Procesamiento del Lenguaje Natural

Introducción

Actualmente, de forma incremental, está ocurriendo que cuando llamas a un banco o tal vez un proveedor de internet lo primero que escuchas del otro lado de la llamada es una voz que dice: "Hola, soy tu asistente digital. Por favor haz tu pregunta". Hoy en día, los robots pueden hablar como humanos gracias al lenguaje natural, y cada vez lo hacen mejor.

Pero solo hay muy pocas personas que entienden como estos robots funcionan y como poder utilizar estas tecnologías en sus propios proyectos.

Por otro lado, cada vez es mayor la cantidad de información que las personas transmitimos cotidianamente a través de las palabras, como pueden ser: canciones, tweets, mensajes de audios, recetas, denuncias, blogs, etc. y que circula por medios digitales.

Usando **procesamiento de lenguaje natural** podemos analizar de forma masiva esta información.

NLP

Si no podes entender lo que otras personas escriben (o la forma que usen para describir cosas), seguramente no puedas tampoco comprender otros aspectos de sus vidas, incluyendo que es lo que hacen y porqué lo hacen.

EL **procesamiento de lenguaje natural** (NLP por sus siglas en ingles) es una rama de la inteligencia artificial que ayuda a las maquinas a entender y responder al lenguaje humano.

> El *lenguaje* que utilizamos los humanos se conoce como *lenguaje natural*, en contraste a otros lenguajes que se conocen como *lenguajes artificiales (ej: Python)*.

Se encarga de procesar y analizar datos que provienen del lenguaje. Pero esto no es tan sencillo, primero los científicos de datos que trabajan con lenguaje deben aprender *cómo* las personas, de donde salen los datos, usan ese lenguaje. Para esto se tiene en cuenta la semántica y reglas gramaticales, para luego construir aplicaciones que puedan *entender* el lenguaje.

Semántica link.

NLP es el proceso por el cual se intenta crear un sistema computacional para entender, analizar gramaticalmente y extraer lenguaje humano (con frecuencia datos en crudo). Existen varias áreas diferentes dentro del *procesamiento natural del lenguaje*:

- Named Entity Recognition
- Part-of-speech tagging
- Syntactic parsing
- Text categorization
- Co-reference resolution
- Machine translation

Adjacente a NLP existe otro campo de la lingüística computacional llamado **Comprensión del Lenguaje Natural (ULP)**. Aquí es donde se entrenan sistemas computacionales para hacer cosas como:

- Question and answering (bots)
- Summarization
- Sentiment analysis

- Paraphrasing

Casos de uso del NLP

- **Resumen de textos**: la idea es diseñar un algoritmo que encuentre la idea central de un artículo, descartando lo que no es relevante.
- **ChatsBots**: algoritmos capaces de mantener una charla fluida con usuarios y responder a las preguntas automáticamente.
- **Reconocimiento de entidades**: para detectar o encontrar en oraciones personas, entidades comerciales, gobiernos, países, ciudades, etc.
- **Análisis de sentimiento**: la idea es entrenar un algoritmo para comprender si un texto presenta algún sentimiento negativo, positivo o neutro y en que magnitud lo hace.
- **Clasificación automática de textos**: organizar textos en categorías pre-existentes ó detectar los temas recurrentes y crear categorías.

¿Cómo las computadoras entienden el lenguaje?

Las computadoras de forma nativa no puede comprender el lenguaje natural. Para esto es necesario crear un sistema que pueda traducir palabras del lenguaje natural en números.

Para esto hay técnicas que permiten la transformación de un texto en palabras o caracteres, en arreglo de números.

Tokenización

Consideremos un texto conformado por una sentencia: ['me gusta el helado'], lo primero que se viene a la mente del programador es separar por espacios quedando ['me','gusta','el','helado'], o bien separando por caracteres ['m','e','g','u','s','t','a','e','l','h','e','l','a','d','o']. Este proceso se llama *tokenization* donde se generan tokens individuales que serán los datos que sirvan de entrada para un modelo de Machine Learning.

