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

Abstract

En este trabajo se abordará un problema de Análisis de Sentimiento, donde nuestra meta será intentar clasificar una review como negativa o positiva. Para ello nos ayudaremos de un tipo de modelo denominado Red Neuronal, perteneciente a la familia de técnicas de *Machine Learning*. En particular, estudiaremos un tipo especial de red neuronal que tiene una estructura recurrente. Comentaremos qué ventajas nos aportan estos modelos para problemas de análisis de texto e ilustraremos matemáticamente cómo funcionan. Finalmente, para poder plasmar los resultados en un ejemplo real, usaremos un conjunto de reviews procedentes de la página IMDb. Con ayuda de la librerías Keras y Tensorflow, construiremos en Python diferentes Recurrent Neural Networks llegando a alcanzar un nivel de 88 % de accuracy.

Índice general

Capítulo 1

Introducción

La era de la información ha hecho que la manera de comunicarnos cambie radicalmente, desde el uso del telégrafo hasta aplicaciones de mensajería instantánea y redes sociales que son tan populares actualmente. De hecho, ya no nos limitamos a comunicarnos solo entre nosotros. Las comunicaciones entre personas y máquinas es cada vez más común, como es el caso de los *chatbots*. Es por ello que hoy en día se generan grandes cantidades de información ligada al lenguaje y, en consecuencia, es necesario desarrollar formas de poder procesarla para diferentes tareas.

Uno de los problemas con bastante relevancia actualmente en el contexto del procesamiento de lenguaje natural es el Análisis de Sentimiento, donde se busca clasificar un documento (un email, un comentario, un tweet, etc) en función de la connotación que haya querido mostrar el usuario que lo ha compartido. En este trabajo se usarán redes neuronales artificiales en un problema de análisis de sentimiento sobre comentarios de películas, donde se usarán diferentes estructuras para estudiar qué modelo es más eficaz clasificando si una opinión tiene una connotación positiva o negativa.

En el *Capítulo 2* se hará una introducción a las redes neuronales artificiales. Profundizaremos en la estructura de uno de los tipos de red neuronal que existen, denominado Recurrent Neural Network, y explicaremos por qué estos

modelos son tan útiles problemas relacionados con el lenguaje. En el *Capítulo 3* se expondrán las características de los datos del problema y qué librerías usaremos para el desarrollo de los modelos. La descripción de la estructura de los los modelos que vamos a usar y los pasos a seguir en su implementación se expondrán en el *Capítulo 4*, mientras que en el *Capítulo 5* se presentarán los resultados conseguidos con los diferentes modelos en los datos de prueba, usando *accuracy* como medida. Para finalizar, en el *Capítulo 6* indicaremos qué tipo de estructura ha sido más eficaz (BRNN en este caso) y añadiremos unas posibles direcciones futuras sobre las que poder seguir trabajando.

Capítulo 2

Análisis de sentimiento mediante redes neuronales

Las redes neuronales artificiales son muy utilizadas hoy en día en diferentes ámbitos, desde el reconocimiento de patrones a partir de imágenes o vídeos hasta predicciones en series temporales en mercados financieros. Se inspiran en la estructura del cerebro y en algunos casos se ha superado la efectividad que pueden tener las personas en una actividad concreta, como en la detección de cáncer hace poco en una competición en China [25].

2.1. Introducción a las Redes Neuronales Artificiales

Las redes neuronales artificiales son una familia de algoritmos 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.

Por ejemplo, imaginemos que queremos desarrollar un filtro de spam para la bandeja de entrada de nuestro correo electrónico. Si usamos programación tradicional, tendríamos que pensar en todas las posibles palabras, frases o indicadores que, a nosotros como usuarios, nos llevarían a pensar que ese email es spam. Después, escribir en el código cada una de esas reglas que se nos

han ocurrido para que la máquina pueda clasificar el email. Es muy probable que no tengamos en cuenta todos los indicadores posibles y además, que estos indicadores vayan cambiando con el tiempo, por lo que habría que estar actualizando las reglas continuamente y no conseguiríamos un filtro de spam eficiente.

En cambio, un algoritmo basado en *machine learning* aprendería automáticamente qué indicadores son los mejores para clasificar un correo como spam mediante el análisis de ejemplos que nosotros hemos considerado spam. Con ello, nos ahorraríamos tener que escribir cada una de las reglas y se iría actualizando en caso de que los indicadores fueran modificándose con el paso del tiempo.

Existen dos grandes ramas en machine learning: *aprendizaje supervisado (o supervised learning)* y *aprendizaje no supervisado (o unsupervised learning).* En el primer tipo, los datos que se usan para entrenar un modelo incluyen la solución que buscamos resolver con el algoritmo. En cambio, en el aprendizaje no supervisado no tenemos esta solución y por tanto el modelo debe aprender sin poder compararse con nada. Usando un conjunto de datos relativamente grande y otorgando al algoritmo una medida para evaluar su predicción, este irá modificando su estructura para obtener un mejor resultado para que después de muchas correcciones, haya aprendido a resolver el problema con bastante efectividad. Quizás una definición más formal 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

2.1.1. 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 potentes 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.

El funcionamiento de las neuronas explicado a muy alto nivel sería algo así: 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".

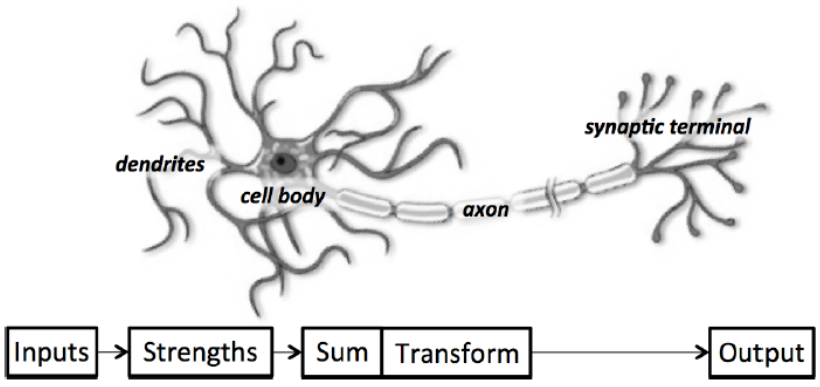


Figura 2.1: Descripción funcional de la estructura de una neurona. [1]

Podemos usar el lenguaje matemático para crear un modelo que intenta representar el funcionamiento de una neurona. Para ello nos ayudaremos del esquema que aparece en la imagen 2.1. 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 $y = f(z)$.

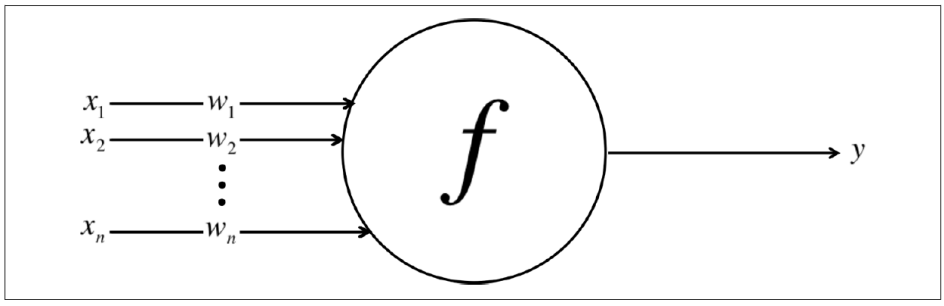


Figura 2.2: Representación matemática de una neurona. [1]

Para simplificar la notación consideramos todas las entradas como el vector $x = [x_1, x_2, \ldots x_n]$ y construimos nuestro vector de pesos como $w = [w_1, w_2, \ldots, w_n]$. 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 $x = [x_0, x_1, x_2, \ldots x_n]$ y $w = [w_0, w_1, w_2, \ldots, w_n]$.

Existen varias alternativas a la hora de elegir una función de activación, que es la 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.

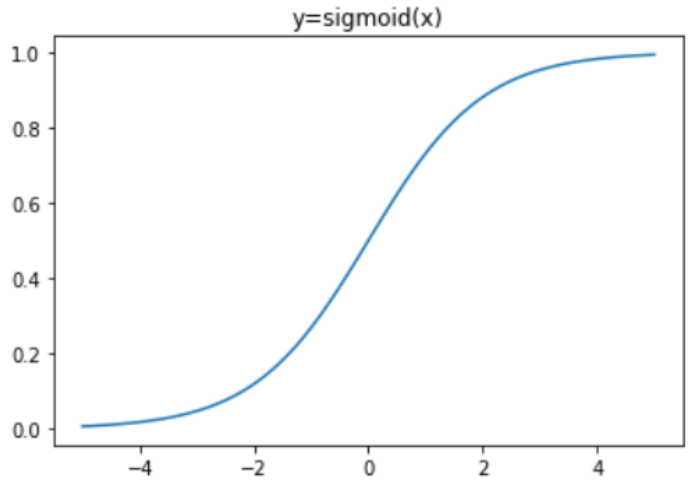


Figura 2.3: Función sigmoide

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.

