

FACULTAD DE MATEMÁTICA, ASTRONOMÍA,
FÍSICA Y COMPUTACIÓN

UNIVERSIDAD NACIONAL DE CÓRDOBA



Aprendizaje Multimodal aplicado al Etiquetado de Imágenes

TESIS PARA OBTENER EL TÍTULO DE

LICENCIADO EN CIENCIAS DE LA COMPUTACIÓN

AUTOR: NICOLÁS JESÚS PERETTI

DIRECTORES: FRANCO LUQUE Y JORGE SÁNCHEZ

CÓRDOBA, ARGENTINA

19 DE DICIEMBRE DE 2019



Esta obra está bajo una licencia Creative Commons “Reconocimiento-CompartirIgual 4.0 Internacional”.

Resumen

El aprendizaje multimodal estudia problemas de aprendizaje automático utilizando datos que combinan información de diferente naturaleza. Un ejemplo de tarea multimodal es el etiquetado de imágenes, donde una imagen debe ser etiquetada con términos (palabras) que describan el contenido de la imagen. En este trabajo proponemos estudiar modelos que permiten etiquetar imágenes a través de funciones que den una ordenación (ránking) de etiquetas posibles a cada imagen dada. Este ránking se obtiene a partir de una puntuación (score) que se obtiene de una función bilineal que combina representaciones de imágenes con representaciones de etiquetas textuales.

Índice general

1. Introducción	1
2. Marco Teórico	3
2.1. Aprendizaje automático	3
2.1.1. Aprendizaje supervisado	3
2.1.2. Aprendizaje no supervisado	4
2.1.3. Ejemplo: Predicción del valor de la vivienda	4
2.1.4. Aprendizaje por descenso por el gradiente	5
2.2. Redes Neuronales	6
2.2.1. Perceptrón	8
2.2.2. Redes Feedforward	9
2.2.3. Activation	9
2.3. Redes Neuronales Convolucionales	10
2.3.1. Convolución	12
2.3.2. Pooling	14
2.3.3. VGG	14
2.4. Word embeddings	16
2.4.1. word2vec	17
2.4.2. Bert	19
2.5. Learning to Rank	24
3. Image Tagging	29
3.1. El modelo	29
3.2. Dataset	30

3.3. Funciones de pérdida	30
3.4. Métricas	32
3.5. Generación de tags ordenados	32
3.6. Resultados	36
4. Experimentos	41
4.1. Experimento 1	41
4.2. Experimento 2	46
4.3. Experimento 3	49
4.4. Experimento 4	50
4.5. Experimento 5	51
5. Conclusiones	55
5.1. Conclusiones	55
 Bibliografía	 57

Capítulo 1

Introducción

Con la avenida de la masividad de las cámaras digitales en un principio y ahora con los dispositivos móviles sumando a las redes sociales las personas capturan y guardan imágenes a un ritmo cada vez más creciente, pero las mismas no están predispuestas a anotarlas con relevantes descripciones ya que es costoso por diferentes razones como lo puede ser la subjetividad del problema y por otra perspectiva aún más simple como lo es el tiempo. Ahora bien ¿Para qué necesitamos tener estas imágenes anotadas?, ¿Qué impacto generaría poder tener un algoritmo que lo realice de manera automática?, bueno la clave para entender la importancia es fijarse en los casos de usos en los cuales se podría automatizar los procesos, como por ejemplo uno de ellos es el etiquetado de imágenes para entrenar algún algoritmo de manera supervisada (clasificadores, detectores), ya veremos más adelante lo que esto significa, otro caso podría ser la búsqueda de imágenes en una base de datos a través de palabras claves. El impacto de contar con este tipo de algoritmos sería una reducción del tiempo humano que se dedican a estas tareas.

En este trabajo se utilizará un modelo *multimodal* que utiliza representaciones vectoriales tanto de imágenes como de palabras, para las primeras utilizaremos redes neuronales profundas pre-entrenadas de uso generalizado en visión por computadora, como por ejemplo

VGG19 (Simonyan and Zisserman, 2014), para la representación vectorial de las etiquetas, utilizaremos word embeddings pre-entrenados como *word2vec* (Mikolov et al., 2013a), *Bert* (Devlin et al., 2018).

En los experimentos, utilizaremos el conjunto de datos COCO Captions (Chen et al., 2015). En este corpus, las imágenes se encuentran asociadas a las denominadas *captions*, que son descripciones cortas en lenguaje natural realizadas por voluntarios. Para obtener etiquetas a partir de estas captions, procesaremos las oraciones utilizando técnicas de Procesamiento de Lenguaje Natural como lematización, etiquetado morfosintáctico (PoS tagging) y análisis sintáctico (parsing). El foco central de los experimentos se basará en el estudio de heurísticas para la detección de sinónimos e hiperónimos que permitan mejorar el conjunto de etiquetas y en particular eliminar etiquetas redundantes. Para ello utilizaremos recursos lingüísticos clásicos como WordNet (Fellbaum, 1998).

Para aprender la función bilineal utilizaremos una función de costo estructurada orientada a ranking (Sanchez et al., 2018), presentada originalmente en un trabajo colaborativo realizado por los directores.

Este trabajo está organizado de la siguiente manera: en la sección 2 se introducirán los conceptos teóricos necesarios para comprender todo lo que se utilizará. En la sección 3 se abordará el enfoque utilizado en el paper (Sanchez et al., 2018) para atacar la problemática del etiquetado de imágenes. Luego en la sección 4 se introducirá en los experimentos realizados, mostrando los resultados obtenidos. Por último se dará una conclusión y planteamientos sobre trabajos futuros.

Capítulo 2

Marco Teórico

2.1. Aprendizaje automático

El aprendizaje automático es un subcampo de la ciencias de la computación y una rama de la inteligencia artificial cuyo objetivo es resolver una tarea basándose en alguna experiencia, algunos ejemplos clarificadores de tareas a resolver pueden ser clasificar si un correo electrónico es spam o no, traducción automática de textos, reconocimiento de voz o reconocimiento facial. Cuando nos referimos a la experiencia estamos hablando generalmente del conjunto de datos que usaremos para poder entrenar los diferentes algoritmos para poder resolver dichas tareas. Los algoritmos que se utilizan en el aprendizaje automático pueden ser categorizados en el mayor de los casos como supervisados o no supervisados, la diferencia de ambos radica en el conjunto de datos con los cuales son entrenados, a continuación vamos a explayarnos y tratar de clarificar la diferencias de ambos.

2.1.1. Aprendizaje supervisado

En el aprendizaje automático supervisado al momento del entrenamiento contamos con un conjunto de datos en los cuales para cada ejemplo del mismo sabemos la predicción correcta, *tag* o *label*, dejan-

do explícitamente la relación entre la entrada y salida del algoritmo. Generalmente las tareas que podemos resolver con este subconjunto de algoritmos pueden ser categorizados en problemas de regresión y clasificación, para los primeros se predicen valores continuos, lo cual significa que estamos tratando de relacionar las variables de entrada en una función continua, para los problemas de clasificación se predicen valores discretos, lo cual significa que tratamos de relacionar las variables de entrada en categorías bien definidas. Ejemplos bien concretos de ambos enfoques pueden ser predecir la edad de una persona dada una imagen de la misma, para una regresión, la edad es una variable continua y para una clasificación predecir si un correo electrónico es spam o no.

2.1.2. Aprendizaje no supervisado

El aprendizaje automático no supervisado difiere del supervisado en que no se dispone de la salida correcta a cada ejemplo de el conjunto de datos. Este tipo de algoritmos nos permiten aproximar problemas sin contar a priori con información de cómo están compuestos nuestros datos, un tipo claro de estos algoritmos es el agrupamiento, mayormente conocido como *clustering* en inglés, un ejemplo de este tipo de problemas o tarea podría ser la segmentación por algún comportamiento en común de usuarios en la red social Twitter.

2.1.3. Ejemplo: Predicción del valor de la vivienda

Vamos abordar diferentes conceptos del aprendizaje automático tales como el modelo, la función de pérdida y el algoritmo de aprendizaje a través de un ejemplo concreto simplificado, este será la predicción del valor de una vivienda. Para empezar vamos a describir el conjunto de entrenamiento, sea $C = \{(x_i, y_i)\}_{i=0}^N$ el dataset donde N son la cantidad de datos que tenemos disponibles y para cada

uno de estos tenemos x_i que va a ser nuestra única variable de entrada que representa la cantidad de metros cuadrados de la vivienda e y_i que representa el valor monetario de la misma. Con la descripción del problema antes mencionado podemos decir que se tratará de un algoritmo de aprendizaje automático supervisado, más aún es un problema de regresión ya que necesitamos predecir el valor de la vivienda y esto es continuo. Ya sabemos como son los datos ahora falta definir el modelo, o sea la función que vamos a proceder a entrenar, queremos tener una $f^* : \mathbb{R} \mapsto \mathbb{R}$ (función de reales en reales) tal que tome como entrada el tamaño de la vivienda y retorne el costo de la misma. La forma más fácil de representarla es con una función lineal como puede ser $f(x; \theta) = x * \theta_1 + \theta_0$ donde θ son los parámetros del modelo. Ya tenemos bien definidos como son los datos y el modelo, ahora bien una pregunta que surge naturalmente sería ¿Cómo sabemos cuán bien está prediciendo el modelo? ¿Cómo podemos decidir qué valores de θ son mejores que otros?, para responder estas preguntas vamos a introducir la función de pérdida o costo, vale aclarar que para cada problemática vamos a utilizar alguna función de costo acorde a la misma, para la situación de nuestro ejemplo utilizaremos la función de error cuadrático medio la cual está definida de la siguiente manera $J(\theta_0, \theta_1) = \frac{1}{2N} * \sum_{i=0}^N (f(x_i; \theta_0, \theta_1) - y_i)^2$ la cual nos dice cual es error del modelo dado ciertos parámetros fijos, nos resta definir cómo elegirlos de manera óptima para ello tendremos que minimizar $\operatorname{argmin}_{\{\theta_0, \theta_1\}} J(\theta_0, \theta_1)$ ya que obtendremos los parámetros con el menor error posible. A continuación vamos a explayarnos en un algoritmo iterativo que se puede utilizar para este fin como lo es el descenso por el gradiente.

2.1.4. Aprendizaje por descenso por el gradiente

Sea $J : \mathbb{R}^d \mapsto \mathbb{R}$ una función de costo y $\theta \in \mathbb{R}^d$, tenemos como objetivo obtener los argumentos que minimicen el valor de salida de la función J , es decir, $\operatorname{argmin}_{\theta} J(\theta)$. Para utilizar este algoritmo vamos a tener que hacer uso del gradiente $\nabla J(\theta)$ que es el vector de derivadas

parciales, donde $\frac{\zeta J(\theta)}{\zeta \theta_i}$ es la derivada parcial de J con respecto a θ_i .

El descenso por el gradiente es un algoritmo iterativo en el cual θ toma valores iniciales los cuales pueden ser aleatorios o fijos, y en cada iteración se actualizan simultáneamente los parámetros de la siguiente manera: $\theta_j := \theta_j - \alpha * \frac{\zeta J(\theta)}{\zeta \theta_i}$ donde $\alpha > 0$ es el coeficiente o tasa de aprendizaje, conocido como *learning rate*, el cual indica el tamaño del paso que damos en cada iteración del algoritmo, notar que con un α demasiado bajo se tardará mucho en converger a un mínimo, y con un learning rate demasiado alto el algoritmo puede llegar a diverger. En cada actualización de parámetros estamos dando un paso en dirección negativa del gradiente en determinado punto.

Ejemplo: supongamos que queremos minimizar $f(w) = w^2$, calculamos la derivada $f'(w) = 2w$, determinemos un $\alpha = 0,1$ y un valor inicial aleatorio de $w = -8$, en la primera iteración del algoritmo vamos a tener un valor de $w = w - f'(w) = -6,4$ siguiendo la iteración del algoritmo vamos a llegar a un valor mínimo, en este ejemplo ese valor sería $w = 0$. En la Figura 2.1 podemos ver cómo se va actualizando el valor de w hasta converger.

