Facultad de Ingeniería y Ciencias Agrarias

Ingenieria Informatica

Trabajo Final

Análisis de Texto Con Redes Neuronales Recurrentes

Investigando Inteligencia Artificial
y Redes Neuronales

Alumno: Santiago Ibañez Musielack
Tutor: Ricardo Di Pasquale

INDICE DE CONTENIDOS

ABSTRACT

Desde el comienzo de la era informática, la humanidad siempre ha imaginado a las máquinas imitando las funciones cognitivas de un ser inteligente. Es decir, que posean inteligencia artificial. El estudio sobre cómo las máquinas podrían aprender como aprenden los humanos ha ido evolucionando y, con las nuevas capacidades que tienen las computadoras de hoy, los resultados son sorprendentes. La detección prematura de cáncer en la piel, autos que se manejan solos, traductores de chino en tiempo real y el uso de rostros como identificación personal; hay evidencia de inteligencia artificial en todas partes.

La inteligencia artificial está cambiando la manera en la que las computadoras resuelven problemas, ya no tratando de conocer todas las decisiones posibles y elegir la mejor, sino aprendiendo mediante la *prueba* y el *error* hasta lograr el resultado deseado.

OBJETIVO

El fin de este trabajo es demostrar que se puede implementar un algoritmo de inteligencia artificial para resolver un problema real, utilizando la teoría básica de *machine learning*, redes neuronales y la ayuda de poderosas herramientas de software libre.

Se va a implementar un modelo para la predicción de sentimiento de texto a partir de una red neuronal recurrente y se va servir el modelo en una aplicación, utilizando las últimas tecnologías y aplicando los conocimientos aprendidos en la carrera de Ingeniería en Informática.

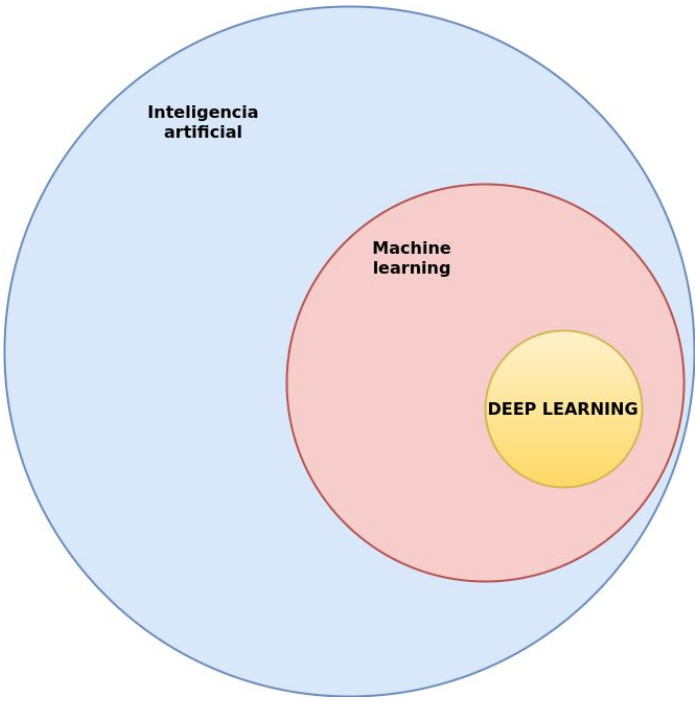
ALCANCE Y LIMITACIONES

Este trabajo tiene como alcance la implementación de un modelo personalizado y una aplicación para la interacción con el usuario. Este proyecto sólo plantea los pasos y componentes necesarios para su desarrollo, se limita a generar un punto de partida para la creación de modelos más precisos con posibilidades de puesta en producción.

MARCO TEÓRICO

Machine learning

Es un campo de las ciencias de la computación que le da a las computadoras la habilidad de aprender a resolver tareas sin haber sidos programadas explícitamente.



El término fue empleado por primera vez en 1959 por Arthur Samuel, un estadounidense pionero en el área de juegos de computadora e inteligencia artificial cuando trabajaba para IBM. El área de estudio evolucionó con el estudio del

reconocimiento de patrones y la teoría del aprendizaje computacional en inteligencia artificial. *Machine learning* explora el estudio y la construcción de algoritmos que pueden aprender de información. Se sobreponen a seguir estrictas y estáticas instrucciones de programación. Crean modelos para la predicción, clasificación y toma de decisiones únicamente a partir de la información recibida y del objetivo a cumplir.

Machine learning es usado para un amplio espectro de problemas en la computación. Muchos problemas son de clase *NP-hard*, por lo que gran parte de la investigación realizada en *machine learning* se centra en el diseño de soluciones factibles para esos problemas. *Machine learning* puede ser visto como un intento de automatizar algunas partes del método científico mediante métodos matemáticos.

Los algoritmos de machine learning se caracterizan en tres grandes grupos: aprendizaje no supervisado, *'Unsupervised Learning'*, aprendizaje supervisado *'Supervised Learning'* y de aprendizaje reforzado *'Reinforcement Learning'*. Este trabajo solamente utilizará algoritmos de aprendizaje supervisado.

Anexo: Categorías de problemas por complejidad

Para definir las categorías primero definimos la función polinómica S:

$$S = f(N)$$

S: es la cantidad de pasos para resolver un problema

N: el tamaño del problema.

P(polynomial time)

Son los problemas que tienen solución en tiempo polinomial es decir que la solución al problema se puede encontrar en S pasos. Ejemplos: suma, multiplicación, MCD, problemas de búsqueda.

NP (non-deterministic polynomial time)

Son los problemas a los que no se les puede encontrar una solución en tiempo polinomial. En particular la dificultad de los problemas escalan proporcionalmente a

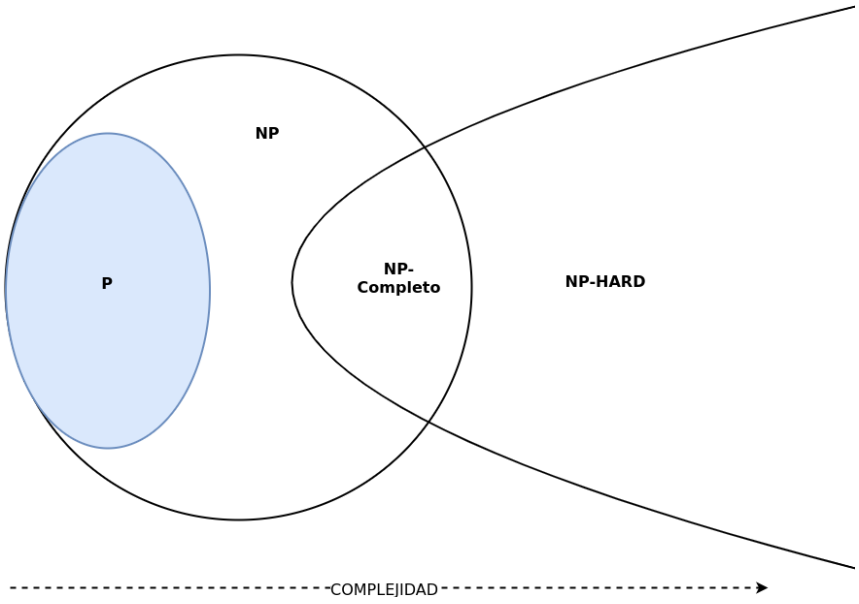
medida que se aumenta el problema. Para hallar la correcta solución a los problemas NP es necesario iterar sobre todas las posibles soluciones hasta encontrar la correcta. Una vez hallada es fácil (en tiempo polinomial) comprobar si es válida.
Ejemplos: sudoku, búsqueda de números primos, encontrar el mejor camino para un conjunto de ciudades, optimizar horarios.

NP-completos

Es una subcategoría de NP. Dado un problema P, se puede reducir en otro problema P_2, si es posible diseñar un algoritmo polinomial que resuelva P cuando exista otro que resuelva P_2. Si un problema se reduce a otro y este se resuelve, también resuelvo el problema original. NP-completo contiene a todos los problemas a los que se reducen todos los problemas de NP. Para estos problemas es difícil (NP) encontrar una solución pero también es difícil (NP) verificarla. Ejemplo: Dado dos secuencias de elementos, ¿cuál es el subsecuencia más grande común a las dos secuencias? (LCS)

NP-hard

Son los problemas que son, al menos, tan difíciles como los problemas más difíciles de NP-completo. Machine learning estima soluciones para estos problemas pero no los resuelve.



Supervised Learning

Es una técnica para deducir una función a partir de datos de entrenamiento. Los datos de entrenamiento consisten de pares de objetos, normalmente vectores: un componente del par son los datos de entrada y el otro, los resultados deseados. La salida de la función puede ser un valor numérico, como en los problemas de regresión, o una etiqueta de clase, como en los de clasificación.

El objetivo del aprendizaje supervisado es crear una función capaz de predecir el valor correspondiente a cualquier objeto de entrada válida, después de haber procesado una serie de ejemplos llamados datos de entrenamiento, o 'training dataset'. Para ello, tiene que generalizar a partir de los datos presentados a las situaciones no vistas previamente. Los algoritmos de aprendizaje supervisado tienen definidos los siguientes componentes:

Entrada
- Dominio X; todos los puntos del conjunto X tiene características observables.
- Conjunto Y de etiquetas o "Labels"
- Dataset, un conjunto de ejemplos
- $S = \{(X_1, Y_1), ..., (X_n, Y_n)\} \subseteq XxY\}$

Salida
- Una regla de predicción/hipótesis
- $h : X \rightarrow Y$

Métrica de performance
- El riesgo/generalización del error:
- $err(h) =: \xi_{x\sim D}[|h(x) - f(x)|]$
- (error = a la salida real - salida de la predicció

Pseudocódigo:

```
Input: Data, Modelo Metal
Output: Modelo Metal Actualizado
while Modelo Metal hace malas predicciones do
  hacer una preddicion
  calcular el error
  if(error es aceptable) then
    break
  else
    proponer ajuste al Modelo Mental
    Modelo Mental <-- Modelo Metal + ajuste
```

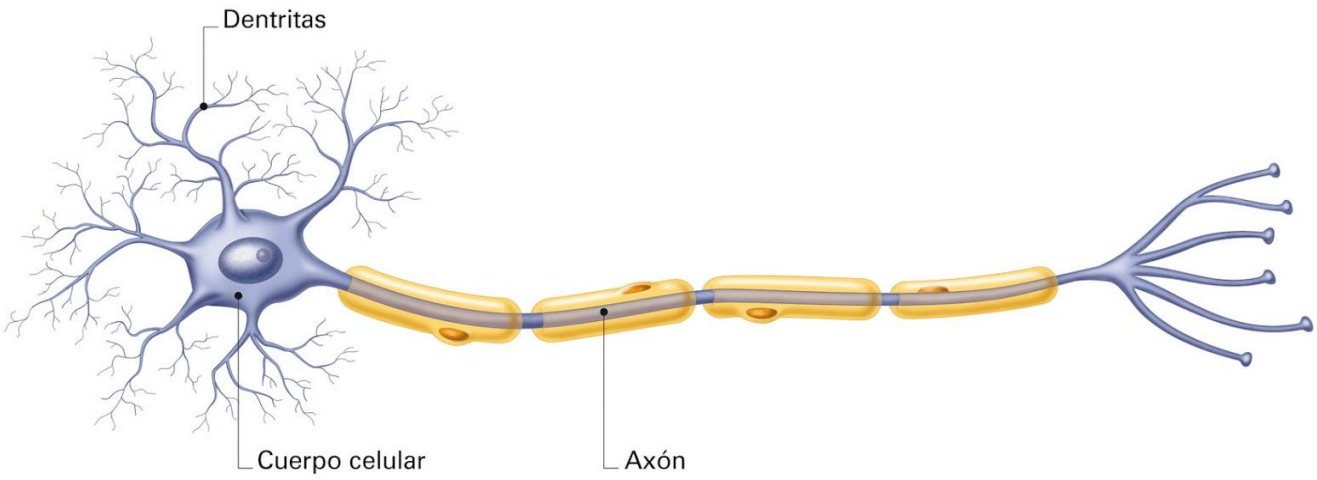
Redes Neuronales

La capacidad que tiene el cerebro humano de recibir información por los sentidos, sintetizarlos y, mediante la *prueba y error* (la experiencia), y poder crear modelos que permiten tomar mejores decisiones a futuro, es lo que llamamos *aprender*.

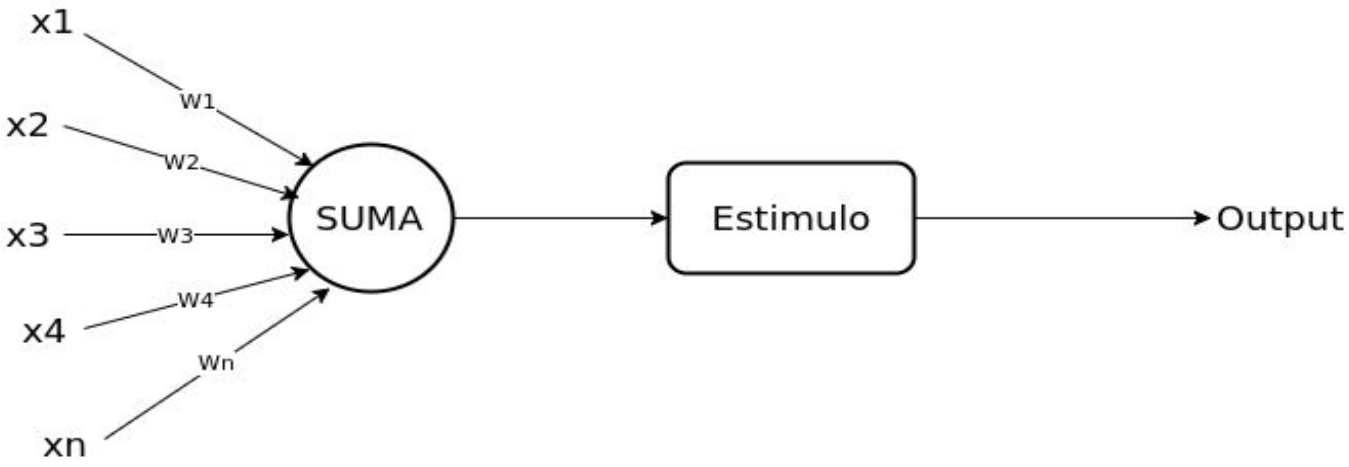
Todo lo que hemos experimentado o sentido, todos nuestros pensamientos y nuestra idea de ser, es producido por el cerebro.

A nivel molecular, sabemos que el cerebro está compuesto por 100 mil millones de células que llamamos neuronas. Cada neurona tiene tres principales funciones:

1. Recibir una señal desde sus dendritas
2. Verificar si la señal debe ser transmitida al cuerpo o soma.
3. Mandar la señal resultante llamada Potencial de acción a través de su axón.



Los primeros científicos en computación exploraron este conocimiento sobre cómo funciona el cerebro e intentaron simularlo. Así fue que, en 1943, MvCulloch & Pitts crearon el primer modelo computacional de una neurona. Fue diseñado para clasificar dos categorías, 1 y 0.