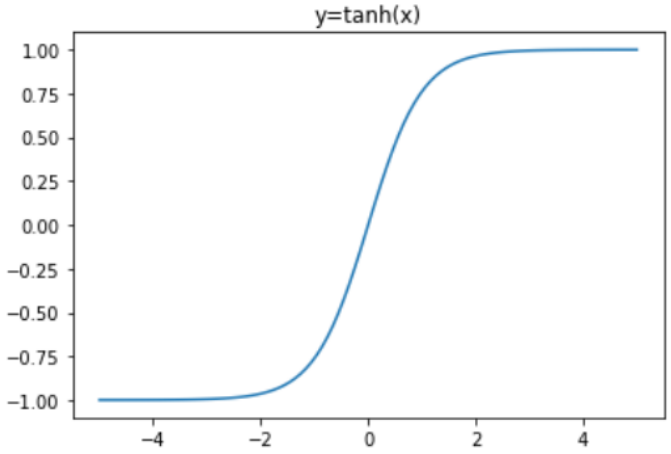


Figura 2.4: Función tangente hiperbólica

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 Restricted Linear Unit o ReLU. Usa la función $f(z) = max(0, z)$. Es muy útil en casos dónde se busca rebajar el coste computacional, pero tiene algunas limitaciones, por ejemplo, no es derivable en $f(z) = 0$ y $f'(z) = 0, \forall f(z) < 0$.

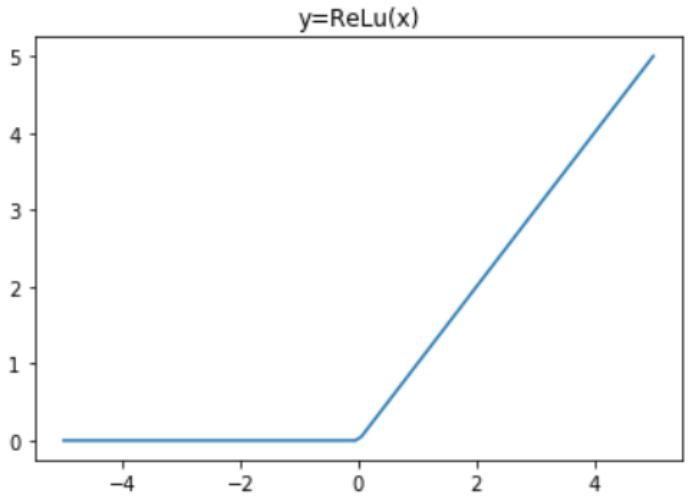


Figura 2.5: Función ReLU

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 $0,01$.

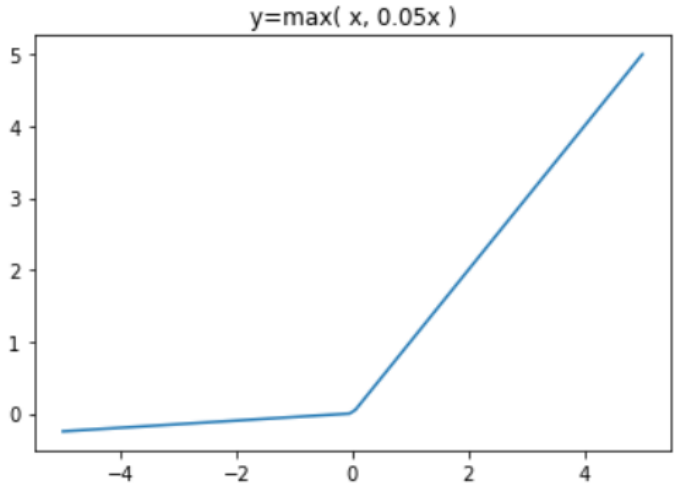


Figura 2.6: Función ReLU modificada

2.1.2. 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, por lo que es interesante crear conexiones entre neuronas. En esta asociación 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.

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

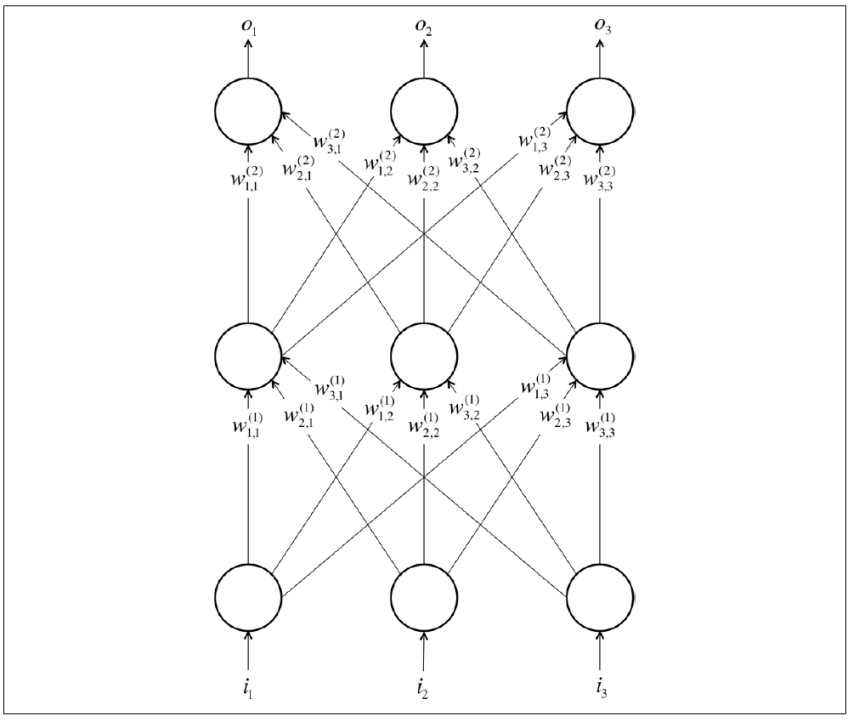


Figura 2.7: Red neuronal con varias capas. [1]

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*. Aquí ya podemos introducir el concepto de *Deep Learning*. Si decidimos añadir a una red neuronal 2 o más capas ocultas se dice que estamos usando algoritmos de aprendizaje profundo o Deep Learning. Como veremos más adelante, introducir más capas ocultas nos puede otorgar ciertas ventajas que una red neuronal simple no tiene.

Aplicando a nuestra red neuronal la definición formal de aprendizaje automático que dimos anteriormente en el capítulo 2.1, 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 $x = [x_0, x_1, x_2, \ldots x_n]$, al

que llamaremos "ejemplo" tiene asociado un valor $\hat{y}$ que es el que intentaremos adivinar o predecir con 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.

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 funciona de manera iterativa y es muy intuitivo, como vemos en la imagen 2.8. Se mide el valor de la función de error en un valor concreto, que consideramos un valor que pertenece al rango de sus parámetros. 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.

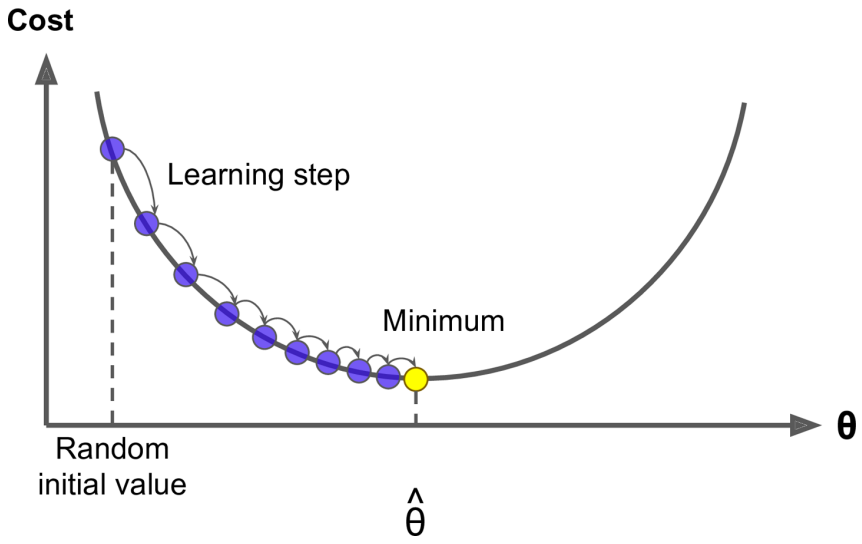


Figura 2.8: Representación del algoritmo Gradient Descent. [3]

Pasamos a reescribir nuestra medida del error 2.1 en función de los parámetros que queremos minimizar:

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

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, que nos asegurarán que en el punto w^* se alcanza un mínimo local:

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

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 todos los ejemplos de todo nuestro dataset. Esto en algunos casos puede ser bastante costoso de calcular computacionalmente. 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 elegido aleatoriamente en cada iteración. Otra variación de SDG muy conocida es en la que, para cada iteración, se usa un mini lote o mini batch distinto de ejemplos elegidos aleatoriamente.

