Trabajo monográfico

Redes neuronales con aplicaciones a procesamiento de lenguaje natural

Santiago Robaina

Junio 2022

Orientadores:

Soledad Villar
Gabriel Illanes

Licenciatura en Matemática
Universidad de la República
Montevideo, Uruguay

Agradecimientos

Quería dar las gracias a todos aquellos que me acompañaron en el proceso de la carrera, a quienes estuvieron desde aquellos primeros días de clase, los que acompañaron durante los años de pandemia y también a quienes llegaron sobre el final.

Un gracias muy grande a mi familia, que desde el primer momento me incentivaron a esforzarme y a perseguir mis metas, gracias por permitirme priorizar mi estudio y formación. A mis amigos, que siempre estuvieron cerca para acompañar estos últimos 4 años, siempre a la orden para una comida o un mate. En particular a aquellos que hicieron que este proceso de estudiar matemática sea tan ameno y disfrutable a pesar de las dificultades, siempre presentes para juntarse alrededor de un pizarrón a pensar y también para compartir una merienda con un truco de por medio.

A mis tutores y a los docentes a lo largo de la carrera que hicieron de la licenciatura y de la matemática un lugar de formación y crecimiento pero también un lugar de encuentro.

Índice general

1. Resumen 5

Capítulo 1. Introducción 7

Capítulo 2. Redes Neuronales 9
1. Modelos lineales 10
1.1. Regresión lineal 10
1.2. Regresión logística 10
2. Ejemplo XOR 11
3. Redes neuronales feedforward 13
4. Partes de una red 15
4.1. Función de costo 15
4.2. Funciones de costo para estadísticos 16
4.3. Neuronas de output 17
4.4. Neuronas de capa oculta 19
4.5. ReLU 19
5. Arquitectura 19
6. Back Propagation 21
7. Descenso de gradiente 24
8. Descenso de gradiente estocástico 25
8.1. Algoritmos de *Batch* 25
9. Redes neuronales recurrentes 26
9.1. Forward propagation 28
9.2. Back propagation 29

Capítulo 3. Procesamiento de lenguaje 31
1. ¿Qué es el procesamiento de lenguaje natural? 31
2. Semántica de palabras 32
3. Modelo de espacio de vectores 32
4. *One-hot encoding* 33
5. Métodos de distribución 33
6. Medida de similitud 34
7. Embeddings 35
8. Métodos de predicción 35
8.1. Modelo de lenguaje con redes feedforward 36
9. *Word2Vec* 36
9.1. Skip-gram 36
9.2. CBOW 38

9.3. Optimizaciones de los modelos 41

Capítulo 4. Codenames 43
1. Introducción 43
2. El Juego 43
2.1. Algoritmo para dar pistas 45
2.2. Función DETECT 46
2.3. Evaluación de los algoritmos 48
3. Conclusiones 48

Capítulo 5. Trabajo futuro 51
1. Transformers 51
1.1. Positional Encoding 52
1.2. Atención 53
2. Ética en los modelos 55

Bibliografía 57

1. Resumen

En este trabajo monográfico buscaremos dar cierto marco teórico al paper [**4**]. Este trabajo propone un algoritmo que permite que un ordenador pueda jugar al juego de caja *Codenames*. Para ello utiliza algunas nociones de procesamiento de lenguaje natural (PLN) como pueden ser los embeddings y la similitud de palabras.

Para realizar este marco aparece la necesidad de describir el funcionamiento de las redes neuronales *feedforward*, desde su motivación y casos de uso hasta su diseño y optimización.

Usando esta estructura definiremos algunas herramientas desarrolladas para el campo del PLN como pueden ser los embeddings y específicamente *Word2Vec* ([**6**]), un método para dar una biyección entre palabras y vectores de un cierto espacio vectorial.

Capítulo 1

Introducción

El 2020 fue un año especial para todos, hubo varios meses de aislamiento, hubo quienes dedicaron sus tiempos libres para aprender a tejer, otros empezaron a dominar el arte de hacer pan, incluso aprender algún idioma. La distancia obligó a que empezáramos a comunicarnos con aquellas personas a las que veíamos más a menudo como lo haríamos con familiares o amigos que viven en otro continente, la virtualidad empezó a ser algo practicado por todos, se organizaron cumpleaños, reuniones de trabajo y hasta casamientos de forma virtual.

En mi caso mi relación con el estudio cambió, estaba acostumbrado a seguir los cursos con mi grupo de estudio, nos quedábamos todos los días después de clase a hacer ejercicios e intentar entender los materiales. Cuando llegó la pandemia esto se vio imposible de hacer, toda esa gente a la que veía regularmente, muchísimas más horas por día básicamente desaparecieron, las clases virtuales con cámaras apagadas no servían como sustituto a las dinámicas que eran habituales en la facultad. Por supuesto que la capacidad de entender la matemática se vio dificultada, pero lo que más faltó en ese tiempo fueron las charlas distendidas sobre cualquier cosa, los ratos de estudio pero sobre todo los ratos de juego. En algún momento de la cuarentena empezamos a hablar en el grupo viendo que estaría bueno tener algún rato, aunque fuera virtual para retomar esos espacios de juego y distensión. El medio que encontramos para esto fue un juego de caja que tiene su versión virtual, el *Codenames*, un juego de palabras en el que podíamos juntarnos y pasar ratos largos encontrándonos como lo hacíamos antes de que el mundo se pusiera en pausa. El juego siempre aparecía como mediador, algunas veces era *Codenames*, a veces era otro.

Todo este preámbulo es para explicar de dónde nace la motivación de este trabajo. Un tiempo después de que incorporamos el hábito del juego, empezó la búsqueda de temas para la monografía, hablando con Gabriel y Soledad pasamos por varias ideas, desde problemas de transporte óptimo hasta exploración de bases de datos genéticas. Hasta que un día Soledad dijo que había estado jugando a un juego y que se le ocurrió que podía ser interesante intentar pensar cómo se podría atacar utilizando técnicas de procesamiento de lenguaje natural, cuando explicó el juego enseguida caí en la cuenta de que era aquel que nos encontraba con mis compañeros. Compartimos algunas ideas sobre cómo se podría hacer para utilizar un modelo de lenguaje para jugar. Sin darnos cuenta estuvimos varios minutos detenidos pensando en el juego y fue ese el momento en el que nos dimos cuenta que ese debía ser la motivación del trabajo.

Encontramos un paper que proponía una solución al juego utilizando distintos embeddings y modelos de lenguaje. Desde ahí la monografía se convirtió en un intento por entender de qué significaba la palabra *embedding* y cómo se utilizaba para jugar.

El objetivo de la monografía es hacer una revisión bibliográfica sobre Redes neuronales y procesamiento de lenguaje natural para comprender la solución propuesta por el paper [**4**] al juego codenames.

Comenzaremos con una introducción al concepto de redes neuronales en el capítulo 1, la referencia de cabecera para este capítulo fue [**2**], en particular los capítulos 6, 8 y 10. En el capítulo comenzamos definiendo las ideas básicas de redes neuronales, *¿Qué son?, ¿Cómo funcionan?, ¿Para qué sirven?*, entendiendo las distintas componentes que hacen a una red. Seguimos entendiendo la forma de optimizar estos modelos con métodos de descenso de gradiente. Por último introducimos la arquitectura de red recurrente, una arquitectura muy útil para procesar secuencias de texto y que son la base de los *Transformers*, de los que hablaremos muy brevemente en la sección de trabajo futuro.

Luego con las herramientas básicas de redes neuronales estamos en condiciones de entender un embedding importante. Para esto en el capítulo 2 teniendo de referencia [**5**] y [**3**] definimos las nociones básicas de procesamiento de lenguaje y algunas de sus herramientas, entre ellas algunos embeddings.

El capítulo 3 está dedicado a explicar el paper [**4**], que nos propone una solución al juego que motiva la monografía. Se explican los algoritmos propuestos y además algunos resultados empíricos sobre el funcionamiento de los mismos.

Capítulo 2

Redes Neuronales

El mundo en el que vivimos puede ser muy complejo y cambiante, es por eso que a la hora de entender las cosas que ocurren a nuestro alrededor hacemos uso de modelos simplificados de la realidad. Estando en un momento de la historia donde tenemos la capacidad de generar y almacenar datos con una abundancia jamás antes vista aparece el campo del aprendizaje automático, una disciplina que encuentra la computación, la matemática y la estadística, con el fin de poder hacer uso de toda esta masividad de datos y desarrollar modelos que nos permitan entender nuestro entorno y también desarrollar herramientas para continuar el desarrollo tecnológico.

Dentro del aprendizaje automático existen dos tipos de problemas que surgen a menudo, que requieren de dos tipos distintos de enfoque dependiendo de cómo sean los datos que tengamos a disposición y cuál sea la tarea que queremos lograr.

El primero estos problemas es el que se conoce como: **aprendizaje supervisado** y se da en aquellos contextos en los que el objetivo que tenemos es aproximar una función a partir de algunos ejemplos de cómo esa función ha respondido anteriormente, los datos que tenemos para aprender el modelo incluyen cómo se comporta esa función que buscamos aprender, de aquí el nombre supervisado. Es importante destacar que a la hora de entrenar un modelo de aprendizaje supervisado tendremos datos sobre los que haremos este entrenamiento, a estos datos los llamaremos datos de entrenamiento. Es a través de estos datos de entrenamiento que buscamos mejorar el modelo.

Una dificultad que tiene el problema es que tenemos como objetivo que nuestro modelo funcione correctamente en datos distintos de los que entrenamos, justamente estos los usamos para entrenar pero buscamos que funcione bien para datos que aún no hemos visto, es decir buscamos que generalice bien. Podríamos encontrar un modelo que responde excelentemente a los datos con los que se entrenó pero que al recibir datos nuevos tenga respuestas distintas de las que esperamos, este fenómeno se conoce como ***overfitting***. Es por esto que se suele separar los datos en un conjunto de entrenamiento y un conjunto de validación. Con el de entrenamiento definiremos los parámetros que luego usaremos, y con los datos de validación nos aseguraremos que el modelo generalice correctamente a datos que jamás vio antes.

Por otro lado tenemos el **aprendizaje no supervisado**, que se da cuando queremos buscar patrones en los datos sin tener a priori una información sobre cómo se agrupan esos datos.

Nos centraremos en los problemas de aprendizaje supervisado. Estos suelen caer en una de dos categorías: **clasificación** o **regresión**. La primera refiere al problema que surge cuando queremos asignarle a cada dato una categoría, por ejemplo clasificar entre una imagen de una cruz y de un círculo. O un caso de aplicación más real podría ser,

recibir una imagen de una persona y determinar si tiene un barbijo puesto. El problema de regresión en lugar de asignarle una categoría discreta (como círculo, cruz, barbijo, no barbijo) lo que busca es dar un valor continuo, como podría ser el caso de recibir las características de una casa y el modelo busca dar una valor del precio en el mercado.

Las redes neuronales tienen muy buenos desempeños tanto en problemas de clasificación como de regresión. Algunos ejemplos se han hecho muy populares en el auge de la inteligencia artificial en el que vivimos, algunos ejemplos son *Alpha Go* [**8**], una red neuronal que fue entrenada, en 2015, para vencer al mejor jugador humano de Go del momento. Este evento marcó un antes y un después para el mundo de la inteligencia artificial y las redes neuronales. El go es un juego tremendamente complejo porque el tablero presenta una cantidad insondable de posibles estados, y es por lo tanto imposible programar un jugador artificial que dado un estado del tablero pueda conocer todos los posibles estados futuros y elegir la jugada más favorable.

En esta monografía presentaremos la forma más básica de red neuronal. Para entender exactamente cómo funciona es conveniente entender primero los modelos lineales que son un primer intento de atacar los problemas de clasificación y regresión. Luego veremos cómo las redes neuronales superan a estos modelos lineales cuando la tarea es un poco más compleja, y explicaremos qué son y cómo funcionan.

1. Modelos lineales

Una forma de modelo consiste en representar la relación entre dos o más cantidades del mundo a través de una función. La función busca reflejar cómo esas cantidades están relacionadas las unas con las otras. Un ejemplo de este tipo de modelo es el modelo lineal, para el caso de regresión lo usual es utilizar la regresión lineal.

1.1. Regresión lineal Se asume que n cantidades $X_1, \cdots, X_{n-1}, Y$ cualquiera sean, su relación puede ser representada con una expresión lineal de la forma:

$$Y = f_{a,b}(X_1, \cdots, X_{n-1}) = a_1X_1 + \cdots + a_{n-1}X_{n-1} + b$$

$$\text{para algunos } (a_1, \cdots, a_{n-1}) = a \in \mathbb{R}^{n-1}, b \in \mathbb{R}$$

Lo que buscamos es encontrar la función de esta forma que mejor se ajuste a la los datos que observamos, y para eso necesitamos una noción de qué tanto se acerca el modelo a la realidad, esta es la **función de costo**. Buscamos los parámetros a, b que logren que lo que nuestro modelo prediga tenga el menor error o costo posible.

Otro lugar de aplicación de los modelos lineales surge cuando en lugar de hacer una regresión queremos clasificar. En ese contexto aparece la regresión logística.

1.2. Regresión logística La regresión logística es un modelo lineal generalizado. Lo que buscamos es encontrar una curva que dado un valor x asigne la probabilidad de que ocurra uno de dos posibles eventos (la probabilidad de que un determinado dato lleve una determinada categoría). Para ajustar esta curva lo que hacemos es transformar la salida en el eje y usando la función logit:

$$logit(p) = \log(\frac{p}{1-p})$$

Esta función es biyectiva entre [0,1] y $\mathbb{R}$ y por lo tanto podemos aplicarla para trabajar en este nuevo espacio y luego deshacer los cambios. Esta transformación lleva el 1/2 a

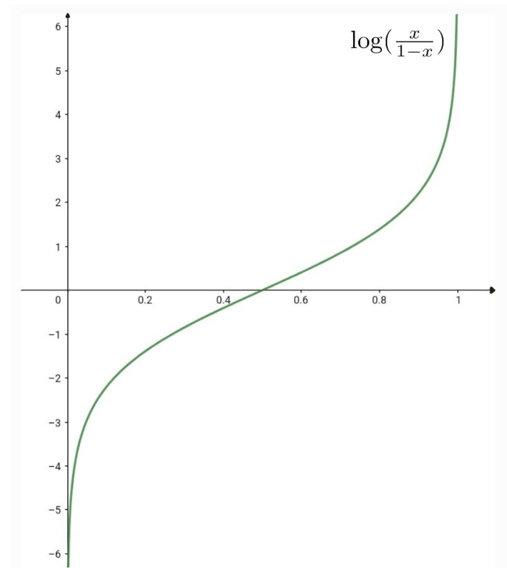

FIGURA 1. Gráfico de la función *logit*, notemos que la función toma valores del $[0, 1]$ y los transforma a valores de todo $\mathbb{R}$

0, el 1 en $+\infty$ y el 0 en $-\infty$. Al transformar el espacio con la función logit los datos que se encontraban clasificados en una de dos categorías (1 o 0) serán mapeados a $+\infty$ o $-\infty$. Esto hace que no sea posible ajustar una recta utilizando el método de mínimos cuadrados. Lo que haremos será considerar una recta y le tomemos la preimagen por la función *logit*, esto será una curva que tendrá en cada punto un valor de probabilidad para cada evento, un ejemplo gráfico se puede ver en la figura 2. Con está curva de probabilidades utilizaremos el método de máxima verosimilitud. En resumen lo que estamos haciendo es ajustando una recta y luego transformándola hasta obtener los valores de las probabilidades de los eventos. La mejor recta será la que maximice la función de verosimilitud de los datos.

Estos tipos de modelos pueden ser muy útiles pues son fáciles de entender y de usar para sacar conclusiones, siempre y cuando se ajuste adecuadamente a lo que queramos modelar. Aparecen mucho para visualizar tendencias en un conjunto de datos, pero tiene como limitante que solo puede expresar una relación lineal entre las cantidades que queramos modelar, si aquello que estamos intentando modelar no sigue esta relación el modelo nos podrá aportar de manera muy limitada o hasta de forma incorrecta.

