Fundamentos matemáticos e implementación de algoritmos Deep Learning para análisis de texto

Master en Visual Analytics y Big Data

Trabajo de fin de master
Curso 2017-2018

Alfonso Javier Arias Lozano

Tutor:
Antoni Munar Ara

Índice general

1. Introducción **7**

2. Fundamentos de las Redes Neuronales **9**
2.1. El cerebro y las neuronas 9
2.2. Estructura básica de una Red Neuronal artificial 10
2.2.1. Funciones de activación 11
2.3. Feed Forward Neural Networks 14
2.4. Entrenamiento de redes neuronales 15
2.4.1. Gradient descent 16
2.4.2. Back Propagation 18

3. Recurrent Neural Networks **21**
3.1. ¿Por qué RNN para análisis de texto? 21
3.2. Estructura de RNNs del tipo *many to one* 23
3.3. BPTT en RNNs de tipo Many to One 24
3.3.1. BPTT en RNN del tipo *Many to Many* 27
3.4. El problema Vanishing Gradients 27
3.5. RNN de tipo Long Short-Term Memory 29
3.5.1. Variantes de LSTM 31
3.6. Bidirectional RNN o BRNN 32
3.7. Deep Recurrent Neural Networks 33

4. Interpretación de texto **37**
4.1. One Hot Vector 37
4.2. Word Embedding 38
4.3. RNNs y Word Embeddings 41

Abstract

Capítulo 1

Introducción

Capítulo 2

Fundamentos de las Redes Neuronales

Las redes neuronales artificiales forman parte de las técnicas de aprendizaje automático más utilizadas hoy en día para diferentes ámbitos, industrias y procesos. Su objetivo es aprender a partir de la experiencia para mejorar el rendimiento en una tarea determinada. Ya existen muchos otros algoritmos de aprendizaje supervisado pero, ¿qué tienen de especial estos algoritmos? Pues que su estructura se basa en simular la arquitectura del cerebro humano y las unidades que lo forman, las neuronas.

2.1. El cerebro y las neuronas

El cerebro humano es capaz de procesar y almacenar gran cantidad de toda la información que recibimos constantemente de nuestro entorno creando y modificando conexiones entre neuronas. De hecho, el cerebro de un bebé es capaz de resolver problemas que nuestros superordenadores más potente son incapaces de resolver. En pocos meses un bebé es capaz de reconocer la cara de sus padres, discernir entre distintos objetos y empezar a procesar el lenguaje. Se calcula que el cerebro humano está compuesto por unas 10^{11} neuronas y que cada una de ellas tiene alrededor de unas 1000 sinapsis o conexiones entre otras neuronas.

Resumiendo el funcionamiento de las neuronas, estas reciben estímulos y transmiten los impulsos nerviosos a otras neuronas u otros tipos de tejidos del organismo mediante una conexión denominada sinapsis. Este impulso nervioso se transmite desde el axón de una neurona hasta las dendritas de otras neuronas, que serían los receptores. Al recibir una dendrita un impulso nervioso, se segrega una determinada sustancia química que hace que la neurona se active, y si supera un determinado umbral, se envía el impulso eléctrico desde su axón a otras neuronas.

Es obvio que sería muy interesante poder simular con algoritmos el funcionamiento del cerebro para conseguir que las máquinas "aprendan".

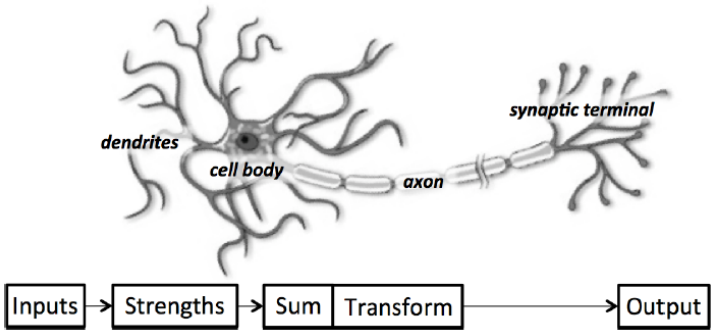


Figura 2.1: Descripción funcional de la estructura de una neurona en [1]

2.2. Estructura básica de una Red Neuronal artificial

Podemos traducir a lenguaje matemático la descripción funcional de nuestra imagen anterior. Denominaremos $x_1, \ldots, x_n$ las entradas o inputs que reciben las dendritas de nuestra neurona. A cada uno de estos inputs le aplicaremos un peso $w_1, \ldots, w_n$ para obtener una suma ponderada de toda la información recibida de la forma

$$z = w_0 + \sum_{i=1}^{n} w_i x_i$$

El término w_0 se denomina *bias* y se añade a este algoritmo como un parámetro inicial que ayuda a regular la activación de una neurona.

Una vez obtenido este valor, aplicaremos una función de activación f para obtener la salida de la neurona $y = f(z)$.

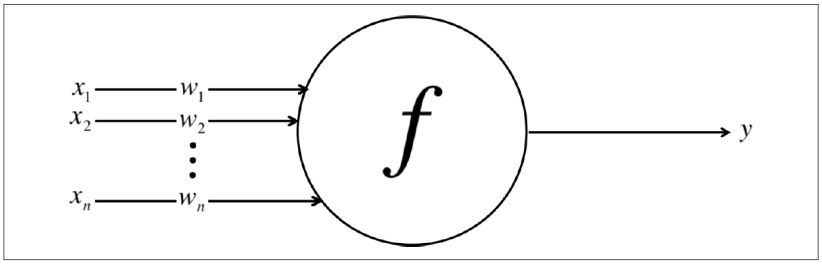


Figura 2.2: Representación matemática de una neurona. Fuente [1]

Para simplificar la notación consideramos todas las entradas como el vector $x = [x_1, x_2, \dots x_n]$ y construimos nuestro vector de pesos como $w = [w_1, w_2, \dots, w_n]$. Así tenemos que

$$y = f(w \cdot x + b)$$

donde b en el término bias. Otra posible notación que se suele usar para evitar tener que arrastrar el término b, es definir $b = w_0 x_0$ con lo que ahora tendremos

$$y = f(w \cdot x)$$

donde $x = [x_0, x_1, x_2, \dots x_n]$ y $w = [w_0, w_1, w_2, \dots, w_n]$.

2.2.1. Funciones de activación

Es la función que usa una neurona para calcular su output. Las más sencillas que se pueden utilizar son funciones lineales de la forma $f(z) = az + b$, pero al ser tan sencillas tienen sus limitaciones a la hora de aprender algunos patrones complejos. Para ello se usan funciones no lineales con algunas propiedades.

La primera de ellas es la función sigmoide, definida como $\sigma(z) = \frac{1}{1+e^{-z}}$. Su principal característica es que $\sigma(z) \in [0, 1], \forall z \in \mathbb{R}$. Esta cualidad la hace

muy útil para determinadas situaciones, como por ejemplo cuando queremos que una neurona nos devuelva una probabilidad.

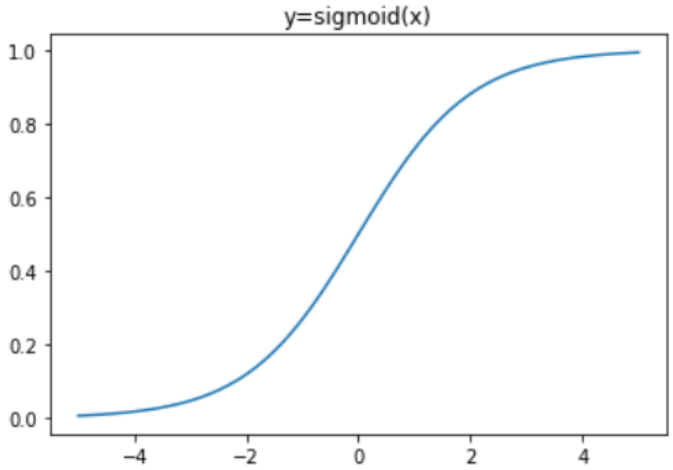


Figura 2.3: Función sigmoide

Otra función bastante común a la hora de construir redes neuronales es la tangente hiperbólica definida como $tanh(z) = \frac{e^x - e^{-x}}{e^x + e^{-x}}$. En este caso, tenemos una función que cumple $tanh(z) \in [-1, 1], \forall z \in \mathbb{R}$. Además, está centrada en 0 al contrario que la función sigmoide.

