Modelos de redes neuronales para identificar entidades en documentos

David Redondo Laencina
Trabajo de fin de grado de Matemáticas
Universidad de Zaragoza

Directores del trabajo: Ricardo López Ruiz
y Vanessa Bueno Sancho
4 de septiembre de 2023

Resumen

Este trabajo surge a raíz de un periodo de prácticas realizado en la empresa NTT DATA. Concretamente en el departamento de *AI Assest Development Center* en el equipo de Dolffia. Dolffia es una plataforma de procesamiento de documentos basada en IA, que extrae y clasifica información de manera rápida y precisa. El objetivo del trabajo es crear una herramienta capaz de identificar las distintas entidades estructurales de un documento. Para ello tuve que aprender sobre NLP (*Natural Language Processing*) y descubrir los diferentes algoritmos usados en el *machine learning*.

El propósito del trabajo es conseguir un modelo capaz de resolver la tarea mencionada de manera eficaz. Para llegar a ese modelo primero hay que entender unos fundamentos teóricos y el problema que hemos de resolver.

En el primer capítulo del trabajo se hace una aproximación al mundo de la Inteligencia Artificial, más concretamente al aprendizaje automático. Diferenciando entre aprendizaje supervisado y no supervisado, se explican algoritmos como la regresión lineal, el gradiente descendiente o la detección de anomalías. A continuación se desarrolla en detalle la definición y conceptos básicos de una red neuronal. Finalmente se hace breve inciso en los modelos habitualmente usados en el NLP, es decir, en el procesamiento del lenguaje natural.

En el segundo capítulo se introduce el problema de identificación de entidades en documentos. Primero se explican sus bases y se comentan dificultades que nos van a surgir al resolverlo. Seguidamente se analizan métodos tradicionales para resolverlo, cómo son los métodos lingüísticos, los métodos basados en diccionarios y los métodos basados en aprendizaje automático. Por último, y de manera extendida, se explica cómo se puede resolver el problema usando redes neuronales convolucionales y la novedosa arquitectura conocida como Transformers.

En el tercer capítulo comienza la parte realmente práctica del trabajo. En primer lugar se explica la forma elegida de resolver el problema: vamos a hacer un *fine-tuning* de un modelo de la biblioteca *Transformers* [1] de *Hugging Face* [2]. El modelo elegido es LayoutLMv2. En la primera sección se explica en detalle y se compara con su predecesor. En la siguiente sección se hace una explicación exhaustiva de DocLayNet, el conjunto de datos elegido para hacer el *fine-tuning*, y cómo lo hemos preprocesado.

Finalmente, en el cuarto y último capítulo se explica cómo se ha realizado el entrenamiento del modelo LayoutLMv2 con el *dataset* DocLayNet, a continuación se analizan los resultados obtenidos. Para acabar se comentan brevemente posibles mejoras a realizar sobre nuestro modelo en un futuro.

[1] `https://github.com/huggingface/transformers`
[2] `https://huggingface.co`

Abstract

This dissertation arises from an internship period in NTT DATA corporation. Specifically at the *AI Assest Development Center* department in Dolffia's team. Dolffia is a document processing platform based on AI, which extracts and classifies information quickly and accurately. The goal was to create a tool that could identify the different structural entities of a document. In order to do that, I had to learn about NLP and find out the different algorithms used in machine learning.

The goal of this dissertation is to develop a model capable of solving the aforementioned task with accuracy. To get that model, it is necessary to understand some theoretical foundations and the problem to solve firstly.

In the first chapter of the work, we make an approach to the artificial intelligence field, more specifically to machine learning. Discerning between supervised and unsupervised learning algorithms such as linear regression, gradient descent or anomaly detection. Moreover, a more in-depth explanation of a neural network can be found, offering a deeper understanding of its workings and mechanisms. Finally, we will make a revision of models in the Natural Language Processing (NLP) field in order to focus on the main models able to solve the proposed task.

The second chapter discusses the problem that concerns us. First of all, we make an explanation and discuss the difficulties encountered in the process. Afterward, we set the traditional methods to solve it, such as linguistic methods, dictionary-based methods and machine learning-based methods. Lastly, and in an extended way, it is explained how the problem can be solved using convolutional neural networks and the new architecture known as Transformers.

The third chapter is where the truly practical part of the work begins. First, we explain the chosen way of solving the problem: we are going to "fine tune" a model from the Transformers [3] library of "Hugging Face" [4]. The chosen model is LayoutLMv2. In the first section it is explained in detail and compared with its predecessors. In the next section, there is an exhaustive explanation of DocLayNet, the data set chosen to do the fine-tuning, and how we have pre-processed it.

The fourth and last chapter explains how the training of the LayoutLMv2 model has been carried out with the DocLayNet dataset, then, the results obtained are analyzed. To conclude, we briefly discus the possible improvements to make about our model in the future.

[3] `https://github.com/huggingface/transformers`
[4] `https://huggingface.co`

Índice general

Capítulo 1

Fundamentos Teóricos

"La inteligencia artificial es un campo fascinante que me apasiona profundamente. La capacidad de desarrollar sistemas y algoritmos capaces de simular y emular el pensamiento humano me maravilla. Desde su capacidad para analizar grandes cantidades de datos y extraer información valiosa, hasta su habilidad para aprender y adaptarse, la inteligencia artificial ofrece un potencial infinito para transformar numerosos sectores de nuestra sociedad."

Este mensaje ha sido generado con ChatGPT (versión GPT 3.5) con el *input* "Necesito que me hagas una introducción de 5 líneas de extensión en la que hables de lo que te apasiona la inteligencia artificial". La Inteligencia Artificial (en adelante IA) es esto, la capacidad de dotar a las máquinas de características similares a las del pensamiento humano.

ChatGPT no es más que la interfaz de un modelo de generación de texto. La IA es mucho más, es un mundo en constante crecimiento que en los próximos años cambiará el paradigma de nuestra sociedad.

En las siguientes secciones se hace una toma de contacto con el mundo de la IA. Para ello se explican los algoritmos más habituales y hacemos un acercamiento al NLP.

1.1. Aprendizaje automático

El aprendizaje automático, más conocido como *machine learning*, es un área de la IA que tiene por objetivo conseguir que las máquinas aprendan. Decimos que una máquina aprende si al realizar en repetidas ocasiones una tarea mejora en su ejecución. Así pues una máquina que dispone de aprendizaje automático, a lo largo del tiempo, varía la forma de resolver el mismo problema para optimizar su resolución.

Dentro del *machine learning* se diferencian dos ramas: el aprendizaje supervisado y el no supervisado.

1.1.1. Aprendizaje supervisado

Los modelos de aprendizaje supervisado, conocidos por su nombre en inglés, *supervised learning models*, son entrenados con datos etiquetados, con el objetivo de que aprenda patrones en la relación entre unos datos y sus etiquetas. Así, una vez entrenado el modelo, cuando reciba unos datos no etiquetados podrá asignarles una etiqueta.

A continuación se explican una serie de algoritmos empleados en el aprendizaje supervisado.

- **Regresión lineal y gradiente descendente**

  El algoritmo se entrena con pares de variables $(x^{(i)}, y^{(i)})$, donde $x^{(i)}$ es el *input* e $y^{(i)}$ es el *output*. Dada la función $f_{w,b}(x^{(i)}) = wx^{(i)} + b = \hat{y}^{(i)}$ el objetivo es ajustar los valores w y b de tal forma que minimicen la función de coste. La función de coste más usada es $J(w,b) = \frac{1}{2m}\sum_{i=1}^{m}(f_{w,b}(x^{(i)}) - y^{(i)})^2$ donde m representa el número de muestras.

  Los valores óptimos de w y b se encuentran con algoritmos como el del gradiente descendente.

El algoritmo del gradiente descendente es tan sencillo como útil, consiste en repetir estos cuatro pasos hasta obtener la convergencia:

$$\begin{aligned} w_{new} &= w - \alpha \frac{\partial}{\partial w} J(w,b) \\ b_{new} &= b - \alpha \frac{\partial}{\partial b} J(w,b) \\ w &= w_{new} \\ b &= b_{new} \end{aligned}$$

El parámetro $\alpha \in (0,1)$ es conocido como *learning rate* y su elección es de vital importancia para el correcto funcionamiento del algoritmo. Si es muy pequeño puede hacer que el algoritmo sea muy lento, si es muy grande puede provocar sobre-ajuste y que el algoritmo nunca alcance un mínimo.