Definiciones:

X_i : *entradas*

W_i : *pesos o weights*

θ : *valor de umbral*

$Suma : \sum_{i=0}^{n}(X_i * W_i) - \theta$

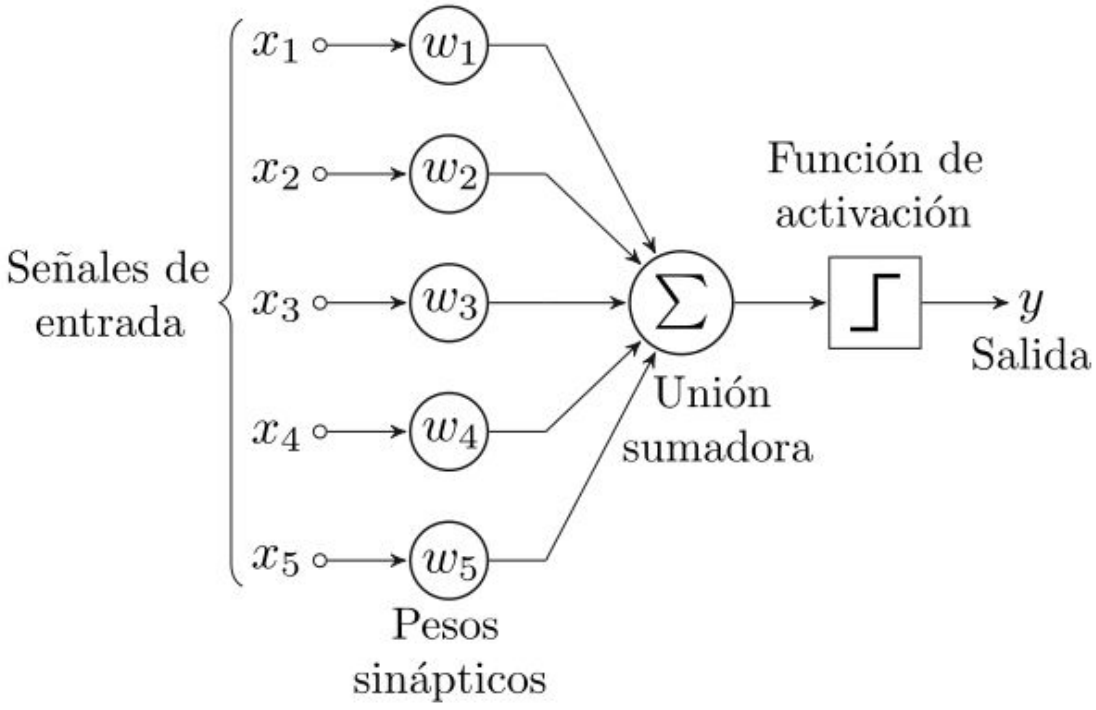
Estímulo : *una función no lineal*. $S(u) = \{1,\ u \geq 0;\ 0,\ u < 0\}$

Output : *clasificación* : $1\ o\ 0$

Perceptron

El segundo gran aporte al área fue la invención del Perceptrón en el laboratorio Cornell Aeronautical por Frank Rosenblatt en 1957. El algoritmo perceptrón corría en una IBM 704 y luego fue implementado en una computadora a medida *'custom'* llamada *Mark 1*. La máquina fue diseñada para hacer reconocimiento de imágenes. Obtenia las imágenes con un arreglo de 400 fotocélulas (400 píxeles) que eran conectadas a las 'neuronas'. Los pesos o *'weights'* eran potenciómetros y las actualizaciones a los pesos eran calculadas por motores eléctricos.

Hoy en dia las unidades básicas de las redes son perceptrones.



x_n : *es el conjunto de entrada*

w_n : *son los pesos o weights*

y : *salida* 0 *o* 1

Lo que se agregó en la creación del perceptrón fue la idea de que esta máquina podía aprender actualizando los *'weights'* y los '*biases*'. *'Bias'* es una constante que se agrega en el momento de la suma.

Definición de la función de activación:

$$f(x) = \begin{cases} 1 & \text{if } \; w \cdot x + b > 0 \\ 0 & \text{en otro caso} \end{cases}$$

Donde:

w : *es un vector de valores en* $\Re$

$w.x$: *es el producto escalar* $\sum_{i=1}^{m} w_i x_i$

m es el número de valores de entrada al perceptrón

b es el bias, una constante que se define para desviar la decisión afuera del origen.

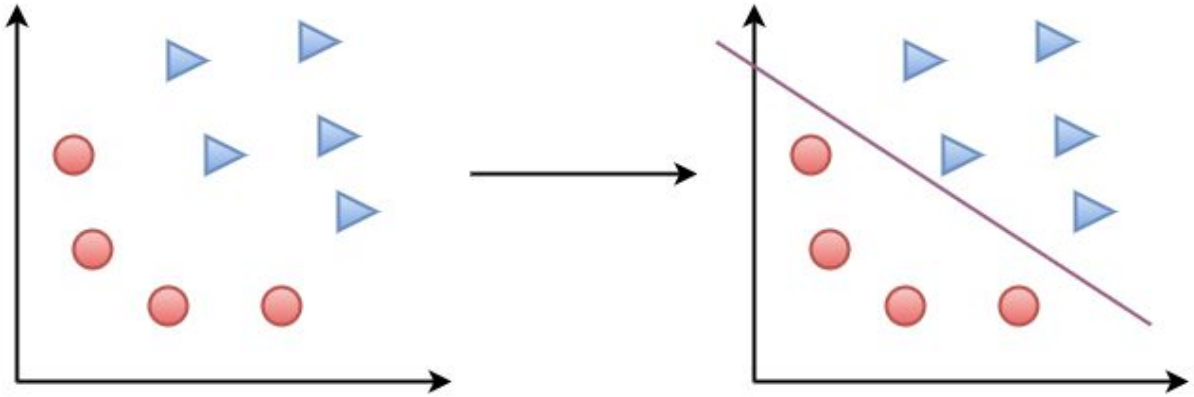
El aprendizaje se basa en ajustar los valores de la matriz de *'weights'* y los *'biases'* para que el error sea cada vez menor. Matemáticamente, se calculan los ajustes observando la tasa de cambio del error para un determinado cambio en el peso.

En cuanto a la convergencia, se demostró que un perceptrón va a converger en una solución solamente si el conjunto de entrada es separable linealmente. Una sistema de un solo perceptrón se puede entrenar para aprender la función AND y la OR pero no así la XOR; XOR no es linealmente separable. Por otra parte, otra limitación es que solamente puede clasificar la entrada en dos clases: 1 o 0.

Algoritmo de aprendizaje para un perceptrón o red de una capa

Ejemplo: Usando un perceptrón para clasificar un conjunto de S que sabemos a priori, y que es *linealmente separable*. Se definen algunas variables:

- Conjunto S Linealmente Separable:



Linealmente Separable

- $y = f(z)$ es la salida del perceptrón para una entrada z
- $S = \{(x_1, d_1), ..., (x_s, d_s)\}$ es el dataset con la que se va a entrenar el perceptrón.
  - x_j es el vector de entrada de dimensión n
  - d_j es el label o la clasificación real para esa entrada.

Cada vector x_j tiene n dimensiones, cada dimensión representa una característica de la entrada.

- $x_{j,i}$ es el valor que tiene esa característica de la entrada.
- $x_{j,0} = 1$

Los pesos, o *'weights'*:

- w_i es el i-ésimo valor del vector de peso. Va a ser multiplicado por la i-ésima característica del vector entrada.
- Como $x_{j,0} = 1,$ el w_0 va a actuar como constante *'Bias'* en vez de tener una aparte.
- $w_i(t)$ es el i-ésimo peso en el tiempo t.

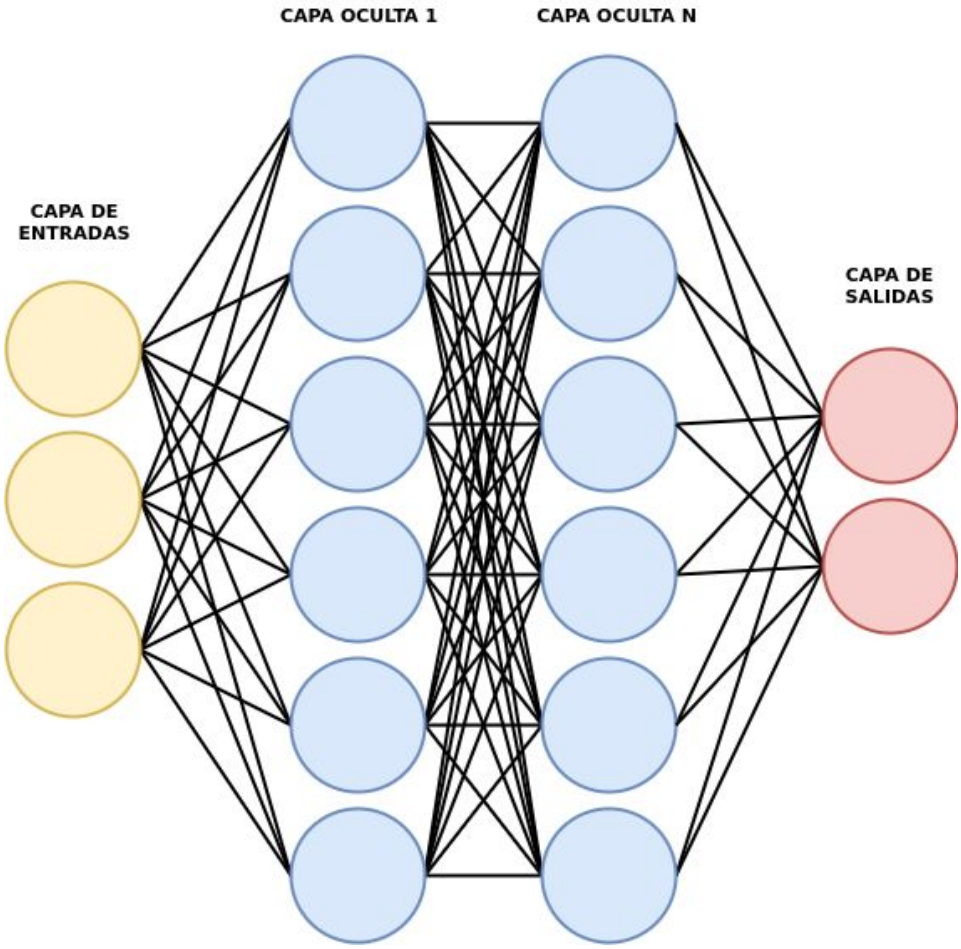
Pseudo algoritmo para el aprendizaje:

1. Se inicializan los pesos, muchas veces 0.
2. Para cada entrada, j del conjunto S, se realizan los siguientes pasos para cada x_j y su correcta clasificación d_j .
    a. Se calcula el output actual.
        i. $y_j(t) = f(w(t) \,.\, x_j$ (producto escalar)
    b. Se actualizan los pesos
        i. $w_i(t+1) = w_i(t) + (S_j - y_j(t)).x_{j,i}$, para cada característica $0 \leq i \leq n$
3. Este procesos de aprendizaje, también llamado entrenamiento se repite por un determinado número de interacciones, o hasta que el error llegue a un límite inferior:
    a. $\frac{1}{L} \sum_{j=1}^{L} \left| S_j - y_j(t) \right|$ donde L es el tamaño del dataset

Redes Neuronales de Multicapa

En los últimos años, la generación de información ha crecido en forma exponencial. Más información ha sido creada en los últimos dos años que en la historia completa de la raza humana. Como consecuencia, existe una urgente necesidad de tener técnicas para controlar esta información.

Una de las técnicas para analizar esta informacion ha sido la de retomar la idea del perceptrón y potenciar su *'performance'* con más capacidad, y la ayuda del poder de cómputo actual. Las redes neuronales no son más que un conjunto de perceptrones agrupados en capas e interconectados entre sí.



Siguen siendo algoritmos de *machine learning* y mantienen la misma idea de aprendizaje que la de un simple perceptron. Se diferencian según la tarea que tienen que realizar. Algunos ejemplos a continuación:

Capa de Entrada

Al igual que con un solo perceptrón, la entrada de una red neuronal es, en primera medida, la información con la que se va a entrar a la red. Una vez entrenada, es la nueva información a clasificar, predecir o agrupar. Esta informacion tiene que estar *vectorizada*.

Vectorización

La representación vectorial de la información es necesaria porque la red es una función matemática. Las redes que se usan para clasificar imágenes o detectar patrones en las imágenes, como pueden ser rostros, objetos, ubicaciones y animales, reciben como entrada la imagen vectorizada. Una de las técnicas es utilizar la matriz de píxeles, en la que cada posición es un número *rgb.* Se concatena cada fila de la matriz en un solo vector y ese vector se usa como entrada de la red neuronal.

Otro ejemplo es el de vectorización de texto. Uno de los métodos es llama *'one hot vectorization',* o 'vectorización de unos calientes', en la que se calcula el tamaño del vocabulario del texto. Cada palabra que va a entrar en la red es un vector de poblado con 0 de dimensión vocabulario y un 1 en la posición de la palabra.

La Capa de Salida

Cuando se diseña una red neuronal, uno de los parámetros más importantes es el tipo de salida, o cuantas características va a tener ese vector. Como ya vimos, si el problema es de clasificación binaria, la dimensión del vector de salida es 1 (solo puede ser 1 o 0). Si la salida tiene múltiples categorías, por ejemplo se clasifica una imagen y se quiere saber si la imagen tiene perros, gatos, humanos y si el clima es agradable o

no, el vector de salida va a tener dimensión 5. En este caso, cada valor del vector va a indicar la probabilidad de que la imagen tenga esa categoria especifica.

Las Capas Ocultas:¿ Por que se agregan más capas a la red?

Vimos que las redes de un solo perceptrón tenían muchas limitaciones al momento de generar funciones/modelos para problemas más complejos que para una clasificación binaria. Las capas se agregan para darles más capacidad a la red, y así poder resolver problemas más complejos. Estos problemas se parten en sub problemas que cada neurona puede resolver.

Otra variable es la cantidad de neuronas por capa. Este aspecto es un problema activo en la investigación sobre redes neuronales y varía según el caso de estudio y el problema a solucionar.

Aprendizaje en Redes Neuronales Multicapa

Las redes neuronales multicapa siguen estando dentro del conjunto de *machine learning* y, en particular, de algoritmos supervisados. Es entonces que, el aprendizaje también se basa en calcular un error y corregir los parámetros internos de la red para que ese error disminuya. En otras palabras, es un problema de optimización.

Dentro de las técnicas para entrenar una red neuronal, el más popular es *'backpropagation'*.

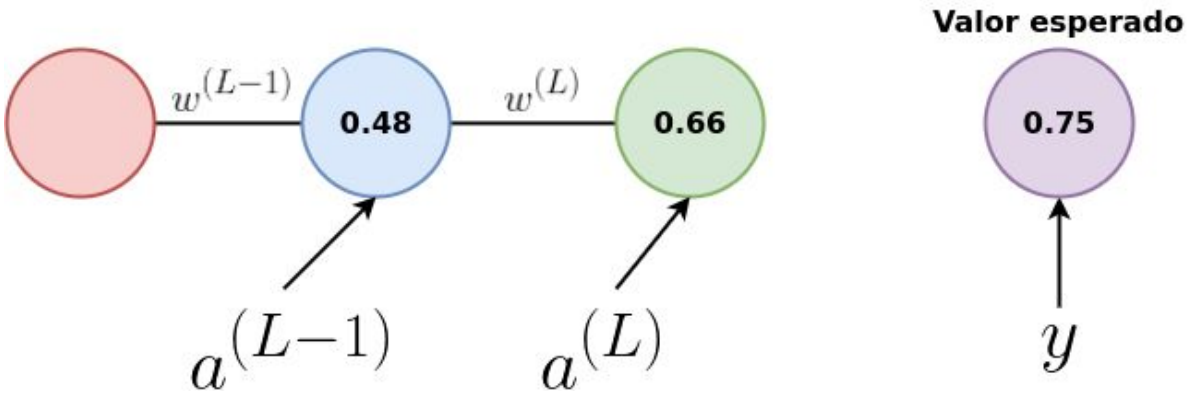
'Backpropagation'/propagación hacia atrás

'Backpropagation' es un método usado en redes neuronales artificiales para calcular la contribución al error de cada neurona, después de que un lote de datos de entrada es procesado. En un análisis de sentimiento de texto, son varias frases por lote.