Ya hemos visto cómo ir modificando los parámetros de nuestra red neuronal 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. Esto se debe a que cada una de las capas influirá en el valor en la medida de error que hayamos definido, por lo que tenemos que ser capaces de

calcular cuánto afectan en el valor de salida de una neurona las variaciones en capas anteriores. De esta manera podremos ir modificando los pesos acorde a nuestros objetivos.

Si echamos un vistazo a [5], podremos ver cómo se define la técnica conocida como *Back Propagation*, que es un algoritmo muy útil para calcular los gradientes necesarios para Gradient Descent en redes neuronales con múltiples capas. Para poder realizarlo, hay que seguir una "cadena hacia atrás" de derivadas de la siguiente manera: Primero se calcula el error que ha generado un input y se calcula cuánto ha influido en ese resultado cada una de las neuronas de la capa que precede (vamos a llamarla capa 1 para esta explicación) a la capa de salida. Después se mide la influencia que ha tenido la capa anterior a la capa 1 para ese mismo error. Si seguimos así sucesivamente, alcanzaremos la capa de entrada, por lo que habremos medido el gradiente de error entre todas las neuronas y pesos de la red neuronal. Realizaremos los cálculos con más detalle más adelante.

2.2. Recurrent Neural Networks

Ya hemos definido la estructura de una red neuronal básica, en la que la información se transmite desde la capa de entrada hasta la capa de salida. Existen varios tipos y variaciones de redes neuronales que buscan ajustarse al tipo de problema que queremos resolver. Por ejemplo, existen muchas situaciones en las que los datos que tenemos no tienen una dimensión fija, al contrario que en casos como el de imágenes. Muchas veces tendremos que tratar con datos de tipo secuencial, es decir, que el orden en el que se presentan las partes de la secuencia influyen en su significado o funcionamiento y además podrán tener longitudes variables. En casos como texto, música o series temporales, el valor o estado de cada posición de la secuencia va variando en función del tiempo y es posible que cada posición tendrá dependencia de posiciones anteriores. Para tratar datos de tipo secuencial es muy común usar redes neuronales con estructura recurrente o Recurrent Neural Networks (RNN) debido a su estructura tan particular.

En 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. En cambio, en RNNs, 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. Como observamos en el modelo de la imagen 2.9, la representación recursiva se puede desplegar en una estructura feed forward en la que cada uno de los inputs se introduce en la red en un tiempo diferente.

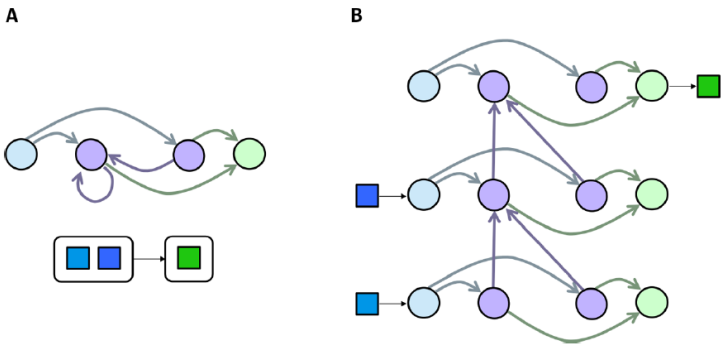


Figura 2.9: Representación de RNN (A) como modelo feed forward (B) respecto al tiempo. Fuente: [1]

En este ejemplo, tenemos dos inputs y una única salida, pero la estructura de una RNN puede cambiar para adpatarse al problema que queramos tratar, como vemos en la imagen 2.10.

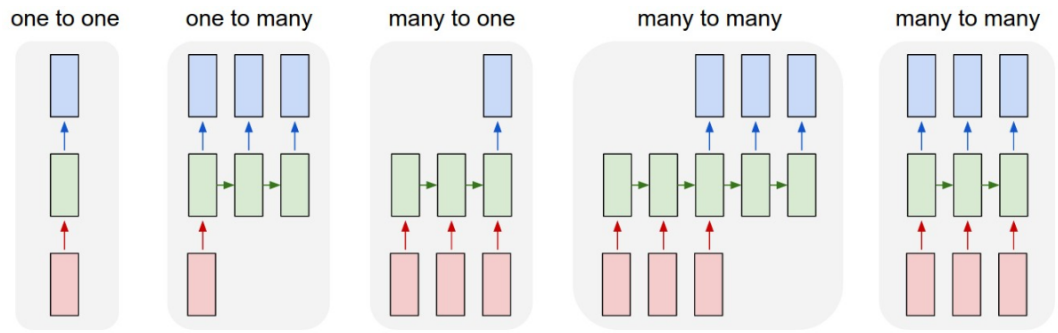
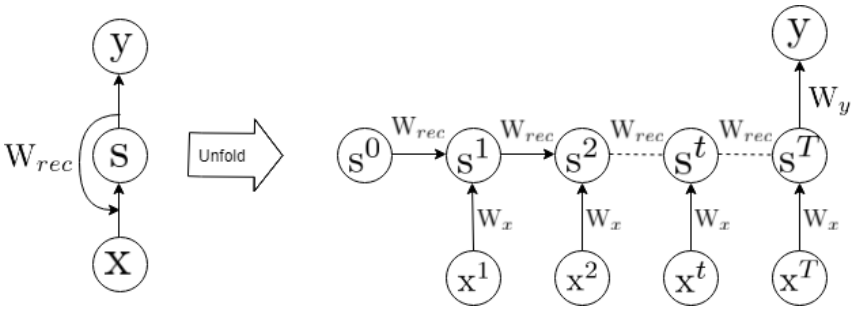


Figura 2.10: Diferentes tipos de RNN. [6]

Para el tipo de problema que trataremos más adelante, vamos a estudiar en mayor profundidad una red neuronal del tipo *Many to One*.



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:

$$S^t = f\left(W_{rec} * S^{t-1} + W_x * X^t\right) \tag{2.4}$$

donde f es una función de activación. Observamos que para el valor del estado de layer en un tiempo 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.

2.2.1. Algoritmo BPTT en RNN

Para poder ir modificando los pesos en una RNN, necesitaremos usar una versión modificada del algoritmo de Back Propagation, en la que deberá tener en cuenta las dependencias respecto al tiempo de los diferentes estados de las neuronas. Esta modificación del algoritmo se denomina *Back Propagation 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:

$$\delta_{z_k} = \frac{\partial E(y)}{\partial z_k} \tag{2.5}$$

$$\delta_{a_h^t} = \frac{\partial E(y)}{\partial a_h^t} \tag{2.6}$$

donde 2.5 es la derivada del error respecto a la entrada de la capa de salida en el tiempo T y 2.6 la derivada del error respecto a la entrada recursiva en un tiempo fijo t.

Hemos definido en 2.2.1 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 2.5 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 2.6

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

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

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

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

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

$$\frac{\partial E(y)}{\partial W_{y_{kh}}} = \frac{\partial E(y_k)}{\partial z_k}\frac{\partial z_k}{\partial W_{y_{kh}}} = \delta_{z_k} s_h^t \tag{2.7}$$

$$\frac{\partial E(y)}{\partial W_{x_{ih}}} = \sum_{t=1}^{T}\frac{\partial E(y_k)}{\partial a_k^t}\frac{\partial a_k^t}{\partial W_{x_{ih}}} = \sum_{t=1}^{T}\delta_{a_k^t} x_i^t \tag{2.8}$$

$$\frac{\partial E(y)}{\partial W_{rec_{hg}}} = \sum_{t=1}^{T}\frac{\partial E(y_k)}{\partial a_h^t}\frac{\partial a_h^t}{\partial W_{rec_{hg}}} = \sum_{t=1}^{T}\delta_{a_k^t} s_g^{t-1} \tag{2.9}$$

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.

Podemos reutilizar estos cálculos para el caso en el que la salida de nuestra red neuronal sea una secuencia. Si usamos una estructura del tipo *Many to Many* obtendremos un y^t en cada time step. Pero como ya vimos anteriormente en 2.10 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:

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

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

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

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*.

Si reescribimos el desarrollo del gradiente 2.9, podemos ver que 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 2.4 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 [10], 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*.

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 [11], dónde se comprobaba que el problema con la inestabilidad de los gradientes se puede evitar mediante 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.