Una variante de este algoritmo es la regresión lineal múltiple. Se diferencia con el anterior en que ahora w y x son vectores así se tiene que $\hat{y}^{(i)} = f_{\vec{w},b}(\vec{x}^{(i)}) = \vec{w}\vec{x} + b$. También ahora se puede minimizar la función de coste con el algoritmo del gradiente descendente, aunque ahora hay que hacer un paso intermedio que consiste en escalar los diferentes valores de $\vec{x}$, una forma de hacerlo es con la normalización *Z-score* tomando $x_i = \frac{x_i - \mu_i}{\sigma_i}$.

- **Regresión logística**

  Es un algoritmo de clasificación binaria, es decir, solo puede haber dos posibles valores de salida, que identificamos como 0 o 1. Este procedimiento se basa en la siguiente función:

  $$f_{\vec{w},b}(\vec{x}) = g(\vec{w}\vec{x} + b) = \frac{1}{1 + e^{-(\vec{w}\vec{x}+b)}}$$

  a la función $g(z) = \frac{1}{1+e^{-z}}$ se la conoce como función logística, o *sigmoid function*, notar que $0 \leq g(z) \leq 1$. La clave de este algoritmo es la construcción de la frontera de decisión, para ello escogemos un valor α y tomamos la siguiente decisión:

  $$\begin{aligned} f_{\vec{w},b}(\vec{x}) \geq \alpha &\Rightarrow \hat{y} = 1 \\ f_{\vec{w},b}(\vec{x}) \leq \alpha &\Rightarrow \hat{y} = 0 \end{aligned}$$

  Para este caso se suele usar la siguiente función de coste:

  $$J(\vec{w},b) = -\frac{1}{m} \sum_{i=1}^{m} [y^{(i)} \log(f_{\vec{w},b}(\vec{x}^{(i)})) + (1 - y^{(i)}) \log(1 - f_{\vec{w},b}(\vec{x}^{(i)}))]$$

  Y se puede aplicar el algoritmo del gradiente descendiente de la misma forma que lo hemos aplicado en el apartado anterior. En este caso también existe una opción múltiple, llamada multi-clase, que tiene como objetivo clasificar en varios grupos en vez de solamente en dos.

Aparte de estos algoritmos hay muchos otros como los árboles de decisión, el algoritmo SVM, los *random forests* o las redes neuronales que se explican más detalladamente en la siguiente sección.

1.1.2. Aprendizaje no supervisado

Más conocido por su nombre en inglés *unsupervised machine learning*, se diferencia con el aprendizaje supervisado en que aquí no sabemos nada sobre el conjunto de datos de salida, este se centra en la búsqueda de patrones, relaciones y estructuras ocultas en los datos para así encontrar agrupamientos naturales.

A continuación se explican algunos de los algoritmos de aprendizaje no supervisado más usados.

- **Algoritmo *K-means***
  Este algoritmo tiene como objetivo agrupar datos en K grupos, para ello se siguen los siguientes pasos:

  1. Se crean K centros aleatorios.
  2. Se asigna a cada dato su centro más cercano.
  3. Para cada grupo se hace la media de todos los puntos, y se toma dicha media como nuevo centro del grupo.

  Se repiten los pasos 2 y 3 hasta que al asignar cada dato a su centro más cercano ninguno cambie de grupo, entonces diremos que el algoritmo ha alcanzado la convergencia. Nótese que la solución final es altamente dependiente de la elección inicial de los centros.

- ***Anomaly detection algorithm***
  El objetivo de este algoritmo es encontrar datos anómalos. A continuación se explica mediante un caso de uso. Queremos saber si el motor de un coche funciona correctamente, para ello se inspeccionan N motores y de cada uno se recogen los siguientes datos:

  $$x_1 = \text{vibración}, \quad x_2 = \text{ruido}, \quad x_3 = \text{temperatura}$$

  Así se obtienen los siguientes $3 * N$ datos $x_1^1, x_2^1, x_3^1, ..., x_1^N, x_2^N, x_3^N$. Ahora se calcula la media y la desviación típica para cada tipo de dato:

  $$\mu_j = \frac{1}{N}\sum_{i=1}^{N} x_j^i \qquad \sigma_j^2 = \frac{1}{N}\sum_{i=1}^{N}(x_j^i - \mu_j)^2 \qquad j = 1,2,3$$

  Con estos valores se define la función de probabilidad:

  $$p(x^i) = \prod_{j=1}^{3} p(x_j^i, \mu_j, \sigma_j^2) = \prod_{j=1}^{3} \frac{1}{\sqrt{2\pi}\sigma_j} exp(-\frac{(x_j^i - \mu_j)^2}{2\sigma_j^2})$$

  Entonces diremos que el motor i es defectuoso si $p(x^i) < \varepsilon$. Ahora bien, ¿cómo se elige el valor ε? Se puede elegir una vez vistas las distintas probabilidades, pero en la práctica se suele elegir tras ver el resultado de unos datos anteriores. En este caso se podrían analizar los valores de unos motores que funcionan correctamente para calcular ε.

1.2. Redes neuronales

Como ya se ha dicho las redes neuronales, también conocidas como *artificial neural networks* o simplemente *neural networks*, son un conjunto de algoritmos de aprendizaje automático supervisado, aunque también hay variantes pertenecientes al aprendizaje no supervisado.

Este algoritmo surgió del intento de imitar el comportamiento del cerebro humano, de ahí su nombre. Ambos tienen varias cosas en común como son una estructuración similar o una gran plasticidad, es decir, la capacidad para adaptarse a distintas situaciones.

Las unidades básicas de las redes neuronales son las neuronas, estas se distribuyen en distintas capas o *layers*, que pueden ser capas de entrada (*input layers*) o capas ocultas (*hidden layers*), que reciben la información de una neurona y la envían a otra, o capas de salida (*output layers*).

El proceso que ocurre en cada neurona es el siguiente, recibe información (*input*), la procesa con la función de activación, y la envía a otra neurona (*output*). La función de activación no es más que una función matemática como la función de la regresión lineal o la *sigmoid function* que hemos visto anteriormente.

A continuación se analiza en profundidad cómo es la estructura de una red neuronal y cómo esta procesa la información.

- **Notación**

  - Se supone una red neuronal con *n layers* $l_1, l_2, ..., l_n$.
  - Se supone una red con solo un *layer* de entrada, que será el l_1, y otro de salida, que será el l_n.
  - Se supone también que el número de neuronas de cada *layer* es desconocido y que no tiene por qué ser igual en los distintos *layers*, se denota como m_i la cantidad de neuronas del *layer* l_i.
  - Se denota como $\vec{x}$ al vector *input* inicial, esta es toda la información que tenemos.
  - Se llama $a^{[i-1]}$ al vector *input* del *layer* l_i. Notar que $a^{[i]}$ es el vector *output* del *layer* l_i y que podemos tomar $a^{[0]} = \vec{x}$.
  - Se denota como g a la función de activación de cada neurona. Notar que cada función de activación puede tener unos pesos diferentes.

Una vez introducida esta notación se puede ver lo que sucede en cada *layer*. Supongamos que estamos en el *layer* l_4 de una red neuronal de 10 capas, y que $m_4 = 3$, es decir l_4 tiene tres neuronas. Luego se tiene:

$$a_1^{[4]} = g(\vec{w}_1^{[4]}\vec{a}^{[3]} + b_1^{[4]})$$
$$a_2^{[4]} = g(\vec{w}_2^{[4]}\vec{a}^{[3]} + b_2^{[4]})$$
$$a_3^{[4]} = g(\vec{w}_3^{[4]}\vec{a}^{[3]} + b_3^{[4]})$$

Así el *output* de l_4 es $\vec{a}^{[4]} = (a_1^{[4]}, a_2^{[4]}, a_3^{[4]})$

A continuación se hace una breve introducción de 3 de las clases de redes neuronales más utilizadas.