2. Ejemplo XOR

Un ejemplo donde vemos la limitación de los modelos lineales es el sencillo caso en el que nuestro objetivo es modelar la relación de 4 puntos

$$\{(0, 0), (0, 1), (1, 0), (1, 1)\}$$

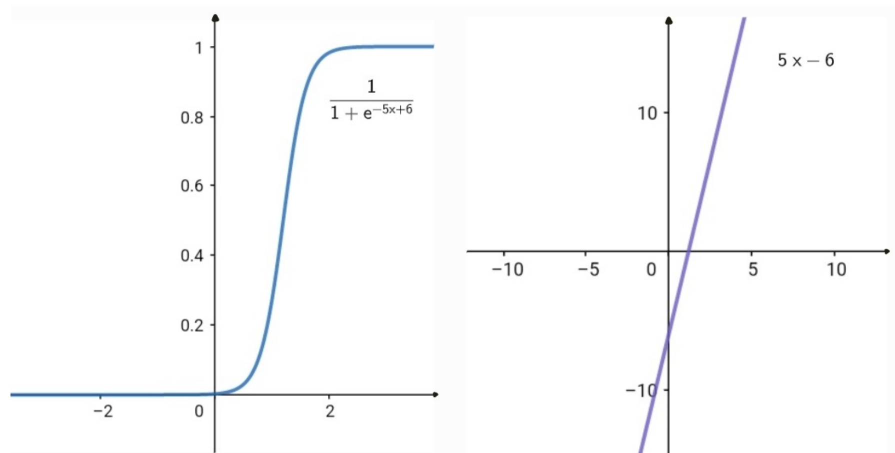

Figura 2. A la izquierda tenemos un ejemplo de curva que toma valores entre 0 y 1, y a la derecha aparece transformada por la función *logit*, la regresión logística busca la recta tal que su preimagen (la curva de la izquierda) maximice la función verosimilitud de los datos.

Queremos que estos puntos se separen en dos posibles categorías: 1 en caso de que las dos coordenadas del punto sean iguales, y 0 en caso de que sean distintas. Diagramado en un dibujo en el plano se ve como muestra la figura 3.

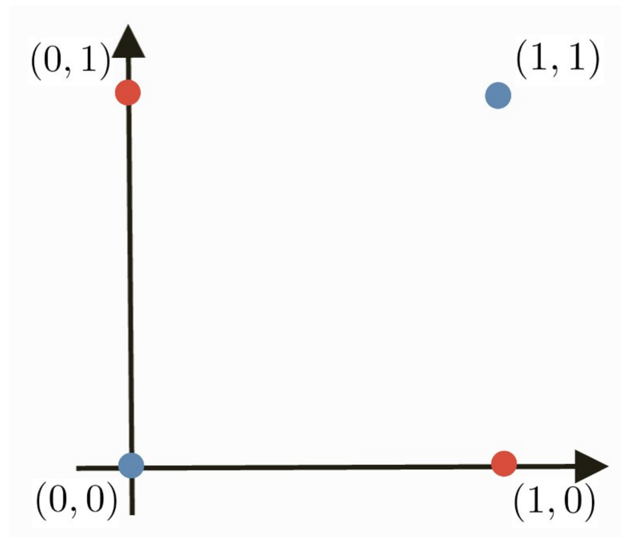

Figura 3. Ejemplo XOR, la relación lógica del *o* excluyente. Tenemos 4 puntos en el plano real, clasificados en dos categorías 1 y 0 representadas con los colores rojo y azul.

Buscamos una función que dado uno de los puntos anteriores me devuelva un 1 o un 0 según corresponda. El problema es que no existe ninguna función lineal que logre esta

sencilla tarea. Esto lo podemos ver de manera gráfica. Si consideramos el gráfico de esta función que buscamos veremos que es una superficie en el $\mathbb{R}^3$, por ser lineal el gráfico de la función aplicada a todo $\mathbb{R}^2$ será un plano afín, en particular si miramos los puntos que estamos buscando modelar este plano debería pasar por 0 en los puntos $\{(0,1),(1,0)\}$ y por 1 en los otros dos. El problema es que dados tres puntos existe un único plano que los contiene. Es bastante claro que si ponemos la condición que estamos buscando para tres de los cuatro puntos obtendremos un plano que no satisface la condición del punto restante. Es decir, no van a existir dos parámetros a, b que nos den un modelo $f_{a,b}$ como el que buscamos.

Es un modelo muy limitado dado que tiene solo dos parámetros, y esa es toda la flexibilidad que tiene, sería demasiado esperar que esto pudiese captar relaciones sofisticadas. Es por esto que surge la necesidad de modelos más complejos con una mayor cantidad de parámetros.

Para superar estas limitaciones podemos utilizar un modelo que en lugar de expresar una relación lineal entre X y Y lo haga entre Y y una transformación de los valores de X, es decir:

$$Y = a\phi(X) + b \text{ con } a, b \in \mathbb{R}$$

Esta es una forma de mantener lo bueno de los modelos lineales pero haciéndolos más flexibles. Lo que queda por definir es de qué manera transformamos esos valores Y, es decir cómo definimos ϕ.

Para esto hay varias posibilidades:

1. Utilizar una ϕ genérica. Un ejemplo de esto es support vector machines, que se usa principalmente para problemas de clasificación, la idea de SVM cuando un modelo lineal no es suficiente es pasar los datos a un espacio de dimensión infinita con el 'Kernel-Trick' a través de una función kernel, este truco lo que nos habilita es tener los datos en un espacio de dimensión mayor donde puedo separar mis las clases por hiperplanos.
2. Y la otra posibilidad es definir ϕ a menos de parámetros y aprender los parámetros buscando minimizar la función de costo. Este es el enfoque de las redes neuronales. Existen varios tipos de redes neuronales según cómo sea su arquitectura, cada red tendrá sus ventajas en las distintas tareas que resuelva. Por ejemplo existen las redes convolucionales, que son muy utilizadas en el campo del procesamiento de imágenes; las redes recurrentes, que se utilizan para procesar datos donde es importante el orden temporal, como puede ser reconocimiento de voz; y las redes feedforward que son el ejemplo más simple de red neuronal, sobre este tipo de redes estaremos ampliando en la próxima sección.

3. Redes neuronales feedforward

Las redes neuronales surgen como una forma de aprender los parámetros de esta última posibilidad.

¿Qué son?

Las redes neuronales justamente son funciones genéricas dentro de una familia y están definidas a menos de parámetros.

¿Por qué redes?

Se les llaman redes pues se pueden representar gráficamente como un grafo direccionado, donde cada uno de los nodos es lo que llamaremos una **neurona**.

La función puede pensarse como dividida en **capas**, en primer lugar la capa de entrada, que es la que recibe los valores de X (tiene tantas neuronas como dimensión de X) y por último la capa de salida que puede tomar distintas formas pero siempre es la que define la salida de la función. Y entre esas capas están las capas ocultas, llevan ese nombre debido a que no está preestablecido que es lo que deberían hacer, y tampoco tenemos una noción completa, a priori, de cómo influyen en la salida de la red. Las capas están conectadas a través de funciones, es decir que la segunda capa es función de la primera, una función $f^1 : \mathbb{R}^n \to \mathbb{R}^m$ donde n es la cantidad de neuronas en la capa 1 y m la cantidad de neuronas en la capa 2. Dichas funciones son las que aparecen representadas en el grafo como las flechas de una capa a la otra, lo mismo sucede con el resto de las capas. Concluimos que la red es una composición de funciones que transforman la entrada capa a capa hasta obtener una salida.

¿Qué forma deben tener las funciones f^i?

Sabemos que no pueden ser lineales, ya que si todas lo fueran, acabaríamos con una composición de funciones lineales, que es una función lineal.

Usualmente para evitar el problema descrito antes se definen las funciones f^i usando una composición: por un lado un mapa lineal afín, regulado por dos tipos de parámetros, los **pesos** que definen la transformación lineal y los **sesgos** que definen como está trasladada la transformación lineal; y por otro lado una función no lineal que llamaremos **función de activación**.

$$f^i(h) = \sigma(Wh + b) \text{ donde } \begin{array}{l} W \text{ es la matriz de pesos} \\ b \text{ es un vector de sesgos} \\ \sigma \text{ es la función no lineal} \end{array}$$

Esta componente no lineal σ es la que nos permite modelar escenarios más interesantes aunque también torna más complejo el problema. La función de costo como función de los parámetros no es convexa por la composición.

Esto dificulta el estudio de la convergencia de los algoritmos, no podemos asegurar a priori que los parámetros convergen siempre a los mismos valores. El caso de modelo lineal se resuelve de manera sencilla porque podemos obtener los parámetros óptimos resolviendo un sistema lineal, mientras que en el caso no lineal nos vemos obligados a optimizar usando descenso por gradiente, que es computacionalmente más costoso. Además como los parámetros los encontramos moviéndonos según el gradiente, tenemos el problema que al no ser convexa la función de costo podríamos terminar encontrando alguno de los múltiples mínimos locales dependiendo de cómo se hayan inicializado los parámetros.

Por qué neuronales?

Se les llama redes neuronales porque su diseño está inspirado en modelos antiguos sobre el funcionamiento de las neuronas que trabajan en nuestro cerebro; estas están conectadas entre sí y se transmiten información a través de impulsos eléctricos, estos impulsos son los que inspiran los pesos que ponderan la suma que llega a una cierta neurona en nuestra red neuronal artificial.

Por qué feedforward?

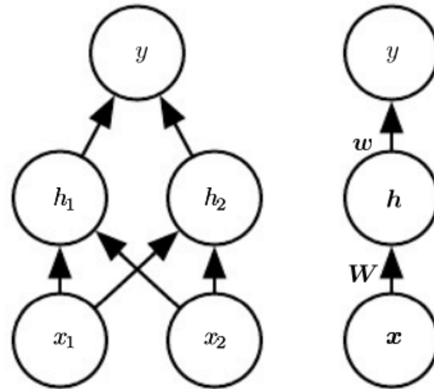

FIGURA 4. Ejemplo sencillo de una red con una capa oculta de dos neuronas

Feedforward hace referencia a cómo fluye la información en la red. Siempre va hacia adelante: es decir, tenemos una determinada entrada de la red y esa misma entrada la vamos transformando con las distintas funciones hasta llegar a la salida, y esta no vuelve a ingresar. En el caso de este tipo de redes tenemos que el grafo que usamos para representarla no tiene ciclos. Existen otras arquitecturas de redes, como las que introduciremos en la sección 9 donde hay algunas aristas de la red formando ciclos.

4. Partes de una red

Una parte fundamental del diseño de la red es la elección de la función de costo. Usaremos el descenso por gradiente estocástico para encontrar los parámetros que definan la "mejor"[1] función con ese diseño de red. La idea del aprendizaje por gradiente es hallar el gradiente de la función de costo según los distintos parámetros y de ese modo ir iterativamente disminuyendo el valor de la función de costo, hablaremos más en detalle sobre esto en la sección 7

4.1. Función de costo La elección de la función de costo dependerá de la salida que tenga la red. A la última capa de la red llegan los valores de la entrada transformados por las capas ocultas, que llamaremos $h = f(X, \theta)$. El rol de la última capa es completar la tarea, es decir lo que haga esta capa con h será la salida que f_θ le asigna a X.

- ***CrossEntropy***
  En algunos casos el modelo que tenemos nos da una distribución condicional $p_{\hat{\theta}}(Y|X)$ como salida dados unos ciertos parámetros $\hat{\theta}$. Podemos usar el principio de máxima verosimilitud para intentar ajustar la función a la realidad que queramos modelar, es decir buscar los parámetros que le asignen mayor probabilidad a los datos que tenemos para ajustar el modelo. Sin embargo lo que buscamos a la hora de entrenar la red neuronal es minimizar una función de costo.

[1] Los parámetros a los que converja la optimización de la función de costo

Observación: Afirmamos que si los datos que estamos usando para entrenar y para estimar la función con máxima verosimilitud son los mismos, y además la familia de posibles soluciones es la misma, entonces si tenemos que la función de costo que utilizamos para el entrenamiento es la de entropía cruzada luego, los parámetros que realizan el mínimo de entropía cruzada son los mismos que realizan el máximo de la función de verosimilitud.

La entropía cruzada para el caso discreto está definida como:

$$H(p,q) = -\sum_{x \in \mathcal{X}} p(y|x)\log(q(y|x))$$

Y la función de verosimilitud con $-\log$ es:

$$-\log(L(q)) = -\sum_{x \in \mathcal{X}} \log(q(y|x))^n$$

Donde $\mathcal{X}$ es el recorrido de X y n_x la cantidad de ocurrencias de x.

Como ejemplo para ilustrar la relación entre estas funciones consideremos el caso discreto y las distribuciones empíricas de los datos p_N y la distribución del modelo q_θ.

Si dividimos la función de verosimilitud entre N, la cantidad de datos que tenemos obtenemos:

$$-\frac{1}{N}logL(q_\theta) = -\frac{1}{N}\sum_{x \in \mathcal{X}} log(q_\theta(y|x))^{n_x} = -\sum_{x \in \mathcal{X}} \frac{n_x}{N} log(q_\theta(y|x))$$

Y por otro lado la entropía cruzada entre la distribución empírica es:

$$H(p_N, q_\theta) = -\sum_{x \in \mathcal{X}} p_N(y|x)log(q_\theta(y|x))$$

Observamos que $p_N(y|x) = \frac{n_x}{N}$ y por lo tanto, La entropía cruzada entre la distribución empírica y la función de verosimilitud dividido N son la misma. Como dividir entre N no varía el argumento mínimo entonces tenemos que los parámetros donde menos logaritmo de máxima verosimilitud se minimiza también minimizan la función de entropía cruzada entre la distribución del modelo y los datos. Por lo tanto es equivalente usar como función de costo la entropía cruzada y utilizar el principio de máxima verosimilitud.

4.2. Funciones de costo para estadísticos No siempre queremos aprender una distribución entera, por lo general para lograr esto necesitamos una cantidad muy grande de datos y la capacidad de cómputo para poder entrenar un modelo que permita está completar esta tarea. Por eso es que a veces aspiramos a un poco menos que la distribución entera y nos conformamos con que la red neuronal nos de como salida algún estadístico de la distribución, como puede ser la media o la mediana. Para cada estimador tendremos una función de costo que logrará que al minimizarla obtenemos el estadístico deseado.

- **Suma de errores cuadráticos** Media

  En el caso de un problema de regresión donde queremos asignar un valor real a cada entrada X en lugar de calcular la distribución $P(Y|X)$ podemos

querer calcular directamente la media de esa distribución, porque esa media es la que daremos como salida. Tenemos resultados teóricos que nos aseguran que si utilizamos como función de pérdida la suma de los errores cuadráticos entonces obtenemos que el valor que minimiza el error es la media.

$$L(f_\theta) = \sum_{i=1}^{N} \|y_i - f_\theta(x_i)\|^2$$

Dónde los pares (x_i, y_i) son datos de entrenamiento y $f_\theta(x_i)$ es la salida de la red para x_i

Por lo tanto, siempre que la función media esté en la familia de funciones sobre las que estamos optimizando obtendremos lo que buscábamos.

- **Suma de errores absolutos** Mediana

  Otro estimador más robusto que la media es la mediana. Este estadístico también es posible estimarlo con una red neuronal. La función de costo con la que obtenemos este estimador es la suma de errores absolutos, también conocido como la distancia $L1$

$$L(f_\theta) = \sum_{i=1}^{N} \|y_i - f_\theta(x_i)\|$$

4.3. Neuronas de output A la hora de modelar un problema siempre necesitaremos algún tipo específico de salida. Si estamos buscando modelar una función con una salida real entonces necesitamos una salida que pueda devolver cualquier valor real. Si queremos modelar la probabilidad de un suceso siempre la salida deberá ser un valor entre 0 y 1. Cuando tenemos un problema de clasificación queremos buscamos tener una salida que represente las probabilidades de cada una de las categorías posibles, es decir queremos un vector de valores entre 0 y 1, y que la suma de todas las entradas dé 1. A continuación exponemos algunas de las neuronas de salida más utilizadas:

- Una posible salida es utilizar una **neurona lineal afín**. Esta neurona realizará la operación:

$$\hat{y} = W.h + b$$

  Siendo $W \in M_{n_h \times m}$ una matriz de pesos y b un vector de sesgos. Esta neurona de salida puede devolver un vector de dimensión m que en cada entrada puede tomar cualquier valor real. Este tipo de neuronas puede resultar útil en caso de querer entrenar la red para resolver un problema de regresión.
- **Sigmoide**

  Otra posible salida aparece cuando buscamos que la red devuelva una probabilidad, por ejemplo si quisiéramos clasificar entre dos posibles categorías, nos serviría que la red estimara con que probabilidad es una de las dos categorías. Estaríamos modelando el problema como una Bernoulli y nuestra red lo que hace es estimar el parámetro.

  En ese caso podríamos pensar en usar una función lineal afín como en el punto anterior pero que en caso de recibir un valor menor a 0 devuelva 0 y en caso de recibir un valor mayor a 1 que devuelva 1, para que la imagen sea $[0, 1]$. Si bien esta idea nos daría una probabilidad tendríamos el problema de que una

vez que los valores son mayores a 1 o menores que 0 entonces tendríamos un gradiente **0** y esto es problemático ya que en para esos valores no podremos descender con el gradiente, y por lo tanto no podremos disminuir la función de pérdida. Por eso buscamos una función que normalice la salida pero que tenga valores de gradiente para poder aprender cuando la salida de la red sea lo que esperábamos y utilizar el algoritmo de descenso por gradiente.