2.2. Redes Neuronales

Las redes neuronales son una composición de funciones no lineales donde el objetivo principal es tratar de estimar de nuevo los parámetros W de $f(X; W, b)$ donde esta f está definida como $f_n \circ \dots \circ f_0(X)$, la particularidad de que sean funciones no lineales es poder representar estructuras más complejas, de hecho en el supuesto de que todas las f_i sean lineales la composición mencionada anteriormente sería otra función lineal, por lo tanto no tendría mucho sentido componerlas. Cada f_i se denomina como capa de la red neuronal, y está compuesta por perceptrones, la unidad básica de procesamiento.

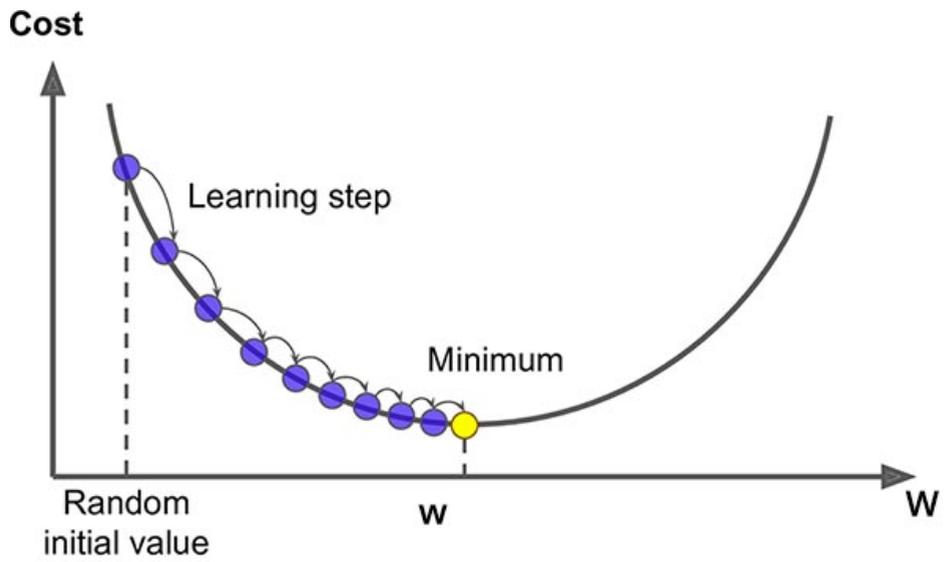


Figura 2.1: Descenso por el gradiente de la función $f(w) = w^2$

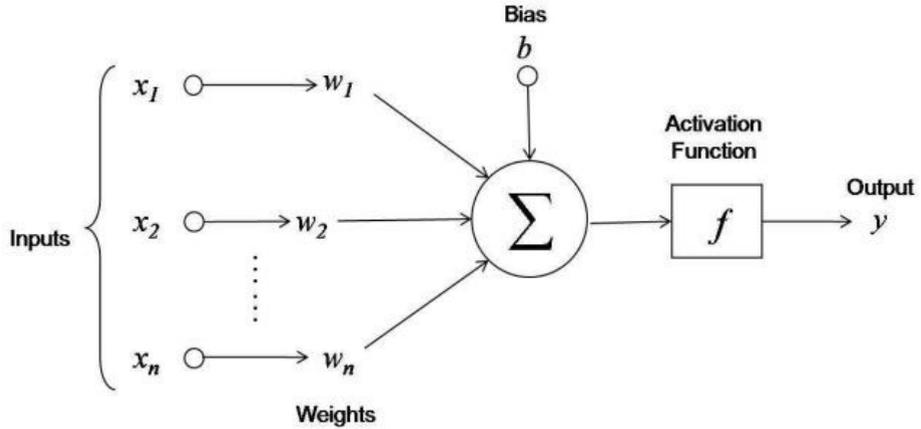


Figura 2.2: Perceptrón

2.2.1. Perceptrón

Como mencionamos/definimos anteriormente un perceptrón también conocido como neurona es la unidad básica de procesamiento que consta de múltiples entradas y una única salida, las operaciones están dadas por las siguientes ecuaciones:

$$z = \frac{1}{n} * \sum_{i=1}^n x_i * w_i + b$$

$$y = f(z)$$

Donde x_1, \dots, x_n son los valores de entrada, w_1, \dots, w_n son los parámetros o pesos del modelo, y f es la función de activación, en la siguiente sección veremos algunas posibles funciones. En la Figura 2.2 vemos una representación de una neurona.

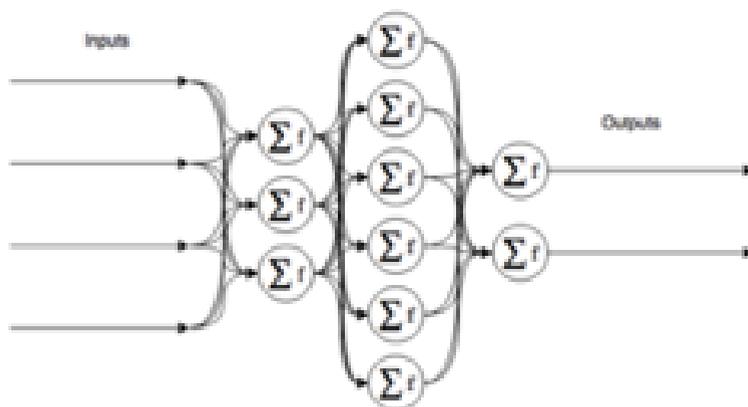


Figura 2.3: Redes FeedForward

2.2.2. Redes Feedforward

Estas redes neuronales se construyen uniendo y componiendo perceptrones en capas fully connected (completamente conectadas), un ejemplo de una red feedforward la podemos encontrar en la Figura 2.3, intuitivamente podemos decir que cada capa realiza una transformación no lineal del espacio vectorial de entrada en otro de salida.

El término de feedforward viene de que solo miran hacia adelante, la información fluye desde la entrada hacia la salida sin mirar hacia atrás, podemos representar el modelo como un grafo acíclico. Para poder entrenar estas redes normalmente se utiliza el algoritmo de backpropagation (Kelley, 1960).

2.2.3. Activation

Como vimos anteriormente en la Figura 2.2 donde se explica el comportamiento del perceptrón se utiliza una función de activación, donde no la definimos. Dependiendo la problemática que deseamos atacar se utilizan diferentes funciones de activación, en esta sección

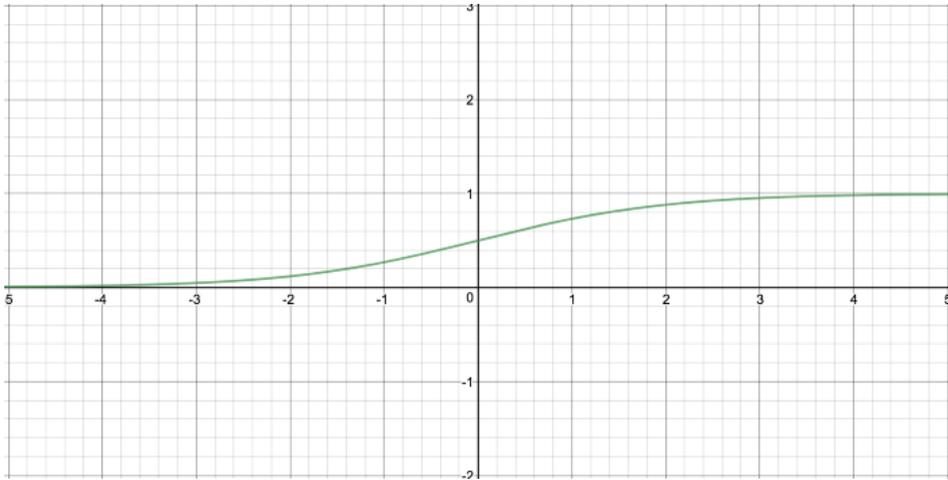


Figura 2.4: Sigmoide

vamos a detallar las más conocidas.

Sigmoide:

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

Es una función continua no lineal donde toma cualquier rango de los números reales y el valor de salida está comprendido entre 0 y 1. En la Figura 2.4 podemos ver la gráfica de la misma.

ReLU:

$$ReLU(x) = \max(0, x)$$

ReLU, *rectified linear unit*, por sus siglas en inglés, es una función muy simple que dado un valor, si es negativo devuelve 0, caso contrario el mismo valor. En la Figura 2.5 se detalla la gráfica.

2.3. Redes Neuronales Convolucionales

Las Redes Neuronales Convolucionales (*CNNs*, del inglés, Convolutional Neural Networks) son una expresión de las Redes Neuronales

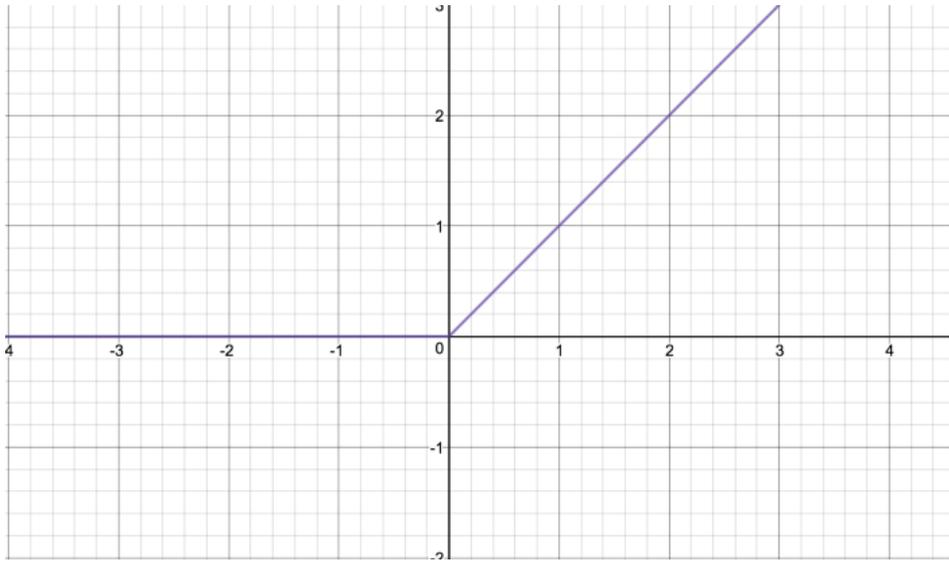


Figura 2.5: ReLU

especializadas en procesar datos de naturaleza de grilla como lo son los datos de series temporales, pueden verse como una grilla $1-D$, y las imágenes que se puede pensar como una grilla $2-D$. El nombre se debe a que se utiliza una operación matemática llamada convolución. Se llaman *CNNs* a una *NN* que utiliza una convolución en vez de una multiplicación matricial en al menos una capa. En esta sección vamos a explicar la convolución y sus ventajas, el pooling y por último vamos ver una arquitectura de CNN como lo es la *VGG* que es la que se utilizará a lo largo del trabajo.

2.3.1. Convolución

La convolución se denota como $s(t) = (x * w)(t)$, donde x y w son funciones, en el dominio de las imágenes puede pensarse que x se refiere al input, w al *kernel*, y la resultante sería el *feature map*, en el aprendizaje automático el input puede pensarse como un *tensor* de datos y el kernel como uno de parámetros, la convolución en su forma discreta se define como $s(t) = (x * w)(t) = \sum_{a=-\infty}^{a=\infty} x(a)w(t - a)$. En la Figura 2.6 podemos ver un ejemplo de una convolución $2-D$ cuya particularización de la definición es $s(i, j) = (I * K)(i, j) = \sum_n^m I(i + m, j + n)K(m, n)$. Notar que el kernel recorre el input de a saltos de 1 fila/columna, $stride = 1$, de la grilla y que la dimensionalidad del output es menor que la del input, estas condiciones se pueden modificar adrede de acuerdo al caso de uso que se necesite, por ejemplo podemos mantener la dimensionalidad haciendo uso del *padding* que consta de agregar una fila y columna al input, existen diferentes tipos de padding que no se verán en el presente trabajo.