2.2.2. RNN de tipo Long Short-Term Memory

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 celda 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:

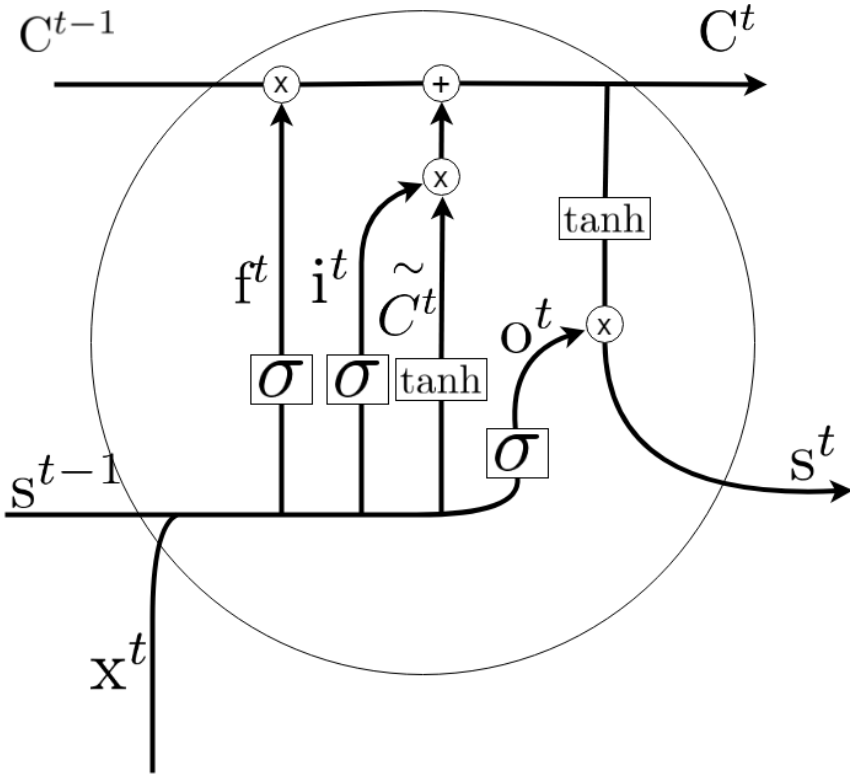


Figura 2.11: Representación de celda LSTM

La aportación que hicieron Hochreiter y Schmidhuber es mantener el estado C^t de la celda, 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 celda 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:

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

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

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

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

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

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

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 celda C^t.

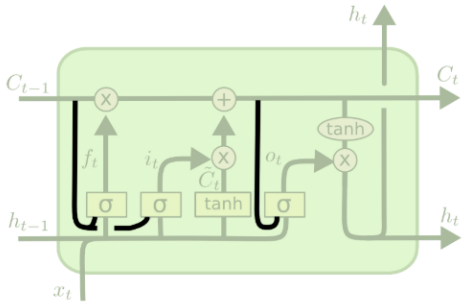
Cada una de las ecuaciones y gates que hemos definido tienen su significado. Siguiendo el camino de las flechas de la imagen, vemos que la primera ecuación que usamos es la denominada *forget gate*, representada por f^t, donde se elige la información del estado anterior que no se va a usar. El siguiente paso 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.

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:

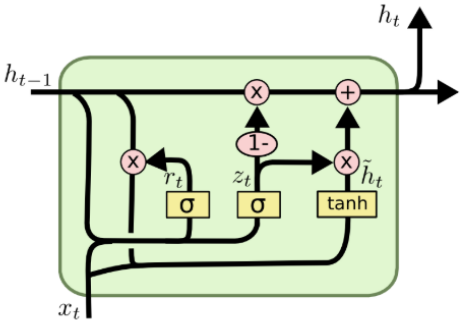
Peephole conections: En el ejemplo de *LSTM* original, cada una de las *gates* que definimos solo usan la información del input x^t y estado de la celda anterior s^{t-1}. Sin embargo, en [14] se observa que puede ser interesante tener en cuenta la información de c^{t-1}



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

Figura 2.12: Representación en [12] de celdas con peephole conections

Gated Recurrent Unit o GRU: Se propone en [15] 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 2.13: Representación en [12] de GRU

Otras estructuras de RNN

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 [16] 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 $t = 1$.

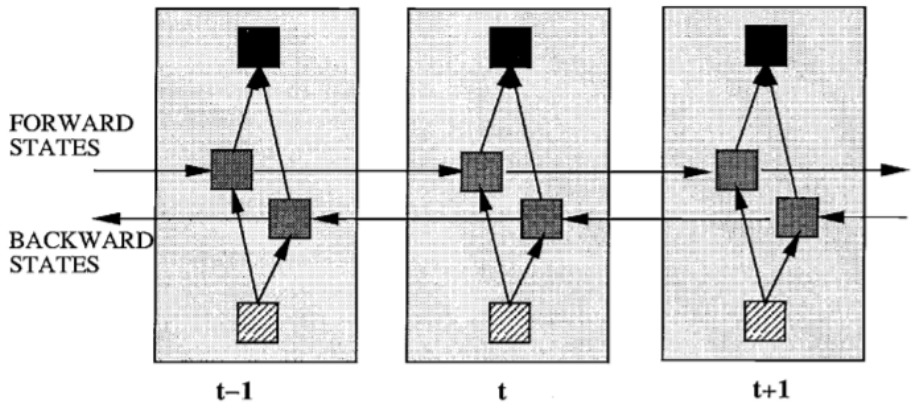


Fig. 3. General structure of the bidirectional recurrent neural network (BRNN) shown unfolded in time for three time steps.

Figura 2.14: Representación de una BRNN. [16]

Además, se puede conseguir que tanto los flujos de propagación hacia delante y hacia atrás tengan estructura de tipo $LSTM$, lo que hará a las redes

neuronales mucho más efectivas.

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. ¿Pero cómo se conectarían las distintas capas de una red neuronal recurrente? En las imagen 2.15 observamos cómo sería el esquema de un modelo tipo Deep Learning con estructura RNN simple. En cambio, en la imagen 2.16 se representa una RNN pero con una estructura LSTM.

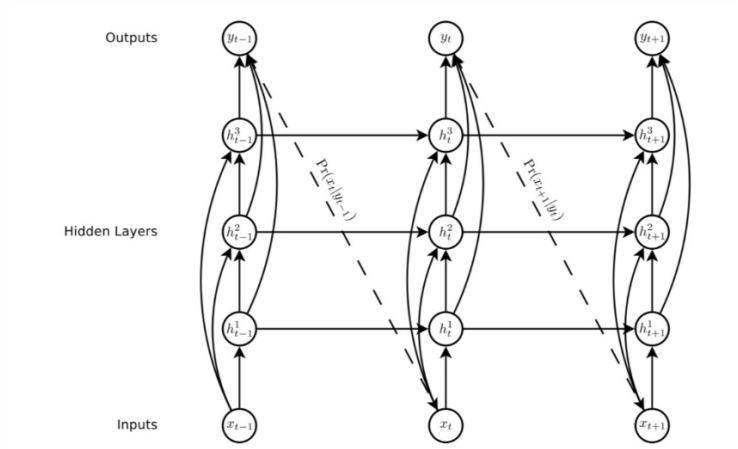


Figure 1: **Deep recurrent neural network prediction architecture.** The circles represent network layers, the solid lines represent weighted connections and the dashed lines represent predictions.

Figura 2.15: RNN con más de una capa oculta. Fuente [24]

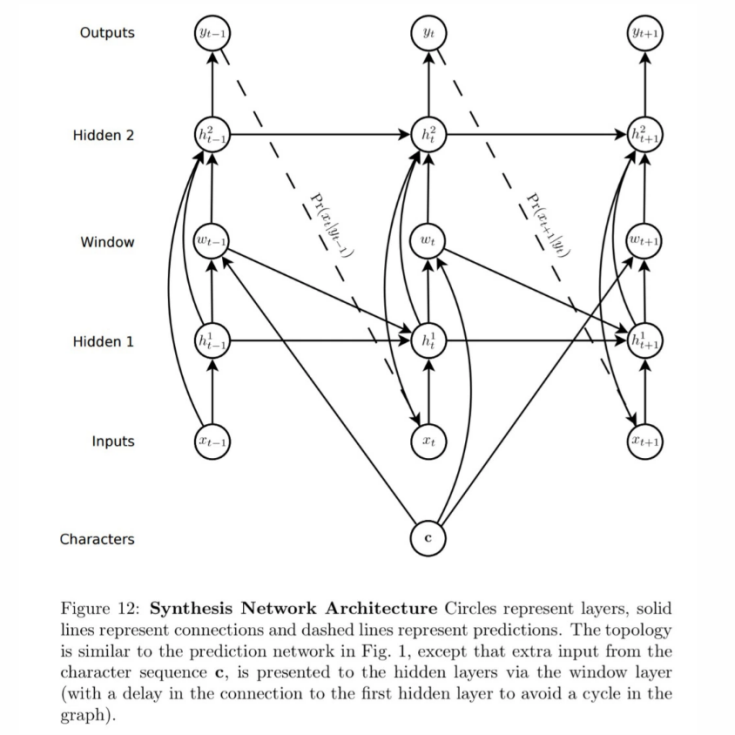


Figure 12: **Synthesis Network Architecture** Circles represent layers, solid lines represent connections and dashed lines represent predictions. The topology is similar to the prediction network in Fig. 1, except that extra input from the character sequence **c**, is presented to the hidden layers via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

