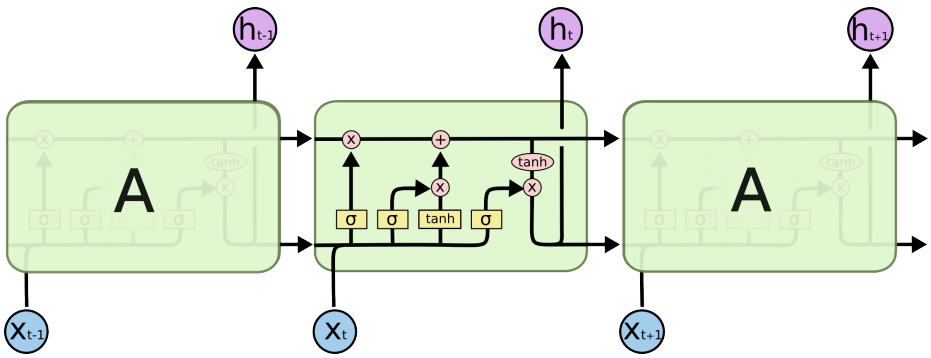


Fig. 2.17. LSTM por dentro [36]

En la figura anterior, cada línea representa un vector, los círculos rosa son operaciones elemento a elemento, los rectángulos amarillos son capas de una red neuronal (lo que implica que hay muchos más parámetros a entrenar) y las líneas que se juntan simbolizan la concatenación de dos vectores.

Desglosando su funcionamiento podemos ver en la figura 2.18 que en la parte superior hay un vector que pasa a través de la red sin ser modificado demasiado. La LSTM tiene la capacidad de añadir o quitar información de este vector que actúa como memoria "extra" o a largo plazo de la red.

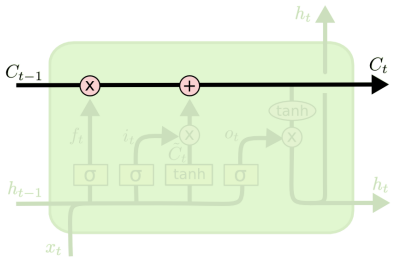
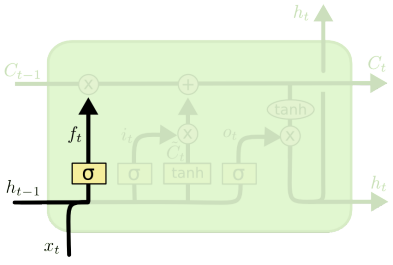


Fig. 2.18. Flujo superior de información de una LSTM [36]

Esta regulación la hace mediante una capa sigmoide. Lo que hace la sigmoide es transformar el vector de entrada en un vector con valores entre 0 y 1, determinando así cuánto de cada vector debería entrar arriba. De esta forma, al hacer el producto elemento a elemento del vector con números entre 0 y 1 y el vector superior de la LSTM, se elige cuánto de cada se conserva, desde nada hasta todo.

Vamos ahora a ver el recorrido de la información en una LSTM paso a paso. Lo primero que hace la LSTM es decidir qué información del vector superior se conserva o se olvida. Esta decisión la hace la primera capa sigmoide, conocida como "puerta de olvido". Esto lo hace a partir del *input* de ese elemento de la secuencia x_t y lo que le viene de la anterior h_{t-1}. Esto generará un vector con números entre 0 y 1 que regulará C_{t-1} como hemos explicado en el párrafo anterior.

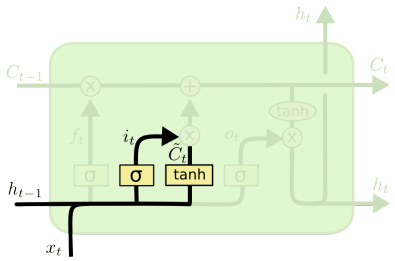


$$f_t = \sigma\left(W_f \cdot [h_{t-1}, x_t] \; + \; b_f\right)$$

Fig. 2.19. Puerta de olvido [36]

A continuación hay que decidir que información nueva (de x_t) se guarda en el vector superior. Este proceso tiene dos partes: primero una capa sigmoide llamada "puerta de *input*" y que decide qué valores del vector superior se actualizan (y el peso de dicha

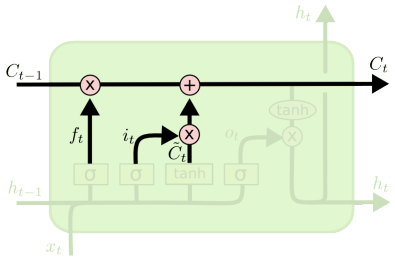
actualización) y después una capa de tangente hiperbólica que crea un vector con nuevos valores candidatos $\tilde{C}_t$ de estar en el vector superior.