Un ejemplo de este tipo de funciones son las **sigmoides**. Las sigmoides son una familia de funciones que tienen en común tener una forma curva en forma de S y que sus valores están entre 0 y 1, una de las sigmoides más comunes es la función logística que está definida de la siguiente manera:

$$\sigma(x) = \frac{1}{1 + e^{-x}}$$

Estas funciones normalizan entre 0 y 1 pero además tienen la característica de tener gradiente en todo $\mathbb{R}$.

Definimos una **neurona sigmoide** como la composición de una transformación lineal afín compuesta con la función logística:

$$\hat{y} = \sigma(W.h + b)$$

Podemos pensar que primero la neurona computa una transformación lineal y luego eso lo pasa por la sigmoide para obtener una probabilidad.

Interpretaremos que si la entrada de la red es un valor X luego la salida de la red con neurona sigmoide es: $\hat{y} = P(Y = 1|X)$

- ***SoftMax***

  Un problema que surge muy a menudo es el de buscar clasificar entre varias categorías, es muy común buscar clasificar imágenes según su contenido, o en problemas de procesamiento de lenguaje natural cuando buscamos clasificar el sentimiento de una oración, o al predecir la siguiente palabra podemos pensar el problema como intentar clasificar una oración con su siguiente palabra.

  En estos casos lo que esperamos de la red es obtener un vector de dimensión tan grande como la cantidad de categorías en las que queramos clasificar. En cada uno de los índices, el vector tendrá un valor entre 0 y 1 con la probabilidad de esa categoría de ser la clase de la entrada. Por ser un vector de probabilidades debe sumar uno. Para lograr todos estos requerimientos es que se utiliza la función Softmax. Podemos pensar esta función como una generalización de la sigmoide que discutimos en el punto anterior. La diferencia es que para una entrada X, en lugar de tener una única salida $\hat{y}$ tendremos todo un vector de probabilidades:

$$y_i = P(Y = i|X) \text{ con } i \in I$$

Siendo I el conjunto de las categorías posibles para la entrada X

La neurona softmax está definida de la siguiente forma:

Supongamos que estamos computando los últimos pasos de la red neuronal, tenemos los valores que están ingresando a la última capa a los que llamaremos h. Lo primero que hará la neurona será aplicar la transformación lineal afín definida por los pesos.

$$z = Wh + b$$

Este z es un vector de dimensión igual a la cantidad de categorías que tenemos para clasificar, donde cada uno de los valores nos dirá el "puntaje"que tiene la categoría correspondiente para ser elegida. Luego este vector de "puntajes"lo normalizamos para obtener un vector con las probabilidades de cada categoría. Esa normalización se hará a través de la función que le da nombre a la neurona:

$$Softmax(z)_i = \frac{e^{z_i}}{\sum_j e^{z_j}}$$

Está neurona de salida por lo general se utiliza con la función de pérdida cross-entropy. Esto tiene sentido porque como vimos, usamos cross-entropy cuando la salida de la red es una distribución de probabilidad, y minimizar esta función era lo mismo que maximizar la función de verosimilitud, un método muy utilizado para aproximar distribuciones.

4.4. Neuronas de capa oculta Como vimos antes para definir una capa debemos definir los pesos que definirán la parte lineal (estos son aprendidos con el gradiente) y también hay que determinar la función de activación que tendrán las neuronas de cada capa.

4.5. ReLU La elección de qué tipo de neurona poner en las capas ocultas no tiene al día de hoy resultados teóricos que la fundamenten. El proceso de elección se hace caso a caso a través de ensayo y error, aunque en la práctica se suele usar como función de activación la ReLU (Rectified Linear Unit), que es una función no lineal definida de la siguiente forma:

$$f(x) = \begin{cases} 0 & x \leq 0 \\ x & x < 0 \end{cases}$$

Problema de no diferenciabilidad en 0

Esta función tiene el problema de ser no diferenciable en 0, y esto podría causar problemas a la hora de utilizar descenso por gradiente cuando el gradiente no está definido. A efectos prácticos esto resulta no ser un problema, se puede definir la derivada como una de sus derivadas laterales y el algoritmo aún obtiene buenos resultados. Una justificación computacional sobre este fenómeno podría ser que realmente la representación numérica dentro de la computadora no es exacta y por lo tanto cuando la computadora utiliza el valor 0 en realidad está usando un valor muy pequeño al que redondea a 0.

Es útil como función de activación ya que tiene derivada 1 en la parte del dominio donde no es 0 y por lo tanto es buena para el algoritmo de descenso por gradiente.

Otro problema que podría presentar esta función es que el gradiente es nulo en los valores negativos, y esto podría limitar el aprendizaje si el valor que le llega a la función es negativo. Algunas generalizaciones de ReLU le aportan un gradiente cuando $x \leq 0$ aunque en la práctica no da resultados mucho mejores.

5. Arquitectura

Como mencionamos anteriormente las redes feedforward están compuestas de neuronas ordenadas en capas. Una vez que fijamos una función de costo, la cantidad de capas, el tipo y la cantidad de neuronas por capa tenemos una red lista para empezar a aproximar funciones de distribución condicional. Decimos que la profundidad de una

red es la cantidad de capas que tiene, y el ancho es la cantidad de neuronas por capa. A la hora de diseñar entonces debemos definir la función de costo, el tipo de neuronas: es decir cuál será la función de activación, y también la profundidad y el ancho que la red tendrá.

Para tomar estas decisiones de diseño hay algunos resultados teóricos y algunos empíricos. Aparecen resultados interesantes como el **Universal Aproximator Theorem**, que tiene varios enunciados posibles, pero el más general dice lo siguiente:

TEOREMA 1. *Sean $C(X, Y)$ el conjunto de funciones continuas de X a Y y $\sigma \in C(\mathbb{R}, \mathbb{R})$, donde σ se aplica coordenada a coordenada.*

Entonces σ es no polinomial si y sólo si $\forall n, m \in \mathbb{N}$, $K \subset \mathbb{R}^n$ compacto, $f \in C(K, \mathbb{R}^m), \varepsilon > 0$, $\exists k \in \mathbb{N}, W \in \mathbb{R}^{k\times n}, b \in \mathbb{R}^k, C \in \mathbb{R}^{k\times n}/$

$$\sup_{x\in K} |f(x) - g(x)| \leq \varepsilon$$

$$\textit{Donde } g(x) = C \cdot (\sigma \circ (W \cdot x + b))$$

□

Luego surge naturalmente la pregunta: si tenemos este resultado, ¿para qué preocuparnos en agregar más que una capa?

Lo que nos dice que una red con una única capa oculta tiene capacidad de aproximar cualquier función medible Borel si tiene suficientes neuronas en esa capa oculta. Eso no necesariamente significa que pueda hacerlo, porque el algoritmo de optimización podría no alcanzar los valores que se necesitan, por ejemplo quedando en un mínimo local. También puede ocurrir que no sea viable asegurar suficientes neuronas en la práctica.

El teorema nos dice que dada una función existe una red feedforward que aproxima bien la función, no dice que dado un dataset exista un algoritmo que nos permita generalizar los datos y obtener la función que aproxime la distribución de los datos.

TEOREMA 2. *Sea $f : \mathbb{R}^n \to \mathbb{R}^m$ lebesgue integrable y $\varepsilon > 0$*

Entonces existe una red neuronal feedforward con funciones de activación $ReLU$ con ancho exactamente: $d_m = max\{n + 1, m\}$ que cumple:

$$\int_{\mathbb{R}^n} \|f(x) - F(x)\|^p dx < \varepsilon$$

Más aún existe una f como antes y un ε para los cuales no haya una red feedforward $ReLU$ de ancho menor a d_m que satisfaga la condición de arriba

Este segundo teorema nos dice qué ocurre si limitamos la cantidad de neuronas en la capa oculta. Pero no pone un límite en la cantidad de capas de profundidad que puede incluir la red.

Y lo que ocurre es que dada una función f entonces podemos aproximarla por una red neuronal que tiene una cantidad fijada de neuronas en la capa oculta.

De estos dos teoremas podemos concluir que las redes neuronales tienen la capacidad de aproximar funciones si le permitimos tener la suficiente cantidad de parámetros, ya sean en las neuronas de una única capa o en varias capas.

6. Back Propagation

Back propagation o *back prop* refiere al algoritmo utilizado por redes neuronales para hacer el cálculo de gradiente que necesita para poder realizar la optimización de descenso por el gradiente.

Cabe aclarar que back prop refiere únicamente al cálculo del gradiente y no al resto del aprendizaje que sucede en la red con la optimización de los parámetros. Es más, el algoritmo se puede utilizar para calcular el gradiente de cualquier función.

Para describir el funcionamiento del cálculo del gradiente primero debemos introducir el concepto de grafo computacional:

Grafo computacional Un grafo computacional es un grafo dirigido, donde cada nodo representa una variable (podría ser un escalar, un vector, una matriz, etc.). Las aristas que tendrá el grafo representan las operaciones (como puede ser el producto, la suma, el producto vectorial) y utilizaremos estas operaciones elementales para definir funciones más complejas. Pondremos una arista desde el nodo x al nodo y si y la obtenemos aplicando una operación a x.

Esta definición es completamente general y no está vinculada a ninguna red neuronal. De hecho lo que haremos a continuación con el algoritmo es simplemente una implementación del cálculo del gradiente de una función utilizando la regla de la cadena.

Regla de la cadena Sea $f : \mathbb{R} \to \mathbb{R}$ y $g : \mathbb{R} \to \mathbb{R}$, $y = f(x)$ y $z = g(y)$.
Si $x \in \mathbb{R}$ y $h = g \circ f$ entonces

$$h'(x) = g'(f(x)).f'(x)$$

$$\frac{\partial z}{\partial x} = \frac{\partial z}{\partial y}\frac{\partial y}{\partial x}$$

En el caso un poco más general donde $f : \mathbb{R}^n \to \mathbb{R}^m$ y $g : \mathbb{R}^m \to \mathbb{R}$. La regla de la cadena dice:

$$\frac{\partial z}{\partial x_i} = \sum_j \frac{\partial z}{\partial y_j}\frac{\partial y_j}{\partial x_i}$$

Lo que es equivalente en notación vectorial a:

$$\nabla_x z = \left(\frac{\partial y}{\partial x}\right)^{\mathsf{T}} \nabla_y z$$

A continuación veremos el ejemplo de back prop aplicado a un ejemplo sencillo donde el grafo las únicas variables que tiene son escalares.

Sea G un grafo que calcula $u^{(n)}$a partir de los nodos $u^{(i)}$ con $i = 1, \cdots, n_i$. Queremos hallar el gradiente de $u^{(n)}$ respecto de sus inputs. Para eso utilizamos primero el algoritmo de forward prop que nos calculará el valor que tendrán todos los nodos del grafo G a partir de las entradas $x_1, \cdots, x_{n_i}$.

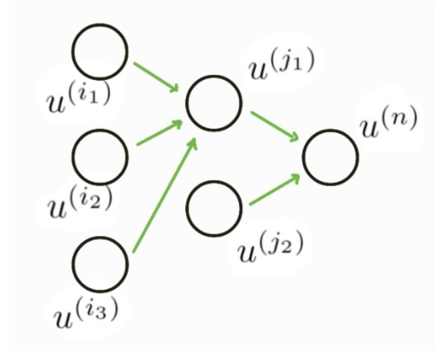

FIGURA 5. Ejemplo de grafo G con 3 nodos de input

Algorithm 1 Front propagation

---

for $i = 1, \cdots, n_i$ **do**
  $u^{(i)} \leftarrow x_i$
end for
for $i = n_i + 1, \cdots, n$ **do**
  $\mathbb{A}^{(i)} \leftarrow \{u^{(j)} | j \in Pa(u^{(i)})\}$
  $u^{(i)} \leftarrow f^{(i)}(\mathbb{A}^{(i)})$
end for
return $u^{(i)}$

Donde notamos $\mathbb{A}^{(i)}$ al conjunto de los nodos padres de $u^{(i)}$ y $f^{(i)}$ a la función que transforma a los padres de $u^{(i)}$ en $u^{(i)}$.

Habiendo calculado todas los valores de las variables en el grafo G solo nos queda calcular el gradiente respecto de los inputs.

Algorithm 2 Back propagation

`grad_table`$[u^{(n)}] \leftarrow 1$
for $i = n - 1, \cdots, 1$ **do**
  La proxima linea calcula $\frac{\partial u^{(n)}}{\partial u^{(j)}} = \sum_{i:j \in Pa(u^{(i)})} \frac{\partial u^{(n)}}{\partial u^{(i)}} \frac{\partial u^{(i)}}{\partial u^{(j)}}$
  `grad_table`$[u^{(j)}] \leftarrow \sum_{i:j \in Pa(u^{(i)})}$ `grad_table`$[u^{(i)}] \frac{\partial u^{(i)}}{\partial u^{(j)}}$
end for
return $\{$`grad_table`$[u^{(i)}] | i \in 1, \cdots, n_i\}$

Este segundo algoritmo es el que se encarga de calcular el gradiente. La regla de la cadena nos da una expresión analítica del gradiente, pero el problema es que a la hora de evaluarlo se vuelve lento por el hecho de que hay muchos términos que se repiten, y computarlos todas las veces sería muy ineficiente. Por esto es que se genera la estructura `grad_table` que es una tabla que irá almacenando las distintas derivadas que vayan apareciendo para tener que calcularlas una única vez. Este proceso tiene como

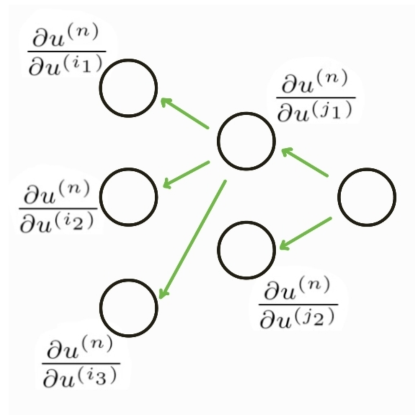

FIGURA 6. Ejemplo de grafo B con 3 nodos de input

desventaja que ocupa más espacio, pero si almacenar los números ocupa poca memoria entonces es una buena opción.

El segundo algoritmo se puede pensar en un grafo alternativo $\mathcal{B}$ que tiene la misma cantidad de nodos y aristas que el grafo G, cada nodo $u^{(j)}$ tiene el cálculo de $\frac{\partial u^{(n)}}{\partial u^{(i)}}$ y las aristas están en sentido opuesto, si en G tenemos arista del nodo $u^{(j)}$ al $u^{(i)}$ entonces en el grafo $\mathcal{B}$ tendremos una arista del i al j y estará asociada con el cálculo del término $\frac{\partial u^{(i)}}{\partial u^{(j)}}$. Esta forma de visualizar el algoritmo es lo que le da el nombre de propagación hacia atrás; hacia adelante computamos los valores de las variables y hacia atrás calculamos los gradientes que describimos antes.

En el caso particular de la red neuronal el valor $u^{(n)}$ será el valor que la función de costo le asocia a un determinado dato. Lo que nos interesa es calcular cómo son los gradientes respecto de los parámetros del modelo que los consideramos como nodos del grafo. Una vez calculados los gradientes podemos proceder al algoritmo de optimización.

A continuación tenemos un ejemplo concreto de una función representada como grafo computacional donde aplicamos el algoritmo de backpropagation:

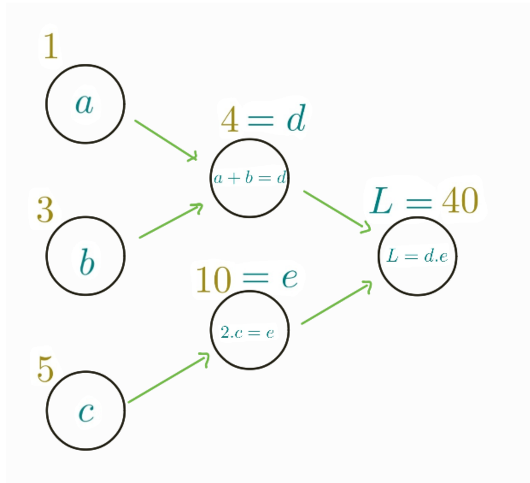

Figura 7. La función representada es $f(a, b, c) = 2c(a + b)$

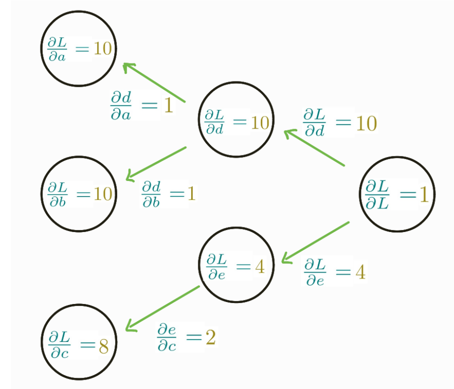

Figura 8. Calculo de back prop

7. Descenso de gradiente

Si tenemos un problema donde lo que buscamos es hallar el máximo o el mínimo de una función dada, nos encontramos frente a un problema de optimización. Existen muchas técnicas para resolver este tipo de problemas, la que indagaremos en esta sección es la de descenso por gradiente.