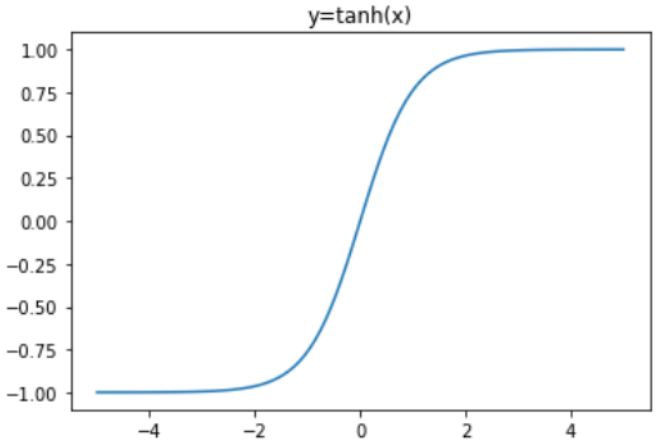


Figura 2.4: Función tangente hiperbólica

Mientras que las funciones anteriores tienen forma de S, se suelen usar otras que tienen una forma algo distinta como es el caso de la función Res-

tricted Linear Unit o ReLU. Usa la función $f(z) = max(0, z)$. Es usada para dar solución en algunos problemas concretos, pero tiene algunas limitaciones, por ejemplo, no es derivable en $f(z) = 0$ y $f'(z) = 0, \forall f(z) < 0$.

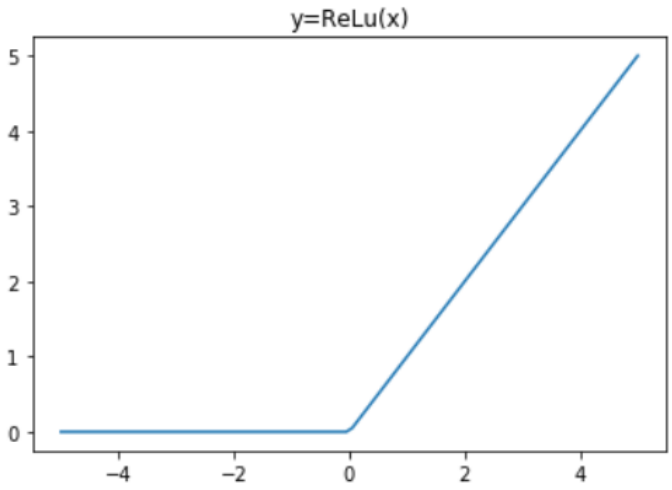


Figura 2.5: Función ReLU

Para solventar el problema de la derivada nula en todos los valores negativos de la función Relu, se usa una versión modificada llamada Leaky ReLU definida como $f(z) = max(\gamma z, z)$, donde γ suele ser un valor pequeño adaptado a nuestras necesidades, como $0,01$.

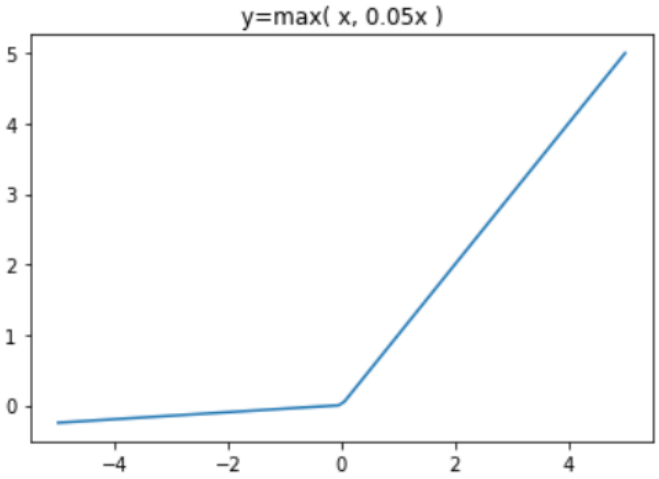


Figura 2.6: Función ReLU modificada

2.3. Feed Forward Neural Networks

Las neuronas que hemos definido anteriormente resuelven muy bien ciertos problemas, pero hay situaciones más complejas en las que una única neurona no es suficiente. Es por ello que nuestro cerebro no está formado por una sola neurona, sino por una red de neuronas interconectadas que se agrupan por layers o capas. En esta asociación o conexión entre neuronas lo que sucede es que el output y de nuestras neuronas de la primera capa pasará mediante sinapsis a las neuronas de la siguiente capa y así sucesivamente. Es decir, los datos de entrada se van propagando por las distintas capas sufriendo ciertas transformaciones hasta que se llega a una capa de neuronas que nos devuelve un objeto matemático como un valor, un vector, una palabra, y en el caso de nuestro cerebro, este output final podrá ser un impulso eléctrico para contraer un músculos o un estímulo para producir una hormona.

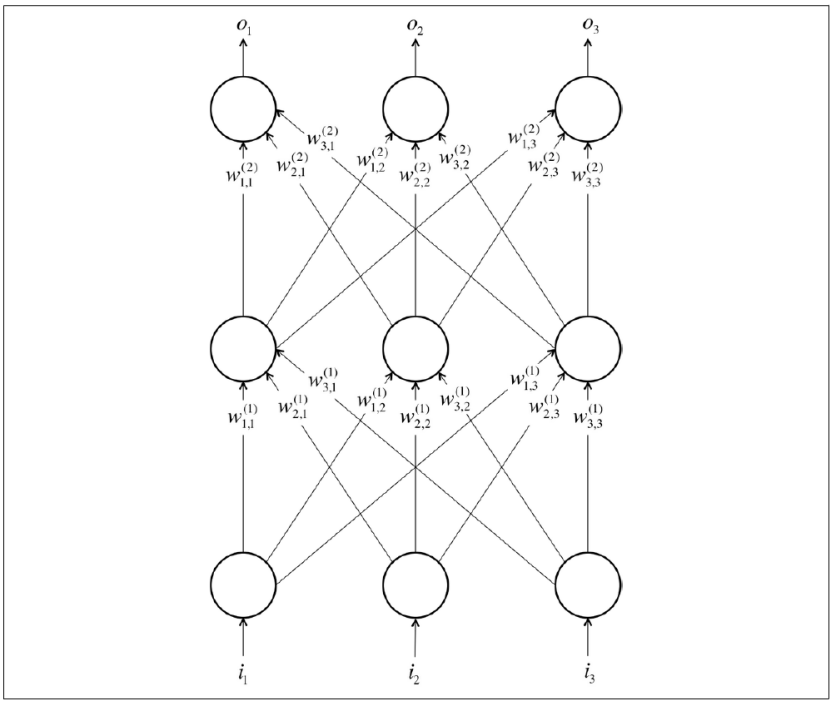


Figura 2.7: Red neuronal con varias capas

En 2.7 vemos un esquema de la construcción de estas redes formadas por varias capas. En este ejemplo, la primera capa estaría formada por las

3 primeras neuronas que reciben los inputs i_1, i_2 y i_3. Después, mediante la ayuda de los pesos w_{ij} que nos indican las conexiones que hay entre las neuronas de distintas capas, la información resultante del procesamiento de la primera capa se propaga hacia la siguiente capa, comunmente denominada como *capa oculta* o *hidden layer*. Estos pesos w_{ij} van a ir cambiando con data iteración en la que se use una red neuronal y este proceso de "aprendizaje" es lo que se llama la fase de entrenamiento de una red neuronal.

Si a una red neuronal añadimos capas ocultas se dice que estamos usando algoritmo de aprendizaje profundo o DEEP LEARNING (PONER MÁS COSAS solo hay conexión hacia alante, output layer...)

2.4. Entrenamiento de redes neuronales

Siguiendo con nuestro intento de simular la estructura del cerebro mediante las redes neuronales artificiales, nos queda imitar la parte más importante, que es la de aprender procesos cognitivos como reconocer un objeto o interpretar un texto. Por ello, las redes neuronales artificiales son una familia de algorimtos que pertenece a una rama del conocimiento o ciencia denominada *aprendizaje automático* o *machine learning*, que se encarga de programar en máquinas de una manera en la cual éstas pueden aprender de los datos. Quizás una definición más fomal nos oriente un poco más en el objetivo de las técnicas de machine learning:

> *"Un programa de ordenador aprende de la experiencia E con respecto a una clase de tareas T y una medida de rendimiento P, si su rendimiento en las tareas T, medido en base a la medida P, mejora con la experiencia E".*
>
> Tom Mitchell, 1997

ESCRIBIR MÁS SOBRE TRAINING Y TESTING SETS

Aplicando esta definición a nuestra red neuronal, debemos definir E,T y P. Podemos intuir que la experiencia E proviene de la cantidad de ejemplos que usemos como input en nuestro algoritmo. La tarea T será obtener un

resultado concreto en la salida y y mediremos el rendimiento P con una función error aplicada en la salida de la red neuronal.

Supondremos ahora que cada input de entrada $x = [x_0, x_1, x_2, \dots x_n]$, al que llamaremos "ejemplo" tiene asociado un valor $\hat{y}$ que es el que intentaremos .ªdivinarçon nuestro algoritmo. Una posible medida del rendimiento que obtenemos podría ser la siguiente función de error o función de coste:

$$E(y) = \frac{1}{2} \sum_{k=1}^{m} (y_k - \hat{y}_k)^2 \tag{2.1}$$

donde m es el número de ejemplos que usamos para medir el error. Para conseguir un rendimiento ótpimo, esta función debería devolvernos el valor 0, o al menos un valor lo más cercano posible a 0. Esto es equivalente a encontrar los parámetros que minimicen el error.