El modelo al recibir la palabra 'me' y seguido 'gusta', va aprendiendo que hay una correlación entre esas dos palabras. Para esto usa celdas de memoria en redes neuronales recurrentes.

Pero así como están los tokens siguen siendo palabras o *strings* que deben convertirse en números, ya que la computadora entiende solo números.

Stop words

En NLP, muchas veces vamos a querer deshacernos de los tokens de menor *carga semántica* para trabajar solamente con aquellos de mayor carga. Nos referimos a los tokens descartados como **stop words**. Para no tener que hacer el trabajo de identificar y filtrar a mano cada una de las stop words, **nltk** nos ofrece un catálogo predefinido de stops words que podemos modificar a gusto según lo necesitemos. Así también **Scikit-Learn** posee un catalogo de palabras incluidas para filtrar. Aunque en ambos casos son en palabras en ingles, por lo que de hacer en *español* requerirá un filtrado más especifico.

También se incluyen en las stops words los emojis que se utilizan en las redes sociales, ya que si se requiere hacer un procesamiento de lenguaje natural basándose en estos datos, seguramente tendrá una carga de muchos símbolos y caracteres especiales.

```
spec_chars = ["!",'"',"#","%","&","'","(",")",
"*","+",",","-",".","/",":",";","<",
"=",">","?","@","[","\\","]","^","_",
"`","{","|","}","~","–","€","!","¡"]
for char in spec_chars:
  data_df = data_df.str.replace(char, ' ')
```

Vectorización

Es el proceso por el cual se asigna un vector por cada palabra. Esto es porque de asignar un número entero por cada palabra, tal vez exista un vocabulario extenso en el texto original, lo cual puede hacer que hayan palabras con un valor de entero muy grande dándole mucho peso a dicha palabra cuando se entrene, lo que generaría un sesgo.

Por este motivo se generan vectores cuyos valores sean entre 0 y 1, con una longitud del espacio vectorial igual a la longitud de la palabra más larga que se encuentre en el texto.

Estos vectores representativos de las palabras o caracteres del texto se puede dar por dos formas:

- One hot encoding.

- Sparse encoding.
- Word Embeddings.
- TF-IDF.

Encodings

Una vez armados los *tokens*, los podemos asociar a números enteros usando **One Hot Encoding**. Esto es cada palabra tendrá un 1 seguida de ceros en un vector de tamaño igual al del vocabulario del corpus o texto. El vocabulario es la cantidad de palabras que contiene el texto.

Otra forma es usando **Sparse Encoding**, es decir, a cada palabra asignándole un valor entero por ocurrencia dentro de cada sentencia. De esta forma reducimos notablemente el tamaño del vector que si usamos *one hot encoding*. Esta codificación la podemos hacer con:

```
from sklearn.feature_extraction.text import CountVectorizer
corpus = ['This is the first document.',
          'This document is the second document.',
          'And this is the third one.',
          'Is this the first document?']
vectorizer = CountVectorizer()
X = vectorizer.fit_transform(corpus)
print(vectorizer.get_feature_names_out())
print(X.toarray())
```

Embedding

Otra forma de asociar un vector de números a una palabra es usando **word embedding**. Esto es muy útil frente a las codificaciones *one hot* y *sparse*, ya que estas últimas generan vectores de muy altas dimensiones y con *word embedding* se puede tener dimensiones mucho menores.

El *word embedding* permite mapear las palabras en vectores de números reales que distribuyen el significado de cada palabra entre las coordenadas del correspondiente vector de palabra. Esta representación de las palabras es en vectores de baja dimensión.