Si miramos a la red neuronal como una composición de funciones, el objetivo de la propagación hacia atrás es minimizar el error de la salida. Como ejemplo vamos a analizar cómo actúa el algoritmo sobre una red neuronal simple.

Algoritmo de backpropagation para una red neuronal de una neurona por capa:

Imagen de la red ejemplo:



Definiciones:

La salida esperada es:

$$y \in \mathbb{R}$$

La activación de una neurona la definimos como:

$$a = (a^{(1)}, a^{(2)}, ..., a^{(L)}, a^{(n)}))) \in \mathbb{R}^n$$

El error también es llamado costo. Lo definimos como:

$$C_0 = (a^{(L)} - y)^2$$

La salida de cada neurona se define como el valor del peso $w^{(L)}$ de esa neurona por la activación de la neurona anterior más el bias $b^{(L)}$.

$$a^{(L)} = \sigma(w^{(L)} a^{(L-1)} + b^{(L)})$$

$$z^{(L)} = w^{(L)} a^{(L-1)} + b^{(L)}$$

$$a^{(L)} = \sigma(z^{(L)})$$

Lo que se quiere saber es qué tan sensible es nuestra función de error o costo, según el peso de la última neurona. O en otras palabras:

$$\frac{\partial C_0}{\partial w^{(L)}}$$

Usando la regla de la cadena, descomponemos las derivadas parciales:

$$\frac{\partial C_0}{\partial w^{(L)}} = \frac{\partial z^{(L)}}{\partial w^{(L)}} \frac{\partial a^{(L)}}{\partial z^{(L)}} \frac{\partial C_0}{\partial a^{(L)}}$$

Calculamos las derivadas:

$$\frac{\partial C_0}{\partial a^{(L)}} = 2\left(a^{(L)} - y\right)$$

$$\frac{\partial a^{(L)}}{\partial z^{(L)}} = \sigma'\left(z^{(L)}\right)$$

$$\frac{\partial z^{(L)}}{\partial w^{(L)}} = a^{(L-1)}$$

Nos queda:

$$\frac{\partial C_0}{\partial w^{(L)}} = a^{(L-1)} \sigma'\left(z^{(L)}\right) 2\left(a^{(L)} - y\right)$$

Ahora tenemos el cambio en el costo con respecto al peso $w^{(L)}$ para un ejemplo. Sin embargo, cuando se entrena, el algoritmo calcula el costo para el promedio de todos los ejemplos de entrenamiento, siendo el tamaño de ese dataset el número n:

$$\frac{\partial C}{\partial w^{(L)}} = \frac{1}{n}\sum_{k=0}^{n-1}\frac{\partial C_k}{\partial w^{(L)}}$$

Los mismos cálculos se realizan para los *'biases'*:

$$\frac{\partial C_0}{\partial b^{(L)}} = \frac{\partial z^{(L)}}{\partial b^{(L)}}\frac{\partial a^{(L)}}{\partial z^{(L)}}\frac{\partial C_0}{\partial a^{(L)}} = 1 * \sigma'(z^{(L)})2(a^{(L)} - y)$$

Ahora podemos calcular el gradiente:

$$\nabla C = \begin{bmatrix} \frac{\partial C}{\partial w^{(1)}} \\ \frac{\partial C}{\partial b^{(1)}} \\ . \\ . \\ . \\ \frac{\partial C}{\partial w^{(L)}} \\ \frac{\partial C}{\partial b^{(L)}} \\ \frac{\partial C}{\partial w^{(n)}} \\ \frac{\partial C}{\partial b^{(n)}} \end{bmatrix}$$

También se puede calcular que tan sensible es el costo con respecto a la activación de la neurona de la capa anterior (L-1):

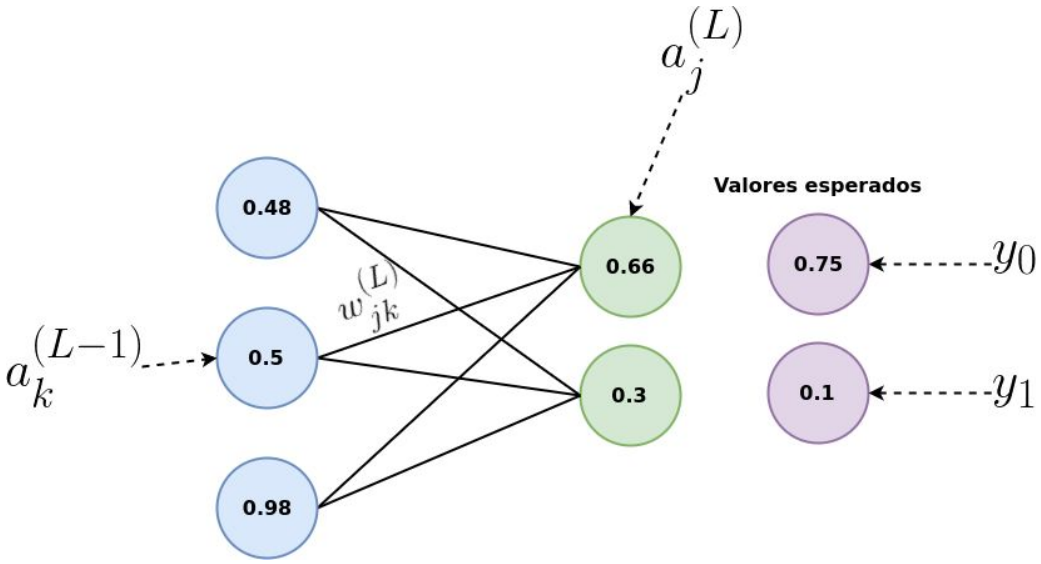
$$\frac{\partial C_0}{\partial a^{(L-1)}} = \frac{\partial z^{(L)}}{\partial a^{(L-1)}} \frac{\partial a^{(L)}}{\partial z^{(L)}} \frac{\partial C_0}{\partial a^{(L)}} = w^{(L)} * 2(a^{(L)} - y)$$

De esta relación es de donde proviene el nombre '*backpropagation*', porque el cálculo de la sensitividad de los pesos y de los bias sobre el costo, empieza desde la última capa hacia las primeras capas. Ahora se puede iterar utilizando la regla de la cadena hacia atrás sobre todas las capas para calcular qué tan sensible es el costo final sobre los pesos y los bias.

Algoritmo de backpropagation para una red neuronal multicapa

Si vemos un caso más complejo donde la red tiene varias neuronas por capa, las ecuaciones no varían mucho, el único cambio es que ahora vamos a tener una matriz de pesos y un vector de bias por cada capa por lo que vamos a tener que seguir con cuidado los índices de los pesos.

Imagen de una red neuronal de múltiples neuronas por capa:



Si la salida tiene más dimensiones, simplemente se calcula la suma?? el cuadrado de las diferencias para cada neurona de la última capa

$$C_0 = \sum_{j=0}^{n_L-1} (a_j^{(L)} - y_j)^2$$

$$z_j^{(L)} = w_{j0}^{(L)} a_0^{(L-1)} + w_{j1}^{(L)} a_1^{(L-1)} + w_{j2}^{(L)} a_2^{(L-1)} + b_j^{(L)}$$

$$a_j^{(L)} = \sigma(z_j^{(L)})$$

La derivada con respecto al un peso w_{jk}:

$$\frac{\partial C_0}{\partial w_{jk}^{(L)}} = \frac{\partial z^{(L)}}{\partial w_{jk}^{(L)}} \frac{\partial a^{(L)}}{\partial z^{(L)}} \frac{\partial C_0}{\partial a_j^{(L)}}$$

La derivada con respecto al un bias b_j:

$$\frac{\partial C_0}{\partial b_j^{(L)}} = \frac{\partial z^{(L)}}{\partial b_j^{(L)}} \frac{\partial a^{(L)}}{\partial z^{(L)}} \frac{\partial C_0}{\partial a_j^{(L)}}$$

Sabiendo que el costo con respecto a la siguiente capa es igual a:

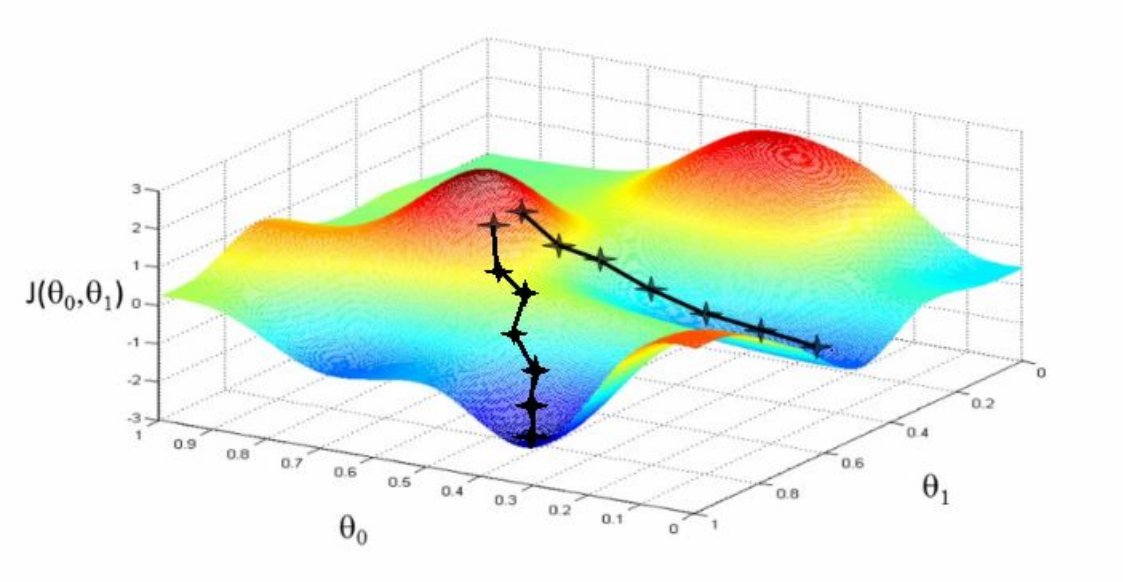
$$\frac{\partial C}{\partial a_j^{(l)}} = \sum_{j=0}^{n_l-1} w_{jk}^{(l+1)} \sigma'(z^{(l+1)}) \frac{\partial C}{\partial a_j^{(l+1)}}$$

O a la última capa:

$$\frac{\partial C}{\partial a_j^{(L)}} = 2(a_j^{(L)} - y_i)$$

Gradient Descent

Con estas últimas ecuaciones se calcula cada componente en el gradiente $\nabla C,$ y se pueden actualizar los pesos y buscar el mínimo de la función. Este método de actualizar los pesos y los bias se llama *'gradient descent'* o "descender por el gradiente". Hay muchas variantes del método que utilizan otras estrategias para alcanzarlo en mayor velocidad y evitando mínimos locales.



Redes Neuronales recurrentes y LSTM

Hasta este punto estuvimos analizando la evolución y el funcionamiento de las redes neuronales desde un simple perceptrón hasta las redes con múltiples neuronas y múltiples capas. Sin embargo, todas las que vimos tienen una característica que las definen: la información sólo 'fluye' hacia adelante y no hay ciclos o loops. Entran en la categoría de las "FEEDFORWARD NEURAL NETWORKS" . Una desventaja de estas redes es analizar información con dependencia temporal (secuencias de datos) como puede ser el análisis de texto, música, video o un traductor en tiempo real. Para resolver problemas de este tipo se necesita que la red tenga propiedades para guardar el contexto de la información que está siendo procesada.

Si ponemos como ejemplo esta frase: "Nací en Francia pero viví toda mi vida en Estados Unidos y manejo perfecto el español", y queremos armar un modelo para saber cual es idioma natal de esta persona, necesitamos que la red neuronal analice la frase palabra por palabra. Tendria que también guardar la información de la primera parte de la frase ("nací en Francia"), así poder guardar el contexto de la frase, y hacer una correcta predicción. Las redes neuronales carecen de memoria, por eso diseñaron una variación de redes neuronales que se llaman redes neuronales recurrentes o RNN.

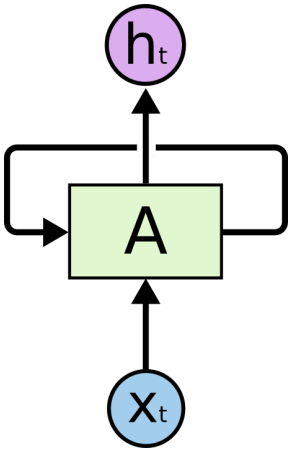
LSTM RNN

Las LSTM RNN son una variación de las RNN que resuelven varios problemas a la hora de entrenar la red, como el desvanecimiento del gradiente o la explosión del gradiente. Son complicaciones que surgen por el hecho de que al aplicar el algoritmo de *backpropagation,* los pesos/*weights* de estados anteriores terminan como poca importancia en el error/costó, cuando se deberían mantener a medida que la secuencia de datos es ingresada.

Como unidad básica, las LSTM no tienen perceptrones. Tienen LSTM *Cells* que internamente son combinaciones de multiplicaciones y sumas del estado anterior y el estado actual. También hacen uso de otros tipos de funciones de activación, como $tanh(x)$ o $relu(x)$. Las LSTM son un objeto actual de investigación y el diseño interno de

la LSTM no tiene un estándar aceptado por la comunidad, por lo cual varia mucho segun la implementación.

LSTM RNN Enrollada

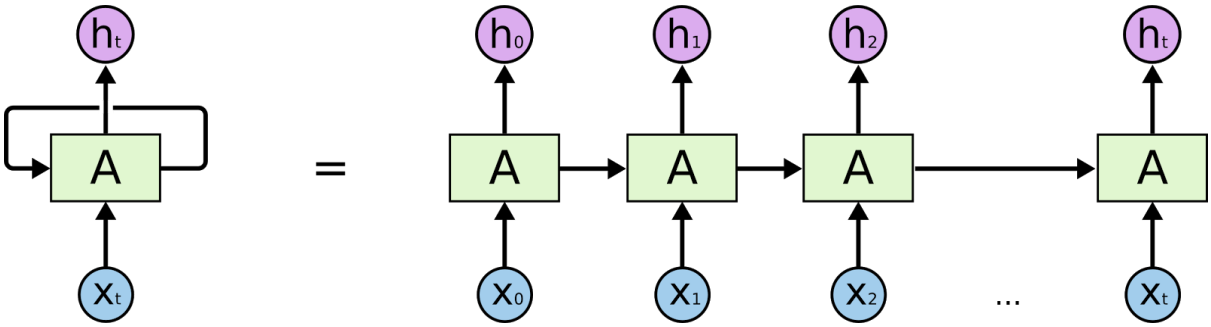


X_t : *es la entrada para un tiempo t* .

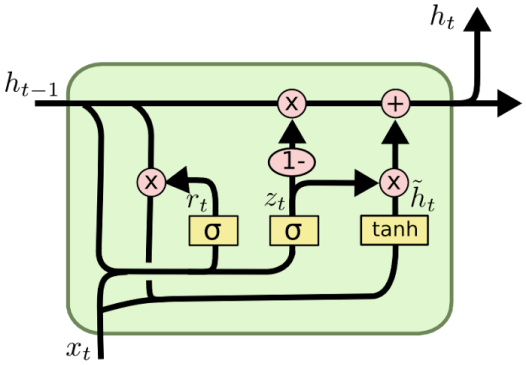
h_t : *es la salida para un tiempo t* .