- **Tipos de redes neuronales**

  - ***Feedforward Neural Networks (FNN)*:** Conocidas en español como redes neuronales de propagación hacia adelante son las redes neuronales más básicas, en ellas la información fluye desde la capa de entrada a la capa de salida en una sola dirección. Entre sus aplicaciones destaca el reconocimiento de patrones.
  - ***Recurrent Neural Networks (RNN)*:** El punto clave de las redes neuronales recurrentes es que la información puede fluir en bucles. Cada neurona tiene la capacidad de recordar el estado de la información en la neurona anterior. La importancia de estas redes se presenta en ámbitos como el modelado de secuencias biológicas.
  - ***Convolutional Neural Networks (CNN)*:** Llamadas redes neuronales convolucionales en español, se caracterizan por el uso de capas de convolución, aquí distintas neuronas procesan la misma información con el objetivo de seleccionar las características más importantes. Este es un concepto complicado que se explicará en profundidad más adelante. Su uso es amplísimo y va desde el reconocimiento facial hasta la conducción autónoma.

1.3. Modelos de Lenguaje en NLP

El procesamiento de lenguaje natural, o NLP por sus siglas en inglés, es la disciplina de la IA que estudia la relación de las computadoras con el lenguaje humano.

Se divide en tres grandes ramas:

- Comprensión del lenguaje: Para, por ejemplo, extraer información de documentos y analizar opiniones o sentimientos.
- Generación del lenguaje: Donde se destaca la traducción automática o el resumen de texto.
- Interacción humano-máquina: Se centra en proporcionar a las máquinas la capacidad de dialogar con los humanos.

Los modelos de lenguaje tratan de resolver tareas como la predicción de palabras o la generación o comprensión de texto, para ello son entrenados en grandes cantidades de documentos con el objetivo de aprender patrones y reglas lingüísticas.

Dentro de estos modelos podemos destacar:

- **Modelos basados en n-gramas:** Son los modelos más básicos. Se basan en la idea de que la probabilidad de que aparezca una palabra solo depende de las palabras anteriores. Un n-grama no es más que una secuencia de elementos de un texto, por ejemplo si tengo el texto "estudio matemáticas en Zaragoza" un 1-grama sería ["estudio","matemáticas","en","Zaragoza"], un 2-grama sería ["estudio matemáticas", "matemáticas en", "en Zaragoza"] , un 3-grama sería ["estudio matemáticas en", "matemáticas en Zaragoza"], finalmente un 4-grama sería el propio texto. Notar que cuanto más grande sea el n más contexto se tiene en cuenta para calcular la probabilidad.
- **Modelos basados en modelos ocultos de Markov (HMM):** Se basan en la idea de que hay una cadena de estados ocultos, estas cadenas son, por ejemplo, los fonemas de una palabra. Estos modelos constan de tres partes, la primera es el conjunto de estados ocultos, la segunda el conjunto observado, y finalmente las matrices de transición y emisión. El objetivo es asignar unas probabilidades a estas matrices a partir de unos datos de entrenamiento.
- **Modelos basados en redes neuronales:** Estos modelos son los más usados en la actualidad, ya que son los que dan mejores resultados. Utilizan los distintos tipos de redes neuronales comentados en la sección anterior. Se explican con más detalle en el siguiente capítulo.

En resumen este capítulo sirve de introducción al *machine learning*. Primero se explica qué es y algunos de sus algoritmos más importantes, se sigue con una explicación más detallada de lo que son las redes neuronales y finalmente se hace una aproximación al mundo del procesamiento de lenguaje natural.

Capítulo 2

Identificación de entidades en documentos

Una nómina, el ticket del supermercado, *papers* científicos, el recibo de la luz... Cada día se generan una infinidad de documentos, extraer datos de ellos es uno de los grandes retos de la actualidad. La información que se puede extraer de un documento es mucha, pero en algunas ocasiones no toda nos es útil. Para poder comprobar la utilidad de un dato y clasificarlo el primer paso es identificarlo, es decir asignarle una categoría.

2.1. Definición del problema

Dentro del *natural language processing* una de las tareas más importantes es la identificación o extracción de entidades. El objetivo de este problema es reconocer entidades en textos para poder estructurarlo y obtener la información más relevante. Las diferentes entidades se pueden elegir en base a diferentes criterios:

- Criterios semánticos y gramaticales: Donde se identifican entidades como personas, ubicaciones, empresas, cantidades o fechas.
- Criterios estructurales y de localización: Las distintas entidades a identificar son, por ejemplo, tablas, imágenes, pie de página, títulos...

Las diferentes formas de afrontar este problema coinciden en sus dos primeros pasos. El primero consiste en el preprocesado del texto con el fin de normalizarlo, este paso engloba la eliminación de caracteres especiales, eliminación de palabras y la *tokenización* o subdivisión del texto en unidades más pequeñas. El segundo paso es la extracción de información característica del texto, ya sean características léxicas, sintácticas, morfológicas o contextuales.

Al realizar los dos etapas arriba mencionadas surgen los primeros problemas:

- El primero surge a la hora de *tokenizar*. Como ya se ha explicado *tokenizar* consiste en subdividir el texto en partes más pequeñas. La forma más sencilla es separar por palabras, es decir cada palabra es un *token*. Aún siendo la forma más habitual tiene problemas en algunos idiomas. Otra forma de *tokenización* es la que está basada en el diccionario, esta elige un diccionario y *tokeniza* únicamente las palabras que aparecen en él. Se explican otras formas, más detalladamente, en los siguientes apartados.
- Otro problema surge a la hora de intentar reducir la cantidad de *tokens*. Hay dos métodos principales: el *stemming* que consiste en truncar los sufijos o prefijos que pueda tener una palabra, así por ejemplo, las palabras “tormenta”, “tormentas” y “atormentar”, se recogerían bajo el *token* “tormenta”. El otro método se conoce como lematización, se basa en asignar como *token* para una palabra derivada, su palabra base. Podemos ver la diferencia de estos métodos con la palabra “amiguito”, con el primero le asignaríamos el *token* “amig”, con el segundo le asignaríamos el *token* “amigo”.

2.2. Enfoques tradicionales

El problema de identificación de entidades es anterior a los avances que se han producido en la inteligencia artificial en estos últimos años. En esta sección se ve cómo se resolvía el problema antes del *boom* de la IA.

A continuación se introducen tres formas de resolver el problema, la primera basada únicamente en técnicas lingüísticas, la siguiente basada en diccionarios y palabras clave, y finalmente una basada en aprendizaje supervisado.

- **Métodos lingüísticos:** Fueron creados por lingüistas en los años 60 y 70, son los más simples, en la actualidad se usan para acompañar y mejorar otros métodos. Se basan únicamente en el análisis gramatical, para ello se crean distintas reglas que sirven para predecir la categoría de cada *token*. Un ejemplo muy simple sería pensar que si tenemos un artículo y seguidamente un sustantivo la siguiente palabra probablemente sea un verbo o un adjetivo.

  Las carencias de estos métodos son evidentes ya que el número de reglas que podemos crear es limitado y hay muchas situaciones ambiguas o que directamente no se pueden contemplar. Otro gran inconveniente es que en muchas ocasiones estas reglas no se pueden extrapolar a otros idiomas.

- **Métodos basados en diccionarios y palabras clave:** Como ya se ha introducido otra posibilidad es hacer la identificación basándonos en un diccionario o bolsa de palabras. Consiste en hacer listas de palabras que sabemos que pertenecen a cada categoría. Una vez tenemos nuestro diccionario se asigna una categoría a cada *token* si coincide total o parcialmente con un elemento de la lista. Para estos métodos es muy común usar el *stemming* o lematización, explicadas en la sección anterior.

  Estos métodos son útiles cuando la información que se quiere identificar pertenece a un ámbito muy específico, así es idóneo para identificar nombres propios, abreviaturas, siglas... Los puntos negativos son muchos más:

  - Aunque es muy fácil de implementar es muy difícil de escalar y adaptar a un campo más amplio.
  - Necesita un continuo mantenimiento y actualización de los diccionarios.
  - No tiene en cuenta el contexto, luego es muy ineficiente para palabras con varios significados. Por ejemplo no sabríamos clasificar la palabra “vino” como un sustantivo o como un verbo (conjugación del verbo venir). Las ambigüedades que pueden surgir con este método son muchas.

- **Métodos basados en aprendizaje automático:** Utilizan características lingüísticas y contextuales para asignar categorías a los diferentes *tokens*. Cabe destacar dos algoritmos que son los modelos ocultos de Markov y el etiquetado basado en secuencias. Ya sabemos algo de ellos, ahora se van a ver en profundidad.

  - **Modelos ocultos de Markov:** Son conocidos como HMM por sus siglas en inglés. Para poder entender este tipo de modelos primero debemos saber qué es una cadena de Markov.