Una forma gráfica de entender la idea de descenso de gradiente es imaginando que estamos en algún terreno montañoso y buscamos llegar a un valle, o al nivel del mar. Para que el problema se parezca aún más al de optimización tenemos que imaginar que estamos en medio de una niebla que no nos permite ver mucho más allá que unos metros. Una estrategia razonable que podríamos seguir es buscar dentro de lo que podemos ver cual es el paso que podemos dar que nos lleve a estar más abajo de lo que estábamos antes. El problema es que una vez que llegamos a un lugar donde parecería que no podemos lograr nada mejor no podemos afirmar que ese lugar sea el punto más bajo del terreno.

Más formalmente, la premisa general es que si no conocemos la forma global de la función y buscamos encontrar un mínimo, entonces lo mejor que podemos hacer dado un punto en el gráfico es buscar localmente hacia donde decrece esta función. El comportamiento infinitesimal de la función lo podemos estudiar a través del gradiente. En particular, el gradiente de la función nos indica el sentido en el que la función tiene mayor crecimiento, por lo tanto si consideramos el gradiente del opuesto de la función, lo que nos indicará es el sentido donde decrece en mayor medida. La idea es buscar valores cada vez más pequeños en la función hasta encontrar un mínimo. Dado un punto x el método nos dará otro punto:

$$\hat{x} = x - \varepsilon \nabla_x f(x)$$

Y sabemos que $f(\hat{x}) \leq f(x)$ si ε es lo suficientemente chico.

El valor ε es lo que se conoce como ***Learning rate*** y este valor determinará el tamaño del paso con el que descendemos por el gradiente. Existen varias formas de tomar este valor. Podría ser fijar un valor constante pequeño para que se cumpla la desigualdad anterior, o podríamos tomar un valor distinto en cada iteración, o evaluar distintos valores y elegir el más pequeño. El método converge una vez que llegamos a un punto de gradiente 0 (en la práctica basta con que sea muy pequeño). Dependerá de la forma que tenga la función si ese punto al que converge es un mínimo global, un mínimo local o un punto silla. Cuando la función es convexa entonces si encontramos un punto de gradiente 0 este tiene que ser un mínimo global. La situación es menos clara cuando tenemos funciones con comportamientos no convexos, más complejos. Esas otras posibilidades (mínimos locales o puntos silla) son las que hacen al problema de optimización en redes neuronales un problema tan complejo.

8. Descenso de gradiente estocástico

8.1. Algoritmos de *Batch* Lo que suele suceder es que la función objetivo se puede escribir como la suma sobre los ejemplos que tenemos disponibles para el entrenamiento.

En el ejemplo de querer alcanzar la función objetivo a través del principio de máxima verosimilitud tenemos lo siguiente:

$$\hat{\theta} = \arg\max_{\theta} \sum_{i=1}^{N} \log(p_{model}(x^{(i)}, y^{(i)}; \theta))$$

Siendo $\{x^{(i)}, y^{(i)} : i = 1, \cdots, N\}$ los datos de entrenamiento

Si dividimos la expresión a maximizar entre N y consideramos p_N como la distribución empírica de los datos obtenemos que lo que queremos maximizar es:

$$J(\theta) = \frac{1}{N} \sum_{i=1}^{N} \log(p_{model}(x^{(i)}, (y^{(i)}; \theta)) = \mathbb{E}_{x,y\sim p_N} \log(p_{model}(x^{(i)}, y^{(i)}; \theta))$$

Una propiedad que se cumple es: Sabemos que calcular el gradiente es hacer la integral respecto de x e y, y el gradiente lo estamos calculando respecto a los valores de los parámetros. Por lo tanto es indistinto en que orden hagamos el gradiente o la esperanza, luego podemos escribir:

$$\nabla_\theta J(\theta) = \mathbb{E}_{x,y\sim p_M} \nabla_\theta \log(p_{model}(x^{(i)}, y^{(i)}; \theta))$$

Este es el gradiente que utilizaremos para el algoritmo de optimización.

La forma de calcular este gradiente es la que diferencia los distintos tipos de algoritmos.

Si calculamos la esperanza con todos los datos entonces tenemos un algoritmo dentro de la categoría de ***Batch***[2]*gradient methods*. Este método no es el más utilizado en la práctica ya que estimar el gradiente a partir de algunos ejemplos suele ser mejor por motivos que luego mencionaremos.

[2]Cabe aclarar que en distintos contextos la palabra batch se interpreta de formas distintas. Cuando aparece en el nombre del método se refiere a hacer uso de todos los datos, sin embargo cuando usamos la palabra sola, por lo general hace referencia a una porción de los datos.

Los algoritmos que estiman el gradiente utilizando un solo dato, se llaman ***Stochastic*** u ***Online***. El término online se utiliza cuando los datos se toman de un ambiente del que estamos aprendiendo, es decir no estamos entrenando con un dataset definido.

Y el caso más común es el que se encuentra entre estos dos extremos, usar algunos datos para estimar, no todos pero tampoco uno solo. Estos llevan el nombre: *algoritmos de* ***Minibatch***. Este tipo de algoritmos presenta algunas ventajas sobre los anteriores, en primer lugar usar solo algunos datos para estimar nos evita tener que hacer una pasada por todos los datos (que son potencialmente miles) y esto lo hace computacionalmente más viable. Aunque también es cierto que el cálculo de la esperanza es más preciso cuantos más datos utilizamos, de hecho la varianza del estimador de la media como variable aleatoria es $\sigma/\sqrt{n}$, siendo n la cantidad de elementos con los que estamos estimando. Observándose desde la perspectiva de la eficiencia, la varianza decrece en orden de $\sqrt{n}$, por ejemplo si calculamos un gradiente con 100 datos y otro con 10,000, el segundo necesita cien veces más capacidad de computación y solo mejora el resultado en un factor de diez. Por otro lado, usar todos los datos no necesariamente es deseable, ya que en una base de datos puede contar con datos repetidos o ciertas redundancias que no le aportan precisión pero sí lo hacen menos eficiente.

Este ruido introducido por la elección del batch también es positivo en tanto es menos probable caer en puntos silla. Esto solo ocurre si caemos en el punto de gradiente 0, un pequeño movimiento hacia alguno de los sectores inestables nos da como resultado que podemos seguir descendiendo por el gradiente. El ruido también aporta a no entrar en mínimos locales pequeños a los que podríamos entrar fácilmente si siguiéramos el gradiente exacto.

Una parte importante del algoritmo es definir el learning rate, antes mencionamos que se pueden elegir de varias formas. En el caso de los métodos batch, podemos determinar un valor constante y si es lo suficientemente chico llegaremos eventualmente a un punto de gradiente 0. En contraste, en los métodos minibatch, la aleatoriedad del muestreo agrega un ruido al cálculo del gradiente que hace necesario considerar un learning rate para cada iteración, y más específicamente que esos valores vayan tendiendo a 0, de este modo iremos convergiendo mientras disminuimos la cantidad de que influye el ruido del gradiente.

Cómo tomar estos valores es para nada trivial, si los tomamos muy pequeños nuestro algoritmo aprenderá lento y tendremos que hacer muchas iteraciones, pero si lo hacemos demasiado grande eso agrega mucha inestabilidad y difícilmente lleguemos a encontrar un mínimo.

No hay criterios claros para definir cómo ir tomando esta sucesión de ε_k, se suele probar distintas posibilidades e ir viendo cómo se comporta la función de costo para determinar cuál de los rates es mejor.

9. Redes neuronales recurrentes

Hasta ahora hemos presentado una única arquitectura de red neuronal. Esta recibía como entrada un determinado valor x y devuelve como salida $f_\theta(x) = y$ donde f_θ es la red feedforward. Recordamos que el nombre feedforward se debía a que la información en la red siempre fluía hacia adelante, hasta llegar a la salida. Una alternativa a esta arquitectura son las redes recurrentes. Estas se utilizan para modelar información en

Algorithm 3 Stochastic Gradient descend update

Require: Valores de learning rate $\varepsilon_1, \varepsilon_2, \cdots$
Require: Parametro θ inicial
$k \leftarrow 1$
while No se cumpla el criterio de parada **do**
Tomar una muestra de minibatch de tamaño m de los datos de entrenamiento $\{x^{(1)}, \cdots, x^{(m)}\}$ con sus correspondientes $y^{(i)}$
$\hat{g} \leftarrow \frac{1}{m} \nabla_\theta \sum_i L(f(x^{(i)}; \theta), y^{(i)})$
Actualizo: $\theta \leftarrow \theta - \varepsilon_k \hat{g}$
$k \leftarrow k + 1$
end while

secuencias de la forma $x(t)$, como podrían ser oraciones o eventos ordenados en el tiempo donde, para cada tiempo t tenemos un valor de x. La recurrencia nos permite que un elemento de cualquier momento de la secuencia influya en la respuesta de la red en un tiempo posterior, esto no ocurría en las feedforward, donde cada input de la red se procesaba de manera independiente y la salida no se ve influida por el orden en el que ingresamos los valores de x.

La red lleva el nombre de recurrente porque el estado de las activaciones de la red dependen de la entrada que reciben, pero también del estado de la red en un momento anterior

La forma en la que se genera esta recurrencia es teniendo una conexión de alguna parte de la red con sí misma. Las recurrencias podrían ser de tres tipos:

1. Una red que tenga una salida en cada tiempo, es decir que para cada $x(t)$ la red nos devuelva un valor $y(t)$ y que tengamos una conexión entre las capas ocultas. Como es el caso de la red ilustrada en la figura 9
2. Una red que tenga una salida en cada tiempo y que las conexiones sean entre la salida de la red en un tiempo y la capa oculta en el siguiente.
3. Recibir una secuencia entera y devuelven una única salida

El funcionamiento general es el mismo, cada neurona tendrá su correspondiente matriz de pesos y sesgos y contará con su función de activación. Es importante destacar que la recurrencia tiene sentido porque las matrices de pesos que utilizamos para multiplicar las entradas y los movimientos de la red no dependen del tiempo, sino sería simplemente una red más profunda o más ancha. Ya veremos que cuando actualizamos los pesos de la matriz estamos teniendo en cuenta su resultado en cada uno de los tiempos, no solo en el último. La diferencia es que existe una nueva flecha en el grafo que agrega una nueva matriz de pesos (que será la que generará la recurrencia) conectando un tiempo con el siguiente, es esta matriz la que nos permite decir que la red tiene una *"memoria"*, esta flecha hará que el estado de la red en el tiempo t influya en el momento $t + 1$.

Desde el punto de vista del grafo que utilizamos para representar la red recurrente lo que tendremos será una red como las que teníamos previamente. Para simplificar la imagen diagramamos cada capa como un único nodo, es decir que la capa de entrada de la red si bien tiene varias neuronas, la graficamos con un único nodo; lo mismo ocurrirá con las capas ocultas y la de salida. La gran diferencia con las feedforward será que algunos

nodos se apuntarán a sí mismos. La arista que forma un ciclo tendrá un cuadrado negro para indicar que cuando recorremos esta arista está pasando un tiempo.

Este tipo de grafo es una forma bastante compacta de diagramar el funcionamiento de la red. La desventaja que tiene es que puede ser algo difícil utilizarla para entender de qué manera se calcula el forwardprop o el backprop, por eso es que se puede llevar adelante un proceso de desplegar el grafo, es decir poner los nodos en los distintos tiempos, justo como se muestra en la figura 9.

La recurrencia es de interés ya que si las secuencias tienen información que se transmite en el tiempo no nos interesa perderla. Por ejemplo, si quisiéramos predecir el precio de un determinado bien, cuánto costaba ayer es importante pero también cuánto costaba la semana pasada. La forma de incluir esta noción en una red feedforward sería incluyendo toda esa historia en cada dato de entrenamiento, pero habría que prefijar cuanta historia queremos incluir, porque la entrada de la red siempre tiene el mismo tamaño. Las recurrentes nos permiten incluir cualquier cantidad de historia ya que es cuestión de incluir un tiempo más en la recurrencia.

9.1. Forward propagation Veremos un ejemplo de una pasada por una red recurrente para fijar ideas y convencernos de que el funcionamiento general es similar al de las redes que ya habíamos visto.

Desde ahora siempre que mencionemos una red recurrente, la recurrencia será como en el punto 1, con conexión entre las capas ocultas, y también supondremos que nos encontramos en un problema de clasificación, donde la salida de la red en cada tiempo es la probabilidad de cada una de las clases en las que buscamos clasificar.

Tendremos para procesar una secuencia $x(t)$ con tiempos entre $t = 1$ y $t = \tau$. La red comenzará con un estado inicial $h^{(0)}$.

Dada la secuencia y el estado inicial, para cada tiempo actualizaremos de la siguiente forma:

$$a^{(t)} = b + Wh^{(t-1)} + Ux^{(t)}$$

$$h^{(t)} = ReLU(a^{(t)})$$

$$o^{(t)} = c + Vh^{(t)}$$

Donde:
- U Matriz que conecta la entrada con la capa oculta
- V Matriz que conecta la capa oculta con la salida de la red
- W Matriz que conecta las capas ocultas en tiempo t y $t + 1$
- b, c vectores de sesgos
- $ReLU$ Función de activación introducida en 4.5

Estas son las actualizaciones que se hacen sobre el grafo de la figura 9

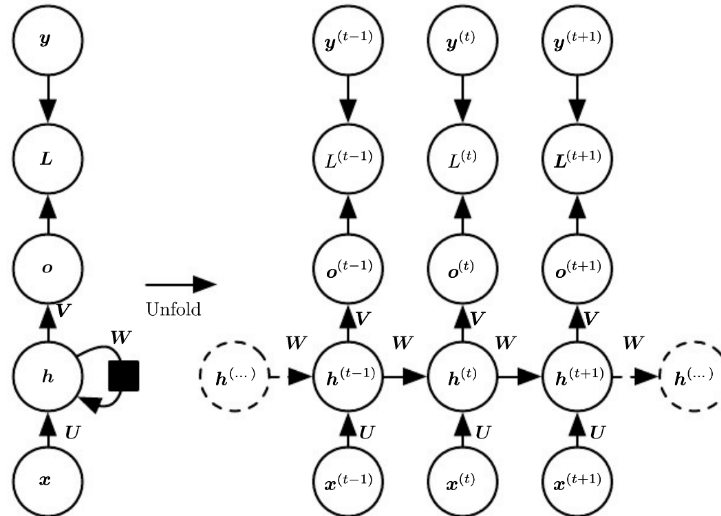

FIGURA 9. Ejemplo de red neuronal recurrente. Tenemos a la izquierda el grafo de la red "plegado" o sin desarrollar, con las capas de la red representadas como un único nodo. La recurrencia ocurre con la arista que nace del nodo h y apunta a ese mismo nodo. Luego a la derecha se despliega el grafo en los distintos tiempos. Notar que todas las matrices U, V, W son las mismas en la versión desplegada del grafo. El vector de sesgos no está incluido en la imagen para mantener la claridad. Notemos también que en la imagen no aparecen los valores $a^{(t)}$ sino que aparece directamente $h^{(t)}$. Imagen extraída del capítulo 10 de [**2**]

Viendo como se hace la actualización de los valores observamos que $h^{(t)}$ depende de $a^{(t)}$, que a su vez depende de $h^{(t-1)}$ y al mismo tiempo $h^{(t-1)}$ depende de $a^{(t-1)}$. Es decir que cada estado de la capa oculta depende en parte de cómo estaba la capa oculta en el tiempo anterior. La dependencia será determinada por la matriz de pesos W. Es esta dependencia a la que nos referimos cuando mencionamos al comienzo del capítulo al decir que la red tiene memoria. Una vez que la red está completamente entrenada tenemos que la capa oculta de la red 'recuerda' cómo estaba la red en los tiempos anteriores. Imaginando que la red se está utilizando en un contexto de procesamiento de texto, no es lo mismo ingresar a la red la secuencia: *"No quiero más"*, que la secuencia *"Quiero más"*. La existencia del *"No"* en el comienzo de la primer frase genera que la capa oculta de la red a la hora de procesar la palabra *"más"* sea distinta en el procesamiento de ambas frases, esto es algo deseable ya que el sentido de las frases es opuesto.

Lo que hacemos es seguir las flechas del grafo de la figura 9 aplicando las matrices y funciones de activación correspondientes, igual que se hacía con las feedforward.

9.2. Back propagation Recordemos que el algoritmo de back propagation lo introdujimos para grafos computacionales generales. Esta nueva arquitectura sigue definiendo un grafo computacional y por lo tanto al algoritmo que definimos en la sección de redes feedforward es perfectamente aplicable. Por tratarse de una arquitectura diferente

lleva el nombre de *Back Propagation Through Time (BPTT)*, pero no hay que perder de vista que el procedimiento es el mismo: recorrer el grafo en sentido opuesto al de las flechas y calcular las derivadas correspondientes en cada paso para así poder calcular el gradiente con la regla de la cadena.