Utilizar convoluciones en vez de capas densas nos da ventajas como lo son las conexiones esparsas y la noción de pesos compartidos, esto se debe a que en las capas dense por cada output tenemos interacciones con cada input, en cambio con la convolución no, otra ventaja es que es menos costoso en memoria ya que posee menos parámetros y también en cálculo ya que se requieren menos operaciones.

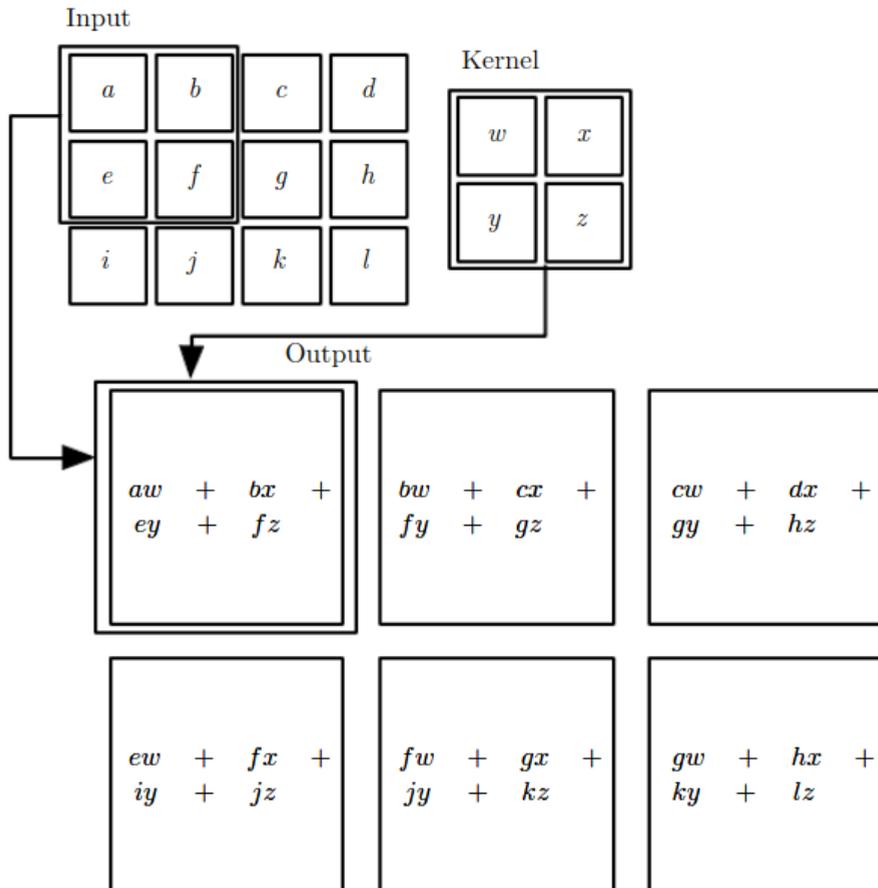


Figura 2.6: Convolución

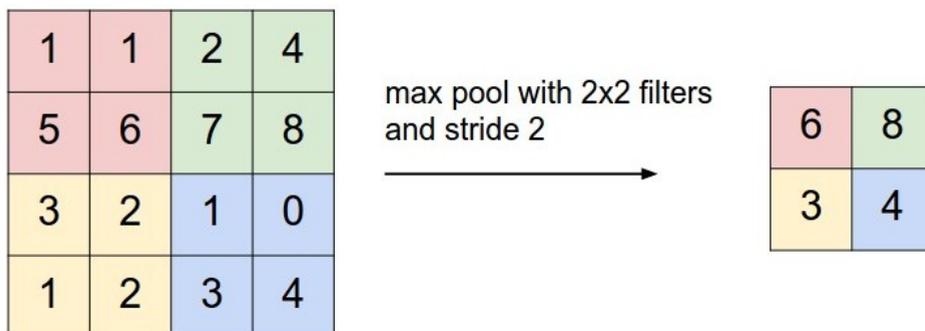


Figura 2.7: Pooling

2.3.2. Pooling

La capa de agrupación, *pooling* en inglés, consiste en reducir la dimensionalidad del output de la capa anterior como lo puede ser en este contexto la convolución, existen diferentes tipos de pooling el más conocido es el max pooling, que consta de quedarse con el valor máximo, podemos ver un ejemplo en la Figura 2.7. La intuición por detrás de este tipo de capas más allá del ahorro en cantidad de parámetros que se necesitarán aprender en las capas venideras es hacer invariante la arquitectura de CNNs a pequeñas traslaciones. Notar que no se aprenden parámetros en el pooling.

2.3.3. VGG

VGG (Simonyan and Zisserman, 2014) es una familia de arquitecturas CNNs entrenadas para ImageNet (Deng et al., 2009) con el objetivo de poder clasificar una imagen entre las 1000 categorías del *dataset*. En la Figura 2.8 podemos ver la arquitectura para una de ellas como lo es la *VGG19*, podemos notar que está compuesta por bloques convolucionales, bloques densos y por último una softmax que nos da la distribución de probabilidad sobre las 1000 categorías, en general

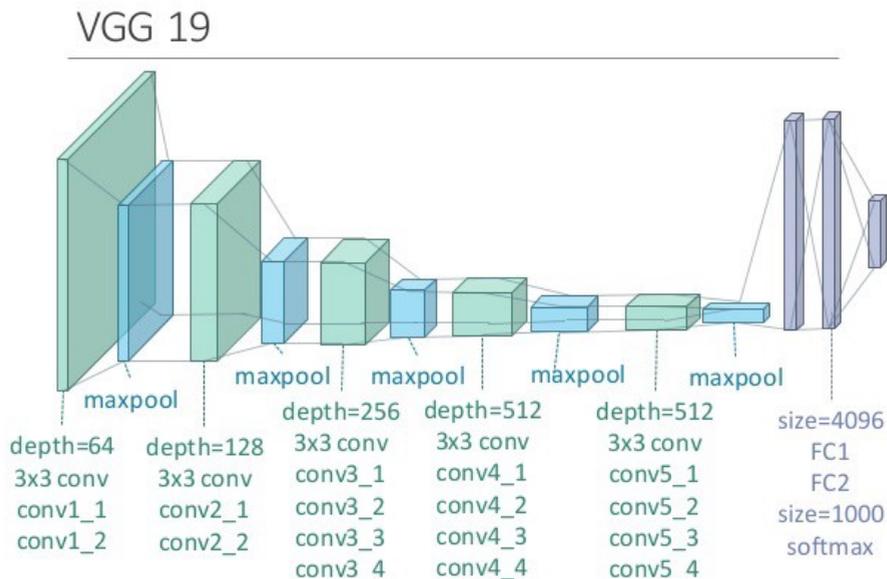


Figura 2.8: VGG19

cada bloque de una *CNN* está compuesta por una capa de convolución seguida de una función no lineal y por último una capa de pooling, para el caso particular de la *VGG19* cada bloque está compuesto por varias capas de Convoluciones de $3 \times 3 + ReLU$ seguido de un *Max pooling*, y al final de los bloques de convoluciones podemos ver que tenemos dos capas densas, en la Figura 2.8 se denota como *FC1* y *FC2*.

En este trabajo se utilizará esta arquitectura para la extracción de features visuales, ¿Ahora bien cómo podemos realizar esto?, hay una noción en imágenes de que una *CNN* entrenada en *imageNet* generaliza bastante a otros dominio que no sea el de la competencia misma

ya que se supone que *ImageNet* son imágenes bastantes generales que han sido insertadas en la taxonomía de WordNet (Fellbaum, 1998), ahora nuestro objetivo es extraer un vector que represente a una imagen, para hacer esto vamos a recortar/tirar la capa de clasificación de la *VGG19* por lo tanto ahora cada vez que le demos como input una imagen vamos a obtener un vector de dimensionalidad 4096.

2.4. Word embeddings

Una de las tareas más importantes dentro de PLN, procesamiento del lenguaje natural, es como representamos las palabras, teniendo como objetivo que estas sean el input, entrada, para cualquier modelo. Una característica fundamental en cómo representamos estas palabras es la noción de similitud entre las mismas por ejemplo la palabra “perro” tiene que tener una mayor similitud a “animal” que a la palabra “persona”, para poder atacar esta problemática vamos a hacer uso de vectores, por lo tanto cada palabra va a tener un vector en un espacio N dimensional asociado que la representa, ahora bien la siguiente pregunta resultante fuera ¿Cómo generamos estos vectores?, el método más simple que podemos utilizar con ese objetivo es **one-hot vector** donde cada palabra $w \in \mathbb{R}^{|V|}$ siendo V el vocabulario ordenado, w consiste en todos ceros y un único uno en el índice que representa la palabra en V , a continuación daremos un ejemplo para clarificar cómo funciona este método: supongamos que tenemos la sentencia *el perro ladra* por lo tanto $V = [el, ladra, perro]$, $|V| = 3$, por lo tanto los vectores de palabras serían los siguientes: “*el*” = $[1, 0, 0]$, “*ladra*” = $[0, 1, 0]$, “*perro*” = $[0, 0, 1]$. Ahora con esta representación vectorial al ser cada palabra una entidad completamente independiente no captura una propiedad que si nos importaba como lo es la similitud entre las palabras. A continuación vamos a ver cómo podemos capturar esta noción mediante dos algoritmos como lo son **word2vec** (Mikolov et al., 2013a) y **Bert** (Devlin et al., 2018).

2.4.1. word2vec

La idea detrás de este algoritmo es diseñar un modelo cuyo parámetros sean los word embeddings, vectores de palabras, luego entrenarlo con cierta tarea, y como nos explayamos anteriormente a cada iteración del entrenamiento, evaluamos el error y actualizamos los parámetros. Para poder generar estos embedding existen dos tipos de algoritmos, **CBOW** y **skip-gram** cada uno con diferentes objetivos.

CBOW (Continuous bag of words): tiene como objetivo predecir la palabra central dentro de un contexto (C), palabras colindantes, por ejemplo en la sentencia *el perro ladra* y un $C = 2$ vamos a tratar de predecir *perro* dado [*el, ladra*]. En la Figura 2.9 podemos ver el modelo *cbow*, donde C es el tamaño del contexto, V es la longitud del vocabulario, N es la dimensión de los word embeddings y W, W' son los parámetros del modelo. En Algoritmo 1 podemos ver el procedimiento.