    Una cadena de Markov es un modelo matemático encargado de describir una secuencia de eventos, en los que la probabilidad de que ocurra un evento depende únicamente del último evento ocurrido. Formalmente definimos $q_1, ..., q_N$ como los N eventos o estados que pueden suceder (en nuestro caso serán las diferentes categorías), definimos como A la matriz de probabilidades donde cada elemento $a_{i,j}$ representa la probabilidad de pasar del estado q_i al estado q_j. Con esta notación tenemos que

    $$P(q_i|q_0...q_{i-1}) = P(q_i|q_{i-1}) \tag{2.1}$$

    ya que como hemos dicho anteriormente la probabilidad de estar en un estado depende únicamente del estado inmediatamente anterior. Habitualmente se añaden dos estados artificiales

q_0, q_F al principio y al final de nuestra cadena. Así por ejemplo en nuestro caso la probabilidad de tener la secuencia (artículo, nombre) sería:

$$P(\text{artículo, nombre}) = P(\text{artículo} \mid q_0) \cdot P(\text{nombre} \mid \text{artículo}) \cdot P(q_F \mid \text{nombre})$$

Donde $P(\text{artículo} \mid q_0)$ representa la probabilidad de que una frase empiece por una palabra perteneciente a la categoría artículo, y $P(q_F \mid \text{nombre})$ representa la probabilidad de que una frase acabe con una palabra perteneciente a la categoría nombre.

Los modelos ocultos de Markov [1] tratan tanto eventos observables como eventos ocultos que se presuponen consecuencia de los observables. Añadimos a la notación anterior $o_1, ..., o_M$ que son las M observaciones realizadas, y la matriz B conocida como matriz de probabilidades de emisión, donde cada elemento $b_i(o_j)$ es la probabilidad de que la observación j sea generada por el estado i. Con esto y con 2.1 tenemos que

$$P(o_i|q_1, ..., q_N, o_1, ..., o_M) = P(o_i|q_i) \tag{2.2}$$

Esta igualdad representa que la probabilidad de una observación depende únicamente de la categoría del *token* donde se ha producido. En nuestro caso las observaciones pueden ser palabras, caracteres o elementos gramaticales.

Se plantean las siguientes cuestiones:

1. Determinar la probabilidad de que se dé una serie de observaciones:

   Como cada observación se corresponde únicamente con una categoría, y gracias a 2.2 deducimos que la probabilidad de una secuencia de observaciones $O = o_1, ..., o_T$ asociada a una secuencia de categorías $Q = q_1, ..., q_T$ es

   $$P(O|Q) = \prod_{i=1}^{T} P(o_i|q_i)$$

   Como no sabemos cual es la secuencia de categorías tenemos:

   $$P(O, Q) = P(O|Q)P(Q) = \prod_{i=1}^{T} P(o_i|q_i) \prod_{i=1}^{T} P(q_i|q_{i-1}) \tag{2.3}$$

   y de aquí se deduce que

   $$P(O) = \sum_{Q} P(O, Q) = \sum_{Q} P(O|Q)P(Q) \tag{2.4}$$

2. Dada una secuencia de observaciones, determinar cual es la secuencia de categorías que tiene la mayor probabilidad asociada, es decir, queremos encontrar la secuencia de categorías que mejor explique una secuencia de observaciones:

   Para este proceso se suele usar el algoritmo de Viterbi. Este algoritmo consiste en asignar valores a los distintos $v_t(j)$, que representan la probabilidad de que tras t observaciones nos encontremos con un *token* perteneciente a la categoría j, es decir:

   $$v_t(j) = \max_{q_0, ..., q_{t-1}} P(o_1, ..., o_t, q_1, ..., q_{t-1}, q_t = j)$$

   Notar que es un algoritmo recursivo ya que, con la notación anterior, también podemos escribir $v_t(j)$ como:

   $$v_1(i) = b_i(o_1)$$

   $$v_t(j) = \left(\max_{1 \leq i \leq N} v_{t-1}(i)(a_{ij}) \right) b_j(o_t) \qquad t = 2, ..., N$$

Supongamos que tenemos M observaciones, entonces denotamos como s_M la categoría más probable para la última observación, es decir la categoría que asignaremos al último *token*

$$s_M = \max_{1 \leq i \leq N} \nu_M(i)$$

Para ver que categorías les asignamos al resto de *tokens* hacemos una recursión inversa, para ello definimos

$$\phi_t(j) = \max_{1 \leq i \leq N} \nu_{t-1}(i) a_{ij} \qquad t = 2, ..., M$$

y finalmente tenemos que

$$s_{t-1} = \phi_t(s_t) \qquad t = M, .., 2$$

3. Calcular los parámetros del modelo, es decir, las matrices de probabilidades A y B:

   Para calcular estas probabilidades se necesitan unos datos de entrenamiento, es decir, *tokens* o palabras que ya estén categorizadas. Para mayor claridad en la explicación supongamos que cada observación es una palabra.
   Los datos de entrenamiento tienen que ser frases en las que sepamos a que categoría pertenece cada palabra. Denotamos como $C(i, j)$ la cantidad de veces que aparece la categoría j después de la i, y como $C(i)$ la cantidad de veces que aparece la categoría i en los datos de entrenamiento. Con esto podremos rellenar las matrices A y B con los siguientes valores:

$$a_{i,j} = P(j|i) = \frac{C(i, j)}{C(i)} \qquad b_j(o_t) = P(o_t|j) = \frac{C(j, o_t)}{C(j)}$$

El gran problema de este algoritmo es que es imposible entrenarlo con todas las palabras que va a tener que etiquetar luego, ya que siempre va a haber vocabulario nuevo, faltas de ortografía o erratas. Esto conlleva que al llegar a una palabra o_t que no tiene una categoría asignada, tengamos que $b_s(o_t) = 0$ lo que implica que $\nu_t(s) = 0$, y debido a su carácter recursivo se tiene que $\nu_t(i) = 0 \; \forall i \geq s$.

Se plantean dos soluciones para este problema:

- **Alisado de Laplace:** Consiste en redefinir $b_j(o_t)$ como:

$$b_j(o_t) = P(o_t|j) = \frac{C(j, o_t) + \alpha}{C(j) + \alpha |V|}$$

  Donde α es un número (próximo a 1) conocido como parámetro de ajuste de alisado, y $|V|$ es el tamaño del vocabulario con el que hemos entrenado el algoritmo. Así si no hemos tenido ninguna observación igual en el entrenamiento tendremos que $C(j, o_t) = 0$ pero $b_j(o_t) = \dfrac{\alpha}{C(j) + \alpha |V|} > 0$ y podremos seguir con nuestro algoritmo con normalidad.

- **Tag más frecuente:** Esta solución es simple, si s es la categoría más frecuente asignamos $b_s(o_t) = 1$ y $b_i(o_t) = 0 \; \forall i \neq s$. Esta aún siendo una solución muy simple resulta muy útil en la práctica.

- **Campo aleatorio condicional:** Es conocido como CRF por sus siglas en inglés o como campo aleatorio de Markov, y es que se fundamenta en los mismos principios que el método anterior.

  La principal diferencia es que en este caso la categoría de un *token* puede depender de las categorías de los *tokens* próximos a él y de una o más observaciones a la vez, en vez de depender únicamente de una observación. Este hecho hace que en el proceso de entrenamiento haya que asignar un peso a cada categoría y observación.

2.3. Modelos de redes neuronales

En los últimos tiempos el avance de la tecnología y la continua investigación en IA y *machine learning*, ha hecho que el progreso en esta rama sea incalculable. Muchos algoritmos, métodos o modelos han sido mejorados o superados por otros que usan herramientas de IA. Un ejemplo son los métodos para resolver el problema de identificar entidades, aunque hemos visto que hay métodos mucho más simples y que dan resultados válidos, la diferencia es abrumadora, y es que como veremos más adelante se consigue llegar a una predicción casi perfecta.

2.3.1. Redes neuronales convolucionales

Anteriormente, en la sección 1.2 se ha hecho un acercamiento a las redes neuronales convolucionales, ahora se explican en detalle y se ve cómo se usan para identificar entidades en documentos.