Figura 2.16: Varias capas ocultas con estructura LSTM. Fuente [24]

2.3. Análisis de sentimiento

Hoy en día es muy común que los usuarios usen Internet para comprar productos online, hacer una reserva en un hotel o reservar mesa en un restaurante. Todo ello, sin haber tenido una experiencia previa en ese hotel e incluso no conocer a nadie de nuestro entorno que haya probado ese restaurante. Pero, ¿qué hace que decidamos entre las múltiples opciones a las que optamos? Muchas de esas veces en las que tenemos varias opciones para elegir, pero no tenemos acceso a más información para poder comparar por nuestra cuenta, nos dejamos guiar por los comentarios o *reviews*. Estos son publicados por personas que no conocemos, pero que ya han tenido una experiencia con algún producto o servicio concreto y comparten su opinión en Internet. Normalmente, se suele compartir un comentario explicando una opinión mediante el lenguaje y se suele acompañar esa opinión con una valoración. Existen diferentes maneras de

representar esa nota. Es muy común usar una puntuación representada por 5 estrellas, donde 1 estrella significa muy mala puntuación y 5 estrellas significan muy buena puntuación. Esta manera de puntuar puede variar, ya que en algunos casos se podrá valorar fracciones de estrella, y otras veces solo se podrán elegir estrellas enteras, lo que hará que una review pueda clasificarse solo de 5 maneras diferentes (1,2,3,4 o 5 estrellas) o 5 clases diferentes, si hablamos en términos de machine learning.

Otra manera de valorar muy extendida, que está relacionada con la forma de interactuar en redes sociales, es indicar si algo "te gusta" o "no te gusta". En este caso, solo tendremos dos posibles clases. Es una manera de puntuar más simple, ya que con la puntuación de estrellas puede haber cierta ambigüedad con resultados cercanos a 3. De hecho, en el caso de tener un problema donde la puntuación puede tener 5 valores diferentes, se puede reducir a un problema de solo dos clases diferentes predefiniendo uno rango que indique "like" y otro rango de puntuaciones que indique "dislike".

Nuestro problema a resolver será de Análisis de Sentimiento, en el que tendremos que etiquetar una review con una puntuación solo usando la opinión escrita por el usuario. En nuestro caso, esta puntuación solo tendrá dos clases, positiva o negativa. Para ello, nos ayudaremos de redes neuronales artificiales, en particular, redes neuronales con estructura recurrente.

2.3.1. Herramientas para problemas de análisis de sentimiento

Existen diferentes métodos para abordar un problema de análisis de sentimiento. En particular, algunas técnicas de machine learning usadas para otros problemas relacionados con procesamiento de lenguaje natural, han conseguido buenos resultados a la hora de clasificar un comentarios.

En [26] podemos observar que se pueden resolver problemas de Análisis de Sentimiento usando diferentes configuraciones en *Support Vector Machines* (SVM). Otro de los métodos que nos pueden ayudar a resolver este problema

son clasificadores *naïve bayes*. Podemos encontrar una comparación de los dos métodos en [27]. Incluso se pueden abordar problemas más complejos, como en [28], donde se usan *árboles de decisión* para clasificar frases subjetivas.

Existen más métodos para dar solución a un problema de Análisis de Sentimiento, entre los que se encuentran las Redes Neuronales Recurrentes. Gracias a la estructura su estructura, este tipo de red es muy útil para problemas relacionados con procesamiento de lenguaje natural o procesamiento de texto.

2.4. Recurrent Neural Networks y procesamiento 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 de texto.

Hoy en día existen problemas de cierta relevancia de procesamiento de lenguaje natural que pueden ser resueltos por RNN. De hecho, cada uno de estas situaciones pueden ser abordadas si escogemos uno de los modelos que vimos en la imagen 2.10, donde se exponían diferentes estructuras de salida y entrada en una red neuronal recurrente. Algunos ejemplos de problemas de procesamiento de lenguaje natural que tienen mucha relevancia hoy en día son los siguientes:

- **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.
- **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.

- **Traducción automática:** Muy similar al problema de reconocimiento de voz, porque tanto la salida como la entrada son secuencias. Pero en este caso, como se busca traducir texto de un idioma a otro, es muy probable que la longitud de la secuencia de salida no sea igual que la de entrada

- **Reconocimiento de entidades nombradas:** Busca detectar ciertas categorías, como nombres propios de personas o ciudades. En este problema también usaríamos una estructura del tipo Many to Many, pero la salida en este caso podría ser un entero, con 0 indicando que no se ha reconocido ninguna categoría en esa palabra o conjunto de palabras y un número distinto de 0 indicando que sí se ha reconocido una categoría y qué tipo es.

Pero no podemos usar el texto de la forma en la que lo conocemos e introducirlo en nuestro modelo, deberemos hacer algún tipo de traducción que nos converta el texto a números, algo que una red neuronal sí es capaz de interpretar.

Como sabemos, el lenguaje humano está representado por un conjunto de palabras al que denominamos *vocabulario* o *diccionario*. 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.

Word Embeddings

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

$$W : vocabulario \rightarrow \mathbb{R}^n \tag{2.10}$$

Al resultado de aplicar W a un conjunto de palabras se le denomina *embedding* como veremos más adelante, su estructura se podría iterpretar como el valor que tiene una palabra para una de las n características. En la imagen 2.17 podemos observar como sería el resultado final del cálculo de un Word Embedding.

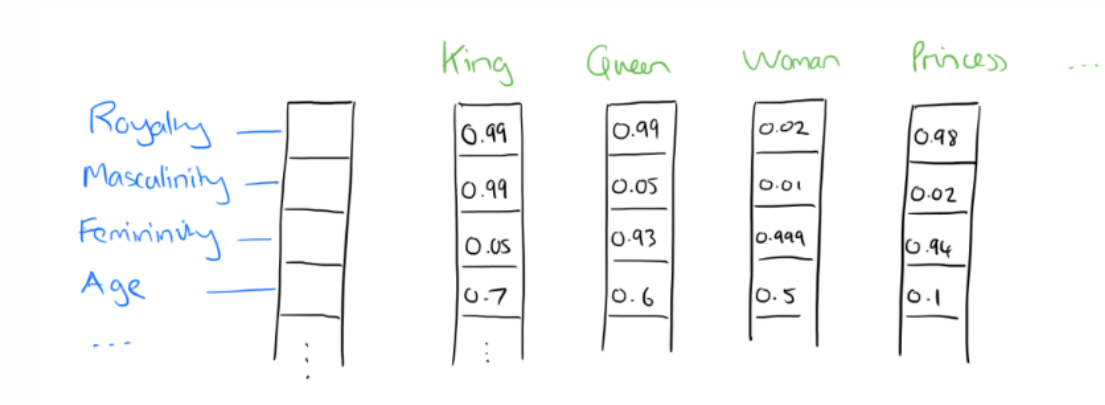


Figura 2.17: https://blog.acolyer.org/2016/04/21/the-amazing-power-of-word-vectors/

Una vez que hemos obtenido una representación en un espacio vectorial de dimensión n de un conjunto de palabras, sería muy interesante poder visualizarlas de alguna manera para poder obtener un feedback de la distribución entre ellas. Para ello existe una técnica propuesta en [17], denominada *t-Distributed Stochastic Neighbor Embedding* o *t-SNE*.

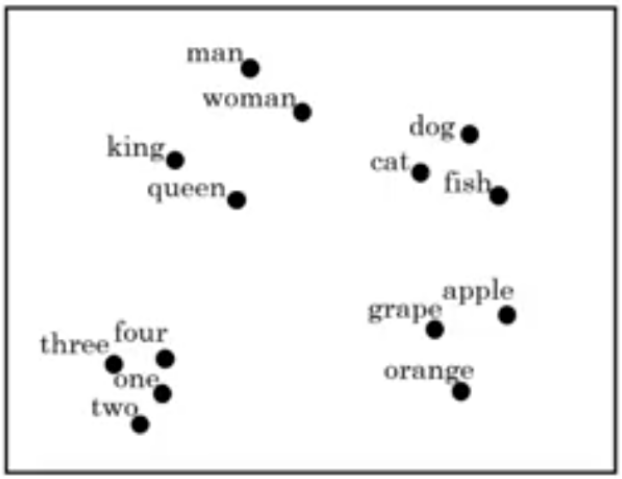


Figura 2.18: Ejemplo de visualización de un Word Embedding usando el algoritmo t-SNE.Fuente: [2]

Podemos observar en la imagen 2.18 como palabras similares presentan menor distancias entre ellas. Otra forma de representar un embedding es usando

una tabla, como la de la imagen 2.19 donde se indica qué palabras están más cerca una palabra dada.