Algorithm 1 CBOW

- 1: Generar los **one-hot vector** con las palabras del contexto, sean x_1, \dots, x_C estos vectores donde $x_i \in \mathbb{R}^{|V|}$. También para la palabra objetivo $y \in \mathbb{R}^{|V|}$.
 - 2: Generar el word embedding promedio $h = \frac{v_1 + \dots + v_C}{C}$.
 - 3: Generar el score vector $z = h \bullet W'$ donde $z \in \mathbb{R}^{|V|}$.
 - 4: Transformar el score vector en probabilidades $\hat{y} = \text{softmax}(z) \in \mathbb{R}^{|V|}$.
 - 5: Computar el error de la función objetivo cross entropy $H(y, \hat{y})$, actualizar los parámetros. Donde $H(y, \hat{y}) = y \log(\hat{y})$ y $\text{softmax}(z) = \frac{e^{z_i}}{\sum_{k=1}^{|V|} e^{z_k}}$.
-

Skip-gram: tiene como objetivo predecir las palabras de contexto a partir de la palabra central, por ejemplo para la sentencia antes mencionada *el perro ladra* y un $C = 2$ vamos a tratar de predecir [*el, ladra*] dado *perro*. En la Figura 2.10 podemos ver el modelo **skip-gram**, donde C es el tamaño del contexto, V es la longitud del vocabulario, N es la dimensión de los word embeddings y W, W' son los parámetros

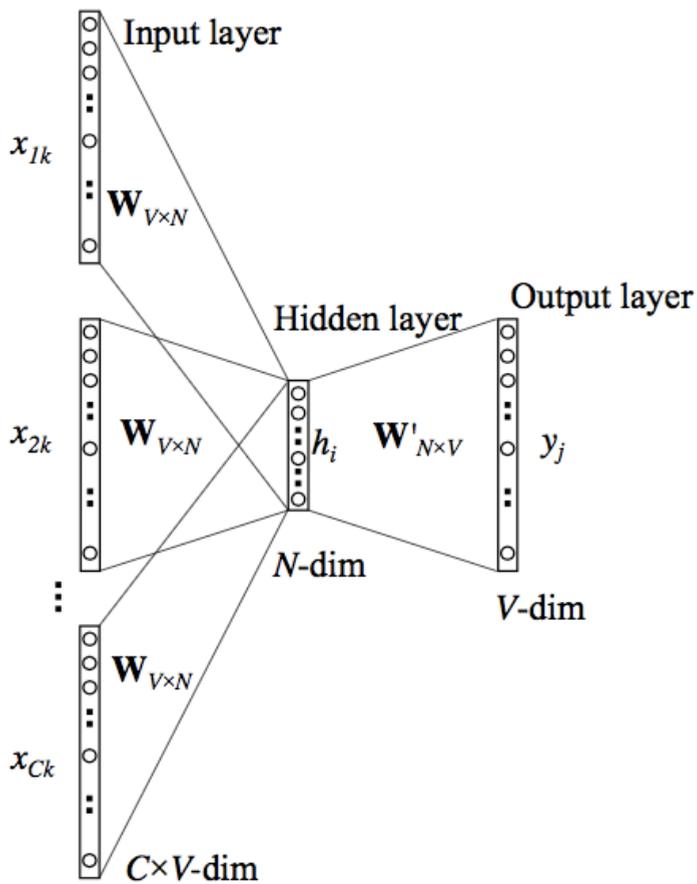


Figura 2.9: CBOV

del modelo. En Algoritmo 2 podemos ver el procedimiento detallado. Existe una optimización al momento del entrenamiento para el modelo skip-gram y este se basa en muestreo negativo (Mikolov et al., 2013b)

Algorithm 2 Skip-Gram

- 1: Generar los **one-hot vector** para la palabra central $x \in \mathbb{R}^{|V|}$ como también para las palabras del contexto que en este caso van a ser las objetivo $y_1, \dots, y_C \in \mathbb{R}^{|V|}$.
 - 2: Generar el word embeddings a partir de la palabra central, $v = x \bullet W$ donde $v \in \mathbb{R}^N$.
 - 3: Generar el score vector $z = v \bullet W'$ donde $z \in \mathbb{R}^{|V|}$
 - 4: Transformar el score vector en probabilidades $\hat{y} = \text{softmax}(z) \in \mathbb{R}^{|V|}$
 - 5: Computar el error de la función objetivo cross entropy $H'(\hat{y}, y_1, \dots, y_C)$, actualizar los parámetros, donde $H'(\hat{y}, y_1, \dots, y_C) = \sum_{i=1}^C H(y, y_i)$
-

Notar que el hiper parámetro más importante en ambos modelos es C , la longitud del contexto. También tenemos que notar que una vez entrenado el modelo ya sea *cbow* o *skip-gram* nos quedamos con la matriz de parámetros W' y W respectivamente, estas van a ser los word embedding finales que se utilizaran con cierto propósito.

2.4.2. Bert

Ciertos lenguajes de modelos pre entrenados han sido efectivos para mejorar algunas tareas de PLN, uno de ellos es Bert (Devlin et al., 2018), Bidirectional Encoder Representation from Transformer, en esta sección veremos la arquitectura del modelo, como representar la entrada del mismo, como ha sido el proceso de entrenamiento y por último lo que realmente nos importa en este trabajo que es como generar embeddings a partir de este modelo pre entrenado.

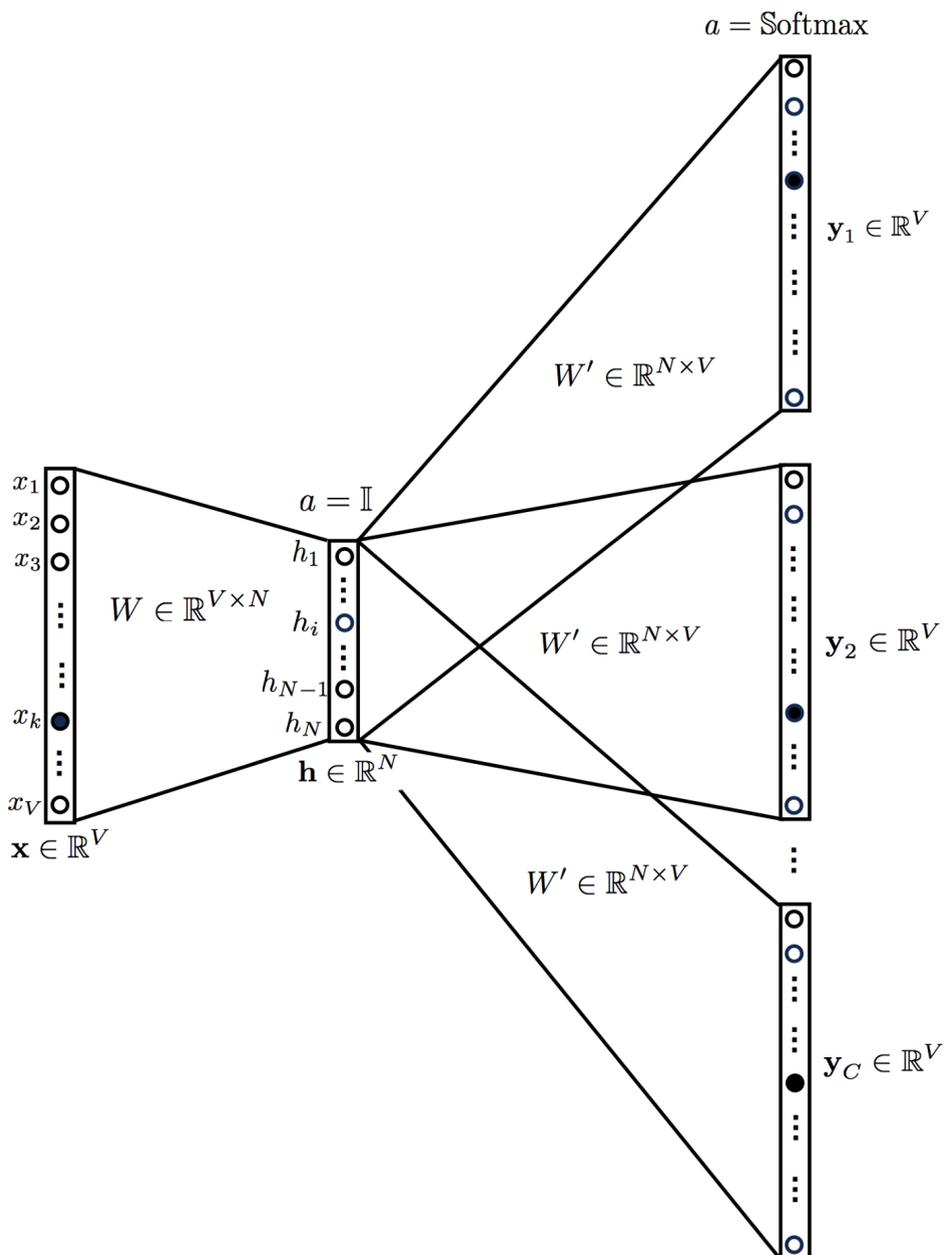
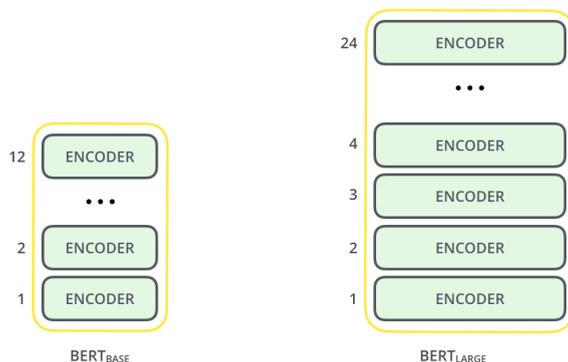


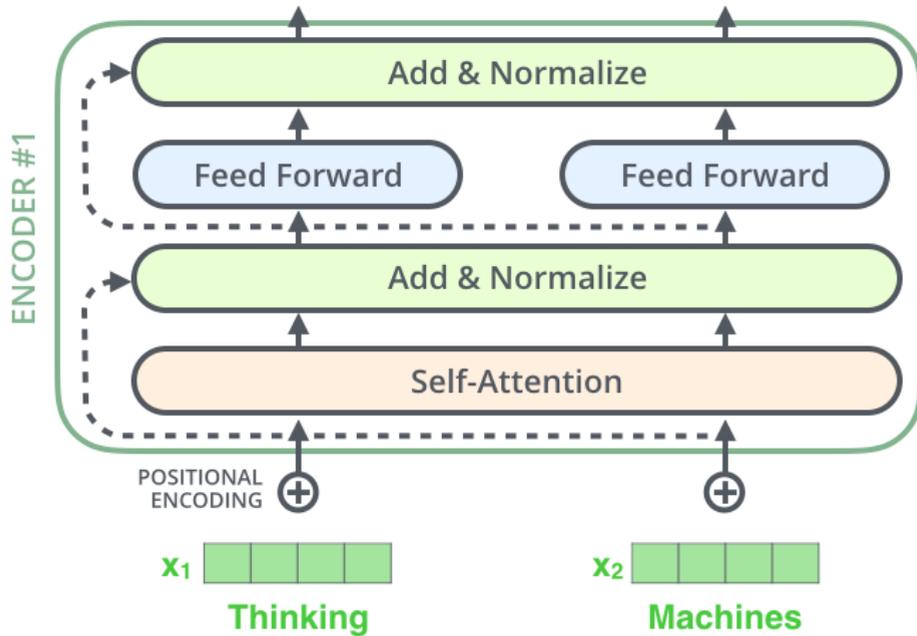
Figura 2.10: SKIP-GRAM

Figura 2.11: Arquitectura de Bert ¹

Arquitectura

Es básicamente una pila de encoders del transformer, implementado y descrito en (Vaswani et al., 2017), como podemos observar en la Figura 2.11 existe dos modelos diferentes. *BERTbase* con una pila de 12 encoder, 768 hidden layer y 12 self-attention heads $L = 12$, $H = 768$, $A = 12$, *parámetros* = 340M. *BERTlarge*, el cual contiene $L = 24$, $H = 1024$, $A = 16$, *parámetros* = 110M.

Ahora que tenemos una noción del modelo vamos a mostrar cómo es un encoder del mismo. Nos enfoquemos en la Figura 2.12 que nos muestra el funcionamiento del encoder para el caso de dos embeddings de entrada, lo primero a notar es que la información fluye hacia adelante en cada bloque del encoder, la primera capa es *bidirectional self-Attention* en la cual no vamos a entrar, para ver su comportamiento se deja una lectura recomendada *Attention is all you need* (Vaswani et al., 2017), otra capa que tenemos es *FFNN*, redes neuronales feed-forward, con $4H$ de capas ocultas cada una, estas ya las describimos anteriormente, por último nos queda explicar que después de cada self-attention o feedforward tenemos una capa de normalización (Ba et al., 2016) la cual consta de conexiones residuales que lo único que

Figura 2.12: Encoder ²

hace es sumar la entrada y salida de la capa en cuestión y retorna esa suma hacia la siguiente capa.