$$i_t = \sigma\left(W_i \cdot [h_{t-1}, x_t] \; + \; b_i\right)$$
$$\tilde{C}_t = \tanh(W_C \cdot [h_{t-1}, x_t] \; + \; b_C)$$

Fig. 2.20. Puerta de *input* y $\tilde{C}_t$ [36]

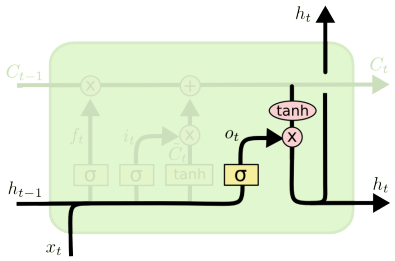
Ahora hay que actualizar C_{t-1} a C_t. Los pasos anteriores han servido para preparar los *inputs* para este paso. De esta forma, solo queda multiplicar elemento a elemento C_{t-1} y f_t y sumarle $\tilde{C}_t * i_t$ (donde $*$ es una multiplicación elemento a elemento).



$$C_t = f_t * C_{t-1} + i_t * \tilde{C}_t$$

Fig. 2.21. Actualización de C_t [36]

Para finalizar, hay que decidir el *output* de la red en la iteración t. Este *output* estará basado en C_t, pero será una versión filtrada. La forma de filtrarlo será mediante una sigmoide a partir de x_t y h_{t-1} y eso se multiplica a C_t pasado por una tangente hiperbólica (solo la función en este caso, no una capa), de esta forma los valores que salen están entre -1 y 1. Nótese que, dependiendo del problema, esta última tangente hiperbólica puede ser otra función de activación distinta.



$$o_t = \sigma\left(W_o \; [\, h_{t-1}, x_t] \; + \; b_o\right)$$
$$h_t = o_t * \tanh\left(C_t\right)$$

Fig. 2.22. Cálculo del *output* en una LSTM [36]

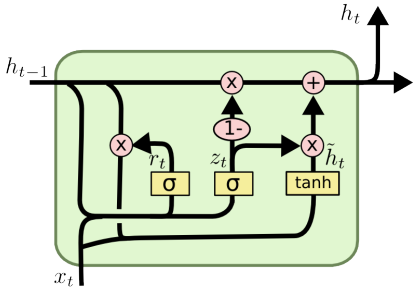
Este tipo de redes se han usado con éxito para crear textos con el estilo de Cervantes, Lovecraft o J.K. Rowling, entre otros. [38]. También pueden incluso aprender a hacer código fuente en C o *papers* en Latex [39]. Obviamente, ni los relatos tienen mucho

sentido a la larga ni el código que se genera es útil. Sin embargo, esto sirve para demostrar como las LSTM pueden aprender complejas estructuras sintácticas.

2.3.7. GRU

Junto a las LSTM, las redes GRU [7] [8] están popularizándose bastante ya que parecen comportarse tan bien como las LSTM en el procesado de secuencias de texto, pero son más simples.

La principal diferencia práctica es que una LSTM es más propensa a cometer sobreajuste (memorizar los datos pero no saber cómo actuar ante datos nuevos) cuando hay menos datos para entrenar. Por otra parte, aunque la GRU sea más simple, con el mismo tiempo de entrenamiento que una red LSTM (pero más iteraciones) se consigue un resultado equivalente a las LSTM [40].



$$z_t = \sigma\left(W_z \cdot [h_{t-1}, x_t]\right)$$
$$r_t = \sigma\left(W_r \cdot [h_{t-1}, x_t]\right)$$
$$\tilde{h}_t = \tanh\left(W \cdot [r_t * h_{t-1}, x_t]\right)$$
$$h_t = (1 - z_t) * h_{t-1} + z_t * \tilde{h}_t$$

Fig. 2.23. Capa de una red GRU [36]

Las redes GRU (*Gated Recurrent Unit*, o Unidad Recurrente con Puertas) son unas redes neuronales recurrentes que se pueden entender como una modificación de las LSTM. Como podemos ver arriba, la primera diferencia con las LSTM es que no hay puerta de olvido, en este caso se combina con la puerta de *input* en una puerta de actualización. Esto, junto con otros cambios menores contribuyen a que el modelo final sea más simple.

2.3.8. Redes bidireccionales

Las redes recurrentes bidireccionales [41] son simplemente dos redes neuronales recurrentes juntas. Los *inputs* de una van en orden normal mientras que los de la otra van en orden inverso. Después, los *outputs* de ambas se concatenan.