Para el algoritmo de backpropagation es necesario definir una función de pérdida, que será la función a la que le calculemos el gradiente. Para esto supongamos que tenemos una red neuronal recurrente que recibe una secuencia $x^{(1)}, \cdots, x^{(\tau)}$, con sus correspondientes valores $y^{(1)}, \cdots, y^{(\tau)}$. Tomaremos la función de pérdida en el tiempo (t) como menos logaritmo de la verosimilitud en $y^{(t)}$ dado $x^{(1)}, \cdots, x^{(t)}$.

La **pérdida total** para la red recurrente quedará definida como la suma de las pérdidas en cada tiempo:

$$\begin{aligned} L\left(\{x^{(1)}, \cdots, x^{(\tau)}\}, \{y^{(1)}, \cdots, y^{(\tau)}\}, \theta\right) &= \sum_t L^{(t)} \\ &= -\sum_t log\left(p_{model}(y^{(t)}|\{x^{(1)}, \cdots, x^{(\tau)}\})\right) \end{aligned}$$

Donde el valor de $p_{model}(y^{(t)}|\{x^{(1)}, \cdots, x^{(\tau)}\})$ está dado por ver en el vector de probabilidades $\hat{y}^{(t)}$ que da como salida la red en el tiempo t y buscar en la entrada correspondiente a la categoría y.

Lo que podemos observar es que el gradiente que tenemos respecto de los parámetros está calculado considerando distintos tiempos de la red. Por lo tanto, al aplicar el descenso por gradiente, se actualizarán los pesos de las matrices teniendo en cuenta su desempeño en más de un tiempo. Esto implica una mejora en varios tiempos de la red.

Una de las desventajas de este método es que si bien el cálculo del gradiente es prácticamente idéntico al de la arquitectura anterior, tiene la dificultad de ser computacionalmente muy costoso, ya que el grafo se extiende con el largo de la secuencia que queramos computar. Un tiempo más implica recorrer el grafo una vez más para calcular ese forwardprop y una vez más para calcular backprop. Para secuencias largas se torna realmente complejo. Un intento de solución podría ser intentar paralelizar los cálculos, esto significa básicamente llevar a cabo varias pasadas en paralelo para ahorrar tiempo de cómputo, pero por la esencia de la arquitectura esto se torna imposible, ya que es estrictamente necesario tener calculadas los estados en todos los tiempos hasta t para calcular el estado en $t + 1$.

Capítulo 3

Procesamiento de lenguaje

> -¿De qué color es la camisa de ese chico?
> -Azul.
> -¿Qué quiere decir azul? Descríbelo.
> Reflexioné un momento, pero no encontré la forma de describirlo.
> -Entonces, ¿azul es un nombre?
> -Es una palabra. Las palabras son pálidas sombras de nombres olvidados. Los nombres tienen poder, y las palabras también. Las palabras pueden hacer prender el fuego en la mente de los hombres. Las palabras pueden arrancarles lágrimas a los corazones más duros. Existen siete palabras que harán que una persona te ame. Existen diez palabras que minarán la más poderosa voluntad de un hombre. Pero una palabra no es más que la representación de un fuego. Un nombre es el fuego en sí.
>
> El nombre del viento

1. ¿Qué es el procesamiento de lenguaje natural?

El procesamiento de lenguaje natural es un área del *machine learning* que tiene como objetivo hacer uso del lenguaje como fuente de información para realizar distintos tipos de tareas, extraer información de las palabras, y que no sean simples secuencias de letras. En los últimos años es un campo que ha ganado gran atención con avances sorprendentes, como puede ser el uso de asistentes virtuales como pueden ser *Siri, Alexa, el asistente de Google,* entre otros. Estas herramientas logran tener una interacción directa con el usuario. La persona se comunica con el programa hablando como si lo hiciera con una persona, y puede solicitarle una variedad diversa de tareas, desde poner una determinada canción, hasta programar una alarma, llamar un contacto de la agenda, controlar el calendario, entre otras tantas funciones. Otra aplicación reciente son los modelos generadores de texto GPT (1, 2 y 3).

Estos modelos se entrenan para generar texto de forma natural bajo diversas circunstancias, se pueden utilizar estos modelos para generar comentarios en foros, diálogos, poesía, y cualquier texto que requiera una alta capacidad de generación y emulación de la capacidad humana de generar texto.

El campo de la inteligencia artificial es el futuro, y el campo de la inteligencia artificial y la generación de texto juntos son la base de lo que será la inteligencia artificial en el futuro, el campo de la inteligencia artificial y la generación de texto están evolucionando rápidamente y están cambiando la forma en que pensamos y generamos texto, el campo de la inteligencia artificial y la generación de texto también están cambiando la forma en que la gente interactúa con la tecnología y la forma en que la tecnología interactúa con nosotros.

Estos últimos dos párrafos, sin ir más lejos, fueron generados automáticamente por *Generative Pre-trained Transformer 3 (GPT-3)* a través de la api de OpenAI.

2. Semántica de palabras

Las palabras, sean del idioma que sean, no son simples combinaciones de símbolos o caracteres, las palabras son la herramienta que surge como medio de comunicación entre los seres humanos. Cada palabra que se dijo alguna vez en la historia tenía de fondo una idea, un concepto, una definición o un sentido. Esto es lo interesante de las palabras, lo que hay dentro de ellas, lo que genera en las personas o seres que la reciben. No es el conjunto de letras {t,e,q,u,i,e,r,o} lo que desata un abrazo, ni es el conjunto de letras {d,i,s,p,a,r,e,n}, el que desata una guerra. Son los significados de las palabras los que son útiles en el mundo. Solamente puedo transmitirles estas ideas que han leído hasta ahora porque cada conjunto de letras que leen despierta en sus cabezas una idea, y es por eso que alguien que no hable español no podrá llevarse nada de este trabajo.

Las palabras tienen varias cualidades que nos interesan a la hora de modelar y entender el lenguaje. A nivel sumamente intuitivo podemos afirmar que existe una cierta similitud entre las palabras, las palabras *“soñar”* y *“dormir”* son similares, o las palabras *”perro”* y *“gato”* son similares a pesar de no ser sinónimos. También podemos estar de acuerdo que *“escuchar”* y *“mesa”* no son palabras similares.

Más allá de la similitud entre las palabras existen palabras que están relacionadas entre sí, aunque no sean similares, como pueden ser *“país”* y *“presidente”* o *“cama”* y *“dormir”*. Estas palabras refieren a objetos, o personas o acciones, algunas a cosas concretas y otras a cosas abstractas, sin embargo sabemos que algunas de ellas están vinculadas, las podemos asociar.

Pueden estar asociadas por pertenecer a un mismo campo semántico, es decir: formar parte de un conjunto de palabras que refieren a algún dominio específico. Como podría ser el campo semántico de la medicina. Las palabras *“Fisiopatología”, “cirujano”, “posología”* o *“anestesia”* sabemos que están vinculadas aunque no sepamos exactamente qué significan ni cómo usarlas.

Otra cualidad interesante de las palabras es que cuentan con connotación, es decir están vinculadas a una sensación o sentimiento. Puede ser una connotación positiva, como *“feliz”* o una negativa como *“triste”*.

Cuando decimos queremos modelar el lenguaje, pensamos en un modelo que logre captar todas estas nociones de significado que tienen las palabras, para poder utilizarlas correctamente.

3. Modelo de espacio de vectores

Los modelos de espacio de vectores para modelar palabras son una idea que surge como encuentro de otras dos ideas:

- La idea de que podemos ordenar palabras en el en un espacio según su connotación
- Y la idea de que podemos entender una palabra por el uso que se le da en el lenguaje. Un ejemplo de esta idea se da en el juego *Tipoteo.* El juego consiste en que una persona cambia un verbo cualquiera por la palabra tipoteo (y sus conjugaciones), el resto de los jugadores deben adivinar cuál era el verbo original.

Para esto los jugadores deben ir haciendo preguntas y usando la palabra tipoteo en distintos contextos para poder usar esta idea de que podemos entender en qué verbo piensa el dueño de la palabra tipoteo a pesar de que haya inventado la palabra para el juego.

A veces puede hablarse de *embeddings* de palabras que son una correspondencia uno a uno entre las palabras que queramos modelar y los vectores de un cierto espacio, esta correspondencia la derivamos según el uso que tengan las palabras en el idioma. Pero se suele usar embeddings cuando los vectores del espacio son densos (pocas dimensiones) como veremos un poco más adelante.

4. *One-hot encoding*

Un primer intento de resolver el problema es poner cada palabra en un vector haciendo que todos los vectores sean ortogonales entre sí. Es decir usando un espacio de dimensión $|V|$ (el tamaño del vocabulario), ordenar todas las palabras y luego indexar. A la palabra con el índice 4 le corresponderá el vector $(0, 0, 0, 1, 0, \cdots, 0)$. El método lleva el nombre *One-hot encoding*

Claramente este método no es una opción muy eficiente para trabajar con el lenguaje, tiene muchísimas dimensiones y no tenemos ninguna información sobre cómo se relacionan las palabras entre sí. De todas formas lo mencionamos porque se utiliza como herramienta para otros métodos que mencionaremos más adelante.

La construcción de modelos más sofisticados suele caer en una de dos categorías: Métodos de distribución o conteo, y métodos predictivos

5. Métodos de distribución

Los métodos de distribución basan su correspondencia entre palabras y vectores en algún tipo de matriz de co-ocurrencia en algún conjunto de documentos. Estos modelos están todos basados en grandes corpus de texto que contienen las palabras que queremos modelar, además de ser la fuente de información de cómo esas palabras son usadas.

Matriz Términos-Documentos

Esta matriz es una de la forma $A \in M_{n \times m}$ dónde n es la cantidad de palabras en el vocabulario que queremos modelar y m la cantidad de documentos que tenemos para considerar en el modelo. La entrada $a_{i,j}$ contiene la cantidad de veces que ocurre la palabra i en el documento j. Esta matriz de co-ocurrencia nos permite dar una representación vectorial de los documentos, en un espacio de dimensión tan grande como palabras incluyamos tomando como vector del documento j su correspondiente columna en la matriz, esta representación es conocida también como bolsa de palabras o ***BOW*** por su sigla en inglés ***bag of words***, lleva este nombre porque en la representación no existe ninguna referencia al orden en el que se usaron las palabras, simplemente se contabilizan, como si estuvieran justamente en una bolsa. Esta representación intuitivamente nos dará una noción acertada de similitud entre los documentos, si dos documentos utilizan muchas palabras muy similares y una cantidad similar de veces entonces esos documentos tiene sentido afirmar que son documentos similares.

Esta misma noción de similitud se puede aplicar a palabras, podemos pensar que dos palabras son similares si se utilizan de manera similar en muchos documentos.

Matriz Términos-Términos Otra matriz que podemos definirnos es considerando las palabras únicamente. Está matriz $A \in M_{n \times n}$ tendrá en la entrada i, j un valor que indicará la cantidad de veces que ocurren las palabras i y j en un mismo contexto. El contexto es algo que debemos definir previamente, podría ser una ventana de 3 palabras (es decir que la palabra i se utilice antes o después de la j), una de 5 palabras, o hasta todo un documento.

	Esto	es	un	ejemplo	de	matriz
Esto	1	1	0	0	0	0
es	1	1	1	0	0	0
un	0	1	1	1	0	0
ejemplo	0	0	1	1	1	0
de	0	0	0	1	1	1
matriz	0	0	0	0	1	1

FIGURA 1. En este ejemplo ilustramos una matriz en un caso simple donde todo el corpus es la frase *"Esto es un ejemplo de matriz"*, la ventana de contextos es de tamaño 3.

"Dime con quién andas y te diré quien eres." Si dos palabras ocurren en contextos similares, podemos entender que las palabras tienen significados similares.

6. Medida de similitud

Una buena pregunta es: ¿cómo podemos medir esta similitud de las palabras en este espacio?

Queremos una medida que tome dos vectores (ya sean documentos o palabras) y que nos devuelva un número que indique que tan similares son. Lo más común es recurrir al producto interno, para generar esta medida. El producto interno a secas tiene el problema de que devuelve valores más grandes para vectores más grandes, y esto no queremos que suceda ya que a priori dos palabras no son más similares porque se utilicen más. Lo que buscamos es normalizar por la norma de los vectores.

Por esto usamos la medida de similitud a partir del **coseno**:

$$\frac{\langle v_i, v_j \rangle}{\|v_i\| \|v_j\|} = \cos \theta$$

El coseno del ángulo entre los vectores es la medida de similitud más utilizada. Claramente no es una distancia debido a que dos palabras podrían tener $\cos(0) = 0$ pero no ser la misma palabra.

Otra medida de similitud que se utiliza en el campo de la teoría de la información es la *Pointwise mutual information (PMI)*. En el caso de querer ver la similitud entre una

palabra w y su contexto c se calcula como:

$$PMI(w,c) = \log_2 \frac{P(w,c)}{P(w).P(c)}$$

El numerador es la probabilidad de que ocurran w en la ventana de c y eso lo estimamos como cuenta la cantidad de ventanas donde aparecen w y c sobre la cantidad de ventanas. El denominador tiene el producto de las probabilidades de las palabras, es decir la probabilidad de que ocurran en un mismo contexto suponiendo que son independientes. Esto nos da una medida de que tanto ocurren las palabras de forma no independiente. Si ocurren en conjunto más de lo esperado podemos decir que las palabras están relacionadas, pero si aparecen menos de lo esperado entonces se podría interpretar que tienen una correlación negativa y la medida tendría signo negativo. Pero se podría dar que las palabras aparezcan poco en un mismo contexto justamente por ser muy similares, por ejemplo dos sinónimos no suelen aparecer en un mismo contexto (si consideramos una ventana pequeña). Como el signo de la correlación puede resultar ambiguo en su interpretación consideraremos solamente la parte positiva de la función PMI, y le llamamos *Positive pointwise mutual information (PPMI).*

$$PPMI(w,c) = max(PMI(w,c), 0)$$

Con esta medida de similitud podemos obtener una transformación de la matriz término-término.

7. Embeddings

Los modelos descritos hasta ahora son modelos esparsos, es decir que tienen muchas dimensiones y en muchas de ellas las entradas del vector es 0 (o un valor muy pequeño). Ya sea en la matriz de términos vectores o en la de términos términos es muy probable que en de todas las palabras en el vocabulario que consideremos la mayoría no aparezca en todos los documentos o no aparezcan en el mismo contexto que otro término.

Por eso se introduce otra forma de pensar vectores, que tengan mayor densidad: pocas dimensiones (comparado con la totalidad de palabras del vocabulario) y la mayoría no nulas. Estas dimensiones ya no tendrán interpretaciones claras. Pero a pesar de esto resulta que en la práctica obtenemos mejores resultados con estos vectores más pequeños.

Si bien no está claro exactamente porque esto es así, existen algunas intuiciones de porqué esto sucede: Usar vectores largos nos lleva a que los modelos tengan una mayor cantidad de parámetros y por lo tanto son más propensos a sobre-ajustar. Otra intuición es que al tener menos dimensiones, el modelo puede captar mejor los sinónimos de una palabra, con más dimensiones.

Si bien se pueden lograr representaciones densas a partir de los modelos de distribución utilizando métodos de reducción de dimensionalidad, los modelos densos más utilizados son los modelos de predicción.

8. Métodos de predicción

Estos métodos como su nombre indica, están basados en la predicción de palabras como veremos a continuación.

8.1. Modelo de lenguaje con redes feedforward Estos modelos constan de una red feedforward. El objetivo de la red será tomar L palabras y predecir la siguiente. Estas palabras para poder ser procesadas por la red necesitan una representación numérica. Usaremos el método de one-hot encoding ya que esto nos servirá para aprender embeddings una vez que esté entrenada la red para predecir la siguiente palabra.

La red consta entonces de $L \times |V|$ neuronas de entrada, siendo $|V|$ el tamaño del vocabulario, y como salida la red devuelve una distribución de las probabilidades de cuál será la siguiente palabra. Por lo tanto la capa de salida tiene $|V|$ neuronas y para que lo que obtengamos sea una distribución utilizamos la función de activación softmax que nos normaliza la salida para que sume 1.

Usando una fuente grande de texto entrenaremos esta red. Cómo estamos tomando los vectores de input en su representación one-hot, tenemos un 1 en el índice de la palabra y un 0 en el resto. Recordemos que la primera capa recibe únicamente una combinación lineal de la capa de entrada, esta combinación la podemos ver cómo multiplicar el vector de entrada por una matriz de pesos W. Cómo multiplicamos todas las columnas por 0 excepto la del índice correspondiente a la palabra que estemos ingresando, entonces esta primera operación podemos pensar que está simplemente seleccionando una representación de la palabra, ya que esa columna solamente aparece en la red si estamos poniendo como entrada esa misma palabra. La matriz de pesos será una matriz $d \times |V|$ siendo d la cantidad de neuronas de la capa oculta. Como los vectores que usaremos como embeddings son las columnas de esta matriz de pesos, entonces la dimensión de los vectores será d, y esa dimensión la elegimos cuando diseñamos la red.