A : *es la célula*

LSTM Desenrollada En El Tiempo



GRU LSTM Cell



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

Aprendizaje en las LSTM

La técnica para entrenar este tipo de redes neuronales sigue siendo el mismo por lotes y con *backpropagation*. Se agregan más funciones pero se sigue viendo como una gran funcion, compuesta por lo que derivada por regla de la cadena nos simplifica el camino.

Deep Learning Y El Futuro:

Con la explosión de la capacidad de procesamiento de las computadoras, los nuevos avances en el procesamiento paralelo, el crecimiento exponencial de la información y la necesidad de poder entenderla clasificarla y controlarla, impulsaron que se volvieran a explorar estos conceptos de machine learning y de redes neuronales. Los resultados son cada vez más prometedores, y es ahí que nace el concepto de *Deep Learning*. Ya no solo con algoritmos supervisados, en los que se necesita data a priori clasificada para poder entrenar las redes, sino con técnicas de aprendizaje no supervisadas y de aprendizaje reforzado. Estos algoritmos tienen la capacidad de entender y abstraer conceptos sobre cantidades de información mas grandes que la que el cerebro humano puede procesar.

En los últimos años se han publicado investigaciones sobre inteligencia artificial en múltiples áreas, incluyendo medicina, computación, educación, economía, medio

ambiente y en las que se destacan que estas herramientas tienen la capacidad para resolver complejos que hasta ahora no tenían solución.

Caso AlphaGO

En al campo de Inteligencia artificial siempre se encuentra la fantasía de que una máquina le gane a un humano. Uno de los más grandes retos era el juego GO.

AlphaGO es un programa de computadora que puede jugar al GO. Fue desarrollado por la empresa GOOGLE DEEPMIND en 2015 y fue el primer programa en ganarle a un jugador profesional. No fue la primera vez que una computadora compite contra un humano en juegos de mesa. En 1997 una supercomputadora llamada *Deep Blue* le gano al campeon del mundo de entonces Kaspárov. Lo que hizo diferente AlphaGo es que no fue diseñada para saber todas las posibles jugadas de ajedrez, y por fuerza bruta elegir la mejor jugada. El juego GO tiene más posibles jugadas que el ajedrez en una proporción mayor a la cantidad de átomos en el universo ($10^{80}\ atomos\ <<\ 10^{170}\ posibles\ jugadas$). AlphaGO aprendió a jugar el juego mediante una red neuronal y un método de entrenamiento llamado *'deep reinforcement learning',* o aprendizaje por refuerzo. Las piezas de GO son blancas o negras, esta red recibe imágenes de un tablero de Go como entrada y se empezó a entrenar mostrándole 100 mil jugadas de jugadores amateur de GO, que estaban disponibles en internet, para que aprenda cómo *imitar* a los jugadores. Luego, para mejorar su rendimiento, se la dejo jugando consigo misma 13 millones de veces y usando reinforcement learning. El sistema mejoro incrementalmente hasta mejorar su taza de partidas ganadas.

En las publicaciones más recientes de la empresa, demuestran que el algoritmo de AlphaGO puede ser utilizado para otros usos fuera de ganar GO, y que están desarrollando inteligencia artificial multipropósito que aprende usando *'reinforcement learning'.* Con este método de aprendizaje, la red se nutre del ambiente en la que se encuentra y se perfecciona según el objetivo que le indican.

Otros desarrollos de DeepMind:

- Lip Reading sentences: un modelo para reconocer los labios y traducirlo a texto
- Wave To Text : una red neuronal para autogenerar texto a partir de audio

SOPHiA Genetics

Es una compañía de medicina que aplica inteligencia artificial para resolver problemas relacionados con la salud. Usando BIG Data sobre secuenciamiento de ADN, crean algoritmos de *machine learning* para detectar cáncer de piel, de pulmón, de ovarios y de mama, detectando errores en el ADN.

Integrando aplicaciones con inteligencia artificial usando API

Estas APIS pagas permiten usar los modelos entrenados por google para integrarlas con una aplicación, sin necesidad de hacer un modelo a medida.

Google Vision Api

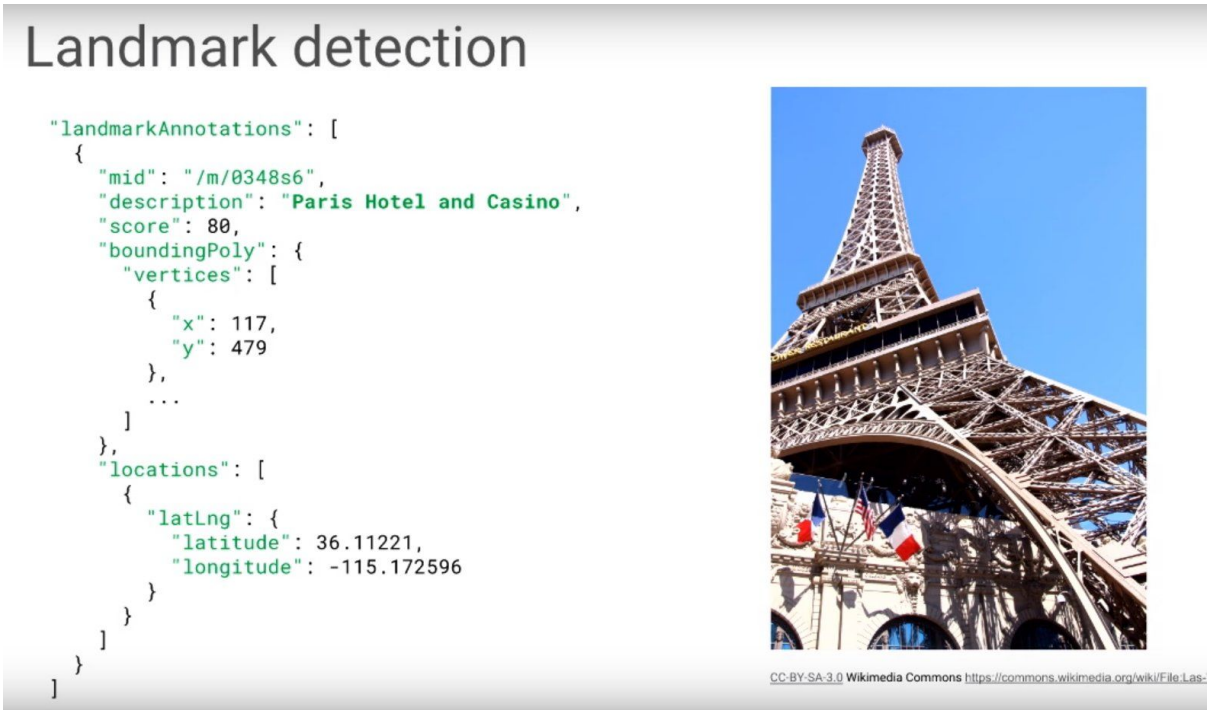
Es una herramienta para el procesamiento de imágenes que sirve para automatizar su clasificación. Envías una imagen el modelo de google, la procesa y te devuelve la clasificación de la imagen.

Permite:

- Detección de tags.
- Detección de rostros.
- Detección de contenido explicito.
- Detección de ubicaciones.
- Detección de logos.
- Reconocimiento de caracteres en la imagen.

Ejemplo:

Una imagen que parece ser de Paris pero en realidad es de Las Vegas, VISIÓN API es tan precisa que detecta la verdadera ubicacion.



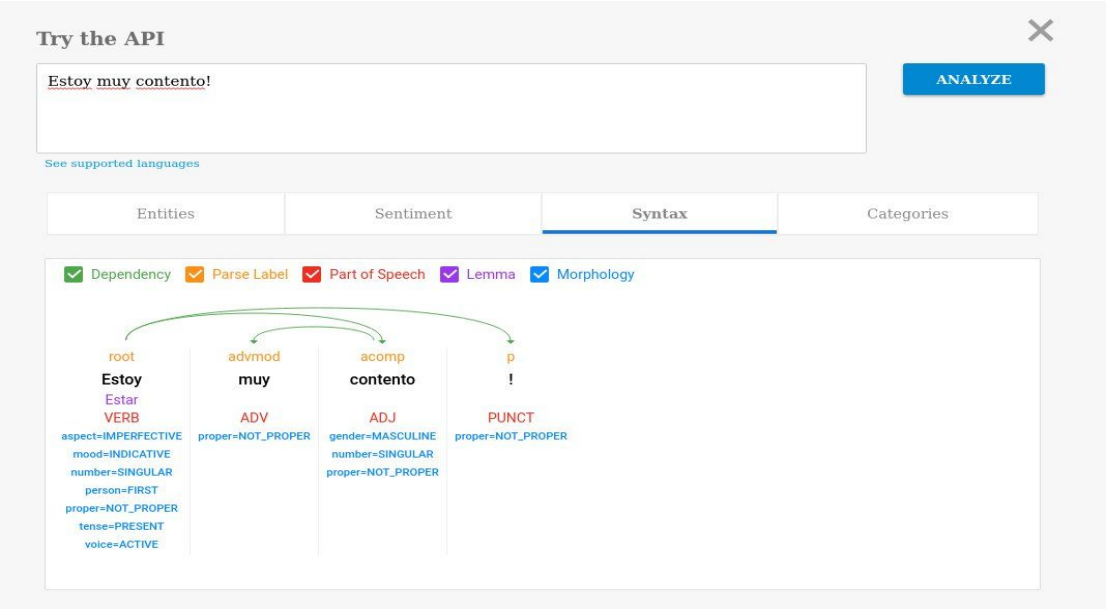
Speech Api

Permite mandar voz y devuelve un json con el texto de lo que mandaste.

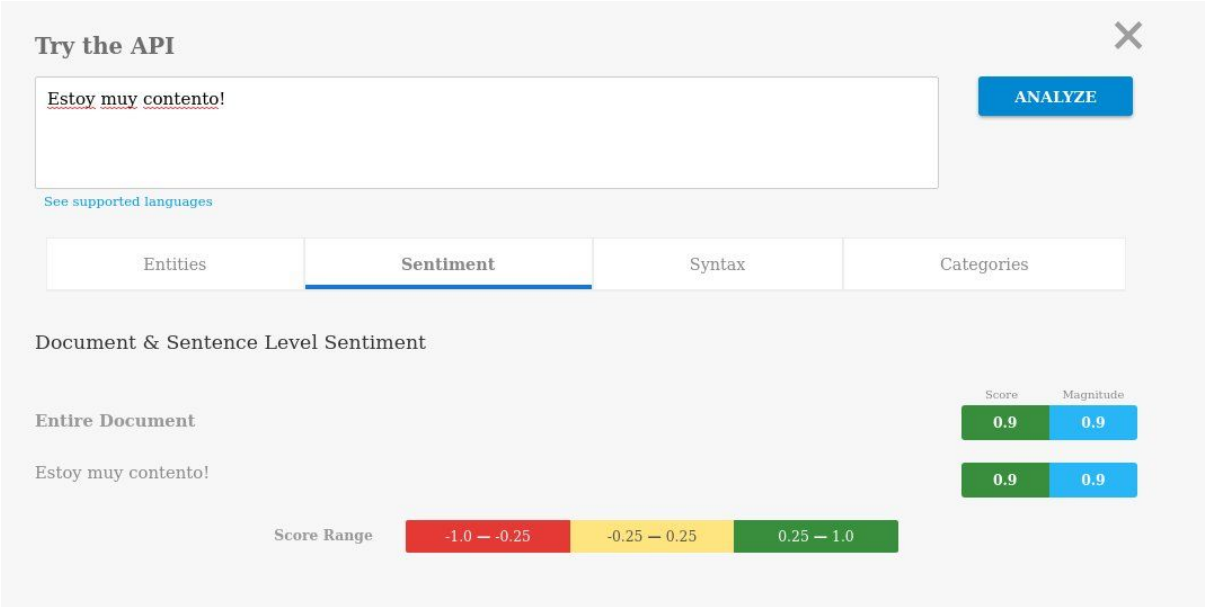
Natural Language Api

Permite analizar texto por sentimiento, semántica y sintáctica.

Sintaxis y semántica:



Sentimiento:



IMPLEMENTACIÓN

Luego de haber aprendido conceptos de *machine learning,* y de haber indagado la inmensa cantidad de problemas a resolver, decidí implementar mi propia red neuronal para el análisis de sentimiento de texto, para entrenar un modelo y luego diseñar una aplicación real para servirlo. El modelo recibe texto y devuelve su "sentimiento" en dos tipos de clases: POSITIVO / NEGATIVO.

Dividí el proyecto en dos grandes partes:

- PRIMERA PARTE: El análisis, diseño e implementación de la red neuronal, entrenamiento y la exportación del modelo.
- SEGUNDA PARTE: *Arquitectura para servir un modelo.*

PRIMERA PARTE: El análisis, diseño e implementación de la red neuronal; entrenamiento y la exportación del modelo

Analisis

Al momento empezar, tuve que analizar estos dos puntos:

- El *dataset*: Necesitaba encontrar un dataset de texto extenso, en español y con la clasificación de cada frase en negativo, positivo y neutral. Sin el *dataset* no me podía proponer a armar el modelo.
- El *framework*: Que framework usar para armar la red, o la posibilidad de implementar una propia con algun lenguaje que domine.

Dataset

El dataset lo arme de la combinación de dos *datasets* de tweets.

- TASS *dataset*
- *Dataset* público de tweets y sentimientos.

En total: 150 mil tweets clasificados.

Aclaración sobre el *dataset*:

El segundo *dataset* era el más grande, pero no estaba formateado. Sobre todo, no era de calidad; los tweets no fueron clasificados manualmente, sino que fue creado a partir de un script que los clasificaba según el emoji que tenía, feliz :), o infeliz :(.

Framework

Como *framework* para implementar la red neuronal, use *TENSORFLOW*. Es un software *open source* para cálculos numéricos que representa la información y las operaciones en un grafo de flujos. Como unidad básica de información utiliza arreglos multidimensionales (*tensores*), y los nodos del grafo se comunican solo por el flujo de estos tensores. El lenguaje de programación de este *framework* es Python. Lo elegí por la documentación propia y por su comunidad.

Luego de leer documentación básica de *tensorflow,* me descargue el repositorio de tutoriales, hice un clasificador de regresión lineal y también un clasificador de imágenes. Luego, me dedique a armar la red con las herramientas que ofrece *tensorflow* y siguiendo y basandome en la teoría de RNN.

Ejemplo básico de un programa en tensorflow

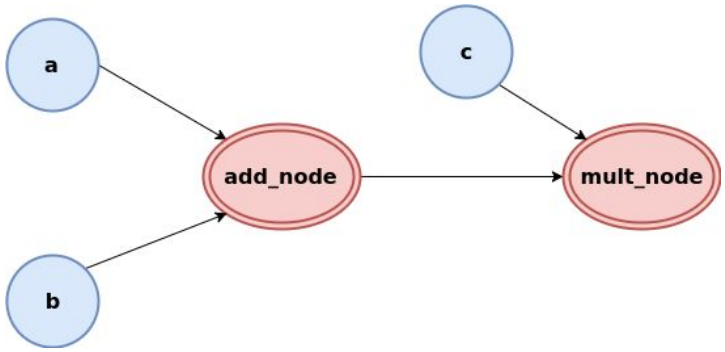
Si queremos representar la siguiente ecuación $(a+b)\,x\,C$

- Definimos las variables: $a\ b\ y\ c$ que van a ser los inputs del programa.
- Definimos también las operaciones $(+\ y\ \ x)$

En tensorflow:

```
3  import tensorflow as tf
4
5  #definimos las variables
6  a = tf.placeholder(tf.float32)
7  b = tf.placeholder(tf.float32)
8  c = tf.placeholder(tf.float32)
9
10 #definimos las operaciones
11 add_node = a + b
12 mult_node = add x c
```

El grafo interno es:



Nodos azules : variables

Nodos rojos : operaciones

Para correr el grafo hay que pasarle las entradas o *inputs:*

```
14 #corremos el grafo insertando los inputs
15 #imprimimos el valor de la multiplicacion
16 print(sess.run(mult_node, {a: 3, b: 4.5, c: 2}))
17
```

```
18
19 #tambien podemos imprimir el valor de la suma
20 print(sess.run(add_node, {a: 3, b: 4.5, c: 2}))
21
```

```
24 outputs:
25 => 7.5
26 => 15.0
```

Vectorización del dataset

ONEHOT

Una de las técnicas para vectorizar el texto es la de “ONE HOT”. Con esta técnica, se calcula el total de las palabras de todo tu *dataset*. Se mapea cada palabra en un vector de dimensión = tamaño del vocabulario, con una sola componente en “1” y las demás en zero.