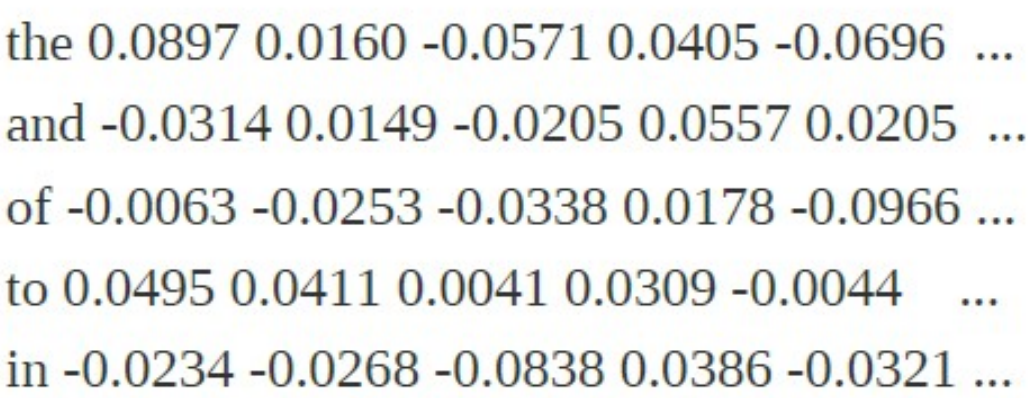


Figura 1: se observa cómo se mapean las palabras "the", "and", "of", "to" y "in" en vectores con distribución normal, extraída de [1].

Una de las limitaciones del *word embeddings* son aquellas palabras con múltiples significados, las cuales entran en conflicto con una sola representación (un solo vector en el espacio semántico). Estos es debido a que la *polisemia* y la *homonimia* no son manejadas adecuadamente.

Hay dos caminos para hacer *word embeddings*: uno es donde las palabras son expresadas como vectores de coocurrencia de palabras, y el otro es expresar palabras como vectores de contextos lingüísticos en donde las palabras ocurren.

Una vez que se tiene la matriz de las palabras mapeadas en vectores numéricos, se puede aplicar técnicas aritméticas con estos vectores. Se puede determinar la similaridad semántica de las palabras, sentencias o del texto entero. También se puede usar esta información para determinar que clase de texto se trata.

Matemáticamente, reducir la similaridad entre dos palabras es calcular el *coseno* de similitud entre los vectores correspondientes, o bien, calcular el coseno del ángulo entre los vectores.

TF-IDF

Esta es otra técnica de vectorización. Su nombre viene de las siglas “Term Frequency - Inverse Document Frequency”, esto nos da una idea del funcionamiento. El significado es : cuantas veces un término aparece en un documento. La importancia de esto es que se supone que mientras más veces aparece, más importante es ese término en ese documento. Por ejemplo, un texto sobre tiburones va a contener muchas más veces la palabra tiburón que un texto que no es sobre tiburones. Mientras mayor frecuencia tenga la palabra en el documento, mayor "puntaje" tendrá. Esto nos sirve, por ejemplo, para detectar sobre qué tópico trata un texto.

$$\text{TF}(term,doc) = \frac{\#\text{ de veces que el } term \text{ aparece en el } doc}{\#\text{ de } terms \text{ diferentes en el } doc}$$

Ejemplo en un documento:

$$\overbrace{\text{Hello}}^{0.125},\ \text{my}\ \overbrace{\text{name}}^{0.125}\ \text{is}\ \overbrace{\text{Brandon}}^{0.375}.\ \text{Brandon Brandon. The}\ \overbrace{\text{elephant}}^{0.125}\ \overbrace{\text{jumps}}^{0.125}\ \text{over the}\ \overbrace{\text{moon}}^{0.125}.$$

Ahora bien, hay términos (como los artículos "la", "el", o las preposiciones "a", "por") que tienen alta frecuencia en todos los textos. Si les asignamos un puntaje alto, no ganaremos información con respecto a cuál es el tópico del documento. Aquí es donde entra en juego la segunda parte del nombre TF-IDF: Inverse Document Frequency. Esto significa que se compara la frecuencia de la palabra en ese documento con la frecuencia en todo el corpus de documentos. A mayor frecuencia de la palabra en todos los documentos, menor puntaje tendrá. Así evitamos puntuar alto a stop words o palabras que no agregan información sobre ese documento específico.