Estamos simultáneamente resolviendo dos problemas: aprendemos a predecir la siguiente palabra y al mismo tiempo aprendemos los embeddings de las palabras.

El problema que tiene este método es que puede ser ineficiente desde un punto de vista computacional.

9. *Word2Vec*

Word2Vec ([**6**]) no es exactamente una método en sí mismo, es más bien una implementación de dos algoritmos: *Skip-gram* y *Continuous bag of words (CBOW)* que están inspirados en el modelo de lenguaje con redes feedforward pero con algunos cambios para hacerlo más eficiente. Ambos modelos tendrán como objetivo generar una representación vectorial de las palabras del vocabulario V.

Para lograr esto ambos algoritmos entrenarán redes neuronales para resolver un problema de clasificación, que será útil para que el entrenamiento de la red nos aporte buenos vectores asociados a las palabras. Entendemos que la representación de las palabras en vectores es buena si dos palabras que se utilizan de manera similar en el lenguaje, es decir, en contextos similares tienen vectores cercanos.

9.1. Skip-gram Como dijimos antes tenemos un problema de clasificación auxiliar que nos ayudará a que está red aprenda los vectores. Para skip-gram el problema de clasificación será: recibir un palabra que llamaremos palabra **objetivo** (está será la palabra de la que aprenderemos el vector) y la red deberá devolvernos cual palabra espera que se encuentre en un mismo contexto con esta palabra objetivo.

La red neuronal que usaremos tendrá dos capas ocultas. Recordamos que las capas de la red neuronal son una función de la capa anterior, esta función cuenta con una parte lineal y una parte no lineal que lleva el nombre de función de activación.

Arquitectura Skipgram:

- La red tomará como entrada un vector one-hot con la palabra objetivo
- **Primer capa**: La parte lineal será la multiplicación por una matriz $W \in M_{d \times |V|}$ (donde $|V|$ es el tamaño del vocabulario) y tendrá como función de activación la identidad. Es por eso que esta primer capa se llama capa de proyección, al multiplicar W con el vector one-hot obtendremos una de las columnas de W, una columna de la matriz por palabra del vocabulario.
- **Segunda capa**: La parte lineal será multiplicar por una matriz $C \in M_{|V| \times d}$ y tendrá como activación la función softmax, de esta forma la salida de la red es un vector con $|V|$ dimensiones que tiene las probabilidades de que cada palabra del vocabulario se encuentre en el contexto de la palabra objetivo.
- Por último nos quedaremos con la palabra de mayor probabilidad.

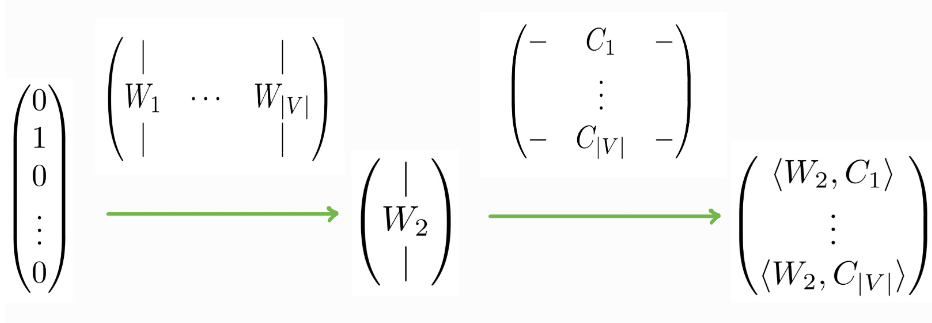

FIGURA 2. Modelo Skipgram. En la primer capa multiplicamos por la matriz W por el vector onehot, y obtenemos la columna que corresponde al índice del vector de entrada. Luego multiplicamos eso por la matriz C y obtenemos el vector de los productos internos entre la columna de la matriz W que obtuvimos en la capa pasada con cada una de las filas de la matriz C. Por último aplicamos la función softmax para obtener de este último vector, un vector de probabilidades

Para el entrenamiento tomaremos como datos los contextos de las palabras, pero no de cualquier forma. Consideraremos una ventana contexto que tendrá un tamaño R, un valor aleatorio entre 1 y S. Consideraremos el contexto R palabras delante de la objetivo y R detrás. Una vez sorteado ese número R tomaremos como ejemplo de entrenamiento el par *(palabra objetivo, palabra contexto)*.

FIGURA 3. Ejemplo de pares objetivo, contexto con $R = 2$

Para interpretar cuál es la idea detrás de este método observemos que la matriz W tiene una columna por cada palabra del vocabulario y C tiene una fila por cada palabra. Al introducir una palabra en la red en la primer capa tenemos un vector de dimensión d que solo aparece cuando se introduce esa palabra, ya que tenemos una correspondencia uno a uno entre palabras del vocabulario y columnas en la matriz.

Por otro lado tenemos que en la segunda capa tenemos una fila de C por cada palabra del vocabulario y las probabilidades serán aplicar softmax al producto interno de las filas de C con las columnas de W. Podemos interpretar que lo que está haciendo nuestra red es calculando el producto interno entre la representación de la palabra objetivo y la de la palabra contexto. El entrenamiento de la red hará que el producto interno entre dos palabras que aparecen en contextos similares es alto (y así la red devolverá que una probabilidad alta de encontrar esas palabras en un mismo contexto), es decir que los vectores son similares. Y de está forma obtenemos que la representación que se hace la red de las palabras respeta esta idea de que dos palabras que se usan en contextos similares tendrán vectores similares.

Lo que nos interesará son las columnas de la matriz W, serán estos los vectores que utilizaremos como embeddings de las palabras correspondientes.

También vemos que dos palabras que se usan en contextos similares deberán activar la red de forma similar (para dar probabilidades altas a las palabras que están en ese mismo contexto). Y de esta forma obtenemos que esos vectores que seleccionamos para las palabras tienen representaciones similares. Por ejemplo *"El perro está comiendo"* es una frase que puede aparecer de manera frecuente en el lenguaje, al igual que *"El gato está comiendo"*. A esto se refiere que las palabras se utilicen en contextos similares, esta similitud en el uso hará que las activaciones de la red tengan que ser parecidas para poder dar como salidas correctas las palabras *Está* y *comiendo*.

9.2. CBOW *Continuous bag of words* es un algoritmo similar al de skipgram que presentamos en el punto anterior. El objetivo volverá a ser asignar una probabilidad a una

palabra dependiendo del contexto en el que se encuentre. Pero será distinta a skipgram en tanto el problema de clasificación anterior era clasificar una palabra objetivo en una palabra contexto, y en este caso el problema es clasificar el conjunto de palabras contexto en la palabra objetivo.

Arquitectura CBOW:

- La red tomará como entrada una lista de vectores one-hot con las palabras objetivo. Y será la suma de estos vectores lo que ingresará a la red.
- **Primer capa**: La parte lineal será la multiplicación por una matriz $W \in M_{d\times|V|}$ (donde $|V|$ es el tamaño del vocabulario) y tendrá como activación la función softmax para devolver una probabilidad por cada palabra como sucedía con skipgram.
- Por último nos quedaremos con la palabra de mayor probabilidad.

La diferencia con skipgram es que en el caso anterior, la red recibía una palabra objetivo y devolvía la probabilidad de que las palabras del vocabulario se encuentren en el contexto; en este algoritmo lo que recibe la red como entrada son los vectores de un contexto y la red devuelve la palabra que se encuentra en ese contexto. Skipgram tomaba una palabra y CBOW toma varias, la cantidad dependerá de cuántas palabras indiquemos que tiene el contexto.

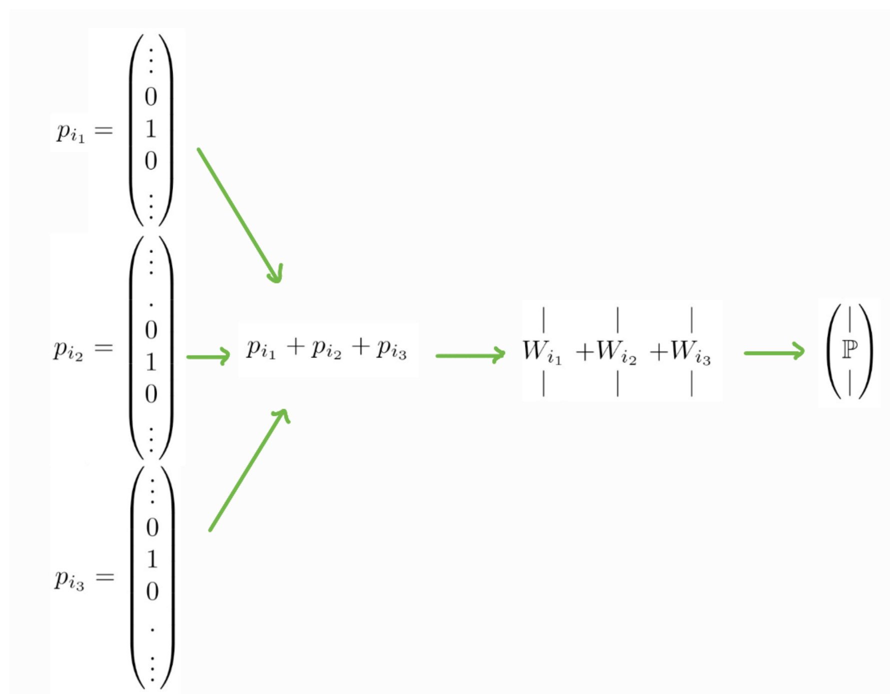

Figura 4. Modelo CBOW. Tomamos las palabras contexto y usaremos como entrada de la red la suma de los vectores. Al multiplicar por la matriz W obtenemos la suma de las columnas y luego aplicando softmax obtendremos las probabilidades correspondientes.

Esto	es	un	posible	ejemplo	de	contextos

(Esto, es,posible, ejemplo), un

Esto	es	un	posible	ejemplo	de	contextos

(es, un, ejemplo, de), posible

Esto	es	un	posible	ejemplo	de	contextos

(posible, ejemplo, de), contextos

Figura 5. Ejemplo de pares (contexto, objetivo) con una ventana de contexto de tamaño 2

Al comienzo del capítulo cuando hablamos de la vectorización de los documentos con la matriz de términos-documentos, dijimos que esos vectores se los llamaba también bolsa de palabras (bag of words) porque no teníamos en cuenta en qué orden se utilizaban esas palabras. Aquí pasa algo similar, recibimos como entrada todos los vectores de las palabras contexto pero luego sumamos todos y eso nos hace perder noción del orden, de aquí que el nombre incluya el término bolsa de palabras. Y decimos que es continuo por la forma en la que consideramos los elementos para entrenar. Iremos avanzando en el texto una ventana, y en la ventana siempre tendremos palabras que serán contexto y una de ellas será la palabra objetivo que entrenaremos a la red para que devuelva.

Esta es otra de las diferencias con skipgram, en el algoritmo anterior tomábamos pares de palabras (objetivo, contexto) y cuantas palabras entrarían en el contexto dependía de la ventana de contexto que se sorteaba. En este caso, tomaremos pares (contexto, objetivo) fijando un tamaño de ventana e iremos desplazándonos y entrenando mientras avanzamos la ventana, como ejemplifica la figura 5.

Ambos algoritmos producen vectores de alta calidad, en el sentido de que capturan bien el significado de las palabras. Esto podemos verlo en que realizar operaciones con los vectores nos da resultados que esperaríamos en las palabras, es decir que semánticamente las operaciones tienen sentido.

Un ejemplo muy conocido es el siguiente:

$$vec(\text{``}Reina\text{''}) \approx vec(\text{``}Rey\text{''}) - vec(\text{``}Hombre\text{''}) + vec(\text{``}Mujer\text{''})$$

Si miramos las palabras la operación podría pensarse como el razonamiento: Hombre es a Rey lo que Mujer a Reina.

También vale que:

$$vec(\text{``}Montevideo\text{''}) \approx vec(\text{``}Madrid\text{''}) - vec(\text{``}España\text{''}) + vec(\text{``}Uruguay\text{''})$$

Este tipo de relaciones que preservan los vectores es lo que muestra que algún significado está quedando registrado en la representación vectorial. Viendo estas operaciones es donde se motivó la última sección de la monografía, ¿será posible usar estos vectores para jugar un juego donde el uso del lenguaje es fundamental?

9.3. Optimizaciones de los modelos Luego de propuestos los modelos originales surgen algunas optimizaciones que buscan mejorar la calidad de los vectores y la eficiencia de los algoritmos.

Las principales optimizaciones son:

- Agregar algunas frases muy repetidas como un único vector en el vocabulario.
- No muestrear tanto palabras muy frecuentes.

Algunos conjuntos de palabras aparecen siempre en conjunto, como pueden ser nombres lugares (Fray Bentos, Bella Vista) o de personas o conjuntos (Ruben Rada, vela puerca). Estas frases se utilizan casi siempre como una única palabra, y tiene sentido unificarlas en un único vector. La idea es calcular algo similar al PMI de la frase para detectar estas frases:

$$score(w_i, w_j) = \frac{cantidad(w_i w_j) - \delta}{cantidad(w_i) \times cantidad(w_j)}$$

El δ aparece para contrarrestar el efecto que pueden tener palabras que aparezcan una única vez, tendrán un score alto con la palabra que viene antes si es poco común también, al restarle esa constante estamos pidiendo que las palabras que consideramos para frases aparezcan más que esa cantidad definida por el parámetro. Y definiremos que el conjunto de palabras se agrega como frase al vocabulario si supera un determinado score prefijado.

El segundo punto busca evitar que palabras como *"y"*, *"entonces"*, *"luego"*, *"es"*, entre otras aparezcan demasiadas veces en el entrenamiento ya que se consideran ruido por el hecho de que aparecen en muchos contextos y por lo tanto no aportan demasiada información. Además se observa que las representaciones vectoriales no cambian demasiado si estas palabras comunes aparecen poco en el entrenamiento. La probabilidad con la que definimos si una palabra debe aparecer menos en el entrenamiento depende fuertemente de la frecuencia con la que la palabra aparezca y está determinada por la siguiente fórmula:

$$P(w_i) = 1 - \sqrt{\frac{l}{f(w_i)}}$$

Donde $f(w_i)$ nos indica la frecuencia con la que se observa la palabra w_i y l es un parámetro que servirá para indicar que tanta frecuencia consideramos como ruido (por lo general es cerca de 10^{-5}).

Capítulo 4

Codenames

1. Introducción

Desde el comienzo de la inteligencia artificial, los juegos han sido un buen lugar donde encontrar desafíos y marcos interesantes para testear distintos modelos.

Existen muchos ejemplos de inteligencias artificiales jugando a distintos juegos muy variados. Uno de los más conocidos ejemplos es el de *AlphaGo*: una inteligencia artificial que no solo logró jugar al complejo y antiguo Go, sino que logró vencer al campeón mundial, también está el icónico caso de *DeepBlue*, una computadora venciendo al campeón mundial de ajedrez.

La motivación de está monografía fue buscar de qué manera se podría implementar un jugador del Codenames.

En este capítulo explicaremos el juego y por qué resulta un problema interesante de resolver. Luego explicaremos los aportes realizados en el paper, presentando una algoritmo concreto para la implementación de un jugador de codenames haciendo uso de la herramientas presentadas en la sección pasada, como lo son los embeddings de palabras. Concluimos con algunos resultados empíricos sobre la efectividad de los modelos probados en jugadores humanos. Cabe destacar que en la propuesta de solución del juego no aparecen explícitamente las redes neuronales, sino que la solución está más bien basada en el uso de los embeddings, y es en la creación de estos que aparecen las redes.

2. El Juego

El *codenames* (o código secreto en español) es un juego de caja en el que dos equipos se enfrentan. Dentro de cada equipo existen dos tipos de jugadores, el *spymaster* (o espía) y el agente. Para comenzar el juego se despliegan 25 cartas con palabras dispuestas en 5 filas de 5 cartas formando una grilla de palabras. Los espías de cada equipo recibirán una tarjeta (una sola carta por partida), que solo ellos pueden ver, que indica a través de colores los tipos de cartas que pusieron en grilla. Hay cuatro tipos de cartas:

- Del equipo azul
- Del equipo rojo
- Neutras
- Una carta negra, la carta bomba

El objetivo de cada equipo es adivinar cuales son las palabras que le corresponden, el primer equipo en conocer sus palabras será el ganador. Para lograr esto, por turnos, el espía de cada equipo debe dar una palabra, y un número. La palabra será la pista que los agentes de su equipo deberán interpretar para poder adivinar sus palabras. El número será la guía de los agentes para saber cuántas palabras son las que el espía busca

FIGURA 1. Ejemplo de tablero desde el punto de vista del espía, extraído de la página del juego.

sugerir. Una vez que el espía elige una pista, los agentes deben seleccionar de forma ordenada las palabras del tablero que creen que pertenecen a su equipo. Si la primer palabra elegida es de su equipo entonces puede elegir una segunda palabra y así tantas palabras como su espía haya sugerido en el número. En caso de cometer un error en la elección de la palabra, es decir que el agente elija una palabra y esta no sea de su equipo pueden ocurrir tres cosas: La palabra que eligió es neutra, en ese caso es el turno del otro equipo; la palabra es del equipo contrario, en este caso también será el turno del otro equipo pero además la palabra se contará para el equipo que le correspondía; un tercer caso ocurre cuando la palabra seleccionada es la palabra bomba, aquí lo que ocurre es que el equipo que eligió esta carta queda automáticamente descalificado.