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 2.19: Visualización de un embedding en una tabla. Fuente: [18]

Pero sin duda, una de las características más importantes y curiosas es la expuesta en [19]. Se puede observar que un embedding puede capturar información sintáctica y semántica, por lo que las analogías existentes entre palabras se pueden expresar, con bastante precisión, mediante la diferencia de los vectores que representan a cada una de las palabras.

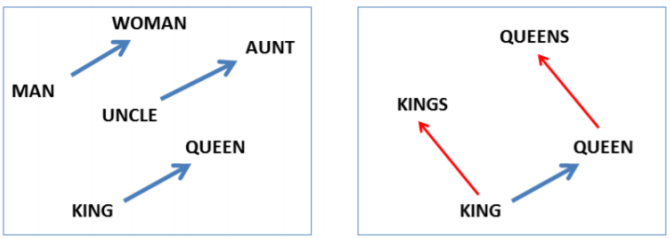


Figure 2: Left panel shows vector offsets for three word pairs illustrating the gender relation. Right panel shows a different projection, and the singular/plural relation for two words. In high-dimensional space, multiple relations can be embedded for a single word.

Figura 2.20: Fuente: [19]

Usando nuestra función W para crear embeddings que definimos en 2.10 en el ejemplo de la imagen 2.20, podemos observar que relaciones como plural/-singular o el género son aprendidas con bastante precisión y podemos incluso realizar ciertas operaciones con ellas:

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

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

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

2.4.1. 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

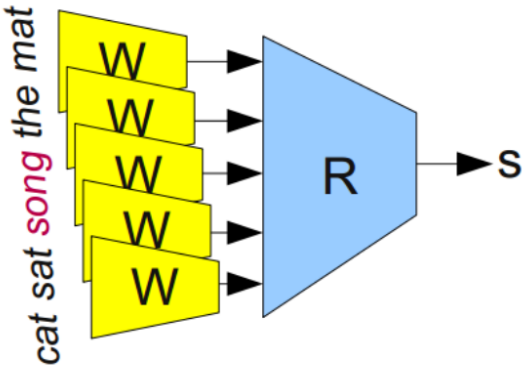


Figura 2.21: Fuente: [13]

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 el comentario es una valoración positiva o no. El módulo R puede estar formado una o varias capas de neuronas, y la red neuronal resultante será del tipo Feed Forward. Podemos entonces encontrar un embedding entrenando la red neuronal mediante el algoritmo de back propagation.

Sin embargo, el modelo anterior tiene ciertas limitaciones, como el número fijo de inputs. En problemas donde el número de palabras es demasiado extenso, como los comentarios en un caso de Análisis de Sentimiento, tenemos un número de inputs demasiado grande y quizás sea complicado representar el significado usando frases largas. Podemos solventar este problema mediante una red neuronal con otra estructura diferente.

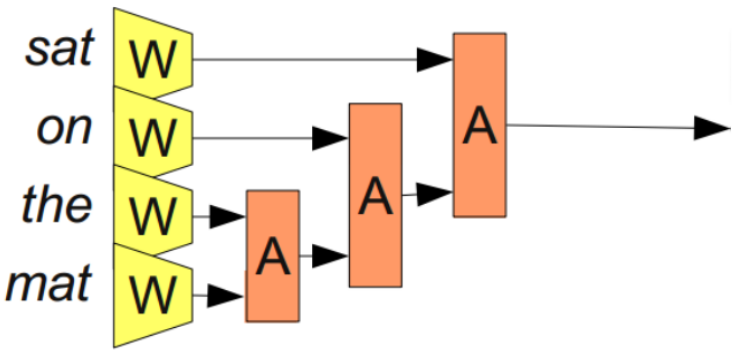


Figura 2.22: Fuente: [13]

Añadiendo un módulo A que vaya asociando palabras de manera recursiva, nos evita el problema de tener un número de inputs fijo, ya que el mismo modelo servirá para comentarios de distinta longitud. De hecho no hace falta ir uniendo las partes de la frase de forma lineal, se pueden ir uniendo de una forma en la que se respete la estructura sintáctica para obtener un mejor resultado.

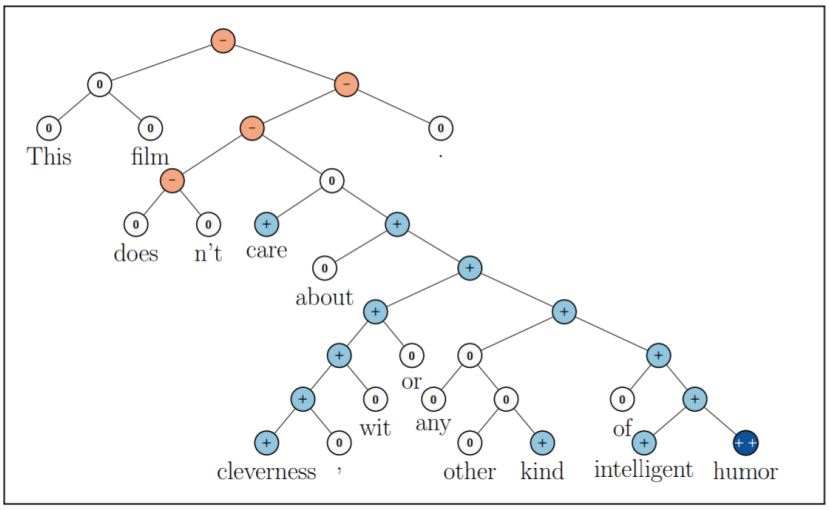


Figura 2.23: Fuente: [20]

Podemos ver en [20] que las RNN funcionan muy bien para problemas de Procesamiento de Lenguaje Natural, en particular para análisis de sentimiento. En este caso en particular, se muestra la construcción de una red neuronal que clasifica de manera bastante efectiva 5 tipos de sentimiento.

Capítulo 3

Metodología

3.1. Datos utilizados

La página web IMDb (Internet Movie Database) ofrece mucha información sobre películas, series de televisión y otros tipos de contenido multimedia. Un uso bastante generelizado es consultar la valoración de una película o una serie realizada por otros usuarios. Esta valoración se basa en reviews de estos contenidos, en las que se deja un comentario y una puntuación del 1 al 10 representadas por estrellas. Para el estudio de nuestro problema de Análisis de Sentimiento, usaremos un conjunto de reviews de la página IMDb publicados en [21].

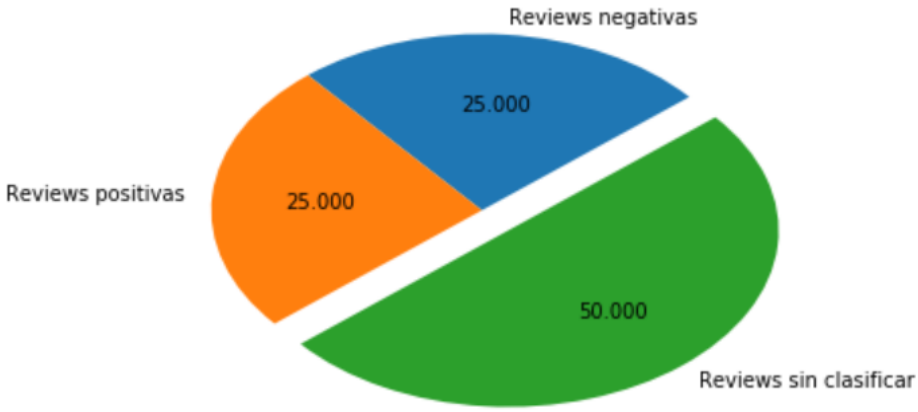
Si descargamos los datos desde [21] obtendremos un conjunto de ficheros con extensión .txt, donde cada fichero contiene un único comentario. El nombre de los ficheros tienen el formato *200_ 8.txt*, donde 200 es un identificador único de cada review y 8 indica una valoración de 8/10 estrellas que se le ha otorgado en este caso concreto. El criterio que se ha seguido para considerar una review como positiva es si tiene una puntuación mayor o igual a 7 estrellas. En cambio, si tiene un número menor o igual a 4 se ha considerado negativa.

Toda la información procedente de los ficheros .txt ha sido condensada en un único fichero .csv en [22], lo que nos ayudará a la hora de trabajar con los

datos.

3.1.1. Características del dataset

Tenemos un total de 100000 reviews en inglés, las cuales están etiquetadas de la siguiente forma:



Observamos que 50000 reviews no tienen puntuación asociada, por lo que no las usaremos en nuestro problema y usaremos la otra mitad del dataset, que como vemos contiene 25000 reviews positivas y 25000 negativas. Cuando partimos de la situación en la que tenemos el mismo número de instancias de cada clase, se dice que tenemos un dataset balanceado.