2.4.1. Gradient descent

El algoritmo Gradient Descent es un algoritmo de optimización que es comunmente usado para modificar parámetros de una función de manera iterativa para minimizarla. En el caso de las redes neuronales se usa para ir modificando nuestro conjunto de pesos w en cada iteración para minimizar la función de error o función de coste.

El funcionamiento de este algoritmo es muy intuitivo. Se mide el valor de la función para un valor concreto de sus parámetros y si no se ha llegado al mínimo deseado (E(y)=0 en nuestro caso), se da un salto (una rectificación de los parámetros) que haga que el valor de la función de error sea más pequeño tantas veces como sea necesario hasta que el algoritmo converja hasta el mínimo.

El valor de este salto se denominará *learning rate* o *tasa de aprendizaje*, que representaremos con γ y tendrá un papel bastante importante en la convergencia del algoritmo. Pasamos a reescribir nuestra medida del error 2.1 en función de los parámetros que queremos minimizar:

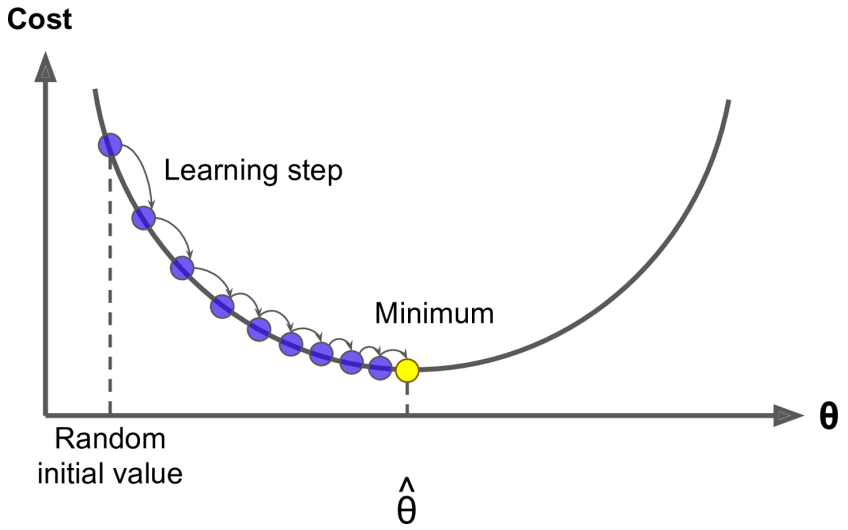


Figura 2.8: Representación del algoritmo Gradient Descent en [3]

$$E(w) = \frac{1}{2}\sum_{k=1}^{m}(f(x_k, w) - \hat{y})^2 \tag{2.2}$$

donde f es la función de activación que elijamos. Hacer inciso en que hemos escogido esta función de coste como ejemplo para ilustrar el algoritmo, pero podríamos elegir cualquier otra medida del error.

Nuestro objetivo entonces es encontrar el conjunto de parámetros w^* que cumpla

$$w^* = \underset{w}{arg\,min}\, E(w)$$

Asumiendo que $E(w)$ es diferenciable dos veces, se tienen que cumplir las siguientes condiciones:

- $\frac{\partial E(w)}{\partial w}|_{w=w*} = 0$
- $H_{mn} = \frac{\partial E(w)}{\partial w_m \partial w_n}|_{w=w^*}$ tiene valores propios positivos, es decir, que H sea definida positiva, siendo H la matriz Hessiana con $m, n \in [1, \ldots, dim(w)]$

—-BUSCAR CITA DE PUBLICACIÓN EN MINIMIZACIÓN EN SUPERFICIES

Ahora estamos en las condiciones óptimas para calcular el gradiente

$$\triangle w = \frac{\partial E(w)}{\partial w} = \frac{1}{2}\sum_{k=1}^{m}\frac{\partial}{\partial w}\left(f(x_k, w) - \hat{y}\right)^2 = \sum_{k=1}^{m}\left(f(x_k, w) - \hat{y}\right)\frac{\partial f(x_k, w)}{\partial w} \tag{2.3}$$

Una vez que ya sabemos como calcular el gradiente, usamos γ para ir actualizando los pesos de forma iterativa:

1. Generar la semilla $w^{[0]}$ y definir $s = 0$
2. $w^{[s+1]} = w^{[s]} - \gamma \triangle w^{[s]}$
3. $s = s + 1$
4. Repetir 2. y 3. hasta alcanzar criterio de parada

Apuntar que para esta fórmula es necesario usar todos los ejemplos de nuestro dataset para realizar cada una de las iteraciones en el algoritmo Gradient Descent. Esta es la razón por la que también es conocido como Batch Gradient Descent, porque usa el lote completo de todo nuestro data set. Esto en algunos casos puede ser bastante costoso computacionalmente de calcular. Un variación de este algoritmo es el Mini Batch Gradient Descent, que lo único que hace es usar un lote más pequeño de ejemplos para minimizar los parámetros.

Otro de los problemas que nos podemos encontrar a la hora de minimizar funciones de coste es que estas no sean totalmente convexas y tengan mínimos locales, lo que nos dificultaría mucho encontrar el mínimo global. Por ello, interesante utilizar Stochastic Gradient Descent (SGD) que consiste en usar un único ejemplo en cada iteración, o una variación del propio del propio (SDG) en la que para cada iteración se usa un mini lote distinto de ejemplos.

2.4.2. Back Propagation

Ya hemos visto como ir modificando los parámetros de nuestra red neuronal para mediante el uso de técnicas de cálculo diferencial. Pero algo que

tener en cuenta es que al introducir más capas ocultas los cálculos del gradiente 2.3 se van a complicar por la dependencia que hay entre las diferentes capas.

En [5] se publicó por primera vez el desarrollo de los cálculos de este algoritmo, basados en derivadas parciales, aplicados a distintos tipos de redes neuronales.

—-EXPLICAR ALGO MÁS DE BACKPROPAGATION—

Capítulo 3

Recurrent Neural Networks

En el capítulo anterior hemos definido la estructura de una red neuronal básica, en la que la información de transmitía desde la capa de entrada hasta la capa salida. Hoy en día existen muchos tipos y variaciones que buscan ajustarse al tipo de problema que queremos resolver. Por ejemplo, para el reconocimiento de patrones visuales o imágenes, es muy común usar Convolutional Neural Networks –EXPLICARLAS POR ENCIMA–.

3.1. ¿Por qué RNN para análisis de texto?

El lenguaje está formado por secuencias de sonidos, palabras, frases o textos. El significado de cada elemento de una secuencia dependerá de elementos anteriores y tendrá influencia en elementos futuros. Por ello una red neuronal de tipo *feed forward*, en la que el flujo de activación solo viaja en un sentido y no tiene conexiones que usen la información de elementos anteriores no es lo más óptimo para datos secuenciales. Por ello, surgen las redes neuronales recurrentes o Recurrent Neural Networks (RNN) donde la salida que devuelve una neurona vuelve a ser usada en el siguiente paso por esa misma neurona en una especie de bucle para que no se pierda información.

Cada una de esta iteraciones que se producen en el bucle donde una misma neurona se retroalimenta, se puede interpretar como un time step t. Con esta idea, podemos "desplegar" la red neuronal y convertirla en una red

feed forward que ya hemos estudiado previamente y nos ahorraría muchos cálculos.

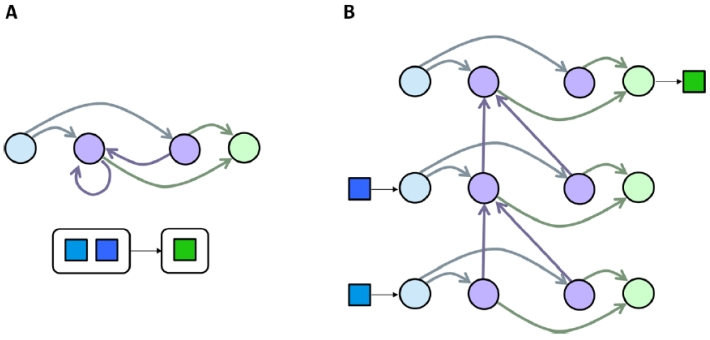


Figura 3.1: Representación de RNN (A) como modelo feed forward (B) respecto al tiempo en [1]

Como observamos en el modelo la imagen anterior, vemos que tenemos dos entradas que se introducen en el modelo en tiempos diferentes y obtenemos una única salida. Aún así, esta estructura es moldeable para dar solución a ciertos problemas sobre secuencias que tienen bastante impacto hoy en día. Algunos de ellos son:

- **Speech recognition o reconocimiento de voz:** Partimos de una entrada X y el objetivo es transcribirla hasta un texto Y, donde tenemos que tanto X como Y son datos secuenciales. X es un audio que se escucha en un periodo de tiempo e Y es una secuencia de palabras.

- **Generación de música:** En este caso buscamos que la salida Y sea de tipo secuencial, ya que la música está compuesta por sonidos que se activan secuencialmente en el tiempo. En cambio, el input X puede ser conjunto discreto indicando el género de música que queremos generar o un conjunto de notas con las que queremos empezar.