Ejemplo:

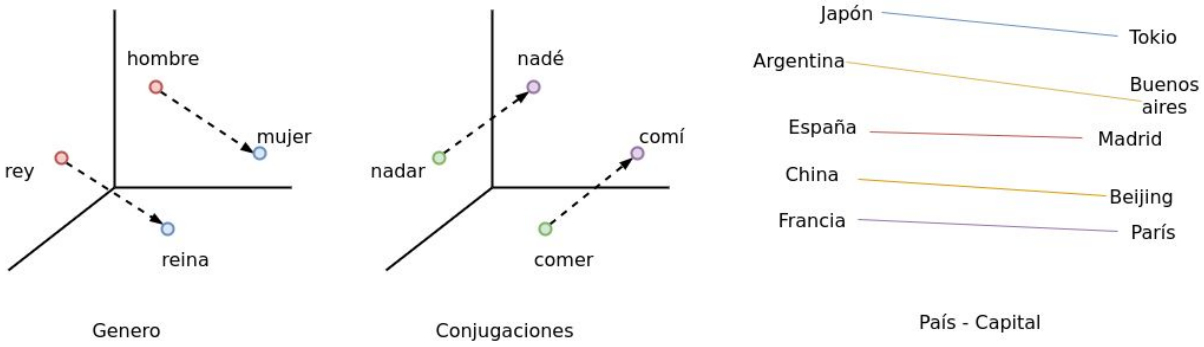
$$Vocabulario = \{feliz, noticia, auto, perro, gato\}$$
$$size(Vocabulario) = 5$$
$$onehot(feliz) = [1, 0, 0, 0, 0]$$
$$onehot(noticia) = [0, 1, 0, 0, 0]$$
$$onehot(auto) = [0, 0, 1, 0, 0]$$
$$onehot(perro) = [0, 0, 0, 1, 0]$$
$$onehot(gato) = [0, 0, 0, 0, 1]$$

Este tipo de vectorización tiene como ventaja la simple implementación. Sin embargo, la desventaja es que es poco eficiente; consume mucha memoria debido a que si aumenta el tamaño del vocabulario, aumentan las dimensiones de los vectores de entrada.

WORD2VEC

El segundo tipo de vectorización es de usar un modelo creado para vectorizar palabras en las que se mantiene constante el tamaño del vector y se mantiene una relación de distancias entre los vectores según el significado de las palabras que representan. El

modelo que use se llama *Word2Vec* y está entrenado con mil millones de palabras en español.



El modelo permite realizar operaciones como:

$$w2v[rey] - w2v[hombre] = w2v[reina]$$

Para el caso de análisis de texto, es importante que la distancia entre vectores (palabras) sea proporcional a su significado. Por eso decidí usar un modelo de *word2vec* para vectorizar las palabras.

Preparacion del dataset

Preprocesamiento del dataset

El preprocesamiento se basa en limpiar el dataset de todas las 'impurezas' que pueda tener el texto. Estas impurezas pueden ser símbolos extraños, palabras que no estén definidas en el modelo *word2vec*, los hashtags y las menciones de los tweets, la duplicación de letras en una palabra, las urls etc. Deje de lado este preprocesamiento en las primeras etapas del desarrollo y, como consecuencia, los primeros resultados de la red neuronal no fueron los esperados.

Vectorización

Luego de programar estas funciones, decidí vectorizar el dataset para tenerlo en el formato correcto con la que la red neuronal se iba a entrenar. Transformé cada vector de texto en un vector numérico de dimensión 300 usando *word2vec*.

Ejemplo de la vectorización de un tweet positivo de 5 palabras.

{"text": "@radio14 Hoy es un excelente dia! #soleado #bicicleta #empezoelverano", "klass": "positive"}

Mapeo de un tweet:

$$texto = \begin{bmatrix} "Hoy" \\ "es" \\ "un" \\ "excelente" \\ "dia" \end{bmatrix} \longrightarrow \begin{array}{c} \\ C_1 \\ C_2 \\ C_3 \\ C_4 \\ C_5 \end{array} \begin{array}{c} \begin{array}{cccc} A_1 & A_2 & \cdots & A_{300} \end{array} \\ \begin{pmatrix} 0.3 & -0.4 & \dots & i_{1,300} \\ -2.1 & 1.6 & \dots & i_{2,300} \\ 1.3 & -0.8 & \dots & i_{3,300} \\ 0.7 & -0.3 & \dots & i_{4,300} \\ 1.4 & 1.2 & \dots & i_{5,300} \end{pmatrix} \end{array}$$

Mapeo de un label:

$$sentimiento = ["positive"] \longrightarrow (1)$$

En cuanto al desarrollo del código, simplemente itere sobre los *tweets*, vectorize cada uno y arme los tensores. Dimensión de tensor: $largo\ del\ tweet\ x\ 300$. Largo de cada *tweet*: 20 palabras (como máximo). Tamaño definido por el modelo *word2vec*: 300.

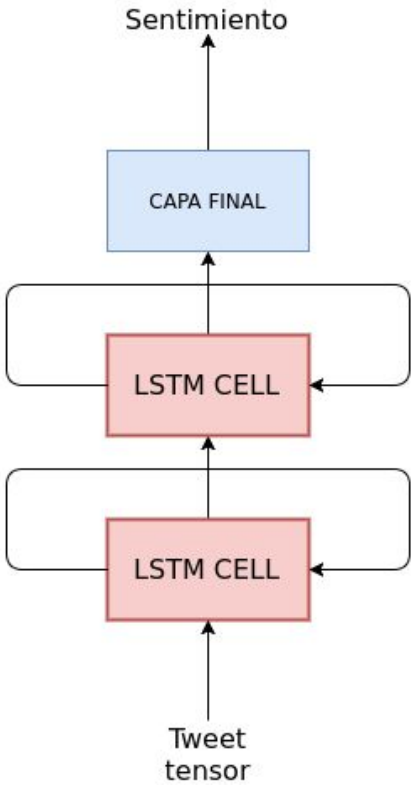
Luego, exporte los tensores que van a ser importados por la red neuronal para el entrenamiento:

```
# load http://crscardellino.me/SBWCE/ trained model
model = gensim.models.KeyedVectors.load_word2vec_format('SBW-vectors-300-min5.bin', binary=True)

shape = (len(tweets), MAX_NB_WORDS, 300)
tweets_tensor = np.zeros(shape, dtype=np.float32)

for i in range(len(tweets)):
  #vectorizing each word in the tweet with a vector shape = (300,)
  for f in range(len(tweets[i])):
    word = tweets[i][f]
    if f >= MAX_NB_WORDS:
      continue
    #if is not in the vocabulary
    if word in model.wv.vocab:
      tweets_tensor[i][f] = model.wv[word]
    else:
      #if it is a mention vectorize a name, for example @michael123 -> would be Carlos
      if word[0] == '@':
        tweets_tensor[i][f] = model.wv[name()]
      #if not append the unknown token
      else:
        tweets_tensor[i][f] = model.wv['unk']
  #End of sentence token
  if(f<MAX_NB_WORDS):
    tweets_tensor[i][f] = model.wv['eos']

labels_array = np.array(list(map(lambda label: label_to_value(label), labels)), dtype=np.int32)

np.save(outputfile + '_vec_tweets.npy',tweets_tensor)
np.save(outputfile + '_vec_labels.npy',labels_array)
```

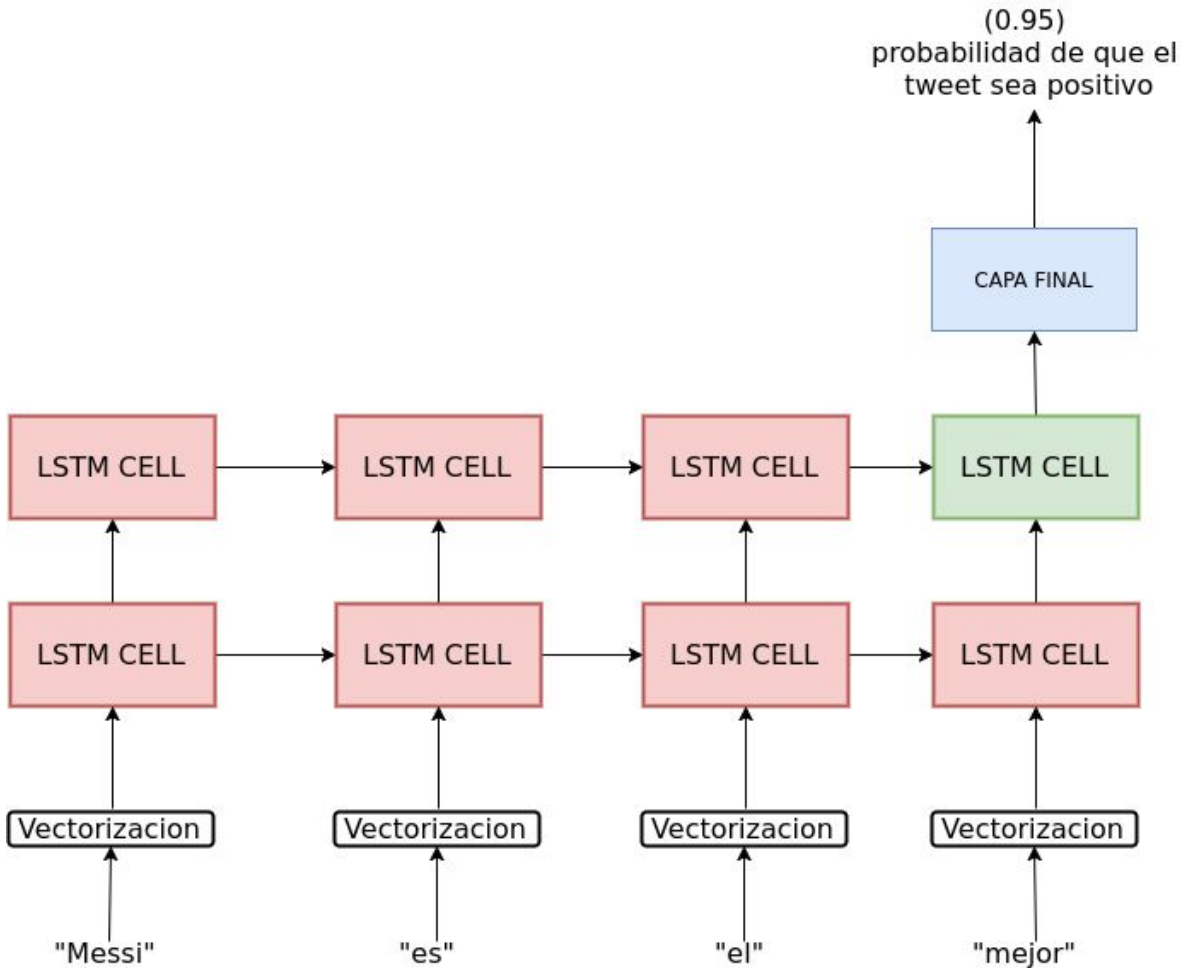
Diseño y desarrollo de la red neuronal:

Basandome en las prácticas utilizadas actualmente, y la documentación de *tensorflow,* utilice una LSTM RNN que experimentalmente tiene los mejores resultados a la hora de analizar secuencia de datos, en particular de texto.

Diseño de la LSTM RNN enrollada:



Para el caso de un tweet específico podemos ver como va a actuar en la secuencia.



Un punto a recalcar es que este ejemplo. Tiene dos capas de células LSTM. La última célula de la segunda capa es la que va a contener la información de todo el tweet. Por eso es que se decide utilizar esa última célula para calcular la predicción.

Inputs

Tensorflow utiliza '*Placeholders'* para definir las entradas a una red neuronal. Los *placeholders* son los primeros nodos del grafo. Hay que definir el tamaño del lote de entrada, la cantidad de palabras por cada tweet y el tamaño del vector y el tamaño de los labels/sentimientos.

'Tweets' y '*labels'* son tensores, es decir, arreglos multidimensionales. Durante el entrenamiento, '*tweets'* va a ser un lote de tweets y '*labels'* va a ser el lote de los sentimientos para cada tweet en el lote. Se separan porque los *'labels'* solo se van a usar al momento de calcular el error. Cuando el lote de tweets pase por la red, va a salir un tensor de predicciones que después será comparado con este tensor de *'labels'* que definimos acá.

RNN multi capa

Se define la cantidad de capas que va a tener la red y el tipo de neurona o célula a usar. En mi implementación, son LSTM cells.

```
# make the lstm cells, and wrap them in MultiRNNCell for multiple layers
def lstm_cell():
    cell = tf.contrib.rnn.BasicLSTMCell(hidden_size)
    return tf.contrib.rnn.DropoutWrapper(cell=cell, input_keep_prob=keep_prob, output_keep_prob=keep_prob)

multi_lstm_cells = tf.contrib.rnn.MultiRNNCell([lstm_cell() for _ in range(number_of_layers)], state_is_tuple=True)
```

Luego defino que a la red va a ingresar como entrada el lote de tweets que defini en los inputs. De esta función puedo obtener las salidas de la red y definir la operación/variable *final_state* que va a tener un tensor, con la concatenación de todas las salidas para cada elemento del tensor de input (*tweets*). En otras palabras, voy a obtener la salida de la red para cada tweet de lote en una colección.

```
# Creates a recurrent neural network
_, final_state = tf.nn.dynamic_rnn(multi_lstm_cells, tweets, dtype=tf.float32, initial_state=zerostate)
```

Ahora tengo que definir la última capa. Esta capa es la que 'moldea' la salida de la red para que tenga el formato deseado. Esta función la había diseñado manualmente, pero a la hora de inicializar los pesos, la función que ofrece *tensorflow* es más eficiente. Básicamente, es una matriz de pesos y *bias* que, por la regla de multiplicaciones de matrices, puedo obtener un vector con la forma deseada.

$$final\_state[-1][-1] \in \mathbb{R}^{(size\ del\ lote \times cantidad\ de\ neuronas\ por\ celula)}$$

$$weight \in \mathbb{R}^{(cantidad\ de\ neuronas\ por\ celula \times cantidad\ de\ clases)}$$

$$bias \in \mathbb{R}^{(cantidad\ de\ clases)}$$

$$sentiments = (final\_state \times weight) \oplus bias$$

Aclaración: La suma del *bias* no es una suma directa. En cambio, se llama suma *broadcast*, en la cual se suma ese vector a todos los elementos del otro vector.

```
44
45 sentiments = tf.contrib.layers.fully_connected(final_state[-1][-1],
46                                                num_outputs=number_of_classes,
47                                                activation_fn=None,
48                                                weights_initializer=tf.random_normal_initializer(),
49                                                biases_initializer=tf.random_normal_initializer(), scope="fully_connected")
50
```

Finalmente, ahora que tengo los sentimientos de cada tweet, tengo que transformarlos en valores de 0 a 1 (probabilidades). Como tengo solo dos clases, y el vector para cada tweet es de dimensión 1, se usa *sigmoid(x)*. Si tuviera mas clases, y estuviera clasificando los tweets por su contenido (deportes, música, política, cine, arte), se usaría una función que transforma un vector de varias dimensiones a una distribución de probabilidades. Esa función es muy popular; se llama *SOFTMAX(x)*.