Representación de la entrada

Una particularidad de *BERT* es que puede tomar como entrada una o dos sentencias, por ejemplo para la tarea de Pregunta-Respuesta se utilizaría el segundo modelado de la entrada. Como factor común a todas las tareas que se quieran entrenar y/o hacer algún fine-tuning cada sentencia va a tener un token especial al principio como lo es *[CLS]* y cada token es alineado con el algoritmo de *WordPiece* (Wu et al., 2016), para el caso de un modelo que requiera dos sentencias

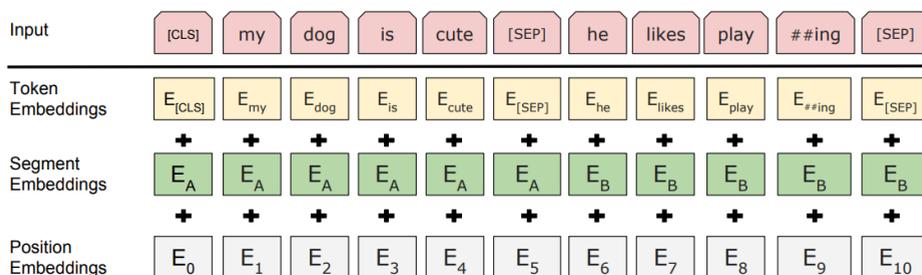


Figura 2.13: Entrada de Bert

cada una de ellas va a estar separado por el token $[SEP]$. Ahora a partir de estos token tenemos que generar un embedding para que sea la entrada al modelo, este embedding está compuesto por la sumatoria de tres diferentes embeddings, el primero es el token propiamente dicho, el segundo es un embedding que representa que sentencia es primera o segunda, y por último un embedding posicional del token con respecto a la sentencia. En la Figura 2.13 podemos ver un ejemplo de esta representación.

Entrenamiento

Lo novedoso de *BERT* es la forma que se entrenó el modelo de lenguaje, a diferencia de los modelos tradicionales donde se utiliza la técnica izquierda-derecha o viceversa, el pre-entrenamiento se basó en dos modelos no supervisados como *MaskedLM* y *predicción de la próxima sentencia*, en esta sección se detallará *MLM*. Los modelos clásicos no utilizaban modelos de lenguajes bidireccional ya que si queremos predecir algún token este indirectamente se puede estar viendo a si mismo, para atacar esta deficiencia el enfoque utilizado por *BERT* fue ocultar, mask en inglés, un porcentaje de tokens de la sentencia de entrada de manera aleatoria y setearlos con el token especial $[MASK]$, de esta manera la tarea sería predecir aquellos token $[MASK]$, específicamente ese porcentaje fue 15%, a partir de esto sur-

gen otros problemas a futuro, como por ejemplo ¿Como hacemos en un hipotético caso de fine-tuning?, esta discrepancia entre el pre entrenamiento y el fine-tuning se debe a que no se tiene el token *[MASK]* en el último de estos, para abordar esta problemática se consideró que de ese 15 % de tokens que se reemplazan con el token especial, el 80 % se siguiera reemplazando de esa forma, otro 10 % que se reemplacen con algún token aleatorio, y el 10 % restante que se mantuviera el mismo token.

Bert Embeddings

Podemos usar el modelo pre entrenado para generar word embeddings contextualizados, lo podemos hacer de diferentes maneras ya que podemos tener un word embedding por capa de la arquitectura del modelo, por ejemplo en la Figura 2.14 si le damos como entrada al modelo la sentencia *[CLS] el perro ladra* podemos extraer la salida de cada encoder y asociar el primer vector para el token *[CLS]*, el segundo para *perro* y el tercero para *ladra*. En el paper (Devlin et al., 2018) los autores comparan diferentes formas de extracción de features para una tarea determinada en *PLN* como lo es *NER*, name entity recognition, y para ello utilizaron diferentes combinaciones que está bueno explayarse como tomar la primera capa del modelo (Embeddings de entrada), la salida del modelo, la ante última capa, la suma de los embeddings de las últimas cuatro capas, la concatenación de las últimas cuatro capas y por último la suma de las 12 capas.

2.5. Learning to Rank

Learning to Rank, *LTR*, es una forma de aplicar técnicas de aprendizaje automático supervisado para resolver ciertas tareas en las cuales los objetivos son obtener un orden parcial. *LTR* es bastante utilizado en áreas tales como Information retrieval, PLN, Minería de Datos. La principal diferencia con las tareas de aprendizaje automáti-

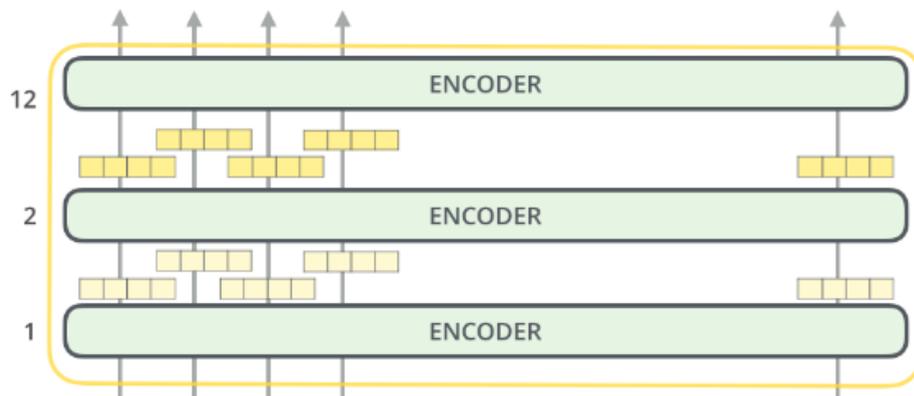


Figura 2.14: Bert Embedding

co supervisado es que en estos el objetivo principal es obtener una predicción, ya sea regresión o clasificación, en cambio en las tareas de LTR el objetivo es obtener un ordenamiento óptimo de los ítems. En la Figura 2.15 podemos ver un ejemplo de un sistema de *LTR* para el caso de Retrieval como puede ser un Motor de Búsqueda, en el ejemplo los d_i serían los documentos o redirecciones a páginas disponibles, q representa la búsqueda del usuario, y $d_{q,i}$ representa los primeros n_q documentos. Nota: $d_q \ll d_N$.

Existen diferentes enfoques en los sistemas de *LTR* tales como *pointwise*, *pairwise* y *listwise*, la principal discrepancia entre estos radica en cuantos documentos consideramos en la función objetivo, loss function, durante el entrenamiento del modelo.

Pointwise

Este enfoque mira un solo documento en la loss function, básicamente se entrena un clasificador/regresor y se trata de predecir cuán relevante es el documento, el ranking final es compuesto ordenando los n_q documentos, cabe destacar que el score de cada documento es

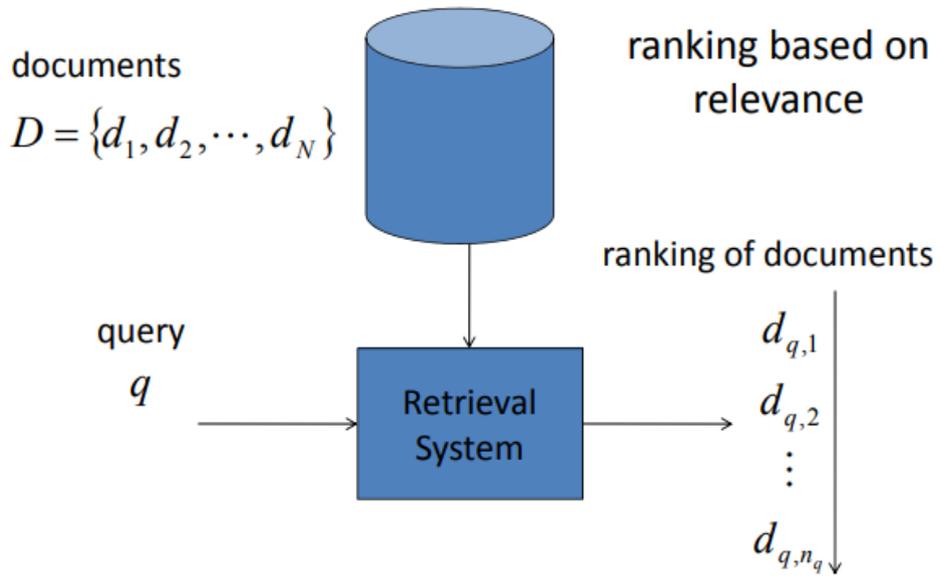


Figura 2.15: Learning to Rank

independiente de los otros documentos en el ordenamiento final.

Pairwise

Este enfoque mira en la loss function documentos de a pares y trata de obtener el ranking óptimo. El objetivo es minimizar el número de inversiones en ordenamiento que el modelo predice, en la práctica este enfoque es mejor que el de pointwise ya que predecir el orden relativo modela mejor la realidad que predecir una relevancia. Unos ejemplos de estos algoritmos son *RankNet*, *LambdaRank* y *Lambda-MART* (Burges, 2010).

Listwise

Este enfoque mira directamente la lista entera de los documentos y trata de obtener el ordenamiento ideal. Unos ejemplos de estos algoritmos son *SoftRank* (Taylor et al., 2008), *ListMLE* (Lan et al., 2014) y *ListNet* (Cao et al., 2007).

Capítulo 3

Image Tagging

En este capítulo nos enfocaremos en detallar el trabajo realizado por los directores del presente trabajo de tesis (Sanchez et al., 2018) que sientan la base de todos los experimentos que se detallarán en la próxima sección, el objetivo final es obtener un sistema de etiquetado automático tal que dado una imagen x , una lista de tags t_1, \dots, t_n , nos retorna esta misma lista de tags pero ordenadas de acuerdo a su correspondencia semántica con lo que nos muestra la imagen. En la Tabla 3.1 podemos ver ejemplos en los cuales queremos etiquetar determinadas imágenes con un conjunto fijo de tags. A continuación veremos el modelo en sí mismo, siguiendo con las diferentes funciones de pérdida que se utilizaron, cómo se generaron los datos para poder entrenarlo, las métricas utilizadas y por último veremos los resultados obtenidos.

3.1. El modelo

Dada una imagen que la representamos con $x \in X$, una etiqueta o tag $y \in Y$ el objetivo del modelo es entrenar una función tal que nos de un score o puntuación de cuan fuerte es la compatibilidad semántica de y con el contenido visual de x . Esta función está dada por

$$s_{(\psi, \varphi)}(x, y; W) = \psi(x)^T W \varphi(y) \quad (3.1)$$

donde $\psi : X \mapsto \mathbb{R}^D$, $\varphi : Y \mapsto \mathbb{R}^E$ representan como generar embeddings a partir de imágenes y palabras respectivamente, un ejemplo de elección puede ser (*VGG19, word2vec*), cabe destacar que estos modelos para generar embeddings tienen que estar fijos en todo momento del pipeline, volviendo a la función, el objetivo del modelo es aprender la matriz de parámetros $W \in \mathbb{R}^{D \times E}$.

3.2. Dataset

Para poder entrenar el modelo será requisito tener un conjunto de datos estructurados de cierta forma, en esta sección vamos a definir esa estructura y luego veremos en la Sección 3.5 cómo a partir de un conjunto de datos existente lo construimos. Sea $D = (x_0, Y_0), \dots, (x_N, Y_N)$ nuestro conjunto de datos donde $x_i \in X$ son imágenes e $Y_i = y_0^i, \dots, y_{|Y_i|}^i$ es un conjunto ordenado de tags; para dar ese ordenamiento supondremos que contamos con una función de relevancia $r : Y \times X \mapsto \mathbb{R}$ tal que $r(y_0^i; x_i) > \dots > r(y_{|Y_i|}^i; x_i) \forall i$, esta función también la definiremos en la Sección 3.5.

3.3. Funciones de pérdida

Sea $L(W; r) = \sum_{n=1}^N l(\hat{Y}(x_n), Y_n)$ la función de pérdida donde Y_n es el conjunto de tags para la imagen x_n ordenados por \leq_r y $\hat{Y}(x_n)$ el mismo conjunto de tags pero ordenados por \leq_s . En esta sección se detallarán las distintas funciones l , específicamente vamos a ver tres que son las que se encuentran en el trabajo realizado en conjunto por los directores (Sanchez et al., 2018).