Las redes neuronales convolucionales son muy eficientes para identificar entidades debido principalmente a tres motivos:

- Son muy buenas para capturar patrones locales lo que hace que su rendimiento en datos estructurados, como las imágenes, sea excelente.
- Son capaces de aprender la jerarquía que existe entre las categorías, ya que las primeras capas procesan características de bajo nivel y las capas más profundas características más complejas y abstractas.
- Reducen la dimensionalidad de los *inputs*, lo que hace que sea un proceso menos pesado computacionalmente y reduce la probabilidad de que se produzca sobreajuste.

Estas ventajas están producidas por la propia estructura de la red, que explicamos a continuación y que podemos observar en 2.3.1. Para mayor simplicidad en esta explicación suponemos que disponemos de la misma información que en los enfoques tradicionales 2.2, es decir, solo disponemos información sobre el texto y no sobre su situación en el documento. En el siguiente capítulo veremos que en nuestro modelo esto no es así, ya que sabremos las coordenadas de las distintas palabras.

1. **Representación de las palabras:**

   El primer paso es ver cuál es el *input* que va a recibir nuestra capa de entrada, es decir, cómo vamos a representar las palabras. En este proceso nos centramos en resolver los problemas producidos por las palabras polisémicas y las desconocidas. Destacamos dos métodos:

   - **Word2Vec:** Pertenece a los algoritmos conocidos como *word embeddings* que son los que asignan a cada palabra un vector de números reales, de esta forma palabras con significados similares o que suelen estar rodeadas de las mismas palabras se encuentran en regiones cercanas del espacio vectorial.

     Word2Vec fue publicado por un equipo de *Google* en 2013 [2][3], es una red neuronal con una sola capa oculta y que está completamente conectada. Dicho vector puede ser producido usando dos arquitecturas diferentes:

     - *Continuous bag-of-words*: Se pregunta qué palabra falta en un contexto de palabras dado. Destaca por su rapidez.
     - Arquitectura *Skip-Gram*: Se pregunta, dada una palabra, cuales son las palabras que más probablemente le acompañen. Es mejor para palabras poco frecuentes.

     En este método se tratan las palabras como vectores, luego se puede calcular la similaridad semántica entre dos palabras w_1 y w_2 como:

$$S(w_1, w_2) = \frac{w_1 w_2}{\|w_1\|\|w_2\|}$$

y la distancia semántica entre dos palabras como:

$$d(w_1, w_2) = 1 - S(w_1, w_2) = 1 - \frac{w_1 w_2}{\|w_1\|\|w_2\|}$$

También podemos hacer operaciones con palabras, un ejemplo sería “Reina”=“Rey”-“Hombre”+“Mujer”.

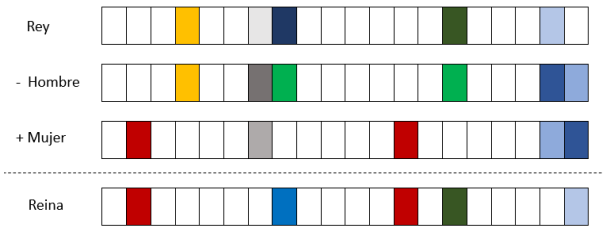


Figura 2.1: Representación de operaciones con vectores.

- **WordPiece:** Es un proceso de *tokenización* que en vez de considerar cada palabra como un *token* único la divide en diferentes subpalabras, conocidas como *subwords*, que pueden ser compartidas por diferentes palabras. Así se puede dar el caso de que el modelo no conozca lo que significa una palabra pero lo pueda inferir del significado de sus *subwords*.

  Por ejemplo la palabra “mesa” *tokenizada* con este método tendría como *subwords* “m”, “##e”, “##s”, “##a”. Notar que “##” representa que ese *token* va precedido por otro.

  Veamos cómo funciona esta forma de *tokenización* con un ejemplo basado en [4][5]:

  Primero definimos la puntuación como

$$S = \frac{\text{frecuencia del par}}{\text{frecuencia primer elemento} * \text{frecuencia segundo elemento}}$$

  El algoritmo se basará en esta puntuación para ver si dos *tokens* han de unirse. Notar que esta fórmula prioriza los *tokens* menos habituales, así mismo será difícil que se una “des” con “##ocupado” ya que son dos *subwords* bastante frecuentes, pero será bastante probable que se una “di” con “##fícil”.

  Supongamos que tenemos el siguiente vocabulario, donde el número representa la cantidad de veces que aparece la palabra

  (“sal”, 10), (“mal”, 5), (“mar”, 12), (“dar”, 4), (“sales”, 5)

  Lo podemos dividir en *tokens* y tenemos:

  [(“s”, 15) (“m”, 17) (“d”, 4) (“##a”, 36) (“##l”, 20) (“##r”, 16) (“##e”, 5) (“##s”, 5)]

  Vemos que la pareja más frecuente es (“##a”, “##l”) pero su puntuación es $\frac{20}{36*20} = \frac{1}{36}$, sin embargo el par con puntuación máxima es (“##e”, “##s”) con $S = \frac{5}{5*5} = \frac{1}{5}$ luego estos *tokens* serán los primeros que uniremos. Ahora tenemos los siguientes *tokens*:

  [(“s”, 15) (“m”, 17) (“d”, 4) (“##a”, 36) (“##l”, 20) (“##r”, 16) (“##es”, 5)]

  El siguiente par con mayor puntuación es (“##l”, “##es”) con $S = \frac{5}{20*5} = \frac{1}{20}$, y obtenemos:

  [(“s”, 15) (“m”, 17) (“d”, 4) (“##a”, 36) (“##l”, 15) (“##r”, 16) (“##les”, 5)]

  Ahora “##a” aparece en todos los posibles pares luego todos los pares tienen la misma puntuación. Cogemos por ejemplo el par (“s”, “##a”) y obtenemos los siguientes *tokens*:

  [(“sa”, 15) (“m”, 17) (“d”, 4) (“##a”, 21) (“##l”, 15) (“##r”, 16) (“##les”, 5)]

Continuamos con este proceso hasta obtener el número de *subwords* deseado o hasta que todas las palabras iniciales sean *subwords*.

Con nuestra última lista de *tokens* la palabra "sala" será ("sa","##l","##a"). Si la palabra a descomponer contiene *subwords* que no podemos encontrar en nuestro vocabulario, ese *token* se representa como "[UNK]". Así *tokenizariamos* "sables" como ("sa","[UNK]","##les").

De la explicación dada se puede deducir que el primer método es mejor para analizar relaciones semánticas y sintácticas entre las palabras, y el segundo es más adecuado para abordar el manejo de palabras desconocidas.

2. **Capa de convolución:**

   La información llega a la capa de convolución en forma de tensor, que no es más que una manera de guardar datos numéricos de forma estructurada. Un tensor de una dimensión lo podemos identificar como un vector y uno de dos dimensiones como una matriz. Sobre esta información se aplica la convolución, que es un operador que transforma dos elementos en un tercero que representa cómo se superponen los dos primeros.

   Cada capa de convolución tiene distintas neuronas, cada una corresponde a lo que conocemos como filtro o *kernel*, que es una matriz de Toeplitz, es decir, una matriz cuadrada en las que las diagonales paralelas a la principal son constantes, además esta matriz ha de ser de dimensión menor o igual a la del tensor. El filtro se va aplicando al tensor haciendo el producto interno de Frobenius con las distintos subtensores de igual dimensión.

   Pongamos un ejemplo, suponemos que nuestro tensor es de dimensión dos, es decir, una matriz, de por ejemplo cuatro filas y cuatro columnas. Suponemos que el filtro es una matriz 3x3, esto producirá una matriz salida 2x2, como vemos en la siguiente imagen:

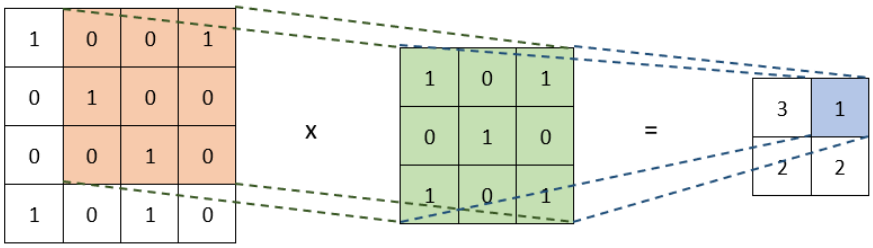


Figura 2.2: Representación de una convolución.

La matriz con submatriz naranja es el tensor de entrada, la matriz verde es el filtro y a la matriz con el recuadro azul se le llama característica convolucionada.