- **Análisis de sentimiento:** En este caso tenemos el ejemplo contrario. Los datos de entrada serán una review de un producto o un comentario formada por texto, lo que hace que X sea secuencial. Sin embargo, la salida Y es un entero indicando una valoración, como una puntuación

del 1 al 10 o una salida binaria indicando si la publicación ha sido valorada con un like o un dislike.

Entonces deberemos elegir el tipo de RNN que mejor se adapte a lo que necesitamos

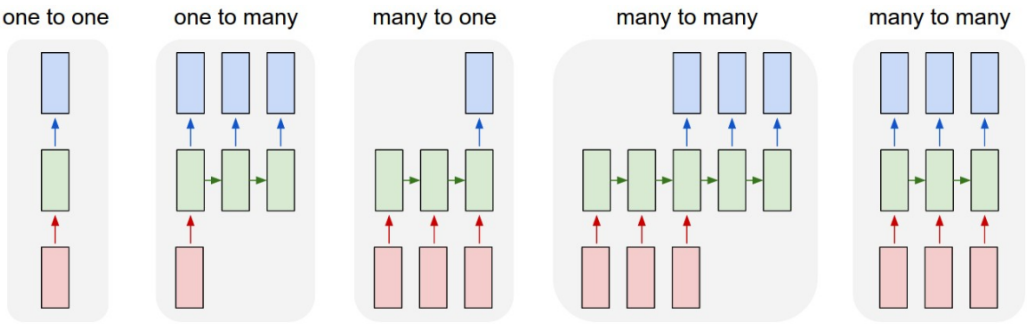
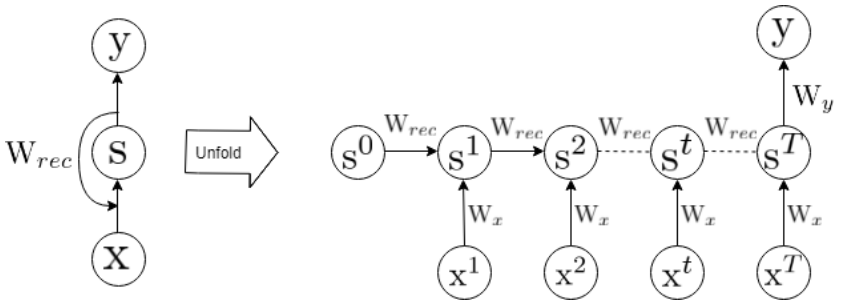


Figura 3.2: Diferentes tipos de RNN. [6]

Una vez valoradas todas estas opciones, en este trabajo abordaremos el problema del análisis del sentimiento o más conocido como *Sentiment Analysis*. En lo que queda de este capítulo, pasaremos a estudiar la estructura de una red neuronal del tipo *many to one* para poder dar solución a nuestro problema y poder implementarlo.

3.2. Estructura de RNNs del tipo *many to one*

Lo primero que haremos será representar con un esquema la RNN que usaremos para nuestro problema de análisis del sentimiento y definir la notación que usaremos.



Como comentamos previamente, para trabajar con datos de tipo secuencial es interesante trabajar con time steps. Esto nos indicará con qué posición de nuestra secuencia estamos trabajando. Introduciremos la variable $t \in [1, \ldots, T]$ como superíndice, donde T será la longitud de cada instancia. Por ejemplo, si fijamos los comentarios de un producto en una longitud de 300 palabras fijaremos $T = 300$. El valor capa oculta s se irá actualizando en cada step de la siguiente forma:

$$S^t = f\left(W_{rec} * S^{t-1} + W_x * X^t\right) \tag{3.1}$$

donde f es una función de activación. Observamos que para el valor del estado de layer en un tiempo t S^t, usamos información del estado anterior S^{t-1} y este a su vez de S^{t-2} de una manera recurrente. Como ya hemos definido nuestra RNN como una red de tipo feed forward a través del tiempo podemos usar las técnicas de entrenamiento que explicamos en el capítulo anterior.

3.3. BPTT en RNNs de tipo Many to One

Esta vez será algo más complicado matemáticamente calcular los gradientes de los pesos, ya que tenemos más dependencias entre time steps. Para ello usaremos el algoritmo de Back Propagation que se usa en redes neuronales del tipo feed forward. Pero ahora la información se propaga en una dirección temporal, por lo que la aplicación de este algoritmo en RNN es denominado *Backpropagation Through Time* o *BPTT*.

Para empezar con los cálculos, debemos establecer cierta notación:

$$a^t = W_{rec} * S^{t-1} + W_x * X^t$$
$$y = F(W_y * S^T)$$
$$z = W_y * S^T$$

Denominaremos $I - 1$ como el tamaño de cada vector input, $H - 1$ será el

número de unidades en nuestra capa oculta y $K-1$ el número de unidades en la capa de salida. El hecho de restar una unidad en cada una de esta cantidades se hace con la intención de introducir los valores del bias para evitar arrastrar un vector b durante todos los cálculos. Usaremos además una función de coste E.

Definimos los siguientes valores:

$$\delta_{z_k} = \frac{\partial E(y)}{\partial z_k} \tag{3.2}$$

$$\delta_{a_h^t} = \frac{\partial E(y)}{\partial a_h^t} \tag{3.3}$$

donde 3.2 es la derivada del error respecto a la entrada de la capa de salida en el tiempo T y 3.3 la derivada del error respecto a la entrada recursiva en un tiempo fijo t.

Hemos definido en 3.3 notación matricial que usaremos. Por tanto, los valores se definirán de la siguiente forma:

$$y_k = F\left(z_k\right)$$

$$z_k = \sum_{h=1}^{H} W_{y_{kh}} s_h^t$$

$$s_h^t = f\left(a_h^t\right)$$

$$a_h^t = \sum_{i=1}^{I} W_{x_{hi}} x_i^t + \sum_{g=1}^{H} W_{rec_{hg}} s_g^{t-1}$$

Si desarrollamos la ecuación 3.2 obtenemos

$$\delta_{z_k} = \frac{\partial E(y)}{\partial z_k} = \frac{\partial E(y_k)}{\partial z_k} = \frac{\partial E(y_k)}{\partial y_k}\frac{\partial y_k}{\partial z_k} = \frac{\partial E(y_k)}{\partial y_k} F'\left(z_k\right)$$

donde hemos usado que y_k solo depende de z_k y hemos aplicado la regla de la cadena en $y_k = F(z_k)$.

Pasamos ahora a desarrollar 3.3

$$\delta_{a_h^T} = \frac{\partial E\left(y\right)}{\partial a_h^T} = \sum_{k=1}^{K} \frac{\partial E\left(y_k\right)}{\partial a_h^T} = \sum_{k=1}^{K} \frac{\partial E\left(y_k\right)}{\partial y_k} \frac{\partial y_k}{\partial z_k} \frac{\partial z_k}{\partial s_h^T} \frac{\partial s_h^T}{\partial a_h^T} = \sum_{k=1}^{K} \delta_{z_k} W_{y_{kh}} f'\left(a_h^T\right)$$

donde hemos tenido en consideración que a_h^T influye en todos los y_k. Además, para calcular esta derivada hemos usado las siguientes igualdades:

$$\delta_{z_k} = \frac{\partial E(y_k)}{\partial y_k} \frac{\partial y_k}{\partial z_k},\ \frac{\partial z_k}{\partial s_h^T} = W_{y_{kh}},\ f'\left(a_h^T\right) = \frac{\partial s_h^T}{\partial a_h^T}$$

Otra cantidad que nos será útil calcular es la siguiente

$$\delta_{a_h^{t-1}} = \frac{\partial E\left(y\right)}{\partial a_h^{t-1}} = \sum_{k=1}^{K} \frac{\partial E\left(y_k\right)}{\partial a_h^{t-1}} = \sum_{k=1}^{K} \frac{\partial E\left(y_k\right)}{\partial a^t} \frac{\partial a^t}{\partial a_h^{t-1}} = \sum_{k=1}^{K} \sum_{l=1}^{H} \frac{\partial E\left(y_k\right)}{\partial a_l^t} \frac{\partial a_l^t}{\partial a_h^{t-1}} =$$

$$= \sum_{k=1}^{K} \sum_{l=1}^{H} \frac{\partial E\left(y_k\right)}{\partial a_l^t} \frac{\partial a_h^t}{\partial s_h^{t-1}} \frac{\partial s_h^{t-1}}{\partial a_h^{t-1}} = \sum_{l=1}^{H} \delta_{a_l^t} W_{rec_{lh}} f'\left(a_h^{t-1}\right)$$

donde hemos usado

$$\delta_{a_l^t} = \frac{\partial E\left(y_k\right)}{\partial a_l^t},\ \frac{\partial a_h^t}{\partial s_h^{t-1}} = W_{rec_{lh}},\ \frac{\partial s_h^{t-1}}{\partial a_h^{t-1}} = f'\left(a_h^{t-1}\right)$$

Ya tenemos todas los cálculos necesarios para obtener como varía la función de coste respecto a los pesos

$$\frac{\partial E(y)}{\partial W_{y_{kh}}} = \frac{\partial E(y_k)}{\partial z_k} \frac{\partial z_k}{\partial W_{y_{kh}}} = \delta_{z_k} s_h^t \tag{3.4}$$

$$\frac{\partial E(y)}{\partial W_{x_{ih}}} = \sum_{t=1}^{T} \frac{\partial E(y_k)}{\partial a_k^t} \frac{\partial a_k^t}{\partial W_{x_{ih}}} = \sum_{t=1}^{T} \delta_{a_k^t} x_i^t \tag{3.5}$$

$$\frac{\partial E(y)}{\partial W_{rec_{hg}}} = \sum_{t=1}^{T} \frac{\partial E(y_k)}{\partial a_h^t} \frac{\partial a_h^t}{\partial W_{rec_{hg}}} = \sum_{t=1}^{T} \delta_{a_k^t} s_g^{t-1} \tag{3.6}$$

Estos son los gradientes que necesitamos para ir actualizando los pesos

en cada iteración mediante el algoritmo de optimización que deseemos, ya sea Gradient Descent o Stochastic Gradient Descent.