Document Frequency

Fracción de todos los documentos en nuestro corpus que contienen el término.

$$\text{DF}(term,corpus) = \frac{\#\text{ de } docs \text{ que contienen } term}{\#\text{ total de } docs}$$

Ejemplo en un documento:

$$\overbrace{\text{Hello}}^{0.125},\ \text{my}\ \overbrace{\text{name}}^{0.125}\ \text{is}\ \overbrace{\text{Brandon}}^{0.375}.\ \text{Brandon Brandon. The}\ \overbrace{\text{elephant}}^{0.125}\ \overbrace{\text{jumps}}^{0.125}\ \text{over the}\ \overbrace{\text{moon}}^{0.125}.$$

Inverse Document Frequency

Logaritmo inversa de DF.

$$\text{DF}(term,corpus) = \text{Log}\left(\frac{\#\text{ total de } docs}{\#\text{ de } docs \text{ que contienen } term}\right)$$

Ejemplo: si está en todos los docs log(N/N) = log(1) = 0

Entonces nos queda que el cálculo del *tf-idf* de una palabra en particular es:

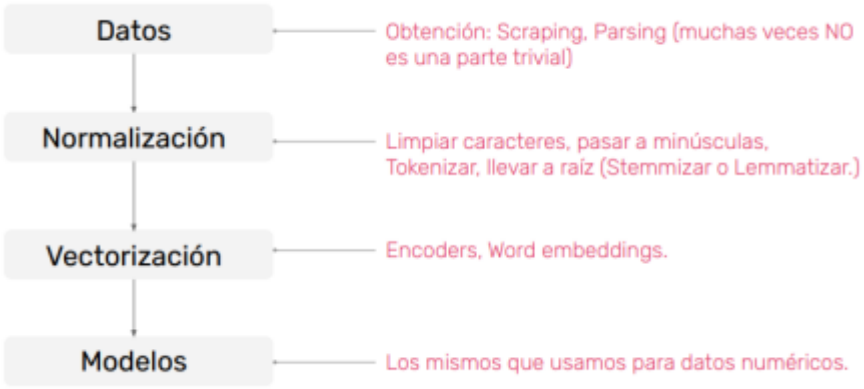
$$TF - IDF = TF * IDF$$

Machine Learning para NLP

Usar *Machine Learning* para construir modelos que comprendan el lenguaje natural no significa que las predicciones que realice este sean declaraciones de hechos. Siempre existe un grado de certeza y de incertidumbre. Mientras mayor exactitud se precise más complicado será el algoritmo pero se perderá eficiencia en la aplicación y en la implementación [1].

¿Si NLP aplicado con ML posee un grado de error en las predicciones, no existe algún otro enfoque para aplicarlo?. La verdad es que no. Podríamos hacer un algoritmo estructurado y muy complejo, y así igual no se podría alcanza la cantidad de palabras y posibilidad de combinaciones que existen.

Los pasos del flujo de trabajo en procesamiento del lenguaje natural es basicamente el mismo para cualquier proyecto con datos.



1. Primero se deben obtener los datos.
2. Estos datos se deberán limpiar, estandarizar y generar los tokens.
3. Los tokens luego serán vectorizados con los métodos anteriores.
4. Para luego ser insertados dentro del modelo de ML.

Más adelante veremos, como se pueden usar distintas capas a las clásicas de una red profunda. En procesamiento de lenguaje natural existe unas redes especiales que se denominan **Recurrent Neural Networks** que se especializan en datos que pueden tener relación con los datos priori y a posteriori.

Modelo estadístico en NLP

Un modelo estadístico en NLP contiene estimaciones de la distribución de probabilidad de las unidas lingüísticas (palabras y sentencias) permitiendo asignarles características lingüísticas.

En estadística y teoría de la probabilidad , la *distribución de probabilidad* de una variable en particular es una tabla de todos los posibles valores que tienen probabilidad de ocurrir en esa variable en un experimento.