Structured Joint Embedding (SJE1)

$$l_{SJE1}(x, Y) = \max_{1 \leq i \leq |Y|} [\Delta(1, i) + s_{(\psi, \varphi)}(x, Y_i) - s_{(\psi, \varphi)}(x, Y_1)]_+$$

donde

$$[z]_+ \equiv \max(0, z)$$

$$\Delta : N \times N \mapsto \mathbb{R}$$

$$\Delta(k, k') = 0 \quad \text{si } k = k'$$

$$\Delta(k, k') = 1 \quad \text{caso contrario}$$

Structured Joint Embedding (SJE2)

$$l_{SJE2}(x, Y) = \max_{1 \leq i \leq |Y|} [\Delta(1, i) + s_{(\psi, \varphi)}(x, Y_i) - s_{(\psi, \varphi)}(x, Y_1)]_+$$

donde

$$[z]_+ \equiv \max(0, z)$$

$$\Delta : N \times N \mapsto \mathbb{R}$$

$$\Delta(k, k') = 1 - \frac{1}{k' - k + 1} \quad \text{si } k' > k'$$

SJE1 y *SJE2* básicamente tratan de seguir el enfoque pairwise que vimos anteriormente en los sistemas LTR Sección 2.5, ya que se podrían ver como que ambas penalizan las inversiones con respecto al primer tag asociado a la imagen.

Listwise Structured Joint Embedding (ListSJE)

$$l_{ListSJE}^{K_{top}}(x, Y) = \sum_{i=1}^{K_{top}} \sum_{i=k}^{|Y|} [\Delta(k, i) + s_{(\psi, \varphi)}(x, Y_i) - s_{(\psi, \varphi)}(x, Y_k)]_+$$

donde

$$\begin{aligned} K_{top} &\leq |Y| \\ \Delta(k, k') &= 1 - \frac{1}{k' - k + 1} \quad \text{si } k' > k' \end{aligned}$$

La diferencia de esta formulación con las anteriores radica en que se tiene en cuenta el orden parcial de los tags y no solamente las inversiones con respecto al primero.

3.4. Métricas

La métrica que se utilizó en el trabajo propuesto de los directores y en los experimentos que detallaremos en la sección 4, es *precisión@k*, popularmente conocida como $p@k$, con $k = 1, 5$, que está definida como el ratio de elementos relevantes entre los primeros k lugares, vamos a tratar de dar un ejemplo para poner la métrica en el contexto de nuestro alcance, supongamos que para una imagen determinada tenemos anotados los siguientes tags [*perro, gato, caballo, oveja*] y el modelo nos da el siguiente ordenamiento [*gato, caballo, oveja, perro*] por lo tanto $p@1 = \frac{0}{1} = 0$, $p@2 = \frac{1}{2}$, $p@3 = \frac{2}{3}$, $p@4 = \frac{4}{4} = 1$.

3.5. Generación de tags ordenados

En la Sección 3.2 hablamos de la estructura que tenía que tener nuestro dataset pero nunca especificamos cual iba a ser concretamente, también supusimos que teníamos una función de relevancia r , en esta sección nos centraremos en tener estas dos definiciones. Recordemos que queremos llegar a tener un conjunto de



Figura 3.1: Captions:

a small fluffy dog sitting on a blue couch.

a white dog is sitting on a couch.

a shot shows pale blue wall over a well-stuffed blue couch that dwarfs the already small, fluffy dog resting face-forward on one of its cushions.

a small adorable dog sitting on a sofa cushion.

a small dog sits on a blue sofa

datos $D = (x_0, Y_0), \dots, (x_N, Y_N)$ tal que $x_i \in X$ son imágenes e $Y_i = y_0^i, \dots, y_{|Y_i|}^i$ es un conjunto ordenado de tags, para la construcción del mismo se basará en el dataset *COCO* (Chen et al., 2015) donde cada imagen viene asociada con cinco sentencias que la describen, en las Figuras 3.1, 3.2, 3.3 podemos ver algunas de ellas con sus respectivas anotaciones, ahora bien la pregunta que sigue es ¿Cómo generar los tags a partir de las sentencias de *COCO*?, para ello primero tenemos que hacer un par de definiciones,

sea $C(x) = \{c_1, \dots, c_Q\}$ el conjunto de sentencias asociado a la imagen x , $t(c_i) \equiv t_i = \{w : w \in c_i \wedge w \in \text{SUSTANTIVO}\}$ para $c_i \in C(x)$. Ahora estamos en condiciones de dar un conjunto de tags para cada imagen x_j , $Y_j = t(c_1) \cup \dots \cup t(c_Q)$. Notar que el conjunto de tags Y_j no están ordenados de acuerdo a ninguna relevancia y nuestro objetivo es tenerlos ordenados, por lo tanto resta definir la función



Figura 3.2: Captions:

white boat navigating on waterway near populated area.

a medium-sized boat cruising away from a harbor.

a boat floats in the water near the shore.

a small white boat in the middle of the water.

a white boat floating down a large body of water.



Figura 3.3: Captions:

a kitchen with a slanted ceiling and skylight.

a small kitchen with a lot of filled up shelves

a small kitchen with low a ceiling

an image of a kitchen loft style setting

a small kichen area with a sunlight and angled ceiling.

r , para ello daremos algunas definiciones predecesoras, sea $loc(w; c)$ que denota la ubicación relativa de la palabra w en la sentencia c , por ejemplo , $loc(gato; el\ gato\ negro) = \frac{2}{3}$, sea $count(w; c)$ la cantidad de veces que la palabra w aparece en la sentencia c , con loc y $count$ podemos definir

$$r_{loc}(w) = \max_{c \in C(x)} \{1 - loc(w; c) : w \in c\}$$

,

$$r_{freq}(w) = \frac{count(w; c_1) + \dots + count(w; c_Q)}{|t_1| + \dots + |t_Q|}$$

r_{loc} trata de capturar la idea de que si una palabra es mencionada al principio de una sentencia entonces es más relevante, r_{freq} trata de capturar la consistencia entre las sentencias ya que estas han sido anotadas por diferentes personas. Ahora con todas estas definiciones estamos en condiciones de definir una función de relevancia,

$$r(w) = \alpha r_{freq}(w) + (1 - \alpha)r_{loc}(w), 0 \leq \alpha \leq 1$$

cabe destacar que α será un hiper parámetro. Para resumir el conjunto de datos quedaría $Y_j = (\hat{Y}, r_\alpha)$, donde $\hat{Y} = t(c_1^j) \cup \dots \cup t(c_Q^j)$ para $j = 0, \dots, N$. Este proceso se aplica tanto para el conjunto de entrenamiento (82000 imágenes), como para el de validación (40000 imágenes) de *COCO* para un determinado α .

3.6. Resultados

En esta sección vamos a pasar a detallar los resultados obtenidos que se muestran en el paper (Sanchez et al., 2018), en la Tabla 3.2 podemos ver los resultados, para llegar a esos resultados en todos los casos se utilizó el conjunto de entrenamiento de *COCO* para valga la redundancia entrenar y validar el modelo y el conjunto de validación de *COCO* a modo de test y reporte siempre bajo el control del hiper parámetro α , también se utilizó $(\psi, \varphi) = (VGG19, word2vec)$ como

extractores de features para todos los casos y por último para el caso cuando se opto por usar *ListSJE* , subsección 3.3, se utilizó con $K_{top} = 5$.

x	t_1, t_2, t_3, t_4	Resultado
	perro, gato, oveja, caballo	caballo, perro, gato, oveja
	perro, gato, oveja, caballo	oveja, perro, gato, caballo
	perro, gato, oveja, caballo	gato, perro, oveja, caballo

Cuadro 3.1: Ejemplos de objetivos a alcanzar

Métrica	Loss	$\alpha = 0$	$\alpha = 0,25$	$\alpha = 0,50$	$\alpha = 0,75$	$\alpha = 1$
p@1	SJE1	0.4621	0.5752	0.6332	0.6457	0.5744
p@5	SJE1	0.6876	0.6814	0.6788	0.6786	0.6634
p@1	SJE2	0.4810	0.5848	0.6395	0.6527	0.5865
p@5	SJE2	0.7162	0.7118	0.7031	0.7003	0.6747
p@1	ListSJE	0.4660	0.5619	0.6250	0.6424	0.5791
p@5	ListSJE	0.7415	0.7411	0.7417	0.7385	0.6899

Cuadro 3.2: Resultados

Capítulo 4

Experimentos

En esta sección vamos a describir los experimentos realizados, el objetivo de cada uno de ellos y los resultados obtenidos.

4.1. Experimento 1

El objetivo de este experimento es buscar un enfoque para tratar de combinar las etiquetas de un conjunto Y , tal que si $y_j \in Y, \dots, y_{j+n} \in Y$ con $n \leq |Y|$ se corresponden semánticamente en la imagen x entonces elegir un representante $\hat{y} \in \{y_j, \dots, y_{j+n}\}$ tal que el nuevo conjunto de etiquetas es $\hat{Y} = \{y_0, \dots, \hat{y}, \dots, y_{|Y|}\}$. Para ponerlo en concreto vamos a dar un ejemplo de lo queremos alcanzar, para ello vamos a elegir una imagen con sus respectivas sentencias del conjunto de datos de *COCO* captions como denota la Figura 4.1, basándonos en la heurística detallada en la sección 3.5 para obtener el conjunto de tags conseguimos que $Y = \{pastry, frosting, hand, good, man, type, close, container, kitchen, baker, chef, person, dessert\}$.

En base a este conjunto de palabras que representan a la imagen podemos decir que $y_1 = \{baker, chef, person, man\}$ semánticamente representan lo mismo al igual que $y_2 = \{pastry, dessert\}$, por lo tanto un conjunto de datos óptimo sería $\hat{Y} = \{\hat{y}_1, \hat{y}_2, frosting, hand, good,$



Figura 4.1: Captions:

- *a chef is preparing and decorating many small pastries.*
- *a man preparing desserts in a kitchen covered in frosting.*
- *a baker prepares various types of baked goods.*
- *a close up of a person grabbing a pastry in a container*
- *close up of a hand touching various pastries.*

$\{type, close, container, kitchen\}$ donde \hat{y}_1, \hat{y}_2 son representantes de y_1, y_2 respectivamente.

Ahora ya sabemos el objetivo del experimento, nos resta dar la heurística utilizada para tratar dar con estas combinaciones de etiquetas, pero para ello primero veremos algunos conceptos claves en el enfoque para aproximar estas combinaciones, veremos mediante ejemplos lo que significa dentro de *PLN* nociones como un árbol sintáctico de dependencias, *POS* tagging y lematización. Primero veremos un tipo de árbol sintáctico de dependencia como lo es el *DependencyParser* de *Spacy*¹ que está basado en las dependencias estándares de Stanford (de Marneffe et al., 2014)², en la Figura 4.2 podemos ver el árbol de parseo sintáctico para la sentencia c_3 de la Figura 4.1 y en la Tabla 4.1 podemos ver las relaciones sintácticas expuestas en el ejemplo, básicamente cada token depende o está conectado con otro de la misma sentencia a través de una relación podemos definir las como $R_c \subseteq \{(t_1, t_2, rel) | t_1, t_2 \in c \wedge rel \in REL\}$ donde REL es el conjunto de relaciones estándares. Otra noción importante y que se utilizará es *POS* tagging³, part of speech tagging por sus siglas en inglés, el cual consiste en predecir una categoría gramatical para cada palabra/token, por ejemplo en el ejemplo de la figura 4.2 que vimos anteriormente en el ejemplo del parser, podemos notar que *baker* tiene como categoría gramatical *NOUN* lo cual es sustantivo en inglés. Por último otro concepto vastamente utilizado es la lematización que consiste en quedarse con la raíz del token un ejemplo de esto puede ser *prepare* para el token *preparing*.