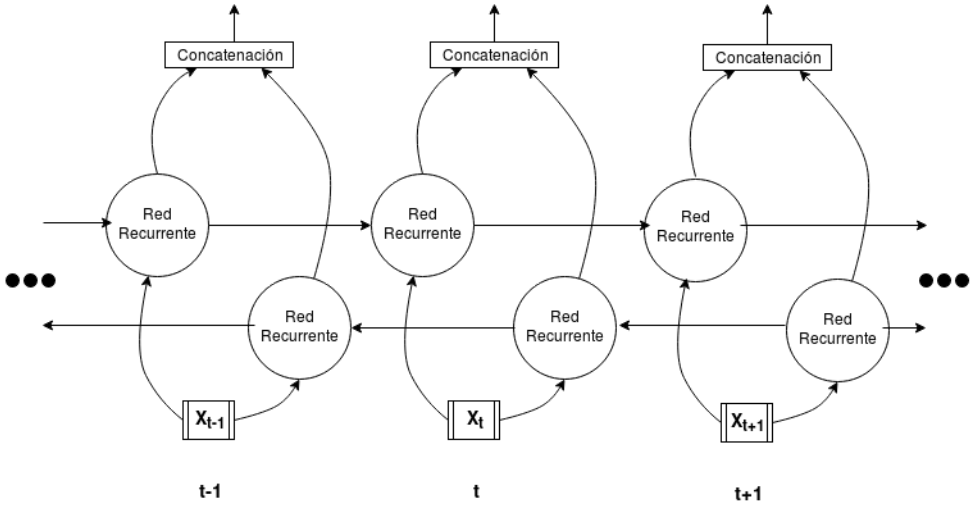


Fig. 2.24. Estructura de una red bidireccional

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

2.4. Métodos para el procesado de lenguaje natural

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

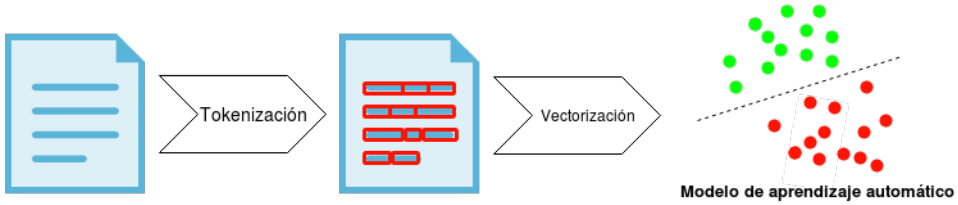


Fig. 2.25. Estructura típica de una solución de PLN

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

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

A continuación, hay que transformar los tokens en vectores de igual longitud de forma para que un modelo pueda aprender de ellos.

2.4.1. Bag of words

Los modelos más comunes para la vectorización de textos son los de *bag of words* o bolsa de palabras [42]. Estos se basan en crear un diccionario de tamaño n, donde n es el número total de tokens en el corpus, en el que a cada token se le asigna un número.

Después, cada token se convertiría en un vector vacío salvo por la posición n-ésima, donde habría un 1. Esto también se conoce como *one hot encoding*.

Para finalizar, cada texto sería la suma de los vectores de todos sus tokens. Estos vectores podrían usarse directamente como *inputs* en una red neuronal.

Sin embargo, los modelos de *bag of words* no recaban nada de información semántica ni de contexto sobre el texto, no saben sobre sinonimia o palabras con significados similares ya que todas tienen el mismo peso.

2.4.2. Modelos de lenguaje

El objetivo de estos modelos es aprender la probabilidad de que una frase dada aparezca en el lenguaje. La construcción de este tipo de modelos es un problema de aprendizaje no supervisado ya que, lo que están haciendo en el fondo, es aprender la distribución de probabilidad de todas las palabras de un lenguaje en base a los textos que se le dan para aprender. Para conseguir esto, lo que se suele hacer es un modelo autorregresivo que usa la probabilidad condicional [43] [44]:

$$\begin{aligned} p(X) &= p(x_1, x_2, ..., x_T) \\ &= p(x_1)p(x_2|x_1)p(x_3|x_1, x_2)...p(x_T|x_1, ..., x_{T-1}) \\ &= \prod_{t=1}^{T} p(x_t|x_{<t}) \end{aligned} \tag{2.43}$$

Donde X es una frase perteneciente al lenguaje.

De esta forma se consigue representar más fielmente el lenguaje y las relaciones semánticas y contextuales entre las palabras.

Usualmente estos modelos se consiguen entrenando redes neuronales con corpus muy grandes de texto. Como suelen ser modelos muy grandes, no se acostumbra a entrenarlos cada vez que alguien los quiere usar, sino que existen algunos ya preentrenados como Word2Vec [45] [46] y FastText [47], entre otros.