Por ejemplo: en un texto en ingles la palabra "count" puede tener las siguientes distribuciones de probabilidad: 78% como verbo y 22% como sustantivo.

Los algoritmos de **Naive Bayes** son algoritmos de clasificación extremadamente rápidos y con pocos parámetros para ajustar, lo cual los hace fácil de implementar en problemas de clasificación en forma de *línea base*, a fin de usarse para comparar algoritmos más complejos cómo son las redes neuronal recurrentes.

Estos algoritmos están basado en el teorema de Bayes donde mediante relaciones de probabilidad conjunta podemos encontrar la probabilidad de una etiqueta dada una característica. Lleva el nombre de *Naive* (sencillo, simple) porque propone una generación de datos de manera sencilla para encontrar dicha probabilidad.

Existen una generación de datos con distribución *Gaussiana* y otra con distribución *Multinominal*. El tipo gaussiano se utiliza en muy pocos casos reales de aplicación ya que los datos de entrenamiento deben tener dicha distribución (muy eventual) y la distribución multinominal es muy común sobre todo en NLP.

Para implementar este clasificador usamos la librería de Scikit-Learn y usamos una vectorización especial que es *term frecuency - inverse document frecuency* (TF-IDF) que a los fines prácticos permite una vectorización con asignación de valores en función de su probabilidad (no solo 1 y ceros):

```
from sklearn.feature_extraction.text import TfidfVectorizer
from sklearn.naive_bayes import MultinomialNB
from sklearn.pipeline import Pipeline
model_base = Pipeline([
                    ("tfidf",TfidfVectorizer()),
                    ("clf",MultinomialNB())
])
model_base.fit(x_train,y_train)
print(model_base.score(x_train,y_train),model_base.score(x_test,y_test))
```

Redes Neuronales Recurrentes

Son muy útiles en automóviles autónomos que trabajan con datos en tiempo real y también con texto, donde una palabra tiene sentido en contexto con las palabras antecesoras.

Una red neuronal recurrente procesa secuencias de datos, iterando a través de los elementos de la secuencia y manteniendo un estado que contiene información relativa a lo que ha visto hasta el momento.

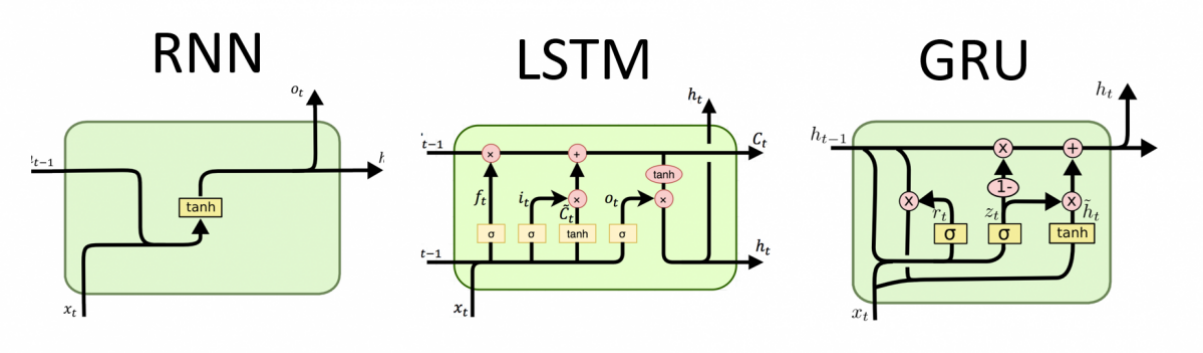
Posee un vector de estado que se restablece entre el procesamiento de dos secuencias independientes diferentes.

En definitiva, una RNN es un **bucle for**, que reutiliza las cantidades calculadas durante la iteración anterior de cada ciclo.

Esta última salida es la que distingue a una RNN de una ANN, ya que el *state vector* permite tener en cuenta las salidas previas para calcular la siguiente salida.