Si echamos un vistazo a una review aleatoria, vemos que tenemos el texto literal que existe en la página, por lo que están todos los signos de puntuación y caracteres que no nos serán válidos para analizar nuestro problema. Más adelante explicaremos qué métodos usaremos para "limpiar" el texto. Otra característica a remarcar es que no hay más de 30 opiniones de la misma película, ya que existirían correlaciones entre las valoraciones.

3.2. Librerías y algoritmos

Vamos a proceder a construir diferentes tipos de Recurrent Neural Networks y usarlas en nuestro dataset. Probaremos redes con diferentes estruc-

turas, empezando por una RNN con una única capa oculta con estructura LSTM. Después implementaremos la misma red neuronal añadiendo una capa más, lo que hará un total de dos capas ocultas y posteriormente, otro modelo con 3 capas ocultas en total. También implementaremos RNN con estructura bidireccional, es decir, con BRNN. Implementaremos tanto un modelo con una única capa como con dos capas.

Desarrollar todo el código a bajo nivel desde cero sería una tarea demasiado compleja y se podría englobar en otro trabajo aparte, por lo que usaremos algunas librerías basadas en Python que nos facilitarán muchísimo esta tarea. Hoy en día, Python es uno de los lenguajes de programación más usado tanto en investigación como en la industria. Uno de los principales factores que ha incrementado su uso es la legibilidad del código resultante en comparación con otros lenguajes. Además, existe una comunidad muy extendida en internet que desarrolla librerías y las publica de manera libre para que las pueda usar todo el mundo.

Una librería que ha tenido mucho impacto en la implementación de técnicas de Machine Learning es TensorFlow. Fue desarrollada por *Google Brain Team*, una división de Google que centraba su investigación en aprendizaje automático, y publicado como Open Source en 2015. Otra librería muy importante es Keras, que es también una librería de Machine Learning, pero centrada en Redes Neuronales. Es una librería de alto nivel y capaz de ejecutarse sobre TensorFlow y otras librerías como CNTK y Theano. En este proyecto usaremos la librería Keras, con TensorFlow como backend.

Además, para realizar algunos pasos, será necesario ayudarnos de otras librerías opensource muy útiles como *pandas*, *scikit-learn* y *matplotlib*. La librería *pandas* nos permite crear y manipular objetos de tipo *data frame* de manera muy sencilla. *scikit-learn* es una librería de machine learning que nos ayudará a separar nuestros conjuntos de datos para entrenamiento y pruebas. Y por último, usaremos *matplotlib* para representar mediante gráficas los datos obtenidos.

Todos los resultados obtenidos están desarrollados en la aplicación de código libre Jupyter Notebook y alojados en el siguiente repositorio público: `https://github.com/alfonarias/tfm/blob/master/Movie_Review.ipynb`

Capítulo 4

Entrenamiento de redes neuronales artificiales para análisis de sentimiento

Vamos a proceder a aplicar algoritmos de redes neuronales recurrentes a nuestro data set. Para ello dividiremos nuestro conjunto de datos en un conjunto de datos de entrenamiento de 47500 y un conjunto de test de 2500 reviews. Además, mantendremos la proporción entre las dos clases, es decir un 50 % de positivas y otro 50 % de negativas. Crearemos un vector representando la clasificación de cada opinión, donde 0 indicará que es una review negativa y 1 indicará que es positiva.

Para empezar, lo primero de todo es la parte de procesamiento de texto donde empezaremos a usar la librería *Keras*. Fijaremos en 1000 el número máximo de palabras que tendrá nuestro vocabulario o diccionario. Después, la función *Tokenizer* de Keras nos ayudará a quedarnos con las 1000 palabras más frecuentes de todas las reviews, eliminando caracteres innecesarios. Y por último, la función *texts_to_sequences* nos convertirá cada review en un vector de enteros, en el que cada entero indica la referencia al diccionario que hemos creado. En otras palabras, traducimos el texto a números. También es recomendable que todas las secuencias tengan la mima longitud, por ello se usa

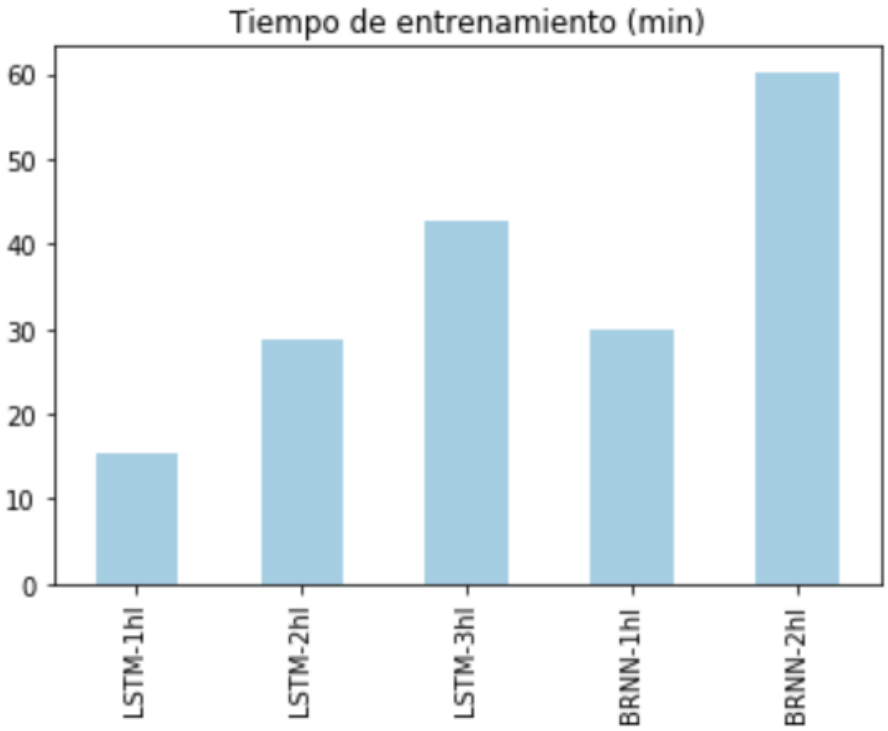
la función *pad_ sequences*. Al indicarle el número deseado, las secuencias serán truncadas y a las que sean más cortas se le añade un valor *null* especial hasta llegar al número deseado.

Pasamos ahora a construir los modelos. Como veremos, la librería Keras nos permite definir redes neuronales en muy pocas líneas de código. Todas las redes neuronales que vamos a construir tendrán una primera capa de tipo emmbedding que nos permitirá representar cada palabra en un vector. En este caso hemos elegido una dimensión de 64 para el word embedding que mantendremos en todos los modelos para poder compararlos correctamente.

El modelo más sencillo que vamos a implementar solo tendrá una capa LSTM que consta de 32 neuronas para aprender sobre las secuencias introducidas en la capa anterior. Si no se le añade ningún parámetro más al LSTM layer, este procesará toda la secuencia hasta el último elemento antes de pasar los valores hacia la siguiente capa. En otras palabras, transforma toda la secuencia en un vector. Además, la función de activación por defecto será la función lineal $f(x) = x$ y tendremos que $bias = 0$. Mantendremos esta configuración en las capas de los demás modelos para poder compararlos entre ellos.

Finalmente, añadimos en todas las redes neuronales construidas una sola neurona de salida usando la capa *Dense* que nos devuelva un valor entre 0 y 1 mediante la función sigmoide, ya que anteriormente definimos un vector indicando la clasificación mediante 0 y 1. Iremos añadiendo más capas ocultas y probaremos qué tal funcionan las redes con estructura bidireccional.

Pasamos a compilar el modelo. Con esto preparamos la red neuronal para ser ejecutada por nuestra librería de backend, en nuestro caso, TensorFlow. Como solo tenemos dos clases, 0 o 1, nuestra medida de pérdida será *binary_ crossentropy* y elegiremos el optimizador *adam*, ya que se recomienda su uso en [29]. A la hora de entrenar el modelo, le indicamos que use 5 veces todos los datos y que use la estrategia de *Mini Batch* que comentamos anteriormente, en este caso, batches o lotes de 32 secuencias.



En esta imagen observamos el tiempo de entrenamiento que ha necesitado cada tipo de red neuronal en función de sus capas ocultas o hidden layers y su estructura, tanto LSTM simple como BRNN. Evidentemente, a más capas o estructuras más complejas se necesita más tiempo de entrenamiento.