Sea $C(x)$, Y el conjunto de sentencias y el conjunto de tags ordenados correspondiente a la imagen x respectivamente que ya se describió en la sección 3.5, vamos a dar un algoritmo para generar \hat{Y} el cual es el conjunto de tags ordenados utilizando la combinación de etiquetas.

¹<https://spacy.io/api/dependencyparser>

²<https://universaldependencies.org/u/dep/>

³<https://spacy.io/usage/linguistic-features#pos-tagging>

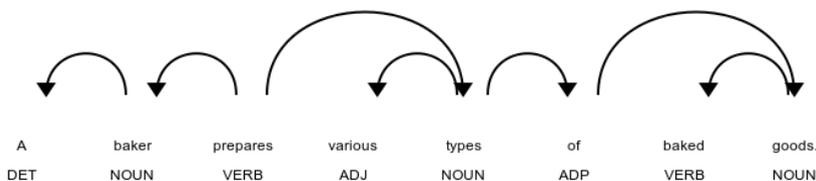


Figura 4.2: Árbol de parseo sintáctico

token	dependencia	ancestro
A	det	baker
baker	nsubj	prepares
prepares	ROOT	prepares
various	amod	types
types	dobj	prepares
of	prep	types
baked	amod	goods
goods	pobj	of

Cuadro 4.1: Dependencias de la sentencia c_3 de la Figura 4.1

Definamos

$$KW_x = \{w | pos(w) \subseteq \{NOUN, VERB\}, w \in c_i, c_i \in C(x)\}$$

donde $pos(w)$ es una función que dado un token w retorna su correspondiente *POS* tagging, sea

$$t_i = \{t_i^1, \dots, t_i^m\}$$

tal que $lemma(t_i^1) = \dots = lemma(t_i^m)$ y $t_i^j \in KW_x$ donde $lemma(w)$ es una función que retorna la lematización del token dado, a partir de estos conjuntos vamos a generar otro conjunto de relaciones basados en la dependencia sintáctica, siguiendo con las definiciones sea

$$RC = \{(rel, c_k) | c_k \in child(t_i^j), t_i^j \in t_i, rel = reldep(t_i^j, c_k), rel \in REL\}$$

donde $reldep(w_1, w_2)$ es una función donde dado dos token retorna su relación basado en el árbol sintáctico de dependencia y $child(w)$ es una función donde retorna los hijos inmediatos en el árbol sintáctico para el token w . Ahora estamos en condiciones de dar un conjunto de tags el cual podemos combinar, sea

$$M = \{y_1, \dots, y_n\}$$

tal que $(rel, y_i) \in RC$, rel es la misma para todos los y_i y por último $y_i \in Y$, pasándolo en limpio esto nos quiere decir que vamos a combinar aquellas etiquetas en las cuales tengan la misma relación en el árbol sintáctico y además están contenidas en el conjunto de tags inicial, solo nos resta la elección del representante $\hat{y} \in M$ y cómo será su score de relevancia r , \hat{y} se selecciona de manera aleatoria entre los elementos de conjunto M , supongamos que $\hat{y} = y_j$ para $1 \leq j \leq n$ y su función r está dada por

$$r_{loc}(\hat{y}) = r_{loc}(y_j)$$

$$r_{freq}(\hat{y}) = \frac{\sum_{i=1}^n \sum_{j=1}^Q count(y_i; c_j)}{\sum_{k=1}^Q |t_k|}$$

por lo tanto:

$$r_\alpha(\hat{y}) = \alpha r_{freq}(\hat{y}) + (1 - \alpha) r_{loc}(\hat{y})$$

En la tabla 4.2 podemos ver los resultados de aplicar la heurística mencionada.

A continuación veremos un par de ejemplos en los cuales estas combinaciones de etiquetas no funcionaron bien el cual nos servirá

Métrica	Loss	$\alpha = 0$	$\alpha = 0,25$	$\alpha = 0,50$	$\alpha = 0,75$	$\alpha = 1$
p@1	SJE1	0,4803	0,5838	0,6285	0,6187	0,5383
p@5	SJE1	0,7043	0,7009	0,6977	0,6957	0,6773
p@1	SJE2	0,4980	0,5931	0,6363	0,6287	0,5530
p@5	SJE2	0,7307	0,7261	0,7206	0,7175	0,6963
p@1	ListSJE	0,4842	0,5706	0,6209	0,6182	0,5456
p@5	ListSJE	0,7539	0,7541	0,7540	0,7497	0,7179

Cuadro 4.2: Resultados

como puntapié para presentar el experimento número dos que veremos en la Sección 4.2.

Aplicando la heurística para la Figura 4.3 llegamos a combinar las siguientes etiquetas $\{egg, bacon\}$ como también $\{ham, spinach\}$ lo cual no es correcto ya que no se corresponden semánticamente, en la Figura 4.4 podemos ver porque se llega a esta combinación para el segundo a través de su árbol de parseo sintáctico, esto nos dice que se combinan ya que $\{ham, spinach\} \in child(egg)$ y ambos están bajo la misma $rel = conj$.

4.2. Experimento 2

El objetivo del presente experimento es tratar de atacar las deficiencias obtenidas como conclusión al experimento detallado en la Sección 4.1, para ello vamos a seguir combinando las etiquetas como lo mencionamos anteriormente pero con la salvedad de hacer un chequeo extra, el cual nos dice que se combinarán las etiquetas si y solamente si son hiperónimos de a pares. Sea $Hyper(w_1, w_2)$ un predicado el cual nos dice si el token w_2 es hiperónimo del token w_1 para realizar este chequeo hacemos uso de *WordNet* (Fellbaum, 1998), recordemos lo visto en la Sección 4.1 que M es el conjunto de potenciales etiquetas a combinar, con esta referencia al experimento anterior vamos a



Figura 4.3: Captions:

- *Plate with breakfast sandwich made with English muffin, egg and ham.*
- *a sandwich on a plate on a table*
- *A white plate topped with a muffin filled with breakfast food.*
- *A sandwich with egg and ham and spinach*
- *Two sandwiches on English muffins featuring greens and cheddar cheese on one sandwich and Canadian bacon and an egg on the other sandwich.*

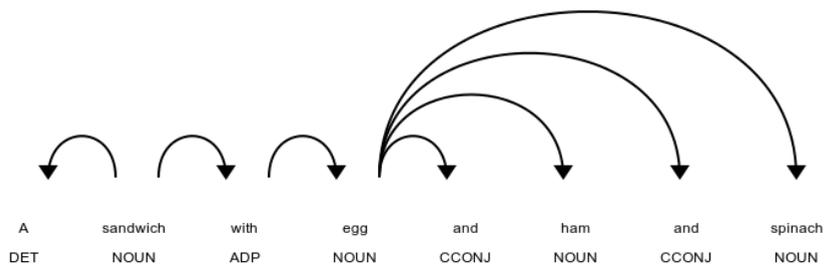


Figura 4.4: Árbol de parseo sintáctico

combinarlas si se cumple la siguiente propiedad:

$$\forall w_1, w_2 \in M : Hyper(w_1, w_2) \vee Hyper(w_2, w_1)$$

Una vez realizado este chequeo extra se procede a elegir el representante $\hat{y} \in M$ y su score $r_\alpha(\hat{y})$ de la misma manera que lo visto en Sección 4.1.

En el Cuadro 4.3 podemos ver los resultados obtenidos.

Métrica	Loss	$\alpha = 0$	$\alpha = 0,25$	$\alpha = 0,50$	$\alpha = 0,75$	$\alpha = 1$
p@1	SJE1	0,4703	0,5788	0,6332	0,6411	0,5646
p@5	SJE1	0,6906	0,6856	0,6811	0,6808	0,6714
p@1	SJE2	0,4885	0,5868	0,6420	0,6498	0,5784
p@5	SJE2	0,7195	0,7145	0,7051	0,7019	0,6882
p@1	ListSJE	0,4758	0,5644	0,6244	0,6412	0,5719
p@5	ListSJE	0,7435	0,7434	0,7435	0,7401	0,7098

Cuadro 4.3: Resultados

4.3. Experimento 3

El objetivo de este experimento es reemplazar $\varphi = word2vec$ por $\varphi = BertEmb_{BASE}$ y $\varphi = BertEmb_{LARGE}$, a diferencia de lo visto en el Capítulo 3, específicamente fijaremos $(\psi, \varphi) = (VGG19, BertEmb_{BASE})$ y $(\psi, \varphi) = (VGG19, BertEmb_{LARGE})$ como extractores de features visuales y textuales para el modelo descrito en la Ecuación 3.1. Para la extracción de features textuales vamos a utilizar la última capa del encoder tanto de $BERT_{BASE}$ como de $BERT_{LARGE}$ utilizándolo como se detalló en la Sección 2.4.2, para el resto de la sección nos vamos a abstraer de estos dos modelos y no vamos a discriminar entre $BERT_{BASE}$ y $BERT_{LARGE}$, luego en la exposición de resultados sí tendremos en cuenta cada modelo por separado. Ahora bien ¿Cómo a partir de $BERT$ llegamos a $BertEmb$?, para ello vamos a detallarlo en el Algoritmo 3, básicamente el diferencial de utilizar $BERT$ en vez de $word2vec$ es la generación de embeddings contextualizados, para un mismo token podemos tener varios embeddings que lo representen dependiendo del contexto de la sentencia en la cual el token ocurrió, en $BertEmb$ nos quedamos con el centroide de cada token del vocabulario en base a todas las ocurrencias del mismo.

Algorithm 3 *BertEmb*

```

1:  $BertEmb := dict()$ 
2:  $temp := dict(list)$ 
3: for  $set \in \{COCO_{train}, COCO_{val}\}$  do
4:   for  $(x, C(x)) \in set$  do
5:     for  $c_i \in C(x)$  do
6:        $w_1^v, \dots, w_k^v := BERT(c_i)$ 
7:        $temp[w_j] : + = w_j^v$  para  $i = 1, \dots, k$ 
8:     end for
9:   end for
10: end for
11:  $BertEmb[w_j] := \mathbf{mean}(temp[w_j])$  para  $i = 1, \dots, |V|$ 

```

Ahora que ya tenemos definido *BertEmb* podemos dar los resultados de entrenar el modelo de la Ecuación 3.1 fijando $(\psi, \varphi) = (VGG19, BertEmb_{BASE})$ y $(\psi, \varphi) = (VGG19, BertEmb_{LARGE})$, dichos resultados se exponen en los Cuadros 4.4, 4.5 respectivamente.

Métrica	Loss	$\alpha = 0$	$\alpha = 0,25$	$\alpha = 0,50$	$\alpha = 0,75$	$\alpha = 1$
p@1	SJE1	0.4581	0.5714	0.6306	0.6416	0.5683
p@5	SJE1	0.685	0.6804	0.6778	0.6788	0.6673
p@1	SJE2	0.4789	0.5832	0.6387	0.6499	0.5825
p@5	SJE2	0.7149	0.7090	0.7018	0.6994	0.6849
p@1	ListSJE	0.4654	0.5604	0.6195	0.6422	0.5786
p@5	ListSJE	0.7400	0.7407	0.7408	0.7373	0.7095

Cuadro 4.4: Resultados

Métrica	Loss	$\alpha = 0$	$\alpha = 0,25$	$\alpha = 0,50$	$\alpha = 0,75$	$\alpha = 1$
p@1	SJE1	0.4598	0.5696	0.6281	0.6397	0.5684
p@5	SJE1	0.6850	0.6802	0.6773	0.6750	0.6637
p@1	SJE2	0.4774	0.5809	0.6349	0.6480	0.5824
p@5	SJE2	0.7130	0.7076	0.6978	0.6930	0.6818
p@1	ListSJE	0.4678	0.5608	0.6201	0.6402	0.5760
p@5	ListSJE	0.7385	0.7380	0.7384	0.7364	0.7073

Cuadro 4.5: Resultados

4.4. Experimento 4

La idea de este experimento es cambiar el modelo bilineal descrito en la Sección 3.1, el cual siempre lo mantuvimos fijo, por uno que vamos a explayarnos a continuación y ver como impacta en los resultados.