El juego plantea un problema interesante para el campo del procesamiento de lenguaje natural, porque la tarea a realizar requiere un manejo del lenguaje muy completo. Dar una buena pista es un problema difícil incluso para jugadores humanos que manejamos el lenguaje. Una buena pista debe 'estar cerca' de las palabras que busca sugerir, o estar vinculadas lo suficiente como para ser una buena sugerencia. Estar cerca en la práctica del juego requiere manejar varias acepciones tanto de la palabra que queremos usar de pistas como también de todas las palabras del tablero para verificar que la pista sugiera palabras de mi equipo y también que no sugiera otra palabra que pueda generar confusión.

La tarea que lleva adelante el agente (el que recibe la pista) no resulta tan difícil de implementar utilizando la noción de distancia que tienen las palabras con sus embeddings, la tarea una vez que tenemos los vectores asociados a las palabras es simplemente buscar los vectores del tablero que se encuentran a menor distancia de la pista sugerida.

Lo más complejo es lograr sugerir una palabra como pista. Tiene la dificultad de tener que elegir cuales palabras son las que queremos que el agente adivine, y simultáneamente que estas palabras se encuentren lejos de otras palabras del tablero que no queremos que el agente elija, ya sean del otro equipo, neutras o la bomba. Por ejemplo en la imagen 2

una buena pista para el equipo azul, sería la palabra *"PiedraLisa"* y sugeriría 3 palabras azules: *Atlántida, Baile* y *Noche*, pero está pista solo es buena si los jugadores del equipo azul saben que en Atlántida hay un establecimiento bailable bajo ese nombre. generar y entender esa pista tiene la dificultad de tener un conocimiento integral sobre esas palabras para poder ver que *"PiedraLisa"* vincula las tres. Una pista para el equipo rojo podría ser: *"Cerradura"* sugiriendo las palabras *"Casa"* y *"Seguro"*, esta pista aparece como una opción si consideramos como definición de *"Seguro"*, el sentimiento de seguridad y además podemos interpretar que la cerradura nos transmite ese mismo sentimiento. Y que una *"Casa"* utiliza cerraduras. A toda esta complejidad se le agrega el hecho de que algunas palabras pueden verse sugeridas de manera equivocada, por ejemplo si fuéramos del equipo azul y diéramos la pista *"Frío"*, podríamos entender que está intentando referirse a *"Guante"* y *"Noche"*, pero también podríamos pensar que Alemania es un lugar donde hace frío.

Un método que se utiliza para entrenar modelos que puedan jugar a juegos es implementar dos jugadores virtuales, por ejemplo con redes neuronales. Luego poner a ambos modelos a competir y de esta forma mejorar ambos modelos jugando en conjunto. Los datos con los que entrenamos surgen de la propia interacción entre los modelos, esto nos da la posibilidad de tener una fuente casi ilimitada de datos que nos permiten entrenar.

Podríamos intentar atacar el problema usando esta idea, implementar un modelo de jugador virtual que sugiere pistas, otro modelo de jugador virtual que las adivine y buscar entrenar esos modelos en conjunto. La idea seguramente funcione y logremos que las redes se entiendan a la perfección incluso. El problema es que la complejidad del juego reside en que el objetivo interesante no es que el modelo colabore con otro modelo, sino que lo haga con un jugador humano. Y quien haya jugado con dos personas distintas a este juego sabe que cuando cambia la persona con la que estamos jugando, una misma pista que era excelente para el primer jugador es terrible para el segundo, como ejemplificamos antes.

En el paper se sugieren algunos procedimientos para seleccionar posibles conjuntos de palabras y además cómo seleccionar las palabras que tomaremos como pistas, utilizando embeddings y ciertas estrategias para que las pistas sean útiles para un jugador promedio.

2.1. Algoritmo para dar pistas En el paper se proponen algunas formas de abordar este problema, pero el problema que se plantean es ligeramente distinto. Simplifican el juego poniendo solamente dos tipos de cartas, las cartas del equipo azul y las del equipo rojo.

El algoritmo propuesto en el paper lleva el nombre de *ClueGiver*, que como su nombre indica tendrá la tarea de, dado un tablero, generar una pista para que un jugador humano pueda adivinar la carta que le corresponde a su equipo.

Formalmente plantearemos que una partida del juego constará de un tablero con dos conjuntos de palabras

- $B = \{b_i\}_{i=1}^n$: las palabras azules
- $R = \{r_i\}_{i=1}^n$: las rojas

Supongamos que el ClueGiver es el espía del equipo azul. El algoritmo cuenta con 2 pasos:

- **Búsqueda de posibles pistas**

En primer lugar el algoritmo hará una preselección de posibles pistas para utilizar. Para eso calculamos los T vecinos más cercanos de las palabras $\{b_i\}_{i=1}^{n}$. Aquí hacemos un fuerte uso de la representación vectorial de las palabras y sus significados. Considerar un entorno de los vectores es pensar en las palabras que se relacionan con las que se encuentran en el tablero

Una vez que tenemos los vectores cercanos a las palabras que nos corresponden, hay que determinar qué conjunto de palabras queremos sugerir a nuestros agentes. Para esto tomaremos todos los posibles subconjuntos I de palabras de B. Y lo que haremos será agregar a un conjunto $\hat{C} = \{(\hat{c}, I)\}$ los pares de palabras pista y subconjuntos de palabras que buscan sugerir. El criterio para agregar la una palabra $\hat{c}$ será que esté en los vecinos más cercanos de alguna de las palabras de I. Es decir que el conjunto $\hat{C}$ tendrá todos las las palabras con las que puedo sugerir los distintos subconjuntos de palabras I.

- **Elección de pista**

  Una vez que tenemos una preselección de pistas con sus correspondientes objetivos debemos evaluar cuál de estas pistas usaremos para jugar. Para eso el paper sugiere la siguiente función de evaluación:

$$g(\hat{c}, I) = \lambda_B(\sum_{b \in I} s(\hat{c}, b)) - \lambda_R(\max_{r \in R} s(\hat{c}, r)) \tag{1}$$

  Nos quedamos con el par $(\hat{c}, I)$ que tenga mayor puntaje. Observamos que para que una pista aparezca bien puntuada no solo necesita ser similar a las palabras que busca sugerir sino que también ser lo menos similar posible a la palabra más próxima del equipo contrario para evitar que los agentes confundan la palabra con una del equipo.

  Podría suceder que no haya intersección entre los conjuntos de vecinos cercanos de las palabras, en ese caso la pista iría dirigida a una única palabra (esto en el juego es válido, pero no es una buena estrategia). Aunque este caso es poco probable si tomamos una cantidad grande de vecinos cercanos, cerca de 500 es suficiente en la práctica.

  Los parámetros λ_B y λ_R regulan cuánto aporta cada una de las componentes al puntaje total de la pista. En la práctica los autores afirman que $\lambda_B = 1$ y $\lambda_R = 0,5$ obtienen los mejores resultados.

2.2. Función DETECT El trabajo incluye una propuesta para mejorar el funcionamiento del algoritmo anteriormente descrito. La propuesta es la incorporación de una función de puntuación llamada DETECT, esta busca evaluar que tan buena es la pista utilizando herramientas que complementan el uso que explicamos de los embeddings. Todo lo propuesto es considerando los documentos de corpus como las páginas de la Wikipedia, salvo que se indique lo contrario.

La función cuenta con dos partes importantes:

- FREQ

  La primer parte es la función *FREQ*. Busca penalizar palabras que son demasiado frecuentes y que no aportan una buena sugerencia como pista, a pesar de ser similar a la o las palabras que intenta sugerir. Y también penaliza palabras que son demasiado raras. Por ejemplo en caso de que busquemos una pista

para las palabras *"Rey"* y *"Egipto"*, no es útil la pista *"El"* porque seguramente tenga una similitud alta para ambas palabras y podría darse el caso en que el término que aparece en la ecuación 1, que penaliza el ser similar a una palabra roja no baste para descartar esta pista genérica. O podría ser una buena pista el nombre de un emperador egipcio de hace miles de años, pero probablemente sea una mala pista para un jugador común que no conoce en profundidad la historia de los emperadores egipcios.

$$FREQ(w) = -\begin{cases} \frac{1}{df(w)} & \text{Si } df(w) \leq \alpha \\ 1 & \text{Si no} \end{cases}$$

Donde $df(w)$ es la cantidad de veces que aparece la palabra w en el corpus de texto y α un parámetro real que definirá un límite de que tan genérica es una palabra.

$FREQ(w)$ tendrá siempre valor negativo. El máximo de la función será $-\frac{1}{\alpha}$ y el mínimo -1. Si una palabra es rara y aparece en pocos documentos su valor de $FREQ$ será menor que una palabra más frecuente, por eso decimos que penaliza las palabras poco frecuentes. Pero si la palabra es muy frecuente (esto lo define el valor de α) entonces también se penalizará dándole el valor mínimo posible.

- DICT

  La segunda parte de la función DETECT es la función DICT, que esencialmente busca utilizar la información que tiene el diccionario sobre el significado de las palabras.

  Dict2Vec es un embedding de palabras generado utilizando como corpus de entrenamiento el diccionario. Es una variante de word2vec que explota la forma en la que el diccionario almacena la información sobre los significados.

  Si llamamos f_d a la función que a cada palabra w le asigna un vector de valores reales. Entonces $DICT$ está definido como:

  $$DICT(w_1, w_2) = cos(\widehat{f_d(w_1), f_d}(w_2))$$

  Esta parte de la función es útil porque la mayoría de los embeddings utilizan el contexto en el que aparece la palabra para definir su vector, pero hay algunas relaciones con otras palabras que no logramos capturar solamente mirando el contexto en el que se usan. Por ejemplo los hipónimos, por ejemplo un hipónimo de *"Color"* es *"Naranja"*, y estas palabras no tienen porque aparecer en contextos similares. Pero en un diccionario *"Color"* siempre aparecerá dentro de la definición de *"Naranja"*.

- DETECT

  La función que combina las dos anteriores está definida de la siguiente forma:

$$DETECT(\hat{c}) = \lambda_F FREQ(\hat{c}) + \lambda_D \left(\sum_{b \in I} 1 - DICT(\hat{c}, b) - \max_{r \in R}(1 - DICT(\hat{c}, r) \right)$$

  Los valores λ_F y λ_D son parámetros que determinan el peso que tiene cada una de las componentes de la función.

  Básicamente una palabra tendrá un valor grande de $DETECT$ si es una palabra común, además debe ser similar a todas las palabras en el espacio de

vectores de Dict2Vec que busca sugerir y no ser similar a la palabra más cercana del equipo contrario en ese mismo espacio.

2.3. Evaluación de los algoritmos Como mencionamos en el comienzo del capítulo es difícil entrenar modelos que aprendan a jugar al Codenames por el hecho de que la única fuente de datos útiles es que la red haga partidas con personas reales. Está limitación también existe a la hora de intentar evaluar estos modelos, para sacar conclusiones no podemos probarlos colaborando con otro modelo que intente adivinar. Nos interesa saber cual es su desempeño jugando con una persona, y la única forma de hacer eso es haciendo que muchas personas jueguen con el modelo y de ahí sacar estadísticas. Todas las pruebas hechas en el paper están basadas en un servicio de *Amazon* llamado *Amazon Mechanical Turk (AMT)*, que es una plataforma donde se pueden plantear tareas sencillas a resolver, luego personas pueden registrarse en la plataforma y resolver estas tareas a cambio de dinero. Es una buena forma de recolectar datos que no podrían ser generados por una máquina, de forma eficiente y económica. En este caso la tarea es jugar una partida con el modelo, de esta forma se junta gran cantidad de datos de partidas con personas reales para evaluar los algoritmos.

La tarea concreta que aparecía en la plataforma AMT era la de recibir un tablero con una pista sugerida por el algoritmo con la información de que esas pistas sugerían dos palabras del tablero. Con esta pista el jugador tenía 4 intentos: 2 intentos oficiales y 2 extras. Estos dos intentos extra se agregaron para distinguir el caso en el que el algoritmo genera una pista buena pero confusa, del caso en el que la pista es mala. Es por eso que en los resultados se grafica *Presición en 2* y *Acertar con 4*.

Las evaluaciones del modelo se hicieron probando distintos embeddings o grafos de lenguaje:

- BabelNet-WSF: Es un grafo de conocimiento, un grafo donde cada nodo es un concepto o palabra y existen aristas entre aquellos conceptos que estén relacionados. La explicación de cómo se utilizó este grafo se encuentra en [**4**]
- GloVe: Embedding presentado en [**7**]
- GloVe-10K: Es el embedding anterior pero quedándonos solo con las diez mil palabras con más frecuencia.
- BERT: Embedding que distingue los distintos contextos de las palabras (para el juego se promediaron los distintos embeddings de una misma palabra). Está basado en Transformers de los que hablaremos en 1.
- Word2vec: Embedding desarrollado en 9.

Observamos que el mecanismo DETECT tiene un fuerte impacto en el rendimiento del algoritmo.

3. Conclusiones

Comenzamos la monografía exponiendo distintas herramientas de redes neuronales, cómo optimizar esas familias de funciones para lograr distintas tareas. Luego pasamos en el segundo capítulo a ver cómo esas redes neuronales trabajaban en problemas donde el texto y el lenguaje eran la fuente de información para los modelos, como esas redes lograban modelar el lenguaje a través, por ejemplo, de word2vec.

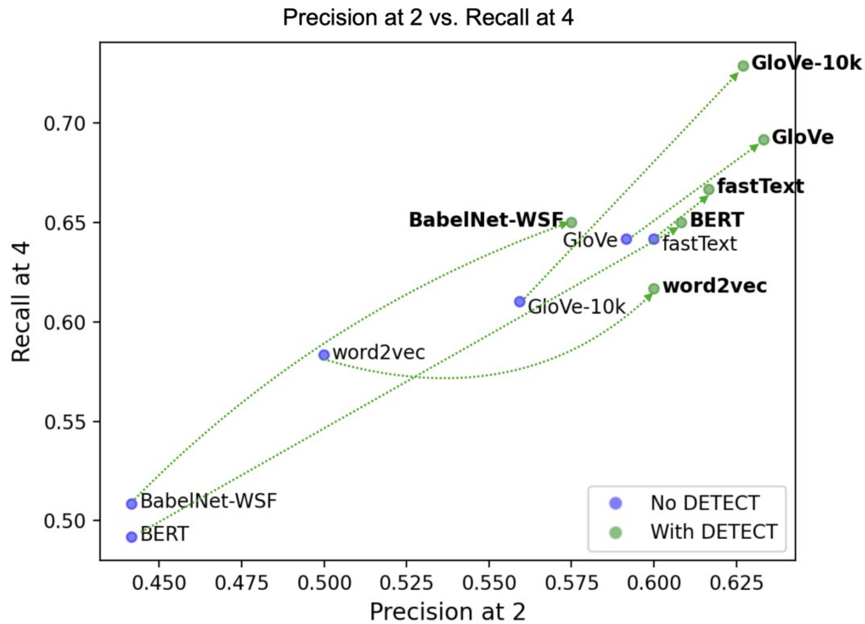

Figura 2. Resultados empíricos presentados en [**4**], se utilizaron distintos tipos de embeddings, con y sin DETECT.

Al comienzo de esta sección mencionamos las dificultades técnicas que el Codenames conlleva como juego, los problemas para buscar buenas pistas, e interpretar correctamente las dificultades que plantea cada tablero. Justamente lo interesante del juego radica en el uso creativo del lenguaje para comunicarse con nuestros compañeros de juego. Creo que esto es lo realmente interesante de las primeros dos capítulos, el hecho de que esas composiciones de funciones acompañadas de algo de optimización y un poco de ingenio a la hora de usarlas pueden llegar a emular un comportamiento humano muy complejo como puede ser pensar en una pista que vincule dos palabras.

Habiendo hecho un recorrido por la bibliografía podemos ver que las herramientas que ha desarrollado el procesamiento del lenguaje natural tienen un gran potencial, y es posible atacar problemas tan complejos como colaborar con un jugador humano en el Codenames, algo que en principio parecía difícil por lo complejo que puede resultar hacer un algoritmo para elegir las pistas, considerando lo complejo que es eso a nivel semántico pero también a nivel de comprensión del otro jugador.