Cálculo del costo/error/loss

Utilice la fórmula más usada; se aplica la función de *sigmoid(x)* a la salida de la última capa, y se calcula la distancia con la salida deseada.

```
51
52 with tf.name_scope("loss"):
53     # define cross entropy loss function
54     losses = tf.nn.sigmoid_cross_entropy_with_logits(logits=sentiments, labels=labels)
55     loss = tf.reduce_mean(losses, name="loss_op")
56     #tensorboard summaries
57     tf.summary.scalar("loss", loss)
58
```

Optimizacion

Tensorflow ofrece varias técnicas de *Gradient Descent*. Algunas tienen mejor resultado que otras. En mi caso, probé con *Adam,* que es una pequeña variación en las últimas investigaciones del tema, y vi que era muy popular.

```
71  # define our optimizer to minimize the loss
72  with tf.name_scope("train"):
73      optimizer = tf.train.AdamOptimizer(learning_rate).minimize(loss)
74
```

Importación del dataset vectorizado

Cabe destacar que, para poder entrenar un algoritmo de aprendizaje supervisado, es necesario separar la información en *entrenamiento* y *validación*. El entrenamiento es la información con la que la red va a aprender. La validación es la información que el ingeniero puede hacer: estudios sobre la *performance* de la red, evaluando el modelo en ejemplos con los que no ha tenido contacto. En general, la red neuronal va a tener una mejor *performance* en el *dataset* de entrenamiento que en el de validación.

```
87  # load our data and separate it into tweets and labels
88  train_tweets = np.load('data_es/train_vec_tweets.npy')
89  train_labels = np.load('data_es/train_vec_labels.npy')
90
91  test_tweets = np.load('data_es/test_vec_tweets.npy')
92  test_labels = np.load('data_es/test_vec_labels.npy')
93
```

El último paso importante a destacar es el ciclo de entrenamiento. Es un loop en el que va a entrar distintos lotes del *dataset* y se van a calcular los costos para poder optimizar la red después de cada vuelta.

Ciclo de entrenamiento

Los *epochs* son un número que define la cantidad de veces que se va a recorrer el *dataset*, y el número de *steps* que se define para separar el *dataset* en lotes.

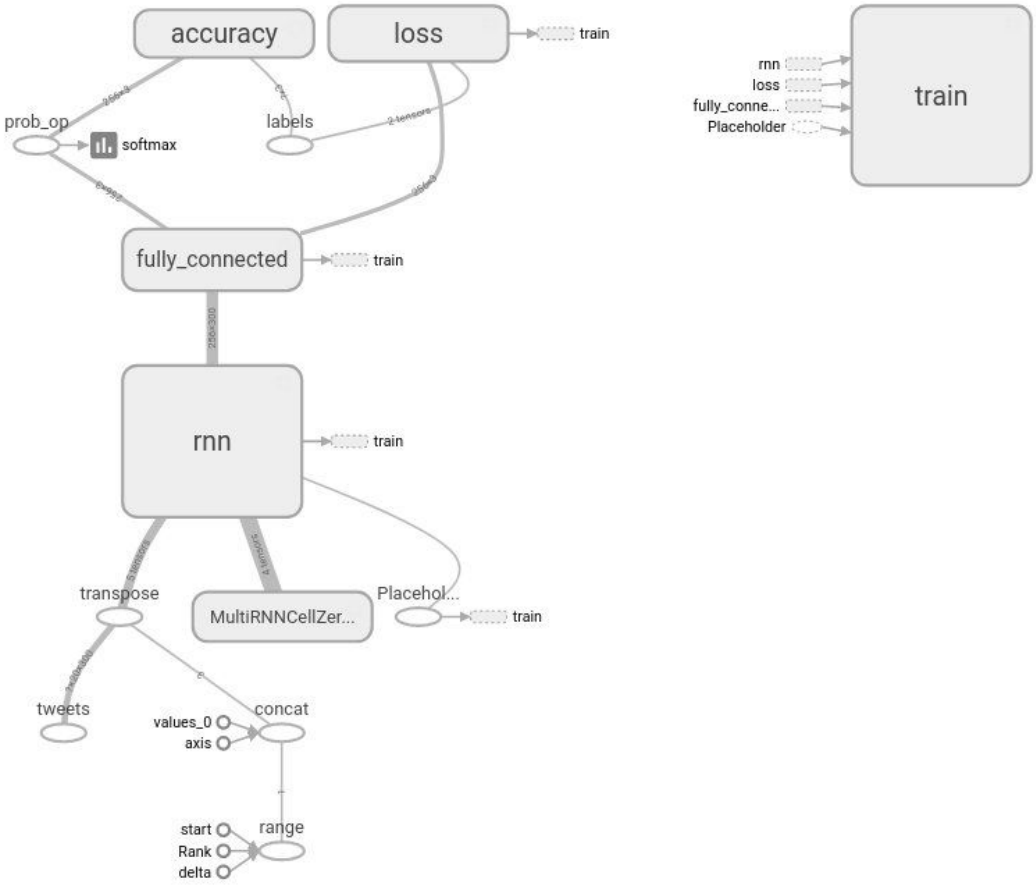
```
 99  for epoch in range(num_epochs):
100      for step in range(steps):
101
102          offset = (step * batch_size) % (len(train_tweets) - batch_size)
103          batch_tweets = train_tweets[offset : (offset + batch_size)]
104          batch_labels = train_labels[offset : (offset + batch_size)]
105
106          data = {tweets: batch_tweets, labels: batch_labels, keep_prob: 0.8}
107
108          #run operations in graph
109          _, loss_train, accuracy_train = session.run([optimizer, loss, accuracy], feed_dict=data)
110
```

El punto clave está en que el grafo se definió pero no se ejecuta hasta que no se le pase información. *session.run()* es la función que llena el grafo con la información, y a la que se le pueden pasar las funciones a ejecutarse durante esa ejecución. En este caso, le pasamos la *variable optimizer* para que optimice, y las de *loss* y *accuracy* para poder medir en un gráfico la *performance* durante el entrenamiento.

LOSS y ACCURACY

Estas dos variables son las que nos van a indicar la evolución del modelo, y cómo aprende conceptos sobre los datos de entrenamiento. Uno desea que el *loss* disminuya y tienda a cero, y que el *accuracy* aumente y tienda al 100%.

El grafo de *tensorflow* final:



Como Fue El Entrenamiento?

Una de las dificultades con las que me encontré fue que mi computadora personal no tiene la capacidad necesaria para entrenar una red de este tipo en tiempos manejables. Un entrenamiento completo tomó aproximadamente 4 horas. Hice alrededor de 40-50 entrenamientos, pero no completos. Cuando veia que habia divergencia, o que no aumentaban el nivel de predicciones, cortaba el entrenamiento. Muchas veces a los 30 minutos de empezar. Aplique cambios al sistema y al manejo del *dataset* hasta obtener un modelo que considero aceptable para los objetivos del trabajo.

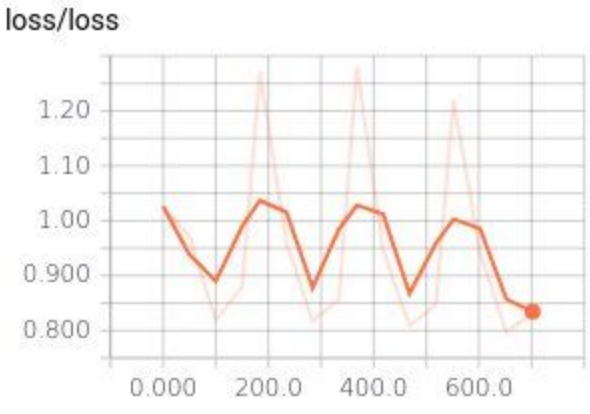
Qué variables entran en juego a la hora de entrenar un modelo?

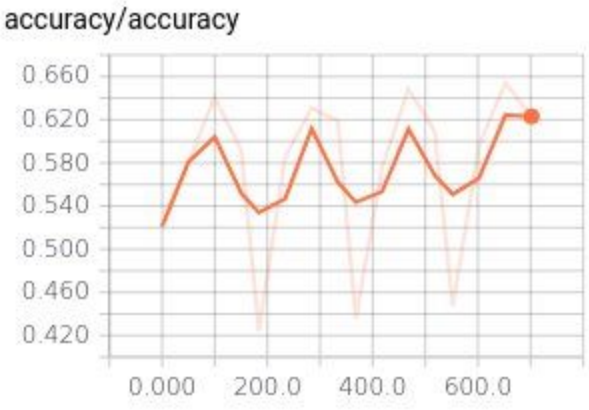
- La cantidad de capas
- La cantidad de neuronas por capa
- El tamaño del lote
- La vectorización y el tratamiento del dataset
- Las funciones de activación
- La cantidad de epochs
- El learning rate (es la agresividad con la que se actualizan los weights)
- El dropout (es una técnica para sobreponerse al *overfitting*)
- Los metodos de optimizacion (*Gradient Descent, Adam, Adadelta*, etc)

Me guié mucho por un paper de *Google* que voy a documentar abajo, sobre la diferencia en performance de los distintos algoritmos de optimización y la capacidad de la red.

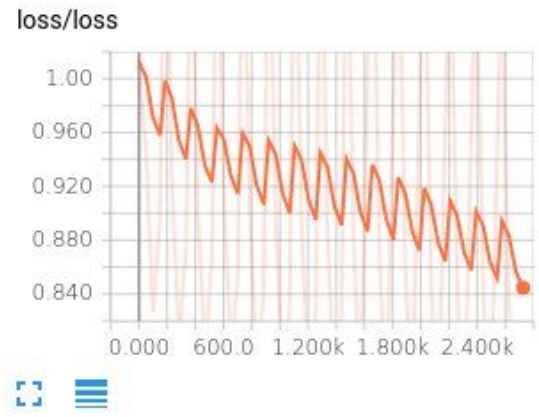
Voy a detallar los principales problemas con los que me encontré:

- Falta de inicialización de los *weights* y *biases*.

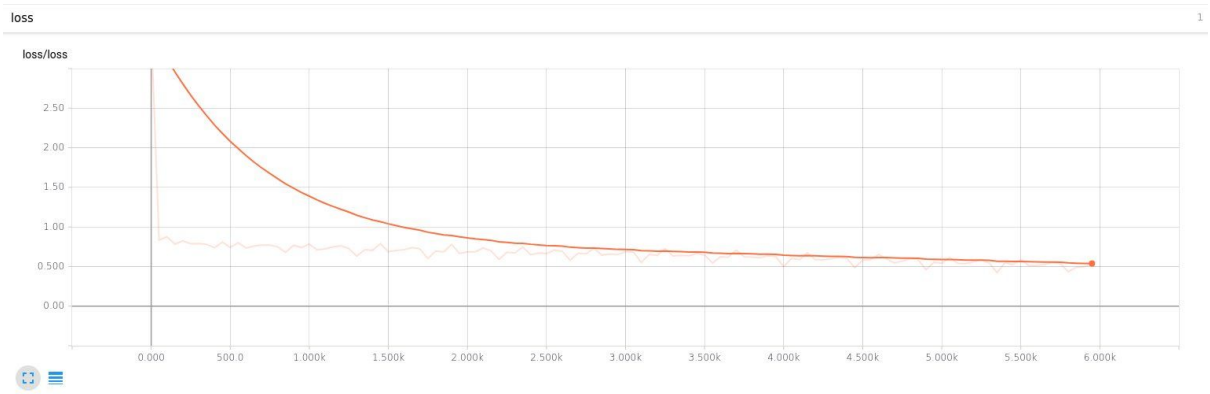


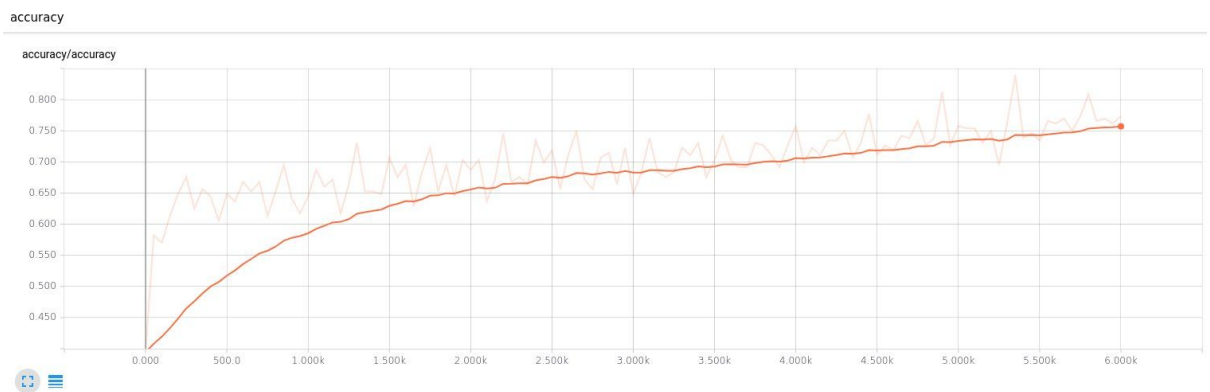


- Falta de ejemplos en el *dataset*: En un comienzo quise clasificar los *tweets* en positivos, negativos y neutrales. Y falle en no verificar que tenia la misma cantidad de ejemplos para cada clase. Tenía 10 veces más ejemplos positivos que neutrales. Lo que me indicó el problema fue que al arrancar el entrenamiento el accuracy era ya del 50%. Claramente, si tengo tres clases para clasificar el entrenamiento no puede arrancar en 50% debería arrancar en 33%, que es la probabilidad correcta de acierto sin entrenamiento.

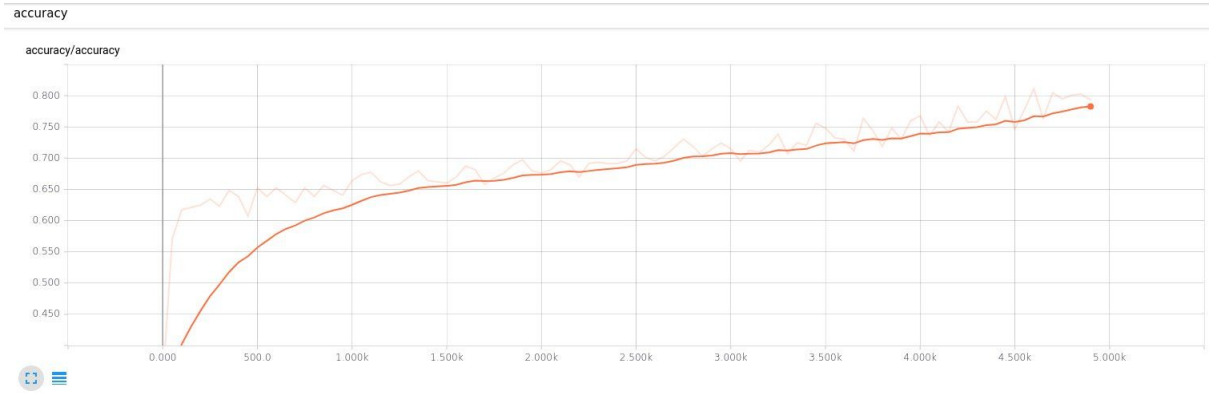


- Después de arreglar ese problema en el *dataset*, elimine los neutrales: Modifique la red para solo soportar dos clases, y por fin me encontré con la correcta forma de la función de *loss* y *accuracy*.