Cada filtro tiene un peso asociado, que se van ajustando con el entrenamiento de la red neuronal. El proceso de ajuste se realiza con algoritmos como los del capítulo anterior. Destacamos el uso del descenso de gradiente ya que su uso es generalizado en esta situación.

3. **Capa ReLU:**

   Para las redes neuronales convolucionales la función de activación elegida suele ser la función ReLU. Esta no es más que la función $f(x) = \text{máx}(0,x)$ así se consigue introducir la no linealidad en la función de decisión.

   En algunas ocasiones también se usan como funciones de activación $f(x) = |\tanh(x)|$ o la función sigmoide $f(x) = \dfrac{1}{1+e^{-x}}$. Se suele preferir ReLU porque es mucho más rápida y la diferencia de precisión no es significativa.

4. **Capa de agrupación o *pooling*:**

   Es una capa sin parámetros entrenables encargada de reducir la dimensionalidad de las características convolucionadas. Captura información relevante sin tener en cuenta su ubicación exacta, para ello utiliza funciones como el promedio o el máximo. En nuestro ejemplo anterior devolvería 2 en el primer caso y 3 en el segundo.

   Es común intercalar capas de agrupación entre las distintas capas de convolución, así se reduce la cantidad de parámetros, el costo computacional y la probabilidad de sobre-ajuste.

5. **Capa totalmente conectada:**

   Es la capa final donde se va a realizar la clasificación. En ella todas las neuronas están conectadas con todas las características convolucionadas. Finalmente devuelve un vector al que se le aplica la función *softmax* que manda cada valor z_j del vector a $\sigma(z_j) = \frac{e^{z_j}}{\sum_{i=1}^{K} e^{z_i}}$, como podemos apreciar la suma de los K valores del vector será 1, representando la probabilidad de pertenecer a cada categoría.

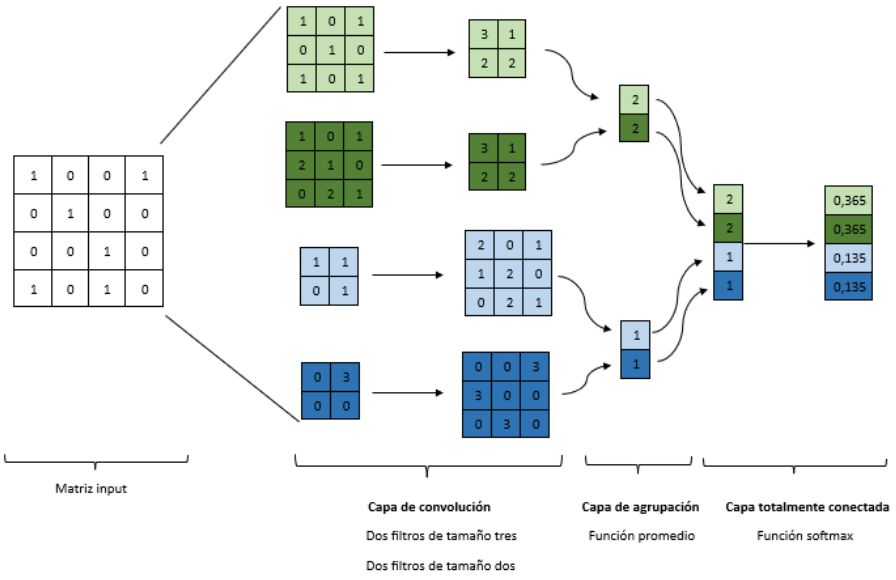


Figura 2.3: Representación de una estructura de una red neuronal convolucional

2.3.2. *Transformers*

Transformers es una arquitectura de red neuronal propuesta en 2017 [6], desde entonces su impacto en el NLP ha sido abrumador. Todos los grandes modelos usados ahora se basan en ellos, desde BERT (*Bidirectional Encoder Representations from Transformers*) de Google [7], pasando por el archiconocido GPT (*Generative Pre-trained Transformer*) de OpenAI [8], hasta el que se ha entrenado para este trabajo, LayoutLM (*Layout Language Model*) de Microsoft [9].

Hasta ahora las redes neuronales vistas se basan en operaciones secuenciales, en cambio los *Transformers* se basan en una estructura de atención. Ahora cada palabra interactúa con todas las palabras de la secuencia, esto permite que el modelo aprenda relaciones de largo alcance y que aprecie dependencia entre elementos distantes.

Para una primera toma de contacto saber que un mecanismo de atención es un algoritmo basado en la idea de asignar pesos a cada elemento de la secuencia de entrada, con el fin de determinar la importancia de cada elemento para las siguientes procesos.

A continuación se explican las distintas partes de la estructura de un Transformer que podemos ver en la figura 2.3.2 :

1. ***Input embedding*:**

   Es el primer proceso, aquí se tokeninzan los *inputs* y se representan en forma vectorial. Para ello se usan métodos como el Word2Vec (1) o WordPiece (1) comentados anteriormente.

2. ***Positional embedding*:**

   Para conocer el significado de una oración es importante saber donde se encuentra cada palabra. Este es el objetivo de este proceso. Se dota a cada vector de un valor añadido que indica la posición de ese *token* en la frase, esto implica que un mismo *token* puede tener representaciones diferentes.

   Esto permite enviar al modelo todos los *tokens* al mismo tiempo, lo que implica un ahorro considerable de tiempo.

3. ***Encoder*:**

   En un transformer hay varios *encoders*, a su vez cada uno cuenta con varios subprocesos que se comentan a continuación:

   - ***Self-attention*:** En este momento tenemos una frase *tokenizada*, en la que cada *token* esta representado por un vector que identifica su posición en dicha frase. El objetivo es saber cómo se relacionan los distintos *tokens* entre sí.

     Para solucionar este problema se introducen los *soft-dictionary* o diccionarios blandos. A diferencia de los diccionarios normales compuestos por pares clave-valor, los *soft-dictionary* estarán compuestos por tres matrices:

     - Matriz de consulta (Q), formada por los vectores de los *tokens* que estamos evaluando.
     - Matriz de claves (K), tendremos los vectores representando las claves del diccionario.
     - Matriz de valores (V), tendremos los vectores correspondientes a los *tokens* de salida.

     La atención vendrá dada por

     $$Z = softmax\left(\frac{QK^T}{\sqrt{d_k}}\right)V$$

     donde $softmax$ es la función comentada en el subsección anterior, y d_k es la dimensión de los vectores, que usamos como parámetro de escala. Cuando dos palabras sean similares tendremos un valor de Z cercano a 1.
   - ***Multi-Head attention*:** Es el proceso que realmente se produce en el *encoder*, consiste en dividir los vectores que representan a los *tokens* en secciones, y aplicar a las distintas secciones la *self-attention* comentada anteriormente. En [6] los vectores tienen 512 valores y se separan en 8 secciones de 64, luego hay 8 cabezas de atención.
   - ***Add & Norm*:** Agrupa varios subprocesos como son las *skip-connections* que son "saltos" entre las distintas capas para así poder guardar información útil, o la normalización de los valores con el objetivo de que no cambien bruscamente en los siguientes procesos.
   - ***Feed Forward*:** Es una red completamente conectada que se encarga de transformar y combinar las representaciones creadas en la capa de atención, para capturar y modelar relaciones más complejas.

4. ***Decoder*:**

   Tiene una estructura similar a el *encoder*. Su "*input*" es el *ouput* del modelo y es que en los *Transformers* se usa la salida como parte del entrenamiento y no solo para ajustar pesos como en otros modelos.

   La gran diferencia con el *encoder* es que no se usa la *self-attention* como capa de atención, aquí usamos:

- ***Cross-attention*:** Utilizamos los valores producidos en el *encoder* que, junto con la matriz V del *output*, nos permiten modelar la atención buscada.
- ***Masked-attention*:** Cuando nosotros generamos el *output* se genera palabra a palabra, luego a priori no podemos saber cómo se va a relacionar la primera palabra con la última. Representamos este hecho "borrando" los datos superiores a la diagonal principal de la matriz QK^T.

5. ***Output*:** El último proceso es una capa lineal para hacer la proyección y transformación final de los datos obtenidos, y finalmente aplicar *softmax* para que la suma de todos los valores sea 1 y obtener así una probabilidad.