También queda abierta la posibilidad de intentar utilizar modelos más recientes como pueden ser los GPT, que están mostrando tener una enorme capacidad de aplicación en tantísimas tareas.

Capítulo 5

Trabajo futuro

1. Transformers

En el capítulo dedicado a redes neuronales hicimos mención de las redes recurrentes como un ejemplo de arquitectura de red que tenía cualidades interesantes a la hora procesar datos secuenciales, como podrían ser secuencias de palabras. Estas redes, si bien presentan una mejoría respecto de las feedforward para encarar ciertos problemas, también tienen algunas desventajas que limitan su capacidad. Para resolver estos inconvenientes aparece en 2017 el paper [**9**] que propone los llamados ***Transformers***, que mencionaremos brevemente en esta sección.

Los transformers han estado desde su primer aparición empujando el estado del arte en la inteligencia artificial. Como ya mencionamos en el capítulo de redes neuronales algunos ejemplos sorprendentes son los modelos GPT, y más reciente aparecen otros ejemplos de aplicación no vinculados a procesamiento de lenguaje únicamente. Un ejemplo de esto es la red *GATO* que tiene una capacidad sorprendente de respuesta a problemas muy diversos, desde jugar distintos videojuegos, hasta mover un robot, generar texto, o generar una descripción de una imagen. La idea principal es que todos estos problemas mencionados anteriormente, pueden ser modelados como secuencias de entradas a las que la red responde con secuencias de salida: Por ejemplo, un juego dado tendrá algunas secuencias de píxeles que nos mostraran el estado del juego en distintos momentos, y esperamos como salida una secuencia de movimientos en el control del juego. Tenemos que el movimiento de un robot también es un una secuencia de señales y así con todos los problemas. Los transformers tienen una gran capacidad para recibir secuencias de entrada y devolver secuencias de salida.

Otro ejemplo de aplicación es el caso de ***AlphaFold***, una transformer que toma como secuencia de aminoácidos y nos devuelve como salida la forma que tendrá la proteína. La forma que tienen las proteínas en el organismo es lo que le aporta distintas funcionalidades. Hasta ahora este problema de predecir cómo se pliega una proteína según los aminoácidos que la componen era un problema abierto, los expertos del área no podían generar buenos modelos. Fue a través de esta arquitectura que se logró predicciones increíblemente acertadas sobre la realidad, superiores a cualquier predicción hecha por expertos. Saber de qué forma se pliega una proteína abre muchas posibilidades en el mundo farmacéutico, poder testear fármacos a través de simulaciones basadas en inteligencia artificial es un gran avance.

Como mencionamos antes las redes recurrentes fueron un buen primer enfoque para procesar secuencias de información. Estas redes tienen aplicaciones en traducción de

texto, generación de música, procesamiento de audio, entre otros. Pero en su diseño aparecen algunas limitaciones importantes:

- **Memoria corto plazo**: Este problema refiere al fenómeno que ocurre cuando procesamos secuencias largas de texto. Cuando damos como entrada una secuencia demasiado larga, por ejemplo dado un problema de predecir la siguiente palabra, la red recibe como entrada una secuencia larga con el contexto de la frase, procesa secuencialmente cada una de las palabras y en el último paso predice la siguiente palabra. Lo que ocurre en la práctica es que las palabras que aparecen al comienzo de la oración no tienen demasiada influencia en la palabra que predecimos al final. Si bien esto puede no ser un problema en algunos ejemplos, sí que lo es en otros. Por ejemplo si estamos procesando la oración:

  "Viví muchos años en Inglaterra, estuve trabajando en una fábrica de alfombras y (. . .), desde entonces hablo fluidamente [`palabra a predecir`]".

  Si miramos el contexto final de la frase sabemos que la palabra que debemos predecir es un idioma, pero cuál idioma solo podemos saberlo si vemos que al comienzo de la oración usamos la palabra *Inglaterra*

- **Imposibilidad de paralelizar procesos**: La información secuencial es utilizada por el modelo ya que procesa los datos de manera ordenada, y la red no responderá igual si ponemos una frase o la misma frase en distinto orden. Esto es útil ya que estamos usando la información secuencial que nos interesa, pero a la vez nos obliga a procesar todo en secuencia, es decir tenemos que calcular el estado de la red para cada tiempo y no podemos calcular un estado futuro hasta no tener el calculado el tiempo anterior. Esto es un problema ya que el tiempo de cómputo puede ser muy alto si no podemos *paralelizar*. Esto significa poder hacer varios cómputos en paralelo en distintas computadoras; por ejemplo a la hora del entrenamiento cuando hacemos el cálculo del gradiente necesitamos hacer un forward prop y un backprop, si para esos cálculos pudiéramos paralelizar, ahorraríamos tiempo. Permite que podamos entrenar modelos más rápidamente y esto da la posibilidad de hacer modelos más grandes (con suficientes recursos).

La arquitectura del transformer es de encoder-decoder, como muestra la figura 1 tiene un bloque dedicado a codificar la entrada y uno para decodificar esa entrada devolver una salida. Nos detendremos en las ideas de *Positional Encoding* y en *Multi-Head Attention.*

1.1. Positional Encoding El transformer tomará como entrada una secuencia, y para poder paralelizar los cálculos tomará todas las palabras simultáneamente, no como las redes recurrentes que procesaban cada palabra por separado en forma secuencial. El problema es que al tener toda la secuencia al mismo tiempo no podemos distinguir el orden de las palabras. En principio sería lo mismo que el input del transformer fuera *"Optimus Prime es un"* que *"Prime un es Optimus"*. Por esto es necesario que la red tenga una forma de distinguir las posiciones que ocupan las palabras dentro de la frase.

Lo que haremos será agregar a cada vector embedding otro vector que nos indique la posición que esa palabra tiene en la secuencia, que nos codifique la posición. La idea será tomar el vector en binario que identifique la posición, es decir: a la primer palabra le

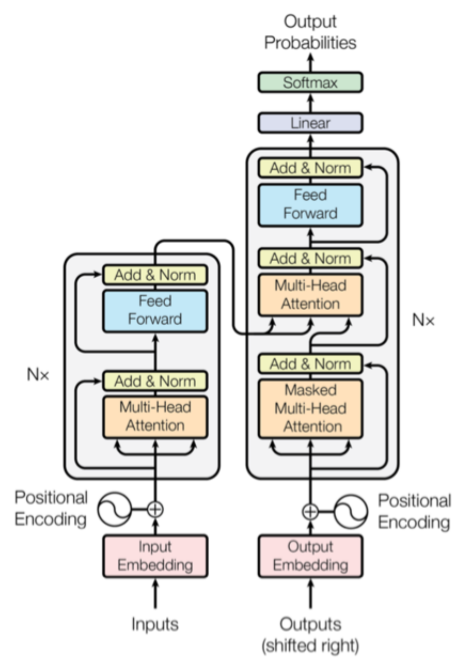

FIGURA 1. Arquitectura propuesta en el paper, lo enmarcado a la derecha es el bloque de encoder, a la izquierda el bloque decoder.

agregaremos el $(1, 0, \cdots, 0)$, a la segunda palabra el $(0, 1, 0, \cdots, 0)$ y así. Esto le permite a la red distinguir las posiciones que ocupan las palabras y por lo tanto usar la información secuencial de la frase. Esta idea en realidad es la simplificación de lo que utiliza el paper original, este método tiene el problema de que agregarle un vector con la posición en binario no es una operación derivable, y esto presenta algunas problemáticas a la hora de calcular los gradientes por ejemplo. Por eso el paper original lo que hace es cambiar lo discreto de 0,1 por una suma de senos y cosenos. Pero la idea de fondo sigue siendo la de distinguirlos por sus valores en binario.

1.2. Atención El nombre del paper [**9**] que introduce esta arquitectura lleva el nombre *"Attention is all you need"*. Esta "Atención"de la que habla el título es un mecanismo que soluciona el problema de la pérdida de memoria que tenían las redes recurrentes.

Dijimos que uno de los problemas que sufrían las redes recurrentes era que en secuencias largas, era usual que la información de los primeros elementos de la secuencia perdiera relevancia a la hora de procesar un dato de la secuencia en un tiempo muy posterior. Para está nueva arquitectura se implementa un sistema de "atención", que permite que la red aprenda cuales palabras son importantes a la hora de procesar alguna de las palabras de la secuencia. Por ejemplo, procesando la frase *"Martinez tiene una biblioteca muy grande donde pone todos sus libros en orden"*. La palabra *"sus"* debería ser procesada dándole importancia a la palabra *"Martinez"*.

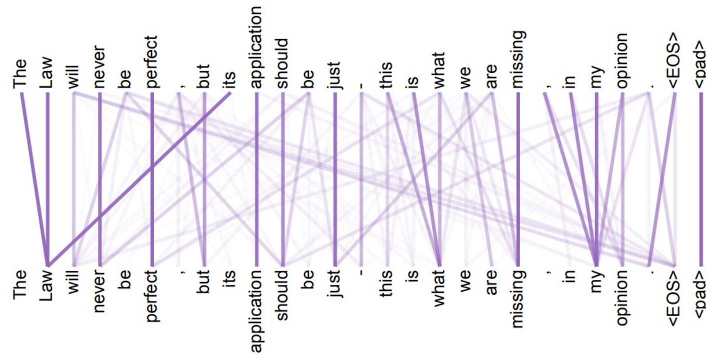

FIGURA 2. Visualización de la matriz de atención en el procesamiento de la frase de input. Imagen de [**9**]

El mecanismo de atención funciona de la siguiente forma: para cada palabra de la secuencia que estamos procesando calcularemos tres vectores a partir del vector de embedding.

- Vector *Query*
- Vector *Key*
- Vector *Value*

Cada uno de estos vectores será calculado multiplicando por tres matrices distintas, cuyos valores serán parámetros del modelo y los aprenderemos durante el entrenamiento.

El vector value será el que asociaremos a la palabra mientras sigamos con el proceso, y los vectores Query y Key serán los que usaremos para aprender el vínculo entre dos palabras. Con estos dos vectores lo que haremos será calcular su producto interno. A la hora de procesar una palabra multiplicaremos su vector query con todos los vectores key de las demás palabras de la oración, esto nos dará lo que llamamos un vector de atención, que nos dice que importancia tienen las palabras de la oración al leer esta primer palabra. Repetir este proceso para todas las palabras nos dará una matriz con todos los vectores de atención. Está matriz es la matriz de atención y visualiza como la red entiende las relaciones entre las palabras, un ejemplo se puede visualizar en la figura 2.

Para cada palabra de entrada el mecanismo de atención devolverá un output que luego se utilizará en lo que sigue de la red. El output de la primer palabra pasada por el mecanismo de atención será hacer la suma ponderada de los vectores valores de la secuencia, multiplicados por el factor de atención que indique el vector de atención. De esta forma la red codificará cada palabra teniendo en cuenta el contexto en el que se encuentra.

Este mecanismo de atención, como muestra la arquitectura de la figura 1 aparece en varios momentos, algunas veces calculando la atención dentro de las palabras de una misma frase, pero a veces también con las frase de output, como muestra la figura 3

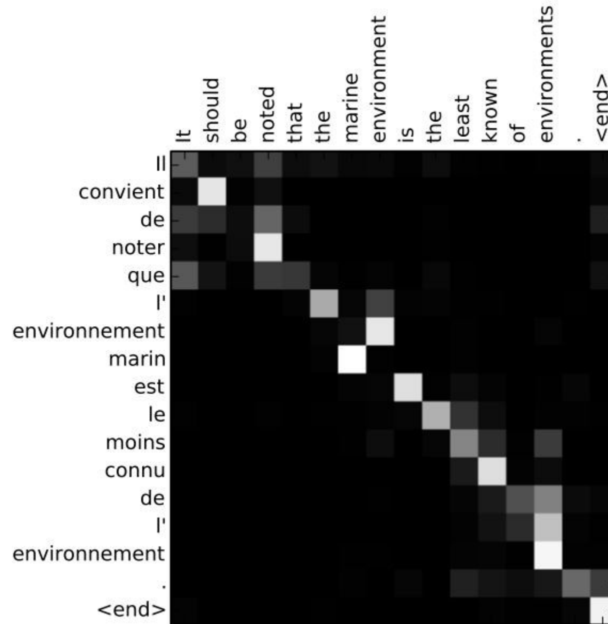

Figura 3. Ejemplo de matriz de atención generada por un transformer con la tarea de traducir una frase del francés al inglés. Es interesante observar que la matriz muestra que el transformer está prestando atención a las palabras en los distintos idiomas a pesar de que aparezcan en distinto orden.

2. Ética en los modelos

Otro ángulo interesante en esta emergente área son los problemas éticos que van surgiendo a medida que los modelos entrenados con datos son utilizados en el mundo real. Surgen naturalmente problemas de sesgos raciales o de género en los datos con los que se entrenan los modelos, y por lo tanto generan que los modelos tengan ciertas tendencias que no son deseables a la hora de utilizarlos.

Los problemas éticos surgen cuando estos modelos con sesgos en su entrenamiento son utilizados para decidir sobre políticas públicas, sobre si dar o no un préstamo a un emprendedor, o sobre qué condena debería tener un recluso según su caso. Si entrenamos modelos basados en datos puede parecer que estamos haciendo algunos procesos un poco más objetivos, no existe una persona detrás con malas intenciones o que pueda equivocarse. Pero lo que sucede con estos modelos que pueden aparentar ser más *"objetivos"* es que están entrenados en base a datos reales, generados en distintos momentos de la historia, donde muy probablemente haya una gran presencia de decisiones racistas, o discriminando alguna minoría. De este amplio tema existe el libro *Weapons of math destruction* escrito por Cathy O'Neil, donde aborda las complicaciones que trae para los individuos el uso de algunos modelos diseñados para procesar enormes cantidades de datos. Desarrolla específicamente el caso de modelos aplicados a seguros, publicidad, educación y patrullaje policial.

Más específicamente en los modelos de procesamiento de lenguaje existen trabajos evaluando los sesgos que pueden haber en embeddings, en algunas tareas específicas como puede ser el análisis de sentimiento [**1**] o en la traducción de texto [**10**]

Dentro de los centros de desarrollo de estas tecnologías se están dedicando equipos al trabajo en la ética de estos grandes modelos, a los que se conocen como *modelos fundacionales.* En la universidad de Standford existe el *Centro de investigación de modelos fundacionales*, que está dedicado a entender estos modelos gigantes para poder emplearlos de forma responsable.

Bibliografía

[1] Mark Diaz, Isaac Johnson, Amanda Lazar, Anne Marie Piper, and Darren Gergle. Addressing age-related bias in sentiment analysis. In *Proceedings of the 2018 CHI Conference on Human Factors in Computing Systems*, CHI '18, page 1–14, New York, NY, USA, 2018. Association for Computing Machinery.

[2] Ian Goodfellow, Yoshua Bengio, and Aaron Courville. *Deep Learning*. MIT Press, 2016. `http://www.deeplearningbook.org`.

[3] Dan Jurafsky and James H. Martin. *Speech and Language Processing (3rd edition draft)*. 2020. `https://web.stanford.edu/~jurafsky/slp3/`.

[4] Divya Koyyalagunta, Anna Sun, Rachel Lea Draelos, and Cynthia Rudin. Playing codenames with language graphs and word embeddings. *Journal of Artificial Intelligence Research*, 71:319–346, 2021.

[5] Waldemar Lopez. Vector representation of internet domain names using word embedding techniques, 11 2019.

[6] Tomas Mikolov, Kai Chen, Greg Corrado, and Jeffrey Dean. Efficient estimation of word representations in vector space. *CoRR*, abs/1301.3781, 2013.

[7] Jeffrey Pennington, Richard Socher, and Christopher Manning. Glove: Global vectors for word representation. volume 14, pages 1532–1543, 01 2014.

[8] David Silver, Aja Huang, Christopher J. Maddison, Arthur Guez, Laurent Sifre, George van den Driessche, Julian Schrittwieser, Ioannis Antonoglou, Veda Panneershelvam, Marc Lanctot, Sander Dieleman, Dominik Grewe, John Nham, Nal Kalchbrenner, Ilya Sutskever, Timothy Lillicrap, Madeleine Leach, Koray Kavukcuoglu, Thore Graepel, and Demis Hassabis. Mastering the game of go with deep neural networks and tree search. *Nature*, 529:484–503, 2016.

[9] Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N Gomez, Ł ukasz Kaiser, and Illia Polosukhin. Attention is all you need. In I. Guyon, U. Von Luxburg, S. Bengio, H. Wallach, R. Fergus, S. Vishwanathan, and R. Garnett, editors, *Advances in Neural Information Processing Systems*, volume 30. Curran Associates, Inc., 2017.

[10] Jieyu Zhao, Subhabrata Mukherjee, Saghar Hosseini, Kai-Wei Chang, and Ahmed Hassan. Gender bias in multilingual embeddings. 04 2020.