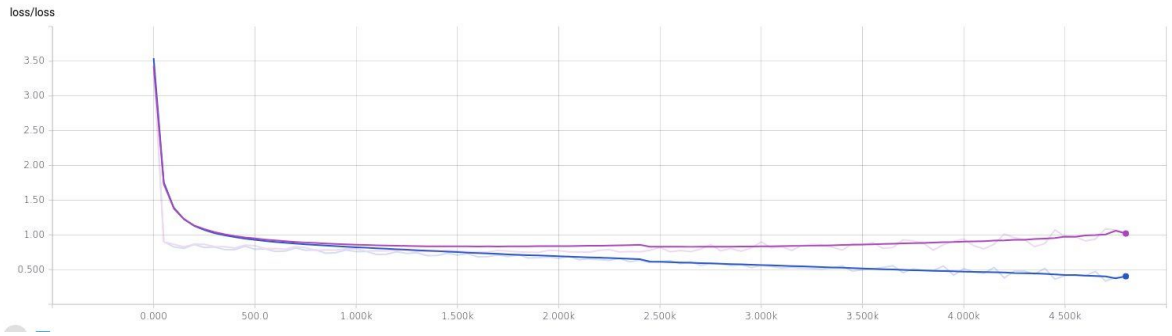


- El siguiente gran problema con el que me encontré es que la red tenía *overfitting*. El gráfico de *accuracy* para la información de entrenamiento se disparaba hacia el 100%, pero el *accuracy* de *testing* no pasaba el 60%. Este problema es muy común. Se debe a que la capacidad de la red es tan grande que se memoriza, o ajusta, los pesos para que sea perfecta para la información de training. Tuve que agregar una función de *tensorflow* que se llama *'dropout'*; lo que hace es 'apagar neuronas' aleatoriamente para forzar aprendizaje

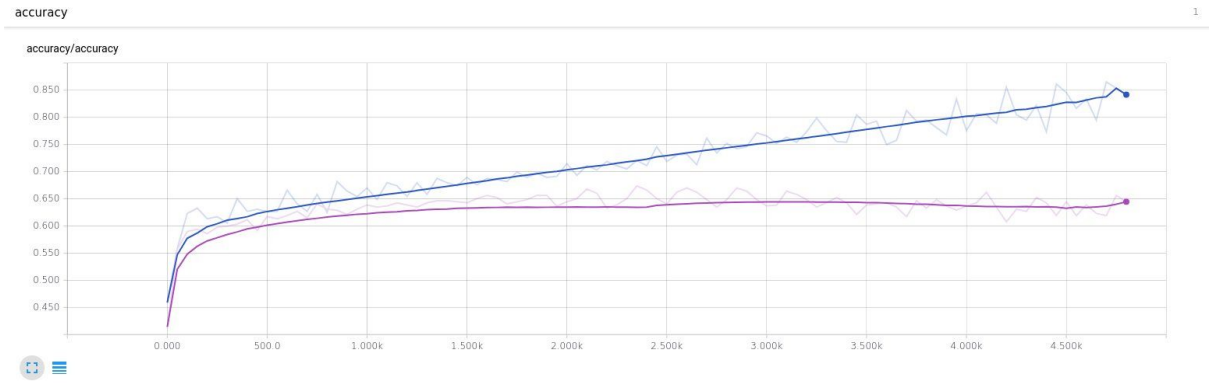


- Luego de esto me dedique exclusivamente a aumentar el *accuracy* de validación/*testing* porque es lo que indica, en una buena medida, como se va a comportar el modelo para data real nunca vista. La red siguió experimentando mucha divergencia entre las variables de *testing* y *training*.

Loss:



Accuracy:



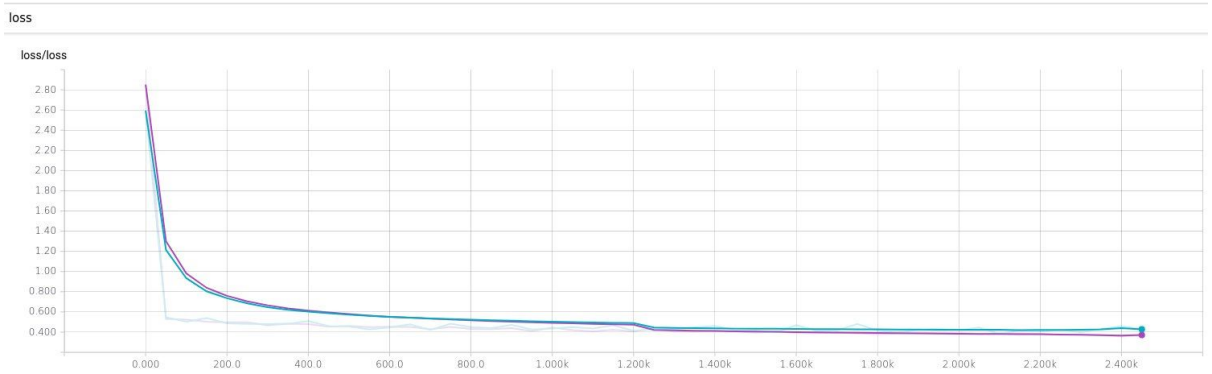
- El ***DATASET***, generador de todos los problemas.

  Después de varios intentos (de cambiar los parámetros, aumentar y disminuir la capacidad, agregar mas controles para analizar la *performance,* e incluso comentar en la comunidad de *tensorflow* y *stackoverflow* sobre mi problema) me dedique a volver a ver la información sobre la cual estaba entrenando la red.

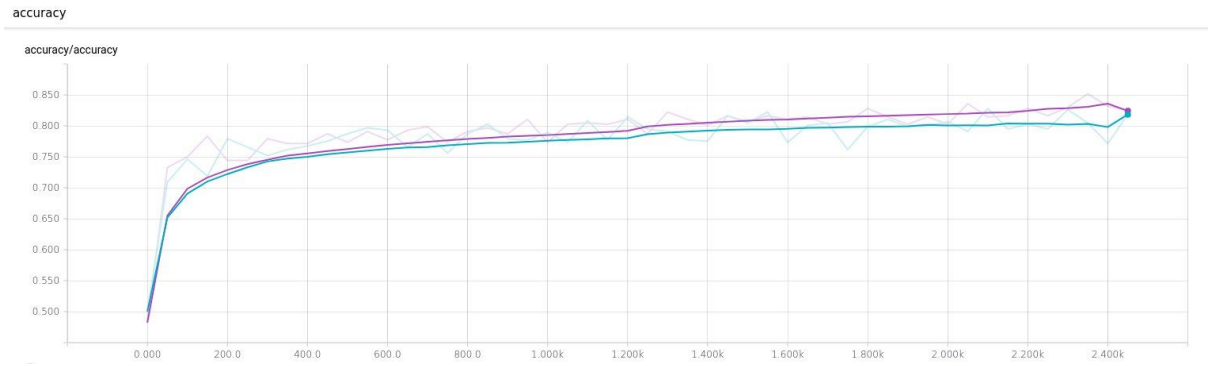
  a. Mejore, con expresiones regulares, los filtros para eliminar URLs, hashtags(#), signos de puntuación, numeros, menciones (@), y reduje palabras con multiples vocales (jajajaaaaa, ajajaaaa, jaajaaa => jaja)
  b. Agregue un *tokenizador* que permite detectar si la palabra está en el idioma, y si no la descarta. Este tokenizador también elimina espacios, tabulaciones y puede separar palabras que estén pegadas sin espacios.
  c. Como el modelo no tiene todas las palabras del idioma, las palabras que no las encontraba las vectorize a un vector *'unknown'*. Cuando se terminaba la frase, agregue un vector para delimitar el fin de la frase *'eos vector'*.

El resultado final fue este:

LOSS:



ACCURACY:



El máximo accuracy que logre en testing fue de 80%. Esto no quiere decir que el modelo va a predecir correctamente el 80% de los tweets que ingresen, está muy lejos de eso. Esto indica que pudo predecir correctamente con esa precisión el *dataset* de testing (20 mil tweets). El punto más importante de todo el trabajo cuando se busca *performance* es el *dataset*. El *dataset* con el que fue entrenada la red es muy limitado. Sin embargo, fue el más grande que pude hallar.

Después de este último entrenamiento, exporte el modelo y me dedique a armar una aplicación para servirlo.

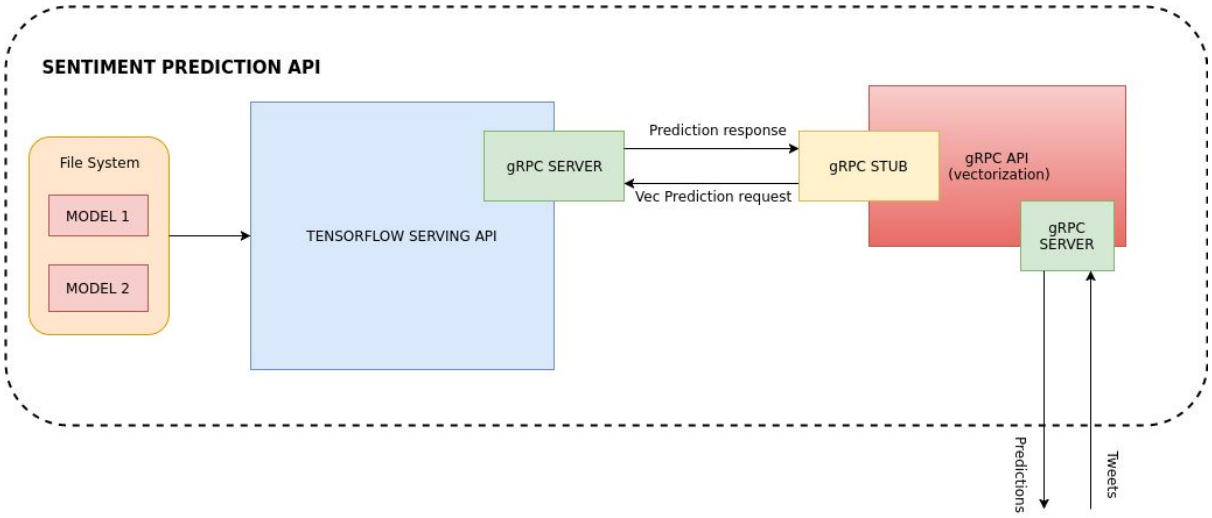
SEGUNDA PARTE: Arquitectura para servir un modelo.

Tensorflow Serving API

Una vez exportado el modelo, utilice la herramienta que ofrece *tensorflow* llamada, *tensorflow serving API,* que se encarga de versionar los modelos y servirlos. Los problema con los que me encontré fueron los siguientes:

1. Para comunicarse con el módulo de *tensorflow,* se utiliza un protocolo llamado gRPC con el que no había trabajado.
2. No puedo enviarle al modelo el texto; primero tengo que vectorizarlo de la misma manera que lo vectoricé en el entrenamiento.

Como solución decidí aprender cómo funciona este protocolo, y crear un pequeño programa en Python que funcione como *middleware*. Con esto, puede recibir texto, transformarlo en vectores, enviarlo al *tensorflow serving API,* recibir la respuesta y enviarla al cliente.

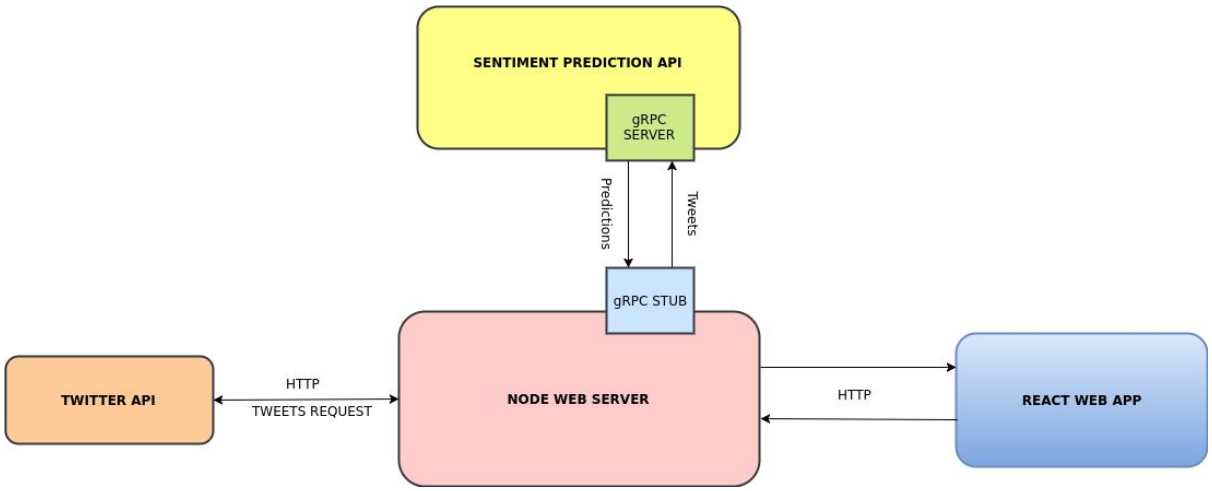


gRPC

gRPC es una librería *open source* multiplataforma que utiliza como capa de transporte a HTTP/2. Es bidireccional y tiene un *throughput* 3 veces mayor que *http*. Podría haber tomado la decisión de que este módulo, 'Sentiment prediction API', reciba

mensajes *http,* pero, al estudiar un poco sobre el protocolo RPC, decidí continuar con la misma estructura que el módulo de *tensorflow*.

El siguiente paso fue crear un servidor web que se comunique con el servicio de predicción, y un *frontend* para la interacción con el usuario.



Servidor WEB

Una vez armado el módulo de predicción, diseñe un pequeño servidor WEB en NODE JS y EXPRESS JS.

Solo defini una ruta:

- HTTP GET /api/predict

Esa ruta recibe como parámetro:

- input=[string]
- input=[string] twitter_api=true

Si solo se envía un input, ese texto es enviado al módulo de predicción y devuelve al cliente la respuesta. Si *twitter_api* es *'true',* el controlador se comunica con la API de Twitter, trae los últimos 20 tweets con el hashtag enviado en el input, los manda a analizar al modelo y luego devuelve la respuesta al cliente.