Capítulo 5

Validación y test

A la hora de probar si nuestro modelo predice bien las puntuaciones a partir de un comentario, es necesario usar alguna medida. Al tener un conjunto de datos balanceado, es decir, con la misma proporción de clases, es útil usar la medida *accuracy* o exactitud definida de la siguiente forma:

$$accuracy = \frac{n^o\ clasificaciones\ correctas}{n^o\ clasificaciones\ realizadas}$$

Al aplicar esta medida a nuestros datos de entrenamiento y los datos del conjunto de test en cada modelo, hemos obtenido los siguientes resultados:

	testAccuracy	trainAccuracy
LSTM-1hl	0.8788	0.893916
LSTM-2hl	0.8808	0.895263
LSTM-3hl	0.7640	0.891347
BRNN-1hl	0.8516	0.892295
BRNN-2hl	0.8832	0.894842

Figura 5.1: Accuracy conseguida con distintas RNNs

La estructura que mejor ha funcionado es la bidireccional con dos capas ocultas. Aún así, observamos que siempre tenemos mejor accuracy en los datos

de entrenamiento que en los de test, destacando en la red neuronal con 3 capas ocultas. Esto indica que existe *overfitting* en nuestro modelos, es decir, que el modelo se ha adaptado demasiado a los datos de entrenamiento y por tanto no es capaz de clasificar correctamente nuevos datos. Existen muchas técnicas para intentar solventar el problema de overfitting, pero una de más útiles y sencillas de implementar es la propuesta en [23]. Consiste simplemente en ignorar, en cada iteración, un porcentaje de neuronas en una capa durante el entrenamiento. Vamos a entrenar ahora la mismas estructuras de redes neuronales pero añadiendo un porcentaje de dropout del 30 % a las capas ocultas.

	testAccuracy	trainAccuracy
LSTM-1hl-drop	0.8612	0.842147
LSTM-2hl-drop	0.8660	0.881474
LSTM-3hl-drop	0.8516	0.867684
BRNN-1hl-drop	0.8308	0.833011
BRNN-2hl-drop	0.8620	0.873684

Figura 5.2: Accuracy conseguida con distintas RNNs con dropout

Observamos que en algunos casos se soluciona el problema de overfitting y en otros casos sigue existiendo, por lo que quizás habría que aumentar el porcentaje de dropout. Pero hacer esto puede que sea contraproducente, ya que los niveles de accuracy que hemos conseguido ahora en el conjunto de datos de test, está por debajo de los conseguidos anteriormente.

Capítulo 6

Conclusiones

En este trabajo hemos probado diferentes modelos de Recurrent Neural Networks para un problema de Análisis de Sentimiento de un conjunto de reviews de películas, alcanzando un 88.32 % de accuracy en los datos de test. La estructura de red neuronal que ha alcanzado este porcentaje y que mejor ha funcionado ha sido una red del tipo BRNN con dos capas ocultas. Indicar que al existir cierto overfitting en los diferentes modelos, hemos aplicado un porcentaje de un 30 % de dropout que nos ayudaba a solventar el problema en algunos casos, pero nos hacía perder cierto nivel de accuracy.

Por otro lado, atendiendo a los tiempo de entrenamiento de los diferentes modelos, hemos observado que la BRNN de dos capas ocultas ha necesitado unos 60 minutos para terminar la fase de entrenamiento, mientras que una RNN con dos capas ocultas ha necesitado la mitad del tiempo alcanzando un nivel de accuracy muy similar.

6.1. Posibles direcciones futuras

A pesar de variar las estructuras de los diferentes modelos que hemos implementado, hemos mantenido ciertas características en todos los modelos para poder comparar resultados de forma equitativa. Una vez que ya hemos visto cuales son las estructuras de red neuronal que mejor funcionan para un pro-

blema de análisis de sentimiento, habría que ir variando los hiper-parámetros para intentar aumentar el nivel de accuracy. Es decir, habría que probar con diferentes tasas de aprendizaje, distinto número de neuronas en cada capa, introducir bias, cambiar la dimensión del word embedding, probar con otras estrategias de optimización o variar las funciones de activación para mejorar la efectividad de nuestra red neuronal.

Apéndice A

Disponibilidad de datos *versus* complejidad del algoritmo

Aprovechando el desarrollo de este trabajo, hemos probado a entrenar algunas redes neuronales con la misma estructura pero disminuyendo el número de datos de entrenamiento a 7500. Atendiendo a las estructuras en las que hemos añadido dropout, hemos observado que los modelos más complejos entrenados únicamente con 7500 instancias no han podido igualar el nivel de accuracy conseguido por los modelos más simples entrenados con 47500 instancias. Todos ellos validados sobre el mismo test dataset.

Este hecho podría indicarnos que la cantidad de instancias que usamos en la fase de entrenamiento de una RNN tiene mucha mayor relevancia que la complejidad de la estructura del modelo, si hablamos de un problema de Análisis de Sentimiento.

Apéndice B

Ejemplo de implementación en Python de una red neuronal

A continuación se muestra un ejemplo en Python sobre cómo construir una red neuronal. El código completo está en `https://github.com/alfonarias/tfm/blob/master/Movie_Review.ipynb`

Listing B.1: Ejemplo de red neuronal con una capa para el dataset de IMDb

```
import pandas as pd
from datetime import datetime

#----LECTURA DE DATOS

data=pd.read_csv('imdb_master.csv',encoding='latin-1',
        index_col=0)
#Se borran las instancias sin clasificar
data_labeled = data[data.label != 'unsup']
#Se transforman las etiquetas
y=data_labeled['label'].apply(lambda x: 0 if x == 'neg' else 1)
```

```
from sklearn.model_selection import train_test_split
#Aislamos nuestro conjunto de test y de train
reviews_large, reviews_test, y_large, y_test =
         train_test_split(data_labeled['review'], y,
         test_size=2500, stratify=y)

#----PROCESAMIENTO DE TEXTO
import keras
#Numero maximo de palabras que tendra nuestro diccionario
max_dic = 1000

#El Tokenizer de Keras nos permite quedarnos con
        las palabras mas frecuentes de todas las reviews
diccionario_large =keras.preprocessing.
        text.Tokenizer(num_words = max_dic)
diccionario_large.fit_on_texts(reviews_train)
#Ahora, por cada review obtenemos un vector de enteros
        indicando la palabra del diccionario
X_train_large=diccionario_large.
        texts_to_sequences(reviews_large)
#Realizamos lo mismo para el data set de test
X_test_large=diccionario_large.texts_to_sequences(reviews_test)
#Es recomendable que todas las reviews
        tengan la misma extension de palabras
max_palabras=300
X_train_large=keras.preprocessing.sequence.
        pad_sequences(X_train_large,maxlen=max_palabras)
X_test_large=keras.preprocessing.sequence.
        pad_sequences(X_test_large,maxlen=max_palabras)

#---- ESTRUCTURAS RNN
red_neuronal=keras.models.Sequential()

#Primera capa tipo embedding.Creamos embedding de dimension 64
red_neuronal.add(keras.layers.embeddings.
        Embedding(input_dim=max_dic, input_length=max_palabras,
        output_dim=64))
```

```
#Segunda capa tipo LSTM con 32 neuronas. Devuelve un vector
        despues de procesar la secuencia completa
red_neuronal.add(keras.layers.recurrent.LSTM(32))
#Ultima capa que devuelve un valor entre 0 y 1
red_neuronal.add(keras.layers.core.Dense(1))
red_neuronal.add(keras.layers.core.Activation('sigmoid'))
red_neuronal.summary()


#Compilacion del modelo
red_neuronal.compile(loss='binary_crossentropy',
        optimizer='adam', metrics=['accuracy'])

#Fase de entrenamiento del modelo
startTime = datetime.now() #medir el tiempo de entrenamiento
history=red_neuronal.fit(X_train,y_train,batch_size=32,epochs=5)
time=datetime.now() - startTime

#Validacion del modelo en el test dataset
validacion=red_neuronal.evaluate(X_test_large, y_large)
print("Test loss", validacion[0])
print("Test accuracy", validacion[1])
```

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] Roelants, Peter. *How to implement a recurrent neural network Part 1* *http://peterroelants.github.io/posts/rnn_implementation_part01/*

[9] 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*

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

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

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

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

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

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

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

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

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

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

[20] Socher, Richard et al., 2013. *Recursive Deep Models for Semantic Compositionality Over a Sentiment Treebank*

[21] Maas, Andrew et al., 2011. *Learning Word Vectors for Sentiment Analysis.* `http://ai.stanford.edu/~amaas/data/sentiment/`

[22] *https://www.kaggle.com/utathya/imdb-review-dataset/version/1*

[23] Srivastava, Nitish et al., 2014. *Dropout: A Simple Way to Prevent Neural Networks from Overfitting*

[24] *https://blog.acolyer.org/2017/03/23/recurrent-neural-network-models*

[25] *https://www.teslarati.com/ai-china-doctors-cancer-diagnosis-test/*

[26] Mullen, Tony; Collier, Nigel, 2004. *Sentiment analysis using support vector machines with diverse information sources*

[27] M. Jadav, Bhumika; B. Vaghela, Vimalkumar, 2016. *Sentiment Analysis using Support Vector Machine based on Feature Selection and Semantic Analysis*

[28] Nakagawa,Tetsuji, 2010. *Dependency Tree-based Sentiment Classification using CRFs with Hidden Variables*

[29] Ruder,Sebastian, 2017. *An overview of gradient descent optimization algorithms*