Para usarlos se crea una matriz de *embeddings* en la que, para cada palabra del vocabulario, hay un vector de tamaño fijo que la representa. De esta manera, para conseguir

la representación vectorial de tamaño fijo de una frase, simplemente hay que transformar todas sus palabras a vectores y hacer la media de los mismos.

FastText

FastText es un modelo del lenguaje de código libre creado por Facebook [48] que se caracteriza por no aprender solo de las palabras, sino también de los n-gramas (partes de palabras) que las componen. Además, hay modelos de FastText preentrenados para más de 157 idiomas diferentes, lo que lo hace ideal para casi cualquier problema de procesado de lenguaje natural, no solo los que están en inglés.

Con esto se consigue que palabras que están fuera del vocabulario como por ejemplo "girafa" puedan ser entendidas por el modelo, al obtener información sobre sus componentes. Incluso si se usasen palabras inventadas, el modelo podría intuir su significado a partir de sus componentes.

3. DESCRIPCIÓN DEL PROBLEMA

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

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

3.1. Objetivos específicos

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

Objetivos técnicos:

- Obtención un modelo de PLN y redes neuronales capaz de generar texto de forma automática.
- Utilización de métodos matemáticos y algorítmicos del estado del arte para obtener los mejores resultados posibles.

Objetivos académicos:

- Aprendizaje de los conceptos del aprendizaje automático.
- Aprendizaje de las técnicas punteras de PLN.
- Aprendizaje de las técnicas punteras de redes neuronales, redes neuronales profundas y redes neuronales recurrentes.

3.2. Solución propuesta

3.2.1. Análisis del problema

La primera complicación que aparece a la hora de idear una solución para el problema descrito más arriba viene dada porque el mismo problema no encaja fielmente en ninguno de los tipos de problemas de aprendizaje automático que hemos definido anteriormente. Como estamos generando texto, no tenemos un conjunto de datos objetivos que nos diga

claramente si nuestra predicción (en este caso, generación) es válida o no. Aun así, como se usan datos sin etiquetar, el consenso es que los modelos de lenguaje pertenecen a la rama de aprendizaje no supervisado [49] si nos enfocamos en la experiencia E sobre la que se aprende. Sin embargo está tomando cada vez más popularidad el termino *self-supervised learning*, algo así como aprendizaje supervisado por el propio modelo. En la práctica, la diferencia con el aprendizaje no supervisado es que este intenta aprender distribuciones de probabilidad y el *self-supervised learning* intenta predecir parte de los datos en base a otros datos (ningunos etiquetados).

Respecto a la tarea T, no estamos solucionando ningún problema de regresión, clasificación, *clustering* o similares. Sin embargo, sí podemos usar algunas de esas técnicas clásicas en nuestro problema. Vamos a entrenar nuestros algoritmos de forma que aprendan a predecir la siguiente palabra de un texto dado. Así, podemos saber si lo han hecho bien o lo han hecho mal.

La cuestión es que hay que buscar una métrica distinta a la precisión ya que no nos interesa que el algoritmo memorice el texto, sino que cree texto nuevo y coherente basándose en lo que le decimos. Es por esto que la elección de métrica es tan importante en este caso.

3.2.2. Métrica elegida

Una de las métricas más usadas en los modelos del lenguaje es la perplejidad [50]. Formalmente la perplejidad de un modelo sobre una distribución de probabilidades discreta p se define como:

$$ppl = 2^{H(p)} \tag{3.1}$$

Donde $H(p)$ es la entropía de una distribución discreta de probabilidades p:

$$H(p) = -\sum_{x} p(x) \log_2 p(x) \tag{3.2}$$

Para entender la ventaja de esta métrica frente a otras en la evaluación de modelos del lenguaje, primero hay que entender qué es la entropía en el contexto de la teoría de la información.

Básicamente, la entropía es el número medio de bits para codificar la información contenida en una variable aleatoria. Así, exponenciar la entropía es la cantidad total de información de una variable aleatoria.

En otras palabras, si quisiéramos predecir una secuencia de números del uno al seis, si predijésemos cada uno de ellos lanzando un dado de seis caras, cada una de esas predicciones acertaría una de cada seis veces. La perplejidad del dado es de seis. Generalizando,

la perplejidad de un modelo del lenguaje viene decirnos "este modelo acierta las mismas veces que lo haría un dado de x caras" donde x es la perplejidad [51].

Es por esto que se usa la perplejidad, si los posibles valores de nuestra distribución discreta de probabilidades p son palabras, la perplejidad es la media de decisiones que el modelo considera razonables tomar en cada momento. A más decisiones (más perplejidad), más incertidumbre tiene el modelo sobre lo que hacer.