3.3.1. BPTT en RNN del tipo *Many to Many*

En el que caso de tener un algoritmo en el que la salida de nuestra red neuronal sea una secuencia, podremos aprovechar los cálculos anteriores para aplicar el mismo algoritmo BPTT.

En este algoritmo obtendremos un y^t en cada time step. Pero como ya vimos anteriormente en 3.1 cuando hablamos de los distintos tipos de RNN, no es necesario que exista un y^t para cada $t \in [1, 2, \ldots, T]$ Por tanto, definiremos el intervalo de time steps donde existe algún output como $t' \in [1, 2, \ldots, T']$. Una vez concretado este punto, basta con aplicar el algoritmo de BPTT teniendo en cuenta únicamente cada elemento $y^{t'}$ e ignorando otros outputs en nuestro nuevo intervalo. De esta manera obtendremos los gradientes $\forall t' \in [1, 2, \ldots, T']$. Ahora toca el paso final, que es realizar una suma en función de t' de todos los gradientes obteniendo:

$$\frac{\partial E(y)}{\partial W_{y_{kh}}} = \sum_{t'=1}^{T'} \frac{\partial E(y^{t'})}{\partial W_{y_{kh}}}$$

$$\frac{\partial E(y)}{\partial W_{x_{ih}}} = \sum_{t'=1}^{T'} \frac{\partial E(y^{t'})}{\partial W_{x_{ih}}}$$

$$\frac{\partial E(y)}{\partial W_{rec_{hg}}} = \sum_{t'=1}^{T'} \frac{\partial E(y^{t'})}{\partial W_{rec_{hg}}}$$

3.4. El problema Vanishing Gradients

Ahora que ya entendemos como funcionan una red neuronal recurrente, es natural preguntarnos de qué manera influye un x^t en el estado de una capa en un tiempo $t + k$ y si es posible aumentar o disminuir esta influencia. De hecho, en [7] se demostró teoricamente que se puede encontrar una RNNcon un número suficiente de capas y neuronas describa la relación existente entre

una salida Y y una entrada X.

Pero tenemos que a pesar de que teoricamente se pueda encontrar RNN perfecta, encontrarla sería bastante complicado si la entrenamos con el algoritmo de BPTT y quizás no lo más óptimo por el problema que explicamos a continuación, al que se le denomina como *Vanishing Gradients*.

Como hemos visto anteriormente, el valor del gradiente 3.6 depende directamente de la cantidad $\frac{\partial S^t}{\partial S^{t-1}}$. Sería interesante calcular cómo varía el término s^t según un estado anterior en un tiempo k. Aplicando la regla de la cadena, tenemos que

$$\frac{\partial S^t}{\partial S^k} = \frac{\partial S^t}{\partial S^{t-1}} \frac{\partial S^{t-1}}{\partial S^{t-2}} \cdots \frac{\partial S^{k+1}}{\partial S^k} = \prod_{i=k}^{t-1} \frac{\partial S^{i+1}}{\partial S^i}$$

Usando ahora la definición de S^t dada en 3.1 y usando la definición del Jacobiano para calcular la derivada de una función recursiva, tenemos:

$$\frac{\partial S^{i+1}}{\partial S^i} = diag\left(f'\left(W_x X^i + W_{rec} S^{i-1}\right)\right) W_{rec}$$

Por tanto, si queremos retroceder i timesteps este gradiente será

$$\frac{\partial S^i}{\partial S^1} = \prod_{1}^{i-1} diag\left(f'\left(W_x X^i + W_{rec} S^{i-1}\right)\right) W_{rec}$$

Como podemos ver en [9], si el mayor valor propio de la matriz W_{rec} es mayor que 1, el gradiente diverge. Por el contrario, si es menor que 1, el gradiente desaparece. Para comprenderlo intuitivamente con un ejemplo, supongamos que nuestra función de activación f es la función sigmoide σ. Siempre tendremos que $f'(x) < 1$. Si los valores de W_{rec} son demasiado pequeños es inevitable que nuestra derivada tienda a 0 por multiplicar varias veces cantidades menores que 1.

Si tenemos en cuenta que $\frac{\partial S^i}{\partial S^1}$ nos indica cómo influye la variación de S^1 en S^i, podemos afirmar con los cálculos anteriores que no tiene ninguna influencia y que por tanto perderemos información.

Como resumen, nos tenemos que quedar con la idea de que uno de los principales problemas que nos encontramos al entrenar una RNN es que

los gradientes diverjan o tiendan a 0, lo que nos impedirá obtener buenos resultados. Existen ciertas técnicas para paliar estos problemas, como por ejemplo, una correcta elección de los pesos iniciales w^0 o cambiar la estructura de nuestra red neuronal con lo que se conoce como RNN de tipo *LSTM*.

3.5. RNN de tipo Long Short-Term Memory

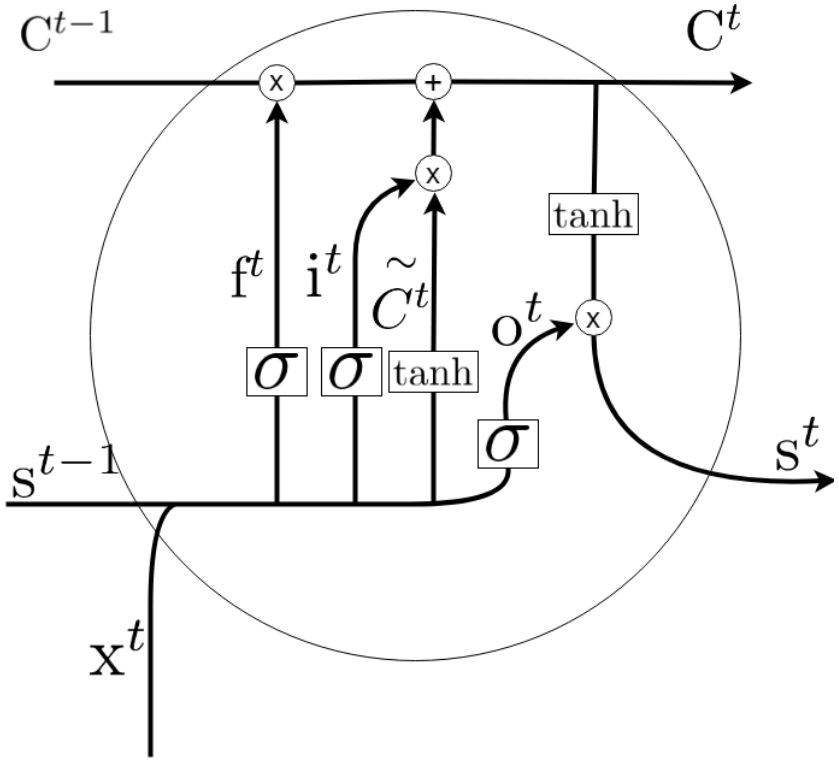
Para evitar el problema anterior, se recurre a un tipo especial de RNN llamado *Long Short Term Memory* o *LSTM*. Fue propuesto por primera vez en [10], dónde se comprobaba que el problema con la inestabilidad de los gradientes se puede evitar medienta la introducción de celdas o *cells* que guardan varios estados de la red y se realiza una combinación entre ellos para optimizar la convergencia. A lo largo de los años se han ido proponiendo algunas variaciones de esta idea con el fin de adaptarla a los problemas que se necesitaban resolver.