Las celdas de memoria pueden ser : celdas simples **RNN**, **LSTM** o **GRU.** Por lo general, cada capa es una celda de memoria que se reutiliza todas las veces necesarias según la secuencia de datos. A su vez cada celda puede componerse de una o más neuronas.

Este tipo de celdas de memoria usan como función de activación *tanh* en lugar de *relu*. Esto es para justamente contener los posibles valores de activación y evitar que se convierta en *inestable* la arquitectura. Pero *tanh* puede entrar en saturación, lo que hace que el entrenamiento sea muy lento.



Long Short Term Memory (LSTM)

Las principales capas recurrentes que se utilizaron para datos en serie de tiempo o procesamiento de lenguaje natural eran las simples **RNN**. Pero en la práctica no se utilizan mucho ya que matemáticamente son muy simples y no poseen la capacidad

de aprender patrones a largo plazo, sobre todo por el problema de desvanecimiento del gradiente por la función *tanh*.

Para eso están las celdas del tipo **GRU** y **LSTM**. Las mismas son variantes de la **RNN** a las que se les ha añadido una manera de contener información a lo largo de muchas épocas.

Las LSTM son un tipo en particular de RNN que poseen algunas mejoras de las simples **RNN:**

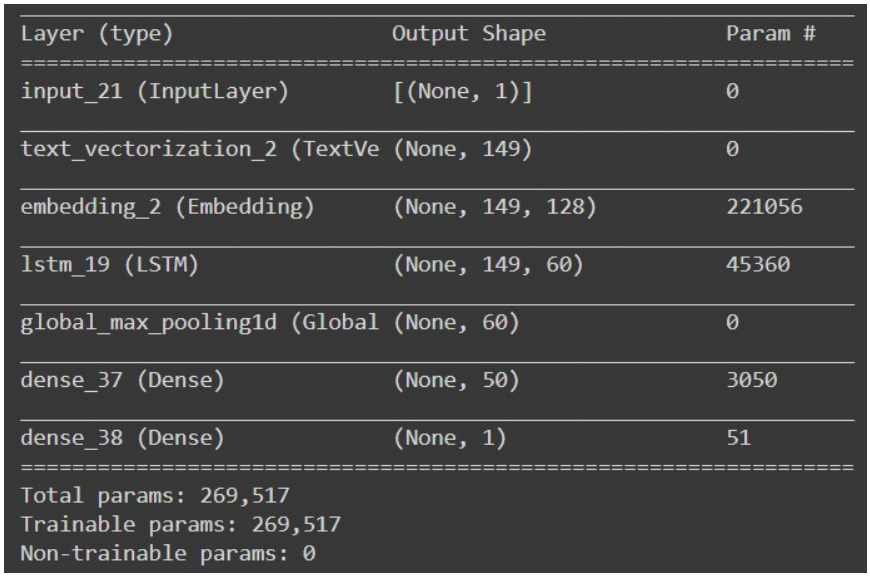
- Funciones de activación que previenen *Vanish Gradient.*
- Funciones de activación que hacen más estable el entrenamiento.

Este tipo de celdas posee un vector de estado (*state vector*) que se propaga una vez por paso al siguiente paso. Este es el término *short-term state*.