Además, se suele elegir la perplejidad frente a la entropía $H(p)$ ya que la entropía no escala de forma lineal (tiene un logaritmo) y la perplejidad (al exponenciar) sí lo hace. Esto facilita la comprensión de la métrica.

De esta forma, si un modelo tiene para elegir entre todas las posibles palabras de un lenguaje (88000 solo en el diccionario de la Real Academia Española), el problema que se nos plantea es muy grande. Es por esto, que la mayoría de modelos que se han publicado, predicen carácter a carácter. De la misma manera, la perplejidad la calculan carácter a carácter.

Esto reduce el tamaño del problema a unas decenas de decisiones, como mucho un centenar si se añaden signos de puntuación. Esta aproximación tiene dos ventajas; la primera es que la perplejidad siempre va a ser sustancialmente más baja y la segunda es que el modelo aprende automáticamente a añadir signos de puntuación y mayúsculas. Sin embargo, es fácil que se cometan faltas de ortografía y se creen palabras sin sentido.

Se ha decidido crear un modelo que haga las predicciones palabra a palabra para introducir una innovación en los modelos del lenguaje actuales. Esto hace que el entrenamiento sea más complicado que si se usara un modelo que fuera carácter a carácter.

3.2.3. Métodos elegidos

Se han hecho pruebas con muchos tipos distintos de arquitecturas de redes neuronales, como se podrá ver en el Trabajo de Fin de Grado de Ingeniería Informática [52], y la conclusión es que la arquitectura que optimizaba el tiempo de entrenamiento y los resultados es una basada en una capa de *embeddings* con FastText y una capa GRU bidireccional de 1024 (2048 al ser bidireccional) neuronas.

Se ha usado un optimizador ADAM [53] ya que ajusta automáticamente el ritmo de aprendizaje η y consigue acercarse rápidamente al mínimo, no como otros algoritmos que son mucho más lentos [54]. Asimismo, ADAM tiene mecanismos que sirven para intentar evitar atascarse en mínimos locales, cosa que a veces no consigue evitar del todo.

Pot otra parte, los *batches* tienen un tamaño de 64 ya que, tras varias pruebas, era la cifra que mejor funcionó para el tamaño de los datos que se estaban manejando.

Respecto a la capa de *embeddings*, no se usa para hacer la media de los *embeddings*, como se ha explicado anteriormente. En este caso se usa para vectorizar directamente las palabras que entran en la red de forma que sean más fáciles de procesar y que conserven

toda la información que ha aprendido anteriormente la red neuronal que se usó en FastText. A esto también se le llama *transfer learning* [55]. La elección de FastText frente a otros *embeddings* como Word2vec [45] [46] viene motivada por sus mejores resultados [47].

Finalmente, la decisión de usar redes GRU está justificada por su facilidad en el entrenamiento y por los ligeramente mejores resultados que ofrecían en nuestro problema concreto ya que, en general, se comportan de forma muy similar a las LSTM u otras variantes de las mismas [56] [57].

3.2.4. Generación de texto

La generación de nuevo texto usa una secuencia de palabras y genera *n* nuevas palabras en base a todas las anteriores.

Para ello, en la última capa de la red neuronal se ha creado una función de activación softmax modificada que predice la probabilidad que tiene cada palabra del vocabulario en ser la siguiente. La función softmax es muy similar a la sigmoidal, salvo por dos diferencias. La primera es que la sigmoidal tiene las pendientes más acentuadas y la segunda es que la softmax produce valores normalizados, de forma que la suma de los mismos da 1, esto permite interpretarlas como probabilidades.

Definida en $\mathbb{R}^K \rightarrow \mathbb{R}^K$, la función softmax es la siguiente:

$$\sigma(\mathbf{z})_i = \frac{e^{z_i}}{\sum_{j=1}^{K} e^{z_j}} \quad para \quad i = 1, \ldots, K \quad y \quad \mathbf{z} = (z_1, \ldots, z_K) \in \mathbb{R}^K \tag{3.3}$$

La parte modificada de la softmax es que al final se utiliza un factor de aleatoriedad (también llamado temperatura) para reescalar las probabilidades originales del modelo, de esta manera se puede conseguir una predicción más uniforme que fuerza al generador de texto a producir una mayor variedad de palabras. Las probabilidades que salen de la softmax quedan de la siguiente manera:

$$q(\mathbf{z})_i = \frac{e^{z_i/T}}{\sum_{j=1}^{K} e^{z_j/T}} \tag{3.4}$$

Donde T es la temperatura.

Este método es usado también en en las técnicas de *Dark Knowledge* (conocimiento oscuro) o destilación de la información, donde se intenta condensar el conocimiento de la red neuronal para que sea más pequeña [58].