Vamos a pasar a explicar la idea original de *LSTM*. Todas las RNN tienen forma de una cadena en la que se repite el cálculo de un módulo o célula mediante la función de activación como vimos en 2.7. Una red neuronal del tipo LSTM tiene esta misma estructura de cadena, pero se introducen algunos parámetros y cambios en el cálculo de los estados de cada capa de la siguiente forma:

La aportación que hicieron Hochreiter y Schmidhuber es mantener el estado C^t de la célula, con ligeras modificaciones de cálculo, para usarlo más adelante. Dicho de otra manera, esto nos ayuda a guardar información de los primeros imputs y que tengan mayor influencia al final de la secuencia. Para regular la información que entra a cada célula o módulo, se usan puertas o *gates* formadas por la función sigmoide σ, ya que nos permite regular con un valor entre 0 y 1 la cantidad de información que queremos dejar pasar. Describiendo este módulo matemáticamente obtenemos las siguientes ecuaciones:

$$f^t = \sigma\left(W_f\left[s^{t-1}, x^t\right]\right)$$

$$i^t = \sigma\left(W_i\left[s^{t-1}, x^t\right]\right)$$



$$\widetilde{C}^t = \tanh\left(W_C\left[s^{t-1}, x^t\right]\right)$$

$$C^t = f^t C^{t-1} + i^t \widetilde{C}^t$$

$$o^t = \sigma\left(W_o\left[s^{t-1}, x^t\right]\right)$$

$$s^t = o^t tanh\left(C^t\right)$$

Apuntar que aquí hemos cambiado algo la notación para hacerla más cómoda ya que hay diferentes matrices de pesos. Esta es la definición de la nueva notación

$$W_f\left[s^{t-1}, x^t\right] = W_{f_s} s^{t-1} + W_{f_x} x^t$$

donde W_f es la matriz de pesos formada por concatenación de W_{f_s} y W_{f_x} asociada a la función f^t de la célula C^t.

Cada una de las ecuaciones y gates que hemos definido tienen su significado. Siguiendo el camino de las flechas de la imagen, vemos que la primera ecuación que usamos es la denominada *forget gate*, representada por f^t, don-

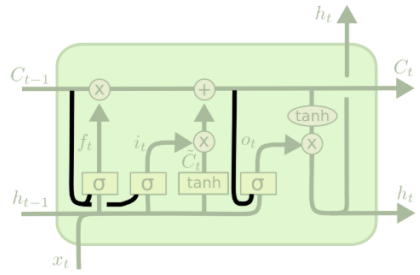
de se elige la información del estado anterior que no se va a usar. El siguiente paseo es elegir qué nueva información vamos a usar en el nuevo estado de nuestra celda con la ayuda de nuestra *input gate* representada por i^t. Combinando las dos informaciones anteriores, podemos calcular un nuevo valor $\widetilde{C}^t$ que indica un posible estado en nuestra celda. Pasamos ahora a calcular C^t, multiplicando C^{t-1} por f^t para olvidar o eliminar la información que consideramos que no era necesaria en los pasos anteriores. Y multiplicando ahora $\widetilde{C}^t$ por i^t, obtener la proporción del cambio en nuestra celda. Finalmente pasamos a usar nuestra *output gate* para decidir qué valor queremos devolver a la siguiente celda. Para ello se usa una combinación de los valores de entrada y nuestro nuevo estado C^t.

En resumen, un estructura *LSTM* puede aprender a reconocer un input importante, guardarlo temporalmente para después eliminarlo u olvidarlo gracias a los diferentes tipos de *gates* que hemos definidos. Este es el motivo por el cual funcionan tan bien al procesar información secuencial en series temporales, textos, audios y muchos más problemas.

3.5.1. Variantes de LSTM

A partir de esta idea, se añadieron más conceptos y variaciones para conseguir un mejor aprendizaje en RNNs. Algunos de los más representativos son los siguientes:

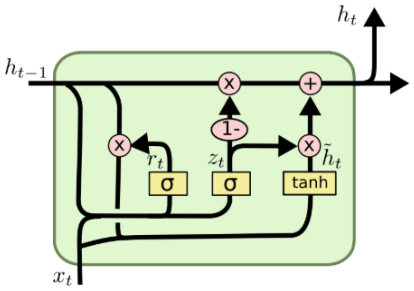
Peephole conections: En el ejemplo de *LSTM* original, cada una de las *gates* que definimos solo usan la información del input x^t y estado de la celda anterior s^{t-1}. Sin embargo, en [13] se observa que puede ser interesante tener en cuenta la información de c^{t-1}



$$f_t = \sigma\left(W_f \cdot [\boldsymbol{C_{t-1}}, h_{t-1}, x_t] \; + \; b_f\right)$$
$$i_t = \sigma\left(W_i \cdot [\boldsymbol{C_{t-1}}, h_{t-1}, x_t] \; + \; b_i\right)$$
$$o_t = \sigma\left(W_o \cdot [\boldsymbol{C_t}, h_{t-1}, x_t] \; + \; b_o\right)$$

Figura 3.3: Representación en [11] de celdas con peephole conections

Gated Recurrent Unit o GRU: Se propone en [14] suna versión simplificada de *LTSM* en la que los dos vectores en los que guardábamos el estado de cada celda se unifican en uno solo. Además, las *forget gate* e *input gate* son combinadas para formar la denominada *update gate*.



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

Figura 3.4: Representación en [11] de GRU

3.6. Bidirectional RNN o BRNN

Aún cabe la posibilidad de darle otra vuelta de tuerca a las RNNs. Hasta el momento, nos hemos centrado en que la información inicial de nuestra secuencia se propagara hasta las últimas iteraciones de nuestra red neuronal para conseguir mayor influencia. Si nos ponemos en el caso de un texto, es obvio que las palabras iniciales tendrán influencia en palabras futuras. Por ejemplo, a la hora de referirnos a una persona a la que vamos describiendo, mantener el género en el significado de la frase o saber si estamos hablando

en singular o plural. Pero, ¿y si consiguiéramos saber qué hay escrito en la posición $t + k$ de la secuencia de texto para aprender correctamente el significado de lo que hay escrito en la posición t? Al fin y al cabo es recurso que usa nuestro cerebro cuando procesamos el lenguaje.

Esta idea fue propuesta en [15] y se basa en introducir *backwards recurrent layers* o *capas recurrente hacia atrás* que tienen la misma estructura que hemos estado estudiando pero la información se propaga desde la iteración T hasta $t = 1$.

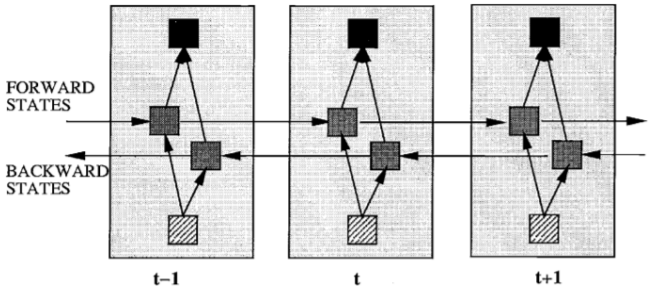


Fig. 3. General structure of the bidirectional recurrent neural network (BRNN) shown unfolded in time for three time steps.

Figura 3.5: Representación de una BRNN en [15]

Incluso podemos dar un paso más y podemos hacer que tanto los flujos de propagación hacia delante y hacia atrás tengan estructura de tipo $LSTM$. Será muy útil cuando estudiemos nuestro problema de análisis de sentimiento.

3.7. Deep Recurrent Neural Networks

Ya hemos estudiado los fundamentos de las RNN y diferentes versiones que nos pueden ayudar a solventar diferentes problemas. Aún así, en algunas situaciones es necesario añadir más capas de neuronas para obtener un mejor resultado.

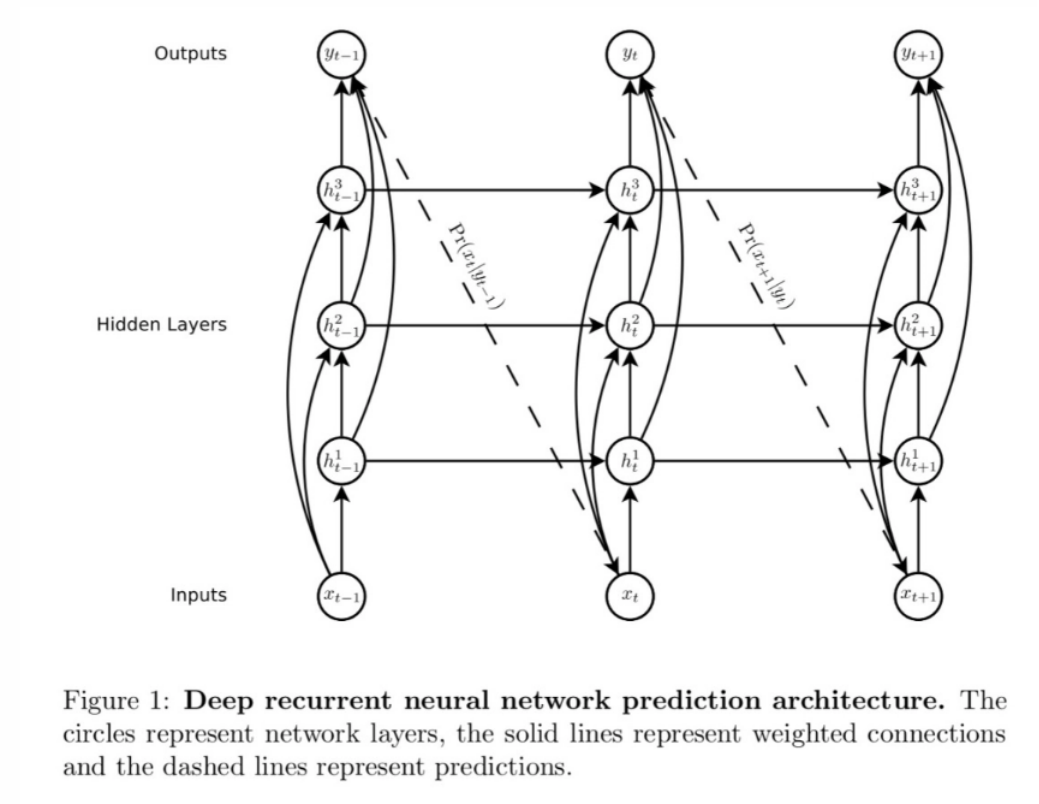


Figure 1: **Deep recurrent neural network prediction architecture.** The circles represent network layers, the solid lines represent weighted connections and the dashed lines represent predictions.