Y también posee un segundo vector de estado que se puede pensar como un vector de estado a largo plazo (*long-term state*).

```
# Modelo LSTM
#Construimos el modelo con Sequential API
inputs = tf.keras.layers.Input(shape=(1,),dtype=tf.string)
x = text_vectorized(inputs)
x = embedding(x)
x = tf.keras.layers.LSTM(128, return_sequences=True)(x)
x = tf.keras.layers.LSTM(60)(x)
x = tf.keras.layers.GlobalMaxPool1D()(x)
x = tf.keras.layers.Dense(50, activation='relu')(x)
outputs = tf.keras.layers.Dense(1,activation='sigmoid')(x)

model_lstm = tf.keras.Model(inputs,outputs)
model_lstm.compile(optimizer=tf.keras.optimizers.Adam(),
              loss=tf.keras.losses.BinaryCrossentropy(),
              metrics=["accuracy"])
model_lstm.summary()
model_lstm.fit(train_dataset,
          epochs=100,
          batch_size=128,
          validation_data=test_dataset,
          callbacks=[tf.keras.callbacks.ModelCheckpoint('Models_review/lstm_model',
                     save_best_only=True),tf.keras.callbacks.EarlyStopping(
                     patience=10,restore_best_weights=True)])
```



```
Layer (type)                 Output Shape              Param #
=================================================================
input_21 (InputLayer)        [(None, 1)]               0
_________________________________________________________________
text_vectorization_2 (TextVe (None, 149)               0
_________________________________________________________________
embedding_2 (Embedding)      (None, 149, 128)          221056
_________________________________________________________________
lstm_19 (LSTM)               (None, 149, 60)           45360
_________________________________________________________________
global_max_pooling1d (Global (None, 60)                0
_________________________________________________________________
dense_37 (Dense)             (None, 50)                3050
_________________________________________________________________
dense_38 (Dense)             (None, 1)                 51
=================================================================
Total params: 269,517
Trainable params: 269,517
Non-trainable params: 0
```

Redes convolucionales para NLP

Una red neuronal convolucional son aquellas que incluyen capas *convolucionales*, las cuales se comparten con etiquetadores de discursos, analizador de dependencias y con reconocimiento de entidades con nombre.

Las capas convolucionales se encargarán de aplicar un conjunto de filtros para detectar las regiones de los datos de entradas donde presentan características determinadas.

Ejemplo: suponga la sentencia : ['Can we count on them']. Podemos considerar la sentencia como una matriz donde cada fila es una palabra en forma de vector. Entonces si cada vector tiene 300 dimensiones y la sentencia tiene 5 palabras entonces la matriz resulta de 5 x 300.

Tal vez en la capa convolucional el filtro tenga un tamaño de 3 (aplicado a tres palabras consecutivas) por lo que resultará de 3 x 300 aplicado al ejemplo.

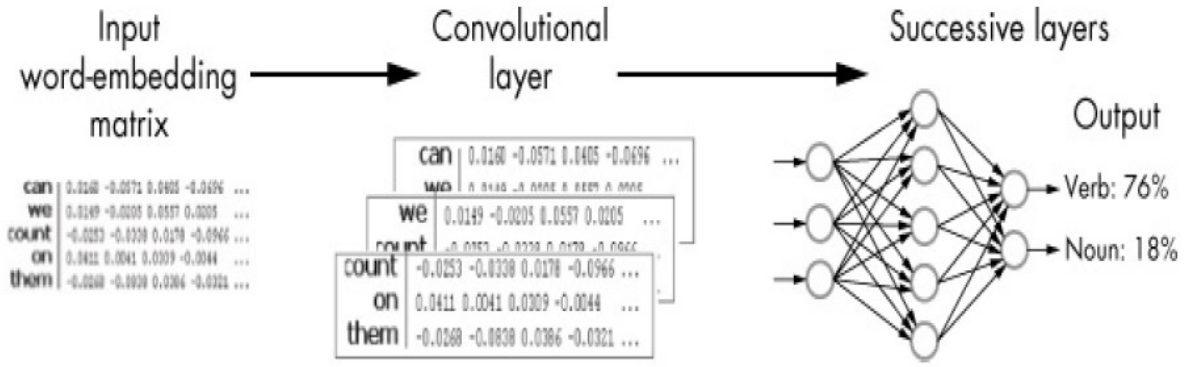


Figura 3: se puede observar el flujo de procesamiento de la sentencia, vectorizada y filtradas para luego entrenar la red neuronal profunda.

Estas redes neuronales convoluciones son conocidas en 2D aplicadas a imágenes para extraer patrones que sirvan para un fin en particular.