4. EXPERIMENTOS

Como se puede ver en el estudio práctico realizado en el Trabajo de Fin de Grado de Ingeniería Informática [52], los resultados de los modelos planteados aquí e implementados allí tienen un gran potencial. A modo de resumen podemos observar en la siguiente gráfica como los resultados se acercan bastante al estado del arte.

Set de datos	WikiText-2	Movie Lines
MaxPooling (Baseline)	76.177	60.490
AveragePooling (Baseline)	76.269	68.012
LSTM-512	31.789	25.969
LSTM-1024	31.360	25.303
GRU-512	33.092	25.058
GRU-1024	24.715	23.221
GRU-Dropout 0.15	28.520	25.681
Grave et al. 2016 [59]	68.9	-
Gong et al. 2018 [60]	39.14	-
OpenAI 2019 [3]	18.34	-

TABLA 4.1. TABLA DE PERPLEJIDADES, CUANTO MENOR, MEJOR EL MODELO

Asimismo, existe una gran variedad de modelos y es importante probarlos todos para poder obtener lo mejor de cada uno de ellos. De esta forma, con más investigación y más recursos *hardware* se podrían mejorar significativamente los resultados.

5. CONCLUSIONES

5.1. Cronograma

En el siguiente cronograma se puede apreciar el esfuerzo dedicado a cada tarea principal de este trabajo. Cabe destacar la gran cantidad de tiempo dedicado a la investigación y el estudio de las técnicas descritas y utilizadas durante toda la memoria, la cual asciende a las 400 horas.

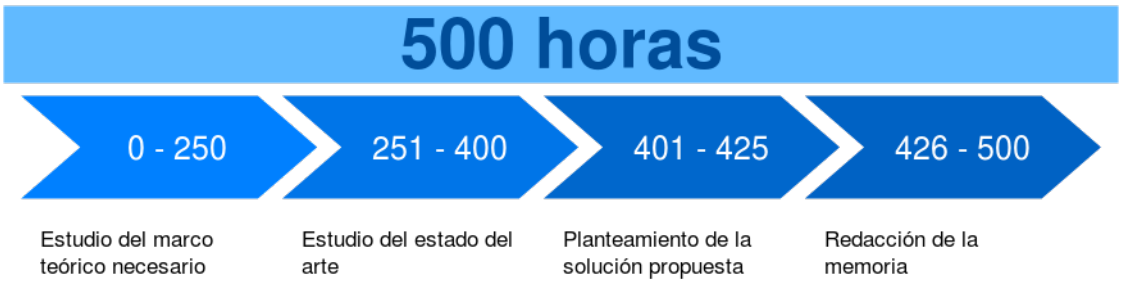


Fig. 5.1. Cronograma del desarrollo del trabajo

Por otra parte, el planteamiento de la solución que se propone durante el trabajo ha sido la tarea más rápida dado que solo se basaba en hacer un planteamiento teórico, obviando las complicaciones técnicas que han sido abordadas en el Trabajo de Fin de Grado de Ingeniería Informática [52]. Para finalizar, se ha dedicado una cantidad considerable de tiempo a la redacción de esta memoria ya que se ha necesitado, por una parte, aprender una gran cantidad de funcionalidades para *LaTeX* y, por otra, se ha tenido que resumir y exponer de forma meticulosa la gran cantidad de información recopilada en el tiempo de estudio.

5.2. Logros

Se han logrado, en gran medida todos los objetivos propuestos durante el trabajo. Primeramente, se han estudiado las bases teóricas del aprendizaje estadístico, de las redes neuronales y redes neuronales recurrentes, el procesado del lenguaje natural y los modelos del lenguaje, lo que ha servido para poder entender el estado del arte y poder hacer posteriormente un estudio práctico de los modelos de secuencias con redes neuronales recurrentes para la generación de texto.

Asimismo, también se ha conseguido un sistema simple, pero lo suficientemente flexible de generación de texto automático. Este hecho es muy relevante porque demuestra que el teorema de aproximación universal para las redes neuronales no es simplemente una demostración teórica, sino que tiene apalicaciones prácticas.

5.3. Trabajo futuro

Hoy en día, el campo de la generación de texto (y no solo texto) de forma automática usando el aprendizaje automático no hace más que crecer. De hecho ya hay modelos que crean imágenes y audio en base a muestras preexistentes con una calidad bastante destacable [61].

Es por eso que como principal objetivo de investigación futura se plantea el estudio del nuevo estado del arte de la generación automática de contenido multimedia, no solo texto.