Figura 3.6: https://blog.acolyer.org/2017/03/23/recurrent-neural-network-models/

Esta sería la estructura que se obtendría al añadir más capas en cada time step. De hecho, si añadimos una celda del tipo *LSTM* nos quedaría una estructura similar a la siguiente

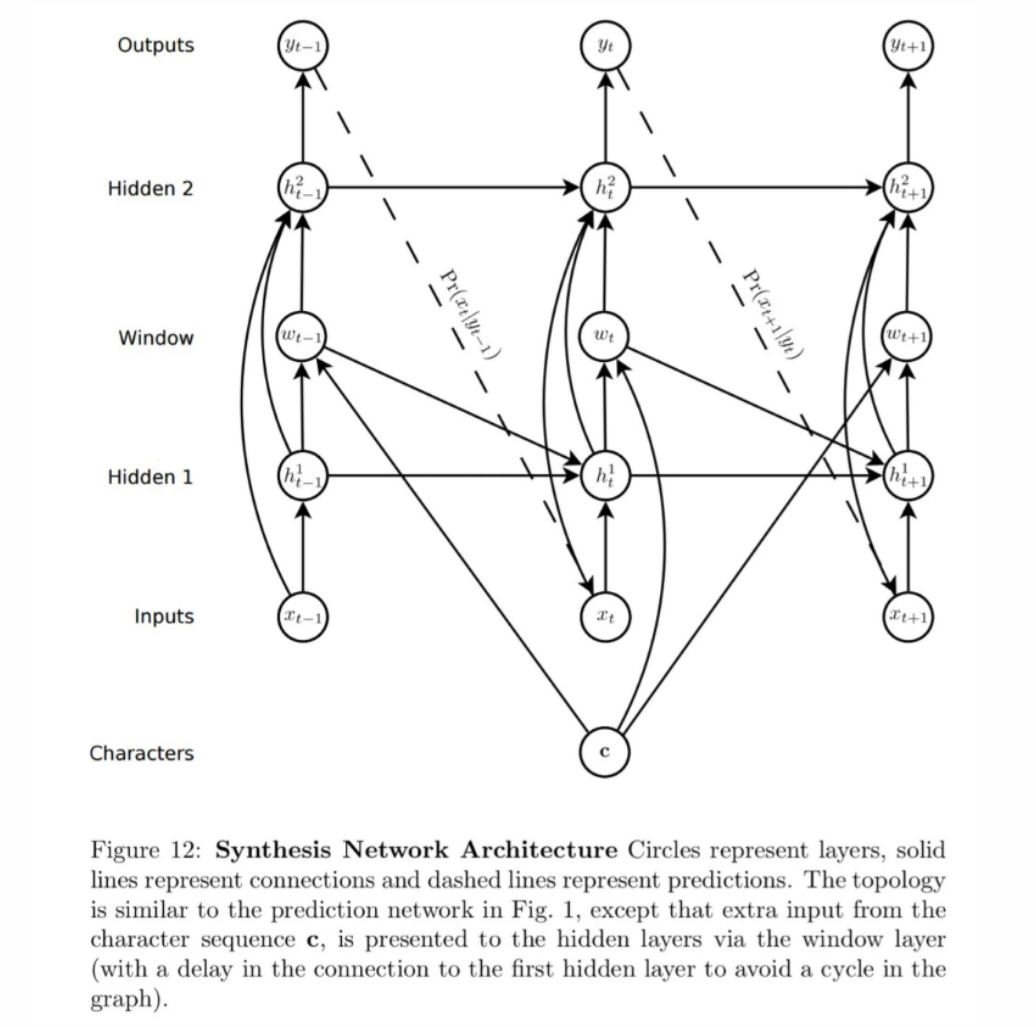


Figure 12: **Synthesis Network Architecture** Circles represent layers, solid lines represent connections and dashed lines represent predictions. The topology is similar to the prediction network in Fig. 1, except that extra input from the character sequence **c**, is presented to the hidden layers via the window layer (with a delay in the connection to the first hidden layer to avoid a cycle in the graph).

Figura 3.7: https://blog.acolyer.org/2017/03/23/recurrent-neural-network-models/

Capítulo 4

Interpretación de texto

Hemos visto algunas ideas clave sobre cómo afrontar problemas para el Procesamiento de Lenguaje Natural mediante Recurrent Neural Networks. Pero aún tenemos que conseguir que los algoritmos que hemos definido sean capaces de "interpretar" el texto que usamos como entrada y establecer ciertas analogías o relaciones entre palabras similares.

4.1. One Hot Vector

Como sabemos, el lenguaje humano está representado por un conjunto de palabras al que denominamos *vocabulario*. El de cada idioma será distinto y cumplirá ciertas propiedades. En nuestro caso, sería interesante crear un vocabulario a partir de todas las palabras que se usan en todas las instancias de las que disponemos. De hecho, podemos incluso limitar el tamaño de nuestro vocabulario a un conjunto menor de un número fijo w de palabras. Al no tener en cuenta todas las palabras de las que disponemos, algunas de ellas tendrán la etiqueta de palabra desconocida.

Una vez definido nuestro vocabulario, podremos representar la palabra que ocupa la posición p en nuestro vocabulario como un vector de ceros de tamaño w y 1 en la posición p de nuestro vector. De esta manera, si la palabra "rojo" ocupa la posición 3451 en nuestro vocabulario, tendríamos que el vector que representa a esa palabra tendría la forma

$$v_{rojo} = [0, 0, \ldots, \underset{pos.\,3451}{1}, \ldots, 0, 0]$$

Esta representación llamada *One Hot Vector* es una primera aproximación bastante simple al problema de vectorización de palabras. Pero al ser tan trivial presenta varios problemas, ya que todas las palabras están a la misma distancia en el espacio $\mathbb{R}^w$ y esto impide que no podamos interpretar significados o funciones sintácticas debido a que la representación será igual en todas las palabras. Por ejemplo, en el caso de palabras que actúen como sinónimos, sería interesante que la representación fuera más parecida entre ellas que en el caso de trabajar con antónimos, que buscaríamos una representación que tuviese poco en común.

4.2. Word Embedding

Si fuésemos capaces de obtener una representación similar entre palabras que están relacionadas sería mucho más sencillo para nuestra red neuronal generalizar lo que ha aprendido sobre una palabra para aplicarlo a otras similares.

La solución más común a este problema es representar cada palabra en nuestro vocabulario usando un vector de cierta dimensión n. Es decir, tendríamos una función de la siguiente forma

$$W : vocabulario \rightarrow \mathbb{R}^n \tag{4.1}$$

Esta función se denomina *word embedding*. La forma resultante de aplicar W a nuestro vocabulario, se podría iterpretar como el valor que tiene una palabra para cada una de las n características que componen cada vector.

En la imagen 4.1 observamos la representación mediante un *embedding* de algunas palabras que pueden tener cierta relación entre ellas. Gracias a esta técnica observamos valores muy cercanos en características similares.