De la misma forma para series de tiempo (texto, forecasting), se utilizan para extraer patrones en una dimensión espacial con *Conv1D*. Donde la dimensión espacial es la dimensión *temporal.*

```
model = tf.keras.Sequential([
    tf.keras.layers.Input(shape=(1,),dtype=tf.string),
    text_vectorized,
    embedding,
    tf.keras.layers.Conv1D(64,5,padding='same',activation='relu'),
    tf.keras.layers.GlobalAveragePooling1D(),
    tf.keras.layers.Dense(64,activation='relu'),
    tf.keras.layers.Dense(1,activation='sigmoid')
])
model.compile(optimizer=tf.keras.optimizers.Adam(),
              loss=tf.keras.losses.BinaryCrossentropy(),
              metrics=["accuracy"])
model.summary()
model.fit(train_dataset,
          epochs=10,
          batch_size=128,
          validation_data=test_dataset,
          callbacks=[tf.keras.callbacks.ModelCheckpoint('Models_review/cnn_model',
                     save_best_only=True)])
```

```
Model: "sequential_1"
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
text_vectorization_2 (TextVe (None, 149)               0
_________________________________________________________________
embedding_2 (Embedding)      (None, 149, 128)          221056
_________________________________________________________________
conv1d_1 (Conv1D)            (None, 149, 64)           41024
_________________________________________________________________
global_average_pooling1d_1 ( (None, 64)                0
_________________________________________________________________
dense_39 (Dense)             (None, 64)                4160
_________________________________________________________________
dense_40 (Dense)             (None, 1)                 65
=================================================================
Total params: 266,305
Trainable params: 266,305
Non-trainable params: 0
```

Las capas convolucionales no tienen memoria como un *RNN.* Cada salida es calculada en base a una *ventana* pequeña de la entrada, que va a correlacionarse con el tamaño del filtro.

Pero si *apilamos* muchas capas de *Conv1D*, esta arquitectura puede realmente capturar patrones largos de secuencia de datos.

Kernel: generalmente es (3,) porque es más eficiente usar pequeño valores de kernel en muchas capas apiladas, en lugar de usar una sola capa convolucional con un kernel muy grande. Además el numero de parámetros a calcular y el tiempo de procesamiento es menor con muchas capas convolucionales con kernel pequeños, que usar una sola capa con kernel muy grande.

Padding: por default no se usa este relleno en absoluto. Esto significa que el largo de la secuencia de salida *filtrada* será menor que la original. Si usamos el argumento *same*, se completa la secuencia con la cantidad necesaria de ceros tanto a izquierda como derecha de la misma, para que al filtrar, no ocurra pérdida de la información, es decir quedará de la misma longitud de entrada. También podemos usar otro argumento que es *casual*, que solo rellenará de ceros a la izquierda de la secuencia, de manera de conservar los primeros valores originales (es decir conserva el pasado) por lo que la salida en un momento dependerá solo de los valores anteriores y no de los posteriores. Esto se usa mucho en series de tiempo para **Forecasting**.

Stride: por default es siempre 1. Lo que significa que el filtro pasará de a una vez por dato de la secuencia. Pero se puede establecer un valor distinto (2,3,4,…) según se precise.

Enlaces complementarios

- NLP: https://colah.github.io/posts/2014-07-NLP-RNNs-Representations/
- RNN: https://karpathy.github.io/2015/05/21/rnn-effectiveness/
- LSTM: https://colah.github.io/posts/2015-08-Understanding-LSTMs/

Referencias

1. Vasiliev, Y. (2020). *Natural Language Processing with Python and SpaCy: A Practical Introduction*. No Starch Press.
2. Géron, A. (2019). *Hands-on machine learning with Scikit-Learn, Keras, and TensorFlow: Concepts, tools, and techniques to build intelligent systems*. O'Reilly Media.
3. VanderPlas, J. (2016). *Python data science handbook: Essential tools for working with data*. " O'Reilly Media, Inc."
4. Chollet, F. (2021). *Deep learning with Python*. Simon and Schuster.
5. Certificación Universitaria en Data Science. Mundos E - Universidad Nacional de Cordoba.

Made with ❤️ Ignacio Bosch