En cuanto a la parte de generación de texto, el objetivo a batir sería replicar el modelo de OpenAI e intentar introducir mejoras para disminuir sus perplejidades reportadas.

BIBLIOGRAFÍA

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

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

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

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

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

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

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

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

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

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

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

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

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

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

[15] R. Bellman, R. Bellman y R. Corporation, *Dynamic Programming*, ép. Rand Corporation research study. Princeton University Press, 1957. [En línea]. Última consulta: Junio de 2019: https://books.google.es/books?id=rZW4ugAACAAJ.

[16] M. Verleysen y D. François, “The Curse of Dimensionality in Data Mining and Time Series Prediction”, en *Proceedings of the 8th International Conference on Artificial Neural Networks: Computational Intelligence and Bioinspired Systems*, ép. IWANN’05, Barcelona, Spain: Springer-Verlag, 2005, pp. 758-770. DOI: 10.1007/11494669_93.

[17] F. R. Bach, “Breaking the Curse of Dimensionality with Convex Neural Networks”, *CoRR*, vol. abs/1412.8690, 2014. arXiv: 1412.8690.

[18] A. Ng. [En línea]. Última consulta: Junio de 2019: https://www.slideshare.net/ExtractConf/andrew-ng-chief-scientist-at-baidu.

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

[20] W. S. McCulloch y W. Pitts, “A logical calculus of the ideas immanent in nervous activity”, *The bulletin of mathematical biophysics*, vol. 5, n.º 4, pp. 115-133, dic. de 1943. DOI: 10.1007/BF02478259.

[21] L. Guesmi, H. Fathallah y M. Menif, “Modulation Format Recognition Using Artificial Neural Networks for the Next Generation Optical Networks”, en. 2018. DOI: 10.5772/intechopen.70954.

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

[23] D. O. HEBB, “Organization of Behavior: A Neuropsychological Theory. Pp. xix, 335. New York: John Wiley & Sons, 1949. $4.00”, *The ANNALS of the American Academy of Political and Social Science*, vol. 271, n.º 1, pp. 216-217, 1950. DOI: 10.1177/000271625027100159.

[24] S. S. Haykin, *Neural networks and learning machines*. Pearson Education, 2009, pp. 50-55.

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

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

[27] K. 1. Ryszard Tadeusiewcz "Sieci neuronowe". [En línea]. Última consulta: Junio de 2019: http://galaxy.agh.edu.pl/~vlsi/AI/backp_t_en/backprop.html.

[28] G. Cybenko, “Approximation by superpositions of a sigmoidal function”, *Mathematics of Control, Signals, and Systems (MCSS)*, vol. 2, n.º 4, pp. 303-314, dic. de 1989. DOI: 10.1007/BF02551274.

[29] Kurt y Hornik, “Approximation capabilities of multilayer feedforward networks”, *Neural Networks*, vol. 4, n.º 2, pp. 251-257, 1991. DOI: 10.1016/0893-6080(91)90009-T.

[30] Z. Lu, H. Pu, F. Wang, Z. Hu y L. Wang, "The Expressive Power of Neural Networks: A View from the Width", en *Advances in Neural Information Processing Systems 30*, I. Guyon et al., eds., Curran Associates, Inc., 2017, pp. 6231-6239. [En línea]. Última consulta: Junio de 2019: http://papers.nips.cc/paper/7203-the-expressive-power-of-neural-networks-a-view-from-the-width.pdf.

[31] Y. Bengio, P. Simard y P. Frasconi, "Learning long-term dependencies with gradient descent is difficult", *IEEE Transactions on Neural Networks*, vol. 5, n.º 2, pp. 157-166, mar. de 1994. DOI: 10.1109/72.279181.

[32] S. Hochreiter, Y. Bengio y P. Frasconi, "Gradient Flow in Recurrent Nets: the Difficulty of Learning Long-Term Dependencies", J. Kolen y S. Kremer, eds., 2001.

[33] Z. Hao. [En línea]. Última consulta: Junio de 2019: https://isaacchanghau.github.io/img/deeplearning/activationfunction/sigmoid.png.

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

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

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

[37] R. Pascanu, T. Mikolov e Y. Bengio, "On the Difficulty of Training Recurrent Neural Networks", en *Proceedings of the 30th International Conference on International Conference on Machine Learning - Volume 28*, ép. ICML'13, Atlanta, GA, USA: JMLR.org, 2013, pp. III-1310-III-1318. [En línea]. Última consulta: Junio de 2019: http://dl.acm.org/citation.cfm?id=3042817.3043083.

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

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

[40] J. Chung, C. Gulcehre, K. Cho e Y. Bengio, "Empirical evaluation of gated recurrent neural networks on sequence modeling", 2014. [En línea]. Última consulta: Junio de 2019: https://arxiv.org/abs/1412.3555.