El controlador del servidor web:

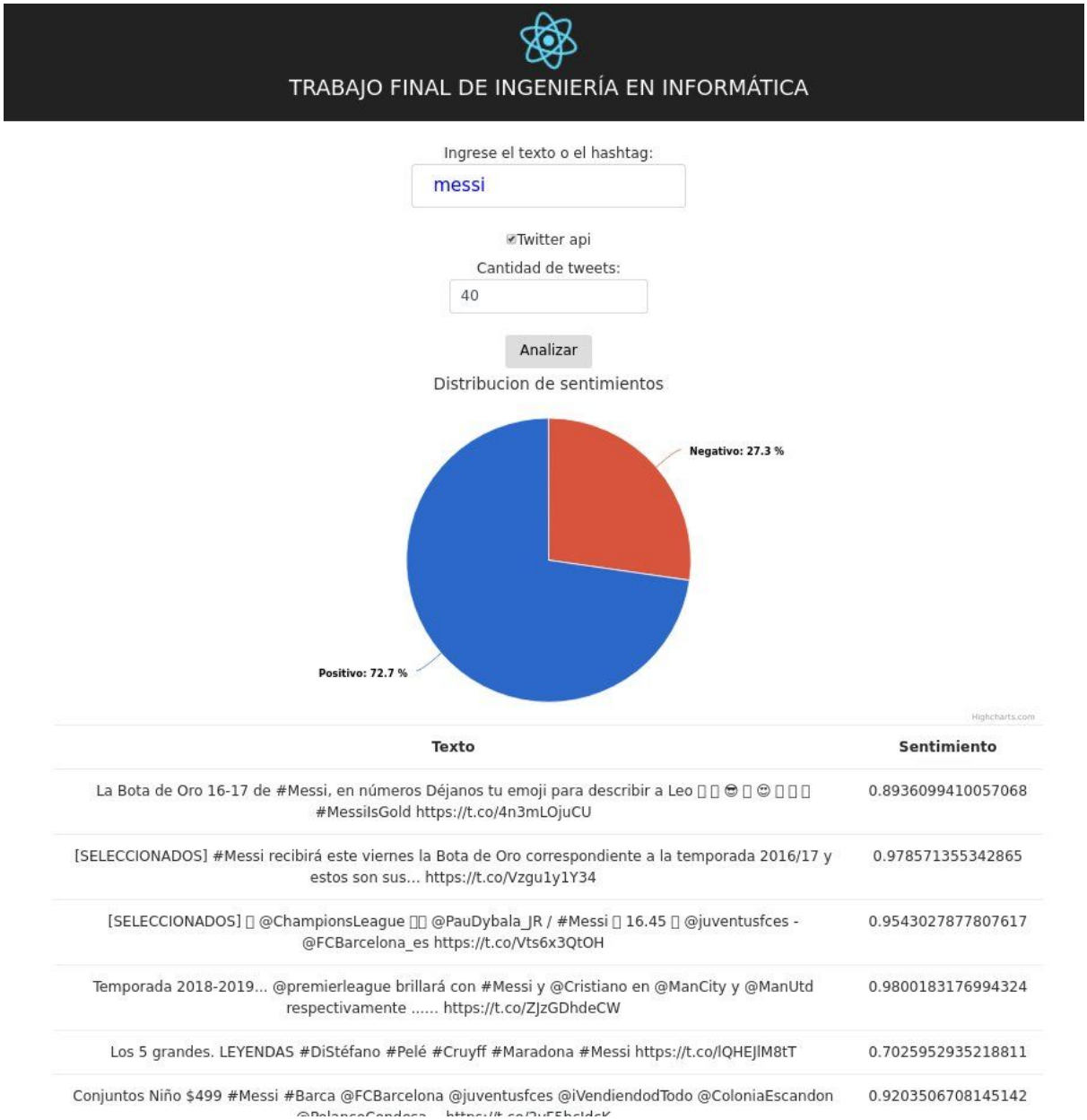
```
exports.predict = (req, res, next) => {
  const {input, twitter_api, n} = req.query
  if(!input){
    return _jsonError(res, { message: "Bad request, no input query." }, 400)
  }
  if(twitter_api === "true"){
    //first retive tweets and then send it to prediction api
    let count = parseInt(n)
    if(isNaN(count) ||  count > 50)
      count = 10
    let hashtag = input
    if(!input.startsWith('#'))
      hashtag = '#' + input
    client.get('search/tweets', {q: hashtag, lang: 'es', count: count, exclude:'retweets', result_type:'mixed'})
      .then((tweets, response) => {
          return tweets.statuses.map((raw_tweet) => raw_tweet.text)
        })
      .then(async (tweets_list)=>{
        let predictions = []
        predictions = await prediction_client.predict(tweets_list)
        return Object.assign({tweets_list}, predictions)
        })
      .then((result)=>{
        res.json(result)
       })
      .catch((err)=>{
        _jsonError(res, error, 500)
      })

  }else{
    //send text to prediction api
    prediction_client.predict(input)
      .then(predictions => res.json(Object.assign({tweets_list: [input]}, predictions)))
      .catch(err => _jsonError(res, err, 500))
  }
}

let _jsonError = (res, err, status) => { res.json({ message: err.message, status: status }) }
```

FRONTEND

El *frontend* lo desarrolle también con otra tecnología ampliamente usada por la industria. REACT JS es un librería de *javascript* para crear interfaces de usuario.

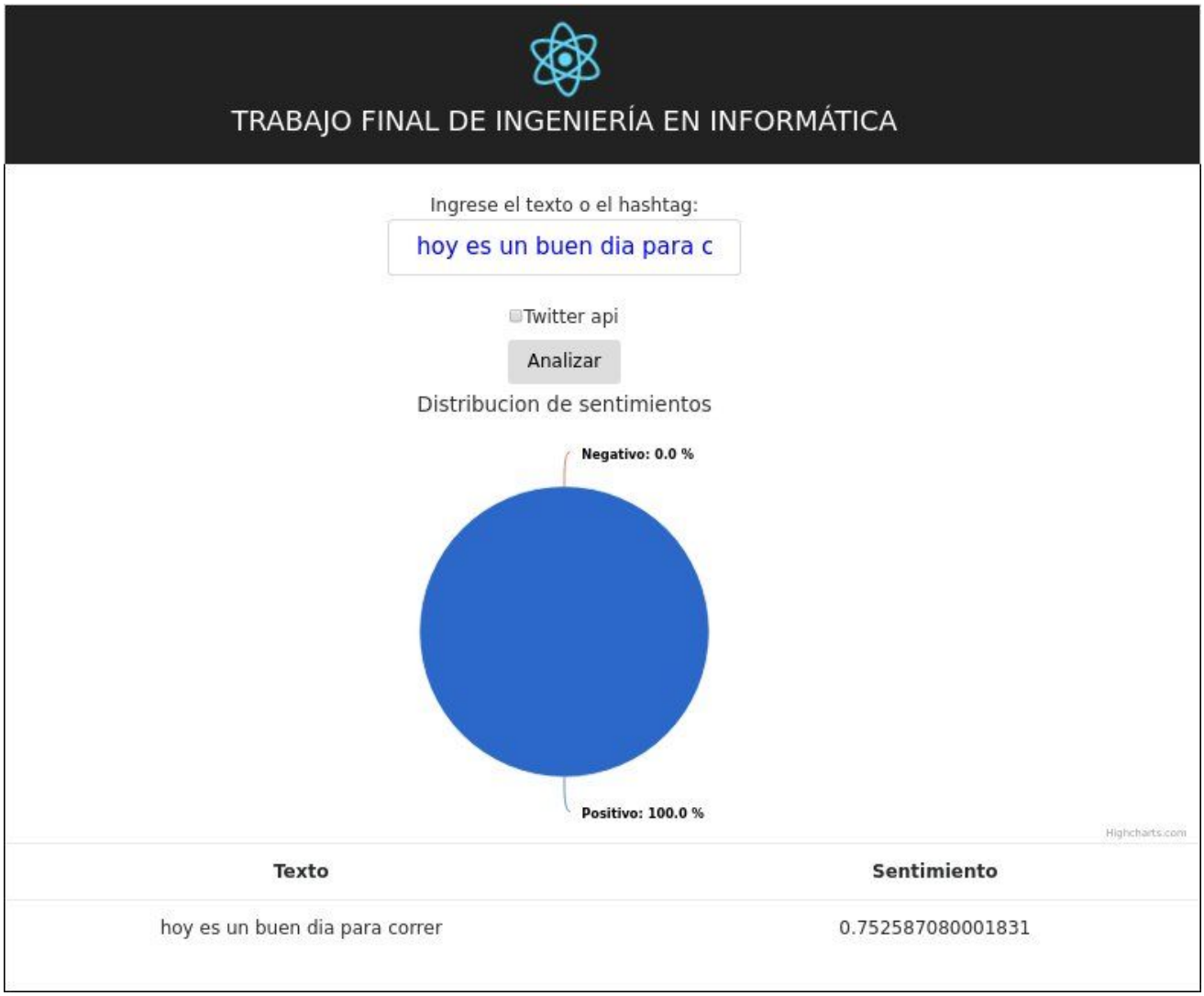
Esta aplicación es la encargada de interactuar con el usuario, recibir los inputs, enviarselos al servidor web, esperar los resultados y presentarlos.

Ejemplo usando la API de Twitter:

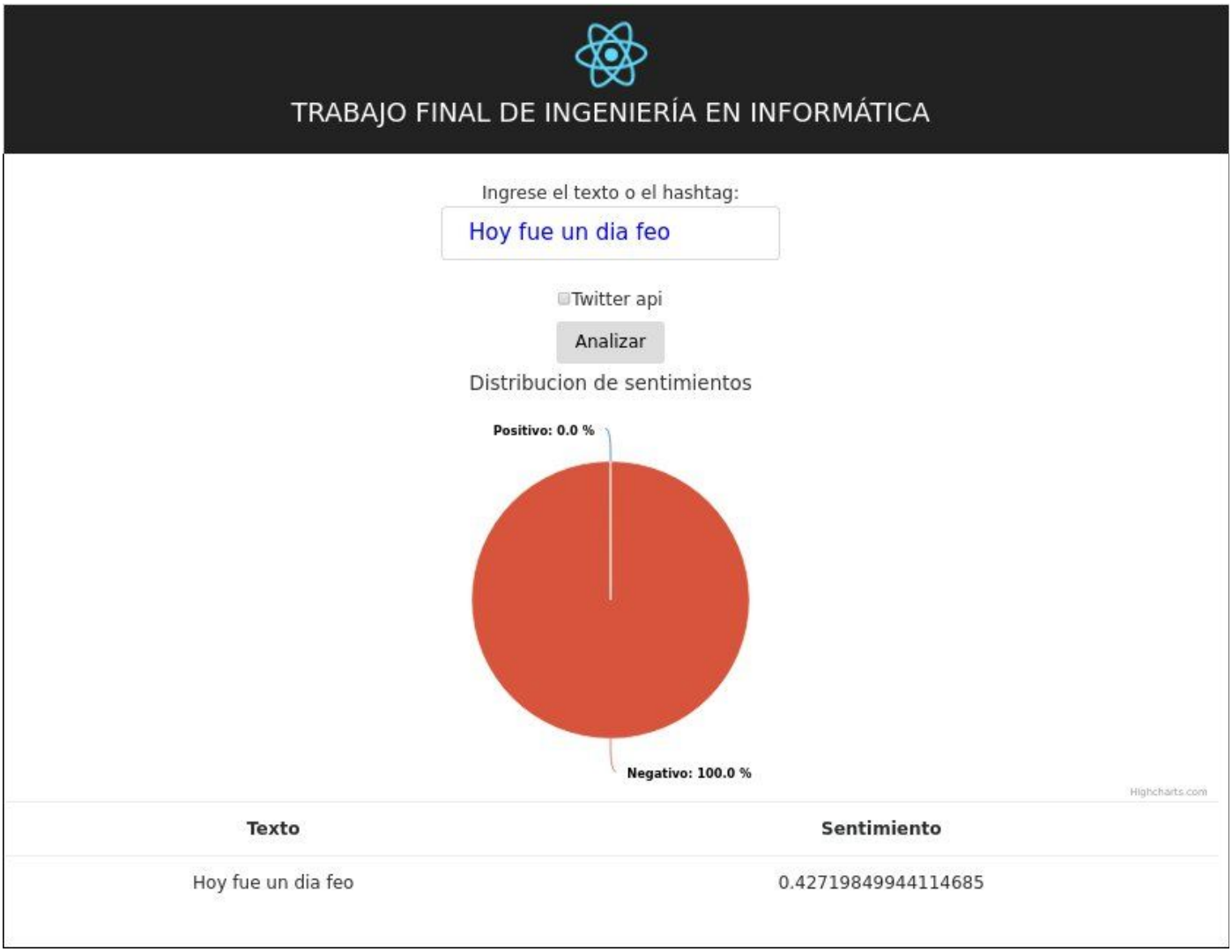


Texto	Sentimiento
La Bota de Oro 16-17 de #Messi, en números Déjanos tu emoji para describir a Leo □ □ 😎 □ 😍 □ □ □ #MessiIsGold https://t.co/4n3mLOjuCU	0.8936099410057068
[SELECCIONADOS] #Messi recibirá este viernes la Bota de Oro correspondiente a la temporada 2016/17 y estos son sus... https://t.co/Vzgu1y1Y34	0.978571355342865
[SELECCIONADOS] □ @ChampionsLeague □□ @PauDybala_JR / #Messi □ 16.45 □ @juventusfces - @FCBarcelona_es https://t.co/Vts6x3QtOH	0.9543027877807617
Temporada 2018-2019... @premierleague brillará con #Messi y @Cristiano en @ManCity y @ManUtd respectivamente ..... https://t.co/ZJzGDhdeCW	0.9800183176994324
Los 5 grandes. LEYENDAS #DiStéfano #Pelé #Cruyff #Maradona #Messi https://t.co/lQHEJlM8tT	0.7025952935218811
Conjuntos Niño $499 #Messi #Barca @FCBarcelona @juventusfces @iVendiendodTodo @ColoniaEscandon	0.9203506708145142

Ejemplo usando texto *positivo*:



Ejemplo usando un texto *negativo*:



Todo el proyecto está subido a github, es público y se puede verificar la evolución del proyecto en el log de commits:

- Red Neuronal: https://github.com/si-m/fproject
- Sentiment Service Module: https://github.com/si-m/fproject_serv
- Web Server: https://github.com/si-m/fproject_node
- FrontEnd: https://github.com/si-m/fproject_front

CONCLUSION

En el trabajo se documentó el avance de las ciencias de la computación a resolver problemas cada vez más complejos con la ayuda de machine learning y la inteligencia artificial. Los resultados en los últimos años han actualizado el estado del arte y marcaron el camino a los avances tecnológicos futuros.

Elegí este tema por el desconocimiento del mismo. Previo a empezar este proyecto, la mayoría de los conceptos me eran ajenos. Plantear el problema no fue difícil, simplemente busque algo en lo que, con técnicas tradicionales, los humanos son buenos haciendo y las computadoras no (análisis de sentimiento de texto). Después, lo extendi a un problema real, como es el análisis de sentimiento para grandes conjuntos de texto (comentarios en una página web de noticias, reseñas de películas, conjuntos de tweets). Finalmente, con el conocimiento adquirido y buscando documentación sobre el tema, desarrolle una solución.

Lo importante de ser ingeniero es la capacidad para resolver problemas usando conocimientos propios, pensamiento crítico y adquiriendo los conocimientos necesarios para llegar a una solución real.

[FIN]

LINKS / REFERENCIAS

---

deepmind:

http://www.davidqiu.com:8888/research/nature14236.pdf

GO is np-hard:

http://webdocs.cs.ualberta.ca/~games/go/seminar/2003/030331/p393-lichtenstein.pdf

Yale LSTM network:

https://yaledatascience.github.io/2016/10/17/learningtolearn-1.html

Tensorflow Serving:

https://www.tensorflow.org/serving/docker

learning rates:

https://towardsdatascience.com/learning-rate-schedules-and-adaptive-learning-rate-methods-for-deep-learning-2c8f433990d1

dropouts in RNN:

http://www.cs.toronto.edu/~rsalakhu/papers/srivastava14a.pdf

word2vec spanish:

http://crscardellino.me/SBWCE/

stanford sentiment analysis with movie reviews:

https://cs224d.stanford.edu/reports/TimmarajuAditya.pdf

tensorflow RNN:

https://www.tensorflow.org/versions/master/tutorials/recurrent

effectiveness of rnn:

http://karpathy.github.io/2015/05/21/rnn-effectiveness/

understanding LSTM:

http://colah.github.io/posts/2015-08-Understanding-LSTMs/

lstm regularization:

https://arxiv.org/pdf/1409.2329.pdf

improving neural networks:

https://machinelearningmastery.com/improve-deep-learning-performance/

lstm for sentiment analysis: http://deeplearning.net/tutorial/lstm.html

reactjs:

https://reactjs.org/

nodejs:

https://nodejs.org/es/

andrea lodi:

https://atienergyworkshop.files.wordpress.com/2015/11/andrealodi.pdf

Juan Jose Salazar:

Programación Matématica (texto)

J.Glenn BookShear:

Computer Science an Overview (texto)