El modelo que se utilizo queda expuesto en la Figura 4.5 donde cada bloque de esta Arquitectura lo podemos ver en mayor detalle en

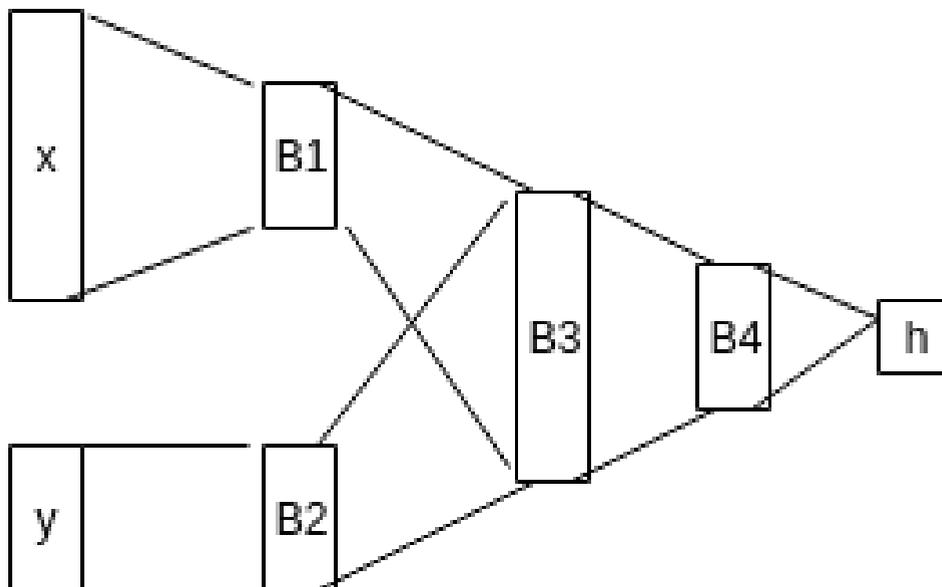


Figura 4.5: Modelo del Experimento 4.4

el Cuadro 4.8, podemos ver que seguimos extrayendo features tanto visuales como de texto con *VGG19* y *word2vec* respectivamente. La noción de concatenar los embeddings transformados en el bloque *B3* se toma desde la arquitectura propuesta en el paper (Galron et al., 2018).

Los resultados obtenidos se muestran en el Cuadro 4.7, como podemos ver no hay muchas diferencias que utilizar el modelo bilineal.

4.5. Experimento 5

El presente experimento se enfocará en encontrar una representación conjunta, tanto para la imagen como para el tag, a través de un *Autoencoder* para luego a partir de esto poder entrenar un modelo lineal, regresor, que nos estima el score que antes obteníamos

Bloque	Composición	entrada	salida
x	VGG19	-	4096
y	word2vec	-	300
B1	<i>Dense + ReLU</i>	4096	300
B2	<i>Dense + ReLU</i>	300	300
B3	<i>ConCat + ReLU</i>	(300, 300)	600
B4	<i>Dense + ReLU</i>	600	300
h	<i>Dense + Linear</i>	300	1

Cuadro 4.6: Bloques

Métrica	Loss	$\alpha = 0$	$\alpha = 0,25$	$\alpha = 0,50$	$\alpha = 0,75$	$\alpha = 1$
p@1	SJE1	0.4336	0.5501	0.607	0.6189	0.5392
p@5	SJE1	0.6749	0.6695	0.6682	0.6702	0.6677
p@1	SJE2	0.4715	0.5728	0.6262	0.6394	0.575
p@5	SJE2	0.7209	0.7169	0.7057	0.7029	0.6915
p@1	ListSJE	0.4632	0.56	0.6188	0.6378	0.5742
p@5	ListSJE	0.7474	0.7482	0.7477	0.7436	0.7141

Cuadro 4.7: Resultados

con $s_{(\psi, \varphi)}(x, y; W)$. En la Figura 4.6 podemos ver la arquitectura del *Autoencoder*, llamemos a este modelo **AutoEnc**, para el diseño de la misma nos basamos en la arquitectura del paper (Wang et al., 2019), la idea es reconstruir la salida tanto la imagen como el tag que esta toma como entrada. En el Cuadro 4.8 podemos ver la definición de cada bloque de la Figura 4.6. Para entrenar este modelo se utilizó el mismo conjunto de entrenamiento y validación con la generación de tags descripta en el Capítulo 3.

Ahora bien, una pregunta interesante es ¿cómo está definida la función de pérdida para este modelo?, se la definió como la suma de los *mse* de cada uno:

$$L_{AutoEnc}(x, y) = mse(x, x') + mse(y, y')$$

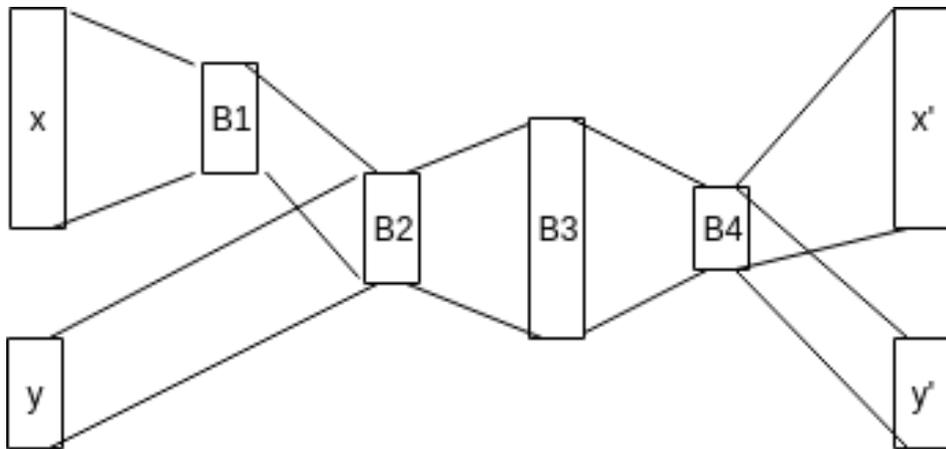


Figura 4.6: Modelo del Experimento 4.5

donde x' , y' son la salida de *AutoEnc* y mse es el error cuadrático medio.

Una vez que tenemos *AutoEnc* entrenado procedemos a extraer esta representación conjunta para una imagen y un tag del bloque $B4$ y a partir de esta representación entrenamos un modelo lineal para estimar el score de relevancia expuesto anteriormente. En el Cuadro 4.9 podemos ver los resultados obtenidos en este experimento.

Bloque	Composición	entrada	salida
x	VGG19	-	4096
y	word2vec	-	300
B1	<i>Dense + ReLU</i>	4096	300
B2	<i>Dense + ReLU</i>	(300, 300)	(300, 300)
B3	<i>ConCat</i>	(300, 300)	600
B4	<i>Dense + ReLU</i>	600	200
x'	<i>Dense + Linear</i>	200	4096
y'	<i>Dense + Linear</i>	200	300

Cuadro 4.8: Bloques

Métrica	Loss	$\alpha = 0,50$
p@1	SJE1	0.2329
p@5	SJE1	0.5643

Cuadro 4.9: Resultados

Capítulo 5

Conclusiones

5.1. Conclusiones

En el presente trabajo de grado se realizó un revistamiento del modelo multimodal propuesto en el paper que dió el puntapié a este trabajo, se plantearon experimentos de diferentes índoles ya sea atacando la parte de la generación de tags ordenados a partir del conjunto de datos, los cuales tuvieron una mejora pero no significativa con respecto a la forma planteada originalmente, también hubo algunos en los cuales se cambió la forma de representar las etiquetas utilizando algoritmos/modelos más modernos en los cuales no se obtuvo mejora significativa. Por último vimos algunos experimentos en los cuales se cambiaba el modelo directamente, específicamente dos modelos, en el primero se obtuvieron resultados similares de los cuales partimos y para el segundo de ellos se obtuvieron resultados negativos pero la idea de generar un Autoencoder para la extracción de features conjuntos puede tener potencial para un trabajo futuro.

Bibliografía

Ba, J. L., Kiros, J. R., and Hinton, G. E. (2016). Layer normalization.

Burges, C. J. (2010). From ranknet to lambdarank to lambdamart: An overview. Technical Report MSR-TR-2010-82.

Cao, Z., Qin, T., Liu, T.-Y., Tsai, M.-F., and Li, H. (2007). Learning to rank: From pairwise approach to listwise approach. Technical Report MSR-TR-2007-40.

Chen, X., Fang, H., Lin, T.-Y., Vedantam, R., Gupta, S., Dollar, P., and Zitnick, C. L. (2015). Microsoft coco captions: Data collection and evaluation server.

de Marneffe, M.-C., Dozat, T., Silveira, N., Haverinen, K., Ginter, F., Nivre, J., and Manning, C. D. (2014). Universal Stanford dependencies: A cross-linguistic typology. In *Proceedings of the Ninth International Conference on Language Resources and Evaluation (LREC'14)*, pages 4585–4592, Reykjavik, Iceland. European Language Resources Association (ELRA).

Deng, J., Dong, W., Socher, R., Li, L.-J., Li, K., and Fei-Fei, L. (2009). ImageNet: A Large-Scale Hierarchical Image Database. In *CVPR09*.

Devlin, J., Chang, M.-W., Lee, K., and Toutanova, K. (2018). Bert: Pre-training of deep bidirectional transformers for language understanding.

- Fellbaum, C. (1998). *WordNet: An Electronic Lexical Database*. Bradford Books.
- Galron, D. A., Brovman, Y. M., Chung, J., Wieja, M., and Wang, P. (2018). Deep item-based collaborative filtering for sparse implicit feedback. *CoRR*, abs/1812.10546.
- Kelley, H. J. (1960). Gradient theory of optimal flight paths. *Ars Journal*, 30(10):947–954.
- Lan, Y., Zhu, Y., Guo, J., Niu, S., and Cheng, X. (2014). Position-aware listmle: A sequential learning process for ranking.
- Mikolov, T., Chen, K., Corrado, G., and Dean, J. (2013a). Efficient estimation of word representations in vector space.
- Mikolov, T., Sutskever, I., Chen, K., Corrado, G., and Dean, J. (2013b). Distributed representations of words and phrases and their compositionality. *CoRR*, abs/1310.4546.
- Sanchez, J., Luque, F., and Lichtensztein, L. (2018). A structured listwise approach to learning to rank for image tagging. In *The European Conference on Computer Vision (ECCV) Workshops*.
- Simonyan, K. and Zisserman, A. (2014). Very deep convolutional networks for large-scale image recognition. *CoRR*, abs/1409.1556.
- Taylor, M., Guiver, J., Robertson, S., and Minka, T. (2008). Softrank: Optimising non-smooth rank metrics. In *WSDM 2008*.
- Vaswani, A., Shazeer, N., Parmar, N., Uszkoreit, J., Jones, L., Gomez, A. N., Kaiser, L., and Polosukhin, I. (2017). Attention is all you need. *CoRR*, abs/1706.03762.
- Wang, C., Tang, L., Bian, S., Zhang, D., Zhang, Z., and Wu, Y. (2019). Reference Product Search. *arXiv e-prints*, page arXiv:1904.05985.

Wu, Y., Schuster, M., Chen, Z., Le, Q. V., Norouzi, M., Macherey, W., Krikun, M., Cao, Y., Gao, Q., Macherey, K., Klingner, J., Shah, A., Johnson, M., Liu, X., Łukasz Kaiser, Gouws, S., Kato, Y., Kudo, T., Kazawa, H., Stevens, K., Kurian, G., Patil, N., Wang, W., Young, C., Smith, J., Riesa, J., Rudnick, A., Vinyals, O., Corrado, G., Hughes, M., and Dean, J. (2016). Google’s neural machine translation system: Bridging the gap between human and machine translation. *CoRR*, abs/1609.08144.