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

[42] Z. Harris, "Distributional structure", *Word*, vol. 10, n.º 23, pp. 146-162, 1954. [En línea]. Última consulta: Junio de 2019: Distributional%20Structure.

[43] F. Jelinek y R. L. Mercer, “Interpolated estimation of Markov source parameters from sparse data”, en *Proceedings, Workshop on Pattern Recognition in Practice*, E. S. Gelsema y L. N. Kanal, eds., Amsterdam: North Holland, 1980, pp. 381-397.

[44] Y. Bengio, R. Ducharme, P. Vincent y C. Janvin, “A Neural Probabilistic Language Model”, *J. Mach. Learn. Res.*, vol. 3, pp. 1137-1155, mar. de 2003. [En línea]. Última consulta: Junio de 2019: http://dl.acm.org/citation.cfm?id=944919.944966.

[45] T. Mikolov, K. Chen, G. Corrado y J. Dean, “Efficient Estimation of Word Representations in Vector Space”, *CoRR*, 2013. [En línea]. Última consulta: Junio de 2019: http://arxiv.org/abs/1301.3781.

[46] T. Mikolov, I. Sutskever, K. Chen, G. S. Corrado y J. Dean, “Distributed Representations of Words and Phrases and their Compositionality”, en *Advances in Neural Information Processing Systems 26*, C. J. C. Burges, L. Bottou, M. Welling, Z. Ghahramani y K. Q. Weinberger, eds., Curran Associates, Inc., 2013, pp. 3111-3119. [En línea]. Última consulta: Junio de 2019: http://papers.nips.cc/paper/5021-distributed-representations-of-words-and-phrases-and-their-compositionality.pdf.

[47] P. Bojanowski, E. Grave, A. Joulin y T. Mikolov, *Enriching Word Vectors with Subword Information*, 2016. eprint: arXiv:1607.04606.

[48] Facebook, *FastText*, 2016. [En línea]. Última consulta: Junio de 2019: https://fasttext.cc/.

[49] A. Radford et al., “Language Models are Unsupervised Multitask Learners”, 2018. [En línea]. Última consulta: Junio de 2019: https://d4mucfpksywv.cloudfront.net/better-language-models/language-models.pdf.

[50] S. Chen, D. Beeferman y R. Rosenfeld, “Evaluation Metrics For Language Models”, mayo de 2001. [En línea]. Última consulta: Junio de 2019: https://www.cs.cmu.edu/~roni/papers/eval-metrics-bntuw-9802.pdf.

[51] A. Schumacher, “Perplexity: what it is, and what yours is”, 2013. [En línea]. Última consulta: Junio de 2019: https://planspacedotorg.wordpress.com/2013/09/23/perplexity-what-it-is-and-what-yours-is/.

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

[53] D. Kingma y J. Ba, “Adam: A Method for Stochastic Optimization”, *International Conference on Learning Representations*, dic. de 2014.

[54] S. Ruder, *An overview of gradient descent optimization algorithms.* 2016. [En línea]. Última consulta: Junio de 2019: http://arxiv.org/abs/1609.04747.

[55] L. Y. Pratt, "Discriminability-Based Transfer between Neural Networks", en *Advances in Neural Information Processing Systems 5*, S. J. Hanson, J. D. Cowan y C. L. Giles, eds., Morgan-Kaufmann, 1993, pp. 204-211. [En línea]. Última consulta: Junio de 2019: http://papers.nips.cc/paper/641-discriminability-based-transfer-between-neural-networks.pdf.

[56] K. Greff, R. K. Srivastava, J. Koutník, B. R. Steunebrink y J. Schmidhuber, "LSTM: A Search Space Odyssey.", *CoRR*, 2015. [En línea]. Última consulta: Junio de 2019: http://arxiv.org/abs/1503.04069.

[57] R. Jozefowicz, W. Zaremba e I. Sutskever, "An Empirical Exploration of Recurrent Network Architectures", en *Proceedings of the 32Nd International Conference on International Conference on Machine Learning - Volume 37*, ép. ICML'15, Lille, France: JMLR.org, 2015, pp. 2342-2350. [En línea]. Última consulta: Junio de 2019: http://dl.acm.org/citation.cfm?id=3045118.3045367.

[58] G. Hinton, O. Vinyals y J. Dean, *Distilling the Knowledge in a Neural Network*, 2015. eprint: arXiv:1503.02531.

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

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

[61] OpenAI, *Generative Modeling with Sparse Transformers*, 2019. eprint: arXiv:1904.10509. [En línea]. Última consulta: Junio de 2019: https://openai.com/blog/sparse-transformer/.