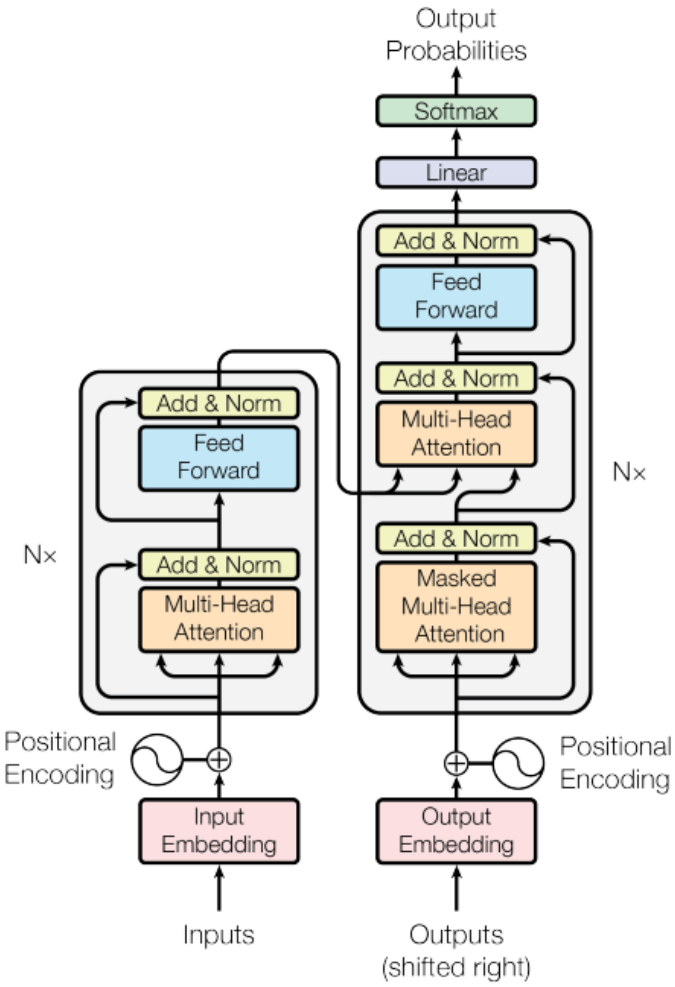


Figura 2.4: Representación de la arquitectura de un Transformer. Fuente: [6].

En resumen, este capítulo sirve como introducción al problema que vamos a intentar resolver. Primero se realiza una breve explicación y pasos comunes en las diferentes soluciones. A continuación se explican distintas formas usadas para resolverlo tradicionalmente. Finalmente, en la última sección, se afronta el problema usando redes neuronales convolucionales y *Transformers* que es la tecnología usada en nuestro modelo.

Capítulo 3

DocLayNet y LayoutLM: Análisis y explicación

En este capítulo comienza la parte práctica del trabajo, en él se explica el modelo que usaremos para predecir las categorías de nuestras entidades y el *dataset* con el que lo hemos entrenado.

Entrenar un modelo desde cero para una tarea específica es un proceso complicado, ya que se necesitan muchos datos, tiempo y ordenadores con gran capacidad computacional. Por esto mismo lo que se suele hacer es coger un modelo preentrenado en una tarea similar, pero más general y entrenarlo con nuevos datos para que pueda resolver la tarea específica deseada.

El objetivo del trabajo es usar el modelo preentrenado LayoutLMv2 y entrenarlo con DocLayNet, para resolver la tarea de identificar entidades estructurales. El proceso en el que entrenamos con un modelo pre-entrenado para una tarea específica se conoce como *fine-tuning*.

3.1. LayoutLMv2

LayoutLM es una familia de modelos basados en *Transformers*. Son multimodales ya que están orientados al análisis de texto, imagen y *layout* (llamamos *layout* a la disposición que ocupa el texto en el documento).

Layout language model más conocido como LayoutLM [9] fue lanzado en 2020 por *Microsoft Research Asia*. En 2021 lanzaron LayoutXML [10] una nueva versión centrada en resolver el mismo problema pero en documentos que mezclan varios idiomas. En enero de 2020 se lanzó LayoutLMv2 [11], una versión mejorada de LayoutLM, que es la que se usa en este trabajo. En julio de 2020 lanzaron LayoutLMv3 [12] que usa menos parámetros y da mejores resultados, es la última hasta la fecha.

Como se ha dicho LayoutLMv2 es un modelo multimodal, orientado a resolver estas tres tareas:

- *Question answering*: Consiste en dotar al modelo de la capacidad de responder a preguntas basándose en un conjunto de documentos.
- *Text classification*: Dota de una categoría al documento completo, por ejemplo, *spam* o *no-spam*.
- *Token classification*: Es el problema de identificación de entidades comentado en el capítulo 2.

Nosotros tratamos el problema de identificación de entidades, pero no nos interesa clasificar una palabra como sustantivo o adjetivo. En nuestro caso queremos clasificarla según sus características estructurales, es decir, por ejemplo queremos diferenciar si una palabra pertenece a un título o a una tabla.

3.1.1. Arquitectura del modelo

Una de las grandes diferencias entre LayoutLM y LayoutLMv2 es la siguiente. En la primera versión la información visual se añadía en el proceso de *fine-tuning*, en la segunda se añade en la fase de preentrenamiento, para así poder tener en cuenta la interacción cruzada entre la información textual y la visual.

El modelo toma como *input* la información textual, visual y de *layout* para establecer las interacciones entre ellas. Añade también un mecanismo de *self-attention* para un mejor modelado de la disposición del documento. A continuación se explica la estructura en base al guión usado en la subsección 2.3.2.

- ***Input embedding*:** Se codifica el texto, la parte visual y el *layout* de formas diferentes.

  - Codificación del texto: Se usa el método WordPiece (explicado en 1), añadiendo el *token* [CLS] al comienzo de la frase, [SEP] al final de cada segmento, y [PAD] hasta completar la longitud máxima, denotada L. La codificación final del texto es la suma del vector que codifica el texto en sí (TokEmb) , más una representación del índice del *token* (PosEmb1D) y una codificación que distingue los diferentes segmentos del texto (SegEmb).

    $$\mathbf{t}_i = \text{TokEmb}(w_i) + \text{PosEmb1D}(i) + \text{SegEmb}(s_i)$$

  - Codificación visual: Usamos un *encoder* basado en una red neuronal convolucional, para convertir la imagen en distintas secuencias de tamaño fijo. El proceso es el siguiente: primero redimensionamos la imagen a 224x224 y la pasamos por el *encoder*, después redimensionamos el *output* a WxH (prefijados), y finalmente hacemos una proyección lineal para unificar la dimensión de la codificación del texto y de la imagen. La codificación final es

    $$\mathbf{v}_i = \text{Proj}(\text{VisTokEmb}(I)_i) + \text{PosEmb1D}(i) + \text{SegEmb}(s_i)$$

    Donde PosEmb1D y SegEmb representan lo mismo que antes.

  - *Layout embedding*: Llamamos *bbox* (*bounding box*) a los cuadros delimitadores de los distintos *tokens*. Los normalizamos para que sus valores estén en el intervalo [0,1000]. Cada *bbox* queda identificada tal que así:

    $$bbox = (x_{min}, x_{max}, y_{min}, y_{max}, width, height)$$

    A continuación se codifican las coordenadas en dos *layers* diferentes y se concatenan, quedando de la siguiente forma:

    $$\mathbf{l}_i = \text{Concat}(\text{PosEmb2D}_x(x_{min}, x_{max}, width), \text{PosEmb2D}_y(y_{min}, y_{max}, height))$$

    A los *tokens* especiales [CLS], [SEP] y [PAD] se les asigan $bbox = (0,0,0,0,0,0)$

- ***Enconder*:** Primero concatena las diferentes $\mathbf{v}_i$ y $\mathbf{t}_i$ obteniendo $X = \{\mathbf{v}_0, ..., \mathbf{v}_{WH-1}, \mathbf{t}_0, ..., \mathbf{t}_{L-1}\}$, después le suma la codificación de las *bbox* obteniendo $x_i = X_i + \mathbf{l}_i$. La estructura es como en los *Transformers*, son *layers* con varias cabezas con *self-attention*, con la diferencia de que añadimos información sobre la posición relativa explícitamente.