Conseguir *embeddings* que sean óptimos es una tarea complicada y se suelen usar técnicas de aprendizaje automático, como veremos más adelante. Una vez que ya lo hemos obtenido, sería interesante poder visualizarlos para

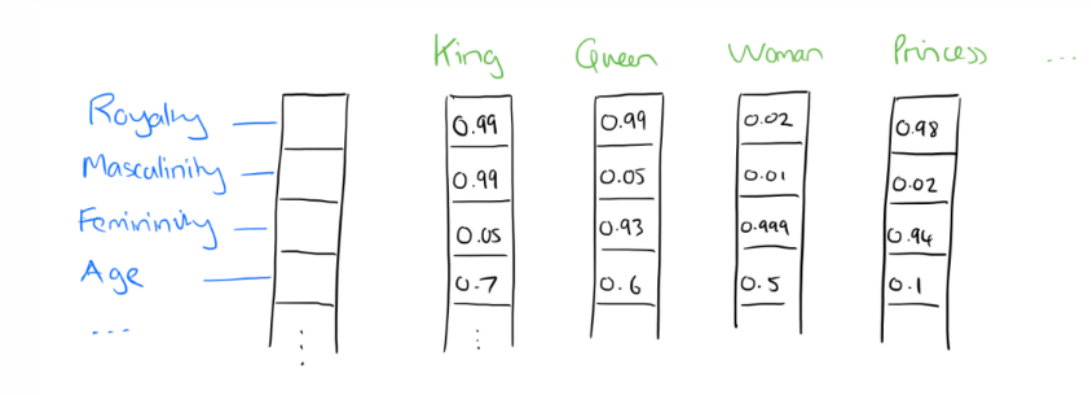


Figura 4.1: https://blog.acolyer.org/2016/04/21/the-amazing-power-of-word-vectors/

obtener un feedback de del resultado final para observar las relaciones entre palabras. Al tener vectores de una dimensión elevada, es necesario usar técnicas avanzadas de representación. Una de las más conocidas es *t-Distributed Stochastic Neighbor Embedding* o *t-SNE*, desarrollada en [16], y que se puede aplicar perfectamete en word embeddings como vemos en 4.2

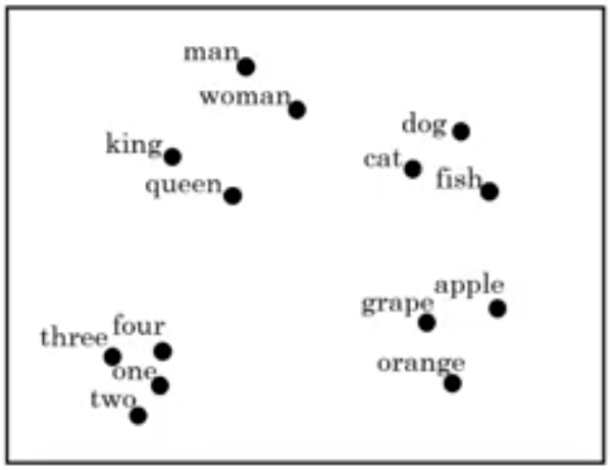


Figura 4.2: Ejemplo de t-SNE aplicado a un word embedding. [2]

Una representación de este tipo es muy intuitiva, ya que palabras que están relacionadas aparecen juntas. Otra visualización que se puede usar es una tabla en la que se exponen las palabras más cercanas a una palabra dada

usando nuestro *embedding* como medida:

FRANCE	JESUS	XBOX	REDDISH	SCRATCHED	MEGABITS
454	1973	6909	11724	29869	87025
AUSTRIA	GOD	AMIGA	GREENISH	NAILED	OCTETS
BELGIUM	SATI	PLAYSTATION	BLUISH	SMASHED	MB/S
GERMANY	CHRIST	MSX	PINKISH	PUNCHED	BIT/S
ITALY	SATAN	IPOD	PURPLISH	POPPED	BAUD
GREECE	KALI	SEGA	BROWNISH	CRIMPED	CARATS
SWEDEN	INDRA	PSNUMBER	GREYISH	SCRAPED	KBIT/S
NORWAY	VISHNU	HD	GRAYISH	SCREWED	MEGAHERTZ
EUROPE	ANANDA	DREAMCAST	WHITISH	SECTIONED	MEGAPIXELS
HUNGARY	PARVATI	GEFORCE	SILVERY	SLASHED	GBIT/S
SWITZERLAND	GRACE	CAPCOM	YELLOWISH	RIPPED	AMPERES

Figura 4.3: Palabras relacionas en un embedding. [17]

Una propiedad muy interesante de los *word embeddings* estudiada en [18] es que se encuentran relaciones o analogías entre las diferencias de varias palabras, como el significado sintáctico, morfológico o semántico.

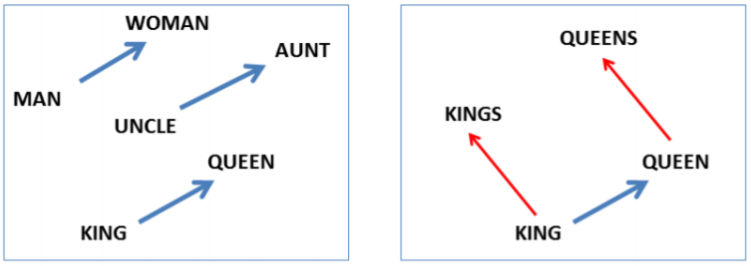


Figure 2: Left panel shows vector offsets for three word pairs illustrating the gender relation. Right panel shows a different projection, and the singular/plural relation for two words. In high-dimensional space, multiple relations can be embedded for a single word.

Figura 4.4: Ejemplo de Word embedding estudiado en [18]

En la imagen 4.4 observamos la representación de vectores obtenida es capaz aprender relaciones como plural/singular y masculino/femenino. Es decir, que usando la función W que definimos en 4.1, se cumplen las siguientes ecuaciones

$$W(kings) - W(king) \simeq W(queens) - W(queen)$$

$$W(woman) - W(man) \simeq W(queen) - W(king)$$

$$W(aunt) - W(uncle) \simeq W(queen) - W(king)$$

4.3. RNNs y Word Embeddings

A la hora de encontrar una representación óptima de nuestro vocabulario, un *word embedding*, es común usar técnicas de deep learning.

En nuestro problema sobre Análisis de Sentimiento, tendremos que interpretar si un comentario indica una valoración positiva o una valoración negativa. Podríamos construir una red neuronal de la siguiente forma

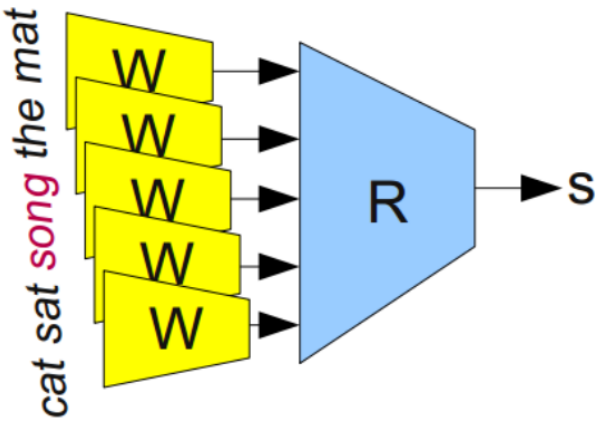


Figura 4.5: Fuente: [12]

donde nuestro W es iniciado con valores aleatorios y es usado para "traducir" el texto a vectores para procesarlo en el módulo R con el objetivo de obtener un valor S que nos indique si

Bibliografía

[1] Buduma, N., 2017. *Fundamentals of Deep Learning: Designing next-generation machine intelligence algorithms*

[2] NG, Andrew *Course on Neural Networks and Deep Learning at Coursera* https://www.coursera.org/learn/neural-networks-deep-learning

[3] Géron, Aurélien, 2017. *Hands-On Machine Learning with Scikit-Learn and TensorFlow*

[4] Barbero, Álvaro; Suárez, Alberto, 2018. *Advanced Statistics and Data Mining Summer School*

[5] Rumelhart, Hinton, & Williams, 1985. *Learning Internal Representations by Error Propagation*

[6] Karpathy, Andrej, 2015. *The Unreasonable Effectiveness of Recurrent Neural Networks* *http://karpathy.github.io/2015/05/21/rnn-effectiveness/*

[7] Kilian, Joe, 1996. *The Dynamic Universality of Sigmoidal Neural Networks*

[8] Weber, Noah, 2015. *Why LSTMs stop your gradients from vanishing: A view from the Backwards Pass* *https://weberna.github.io/blog/2017/11/15/LSTM-Vanishing-Gradients.html*

[9] Pascanu,Razvan; Mikolov,Tomas; Bengio, Yoshua, 2013. *On the difficulty of training Recurrent Neural Networks*

[10] Hochreiter, Sepp; Schmidhuber, Jürgen, 1997. *Long Short-Term Memory*

[11] Olah, Christopher, 2015. *Understanding LSTM Networks* *https://colah.github.io/posts/2015-08-Understanding-LSTMs/*

[12] Olah, Christopher, 2014. *Deep Learning, NLP, and Representations* *http://colah.github.io/posts/2014-07-NLP-RNNs-Representations/*

[13] Gers, Felix; Schmidhuber, Jürgen, 2000. *Recurrent nets that time and count*

[14] Cho, Kyunghyun et al., 2014. *Learning Phrase Representations using RNN Encoder–Decoder for Statistical Machine Translation*

[15] Schuster, Mike; Paliwal, Kuldip, 1997. *Bidirectional Recurrent Neural Networks*

[16] van der Maaten, Laurens; Hinton, Geoffrey, 2008. *Visualizing Data using t-SNE*

[17] Collobert, Ronan; Weston, Jason et al., 2011. *Natural Language Processing (almost) from Scratch*

[18] Mikolov, Tomas et al., 2013. *Linguistic Regularities in Continuous Space Word Representations*