  Supongamos que estamos en un una cabeza de un *layer* de tamaño d_{head}. Definimos $\mathbf{W}^Q$, $\mathbf{W}^K$ y $\mathbf{W}^V$, como las matrices de proyección de las consultas, claves y valores respectivamente. Entonces el mecanismo captura la correlación entre la consulta x_i y la llave x_j así: $\alpha_{ij} = \dfrac{1}{\sqrt{d_{head}}}(x_i\mathbf{W}^Q)(x_j\mathbf{W}^K)^T$

  Denotamos como $\mathbf{b}^{(1D)}$,$\mathbf{b}^{(2D_x)}$,$\mathbf{b}^{(2D_y)}$ los diferentes sesgos de posición, y siendo (x_i, y_i) las coordenadas de la esquina superior izquierda de la *bbox* i-ésima, obtenemos que la puntuación de atención *spatial-aware* viene dada por $\alpha'_{ij} = \alpha_{ij} + \mathbf{b}^{(1D)}_{j-i} + \mathbf{b}^{(2D_x)}_{x_j - x_i} + \mathbf{b}^{(2D_y)}_{y_j - y_i}$. Finalmente los vectores del *output* vienen dados por:

  $$\mathbf{h}_i = \sum_j \frac{\exp(\alpha'_{ij})}{\sum_k \exp(\alpha'_{ik})} \mathbf{x}_j \mathbf{W}^V$$

3.1.2. Preentrenamiento y *fine-tuning*

En [11] se presentaron dos modelos, con la misma estructura pero diferente configuración:

- LayoutLMv2$_{BASE}$: Se usa un *encoder* de 12 *layers* con 12 cabezas y de tamaño 768. Tiene un total de 200 millones de parámetros.
- LayoutLMv2$_{LARGE}$: Se usa un *encoder* de 24 *layers* con 16 cabezas y de tamaño 1024. Tiene un total de 426 millones de parámetros.

Ambos fueron preentrenados con el *dataset* IIT-CDIP Test Collection [13], un conjunto de datos compuesto por mas de once millones de páginas.

El proceso de *fine-tuning* ha sido realizado con los siguientes seis *datasets*:

- FUNSD [14] (199 páginas), CORD [15] (1000), SROIE [16] (973) y Kleister-NDA [17] (540) para la extracción de entidades y la extracción de datos asociados a un conjunto de claves.
- RVL-CDIP [18] (400.000 páginas) para la clasificación de imágenes.
- DocVQA [19] (50.000 preguntas) para la tarea de *question & answering*.

Para comparar estos dos modelos y los dos análogos de la versión uno, vemos su F1 en distintos *datasets*:

Model	FUNSD	CORD	SROIE	Kleister-NDA
LayoutLM$_{BASE}$	0.7866	0.9472	0.9438	0.8270
LayoutLM$_{LARGE}$	0.7895	0.9493	0.9524	0.8340
LayoutLMv2$_{BASE}$	0.8276	0.9495	0.9625	0.8330
LayoutLMv2$_{LARGE}$	0.8420	0.9621	0.9781	0.8520

3.2. DocLayNet

DocLayNet es un *dataset* publicado en 2022 [20] por IBM, con la idea de ser el *dataset* más completo para el análisis de *layout*. Anteriormente había otros *datasets* para esta misma tarea como son PubLayNet [21] (300.000 páginas anotadas) o DocBank [22] (500.000), que son muy válidos para entrenar modelos como LayoutLM, pero tienen un inconveniente, todos sus documentos son obtenidos de repositorios de artículos científicos, por lo que tienen poca variabilidad.

3.2.1. Análisis del *dataset*

Las anotaciones de los dos *datasets* anteriormente mencionados han sido generadas de manera automática por un modelo de segmentación. Esto reduce los costes, tanto de tiempo como de dinero, pero también reduce la variabilidad de los datos.

DocLayNet es un *dataset* en el que cada anotación ha sido realizada a mano, se puede ver la guía de anotación en [23]. En esta guía se explica con todo detalle cómo etiquetar las diferentes estructuras que pueden aparecer en cada tipo de documento, y posibles errores que pueden surgir. Para hacernos una idea de lo que conlleva la anotación manual, se necesitó el trabajo de 32 anotadores durante tres meses para poder etiquetar el *dataset* completo.

DocLayNet está dividido en tres, la parte de *train* conjunto preparado para entrenar el modelo, la parte de *validation* que sirve para evaluar el modelo y ajustar los parámetros, y el conjunto de *test* preparado para testear el rendimiento final del modelo. Cada subconjunto de DocLayNet tiene la misma estructura y solo difiere en la cantidad de documentos que contiene. *Train* tiene 69375 imágenes con un total de 941123 anotaciones, *validation* tiene 6489 imágenes con 99816 anotaciones y *test* tiene 4999 imágenes con 66531 anotaciones. En total DocLayNet está compuesto por 80.863 imágenes y 1.107.470 anotaciones.

El *dataset* está compuesto por documentos pertenecientes a distintas categorías: informes financieros un 32 % del total, manuales un 21 %, artículos científicos un 17 %, leyes y regulaciones un 16 %, patentes un 8 %, y licitaciones del gobierno 6 %.

Las categorías con las que se ha etiquetado son: *caption* (pie de foto), *footnote* (nota en el pie de página), *formula* (fórmula), *list-item* (elemento de una lista), *page-footer* (pie de página), *page-header* (encabezado), *picture* (imágen), *section-header* (encabezado de sección), *table* (tabla), *text* (texto), *title* (título). Los porcentajes de aparición quedan descritos en la siguiente tabla.

		% of Total		
class label	Count	Train	Test	Val
Caption	22524	2.04	1.77	2.32
Footnote	6318	0.60	0.31	0.58
Formula	25027	2.25	1.90	2.96
List-item	185660	17.19	13.34	15.82
Page-footer	70878	6.51	5.58	6.00
Page-header	58022	5.10	6.70	5.06
Picture	45976	4.21	2.78	5.31
Section-header	142884	12.60	15.77	12.85
Table	34733	3.20	2.27	3.60
Text	510377	45.82	49.28	45.00
Title	5071	0.47	0.30	0.50

Figura 3.1: Representación de la arquitectura de un Transformer. Fuente [20].

A continuación se realiza un análisis más exhaustivo de toda la información que nos proporciona DocLayNet. Tenemos dos carpetas, DocLayNet_Core y DocLayNet_Extra:

- DocLayNet_Core: Contiene las 80.863 imágenes en formato “PNG” y tres archivos “JSON”, uno para *train*, otro para *validation* y otro para *test*, todos con la misma estructura:
  - *Categories*: Contiene la lista de categorías arriba mencionadas y le asigna un id a cada una (que denotamos **l_id**).
  - *Images*: Contiene el id de la imagen (que denotamos **im_id**), su anchura y altura (todas tienen 1025x1025 píxeles), el nombre que tiene el archivo “PNG”, la categoría del documento al que pertenece y cierta información que no es de nuestro interés.
  - *Annotations*: Contiene el id de la anotación (que denotamos **ann_id**), el **im_id** de la imagen a la que pertenece, su **l_id**, y su *bbox* en formato COCO, es decir, en una lista de la forma $[x, y, w, h]$ dónde:
    - x es la distancia entre el lado izquierdo de la celda y el lado izquierdo de la hoja.
    - y es la distancia entre el lado superior de la celda y el lado superior de la hoja.
    - w representa la anchura de la anotación.
    - h representa la altura de la anotación.
- DocLayNet_Extra: Contiene las 80.863 imágenes en formato “PDF” y un “JSON” para cada imagen que contiene información sobre el texto de la página:
  - *Metadata*: contiene el nombre de la imagen, la categoría del documento y más información que no nos es útil.
  - *Cells*: Es la información que devuelve el OCR, en este caso se ha usado Detectron2 [24]. De toda la información solo nos interesa la *bbox*, que está en el mismo formato que antes, y el texto que contiene la celda.

En el anexo A (5.1) se puede ver un ejemplo de las *bbox* de *annotations* y de las de *cells*.

3.2.2. Preprocesado

En esta subsección se explica el proceso seguido para adaptar DocLayNet a nuestras necesidades y simplificarlo para usos futuros. Como ya se ha comentado anteriormente dentro de DocLayNet hay mucha información que no nos interesa, también hay una serie de imágenes que por su naturaleza no son adecuadas para entrenar el modelo. En las siguientes líneas se detallan las modificaciones hechas:

1. Ajuste de las *bbox*. Tras estudiar el *dataset* se observó que alguna coordenada superaba los límites de la imagen. Por ello el primer paso fue ajustar las coordenadas en los límites [0,1250]x[1,1250].

2. Para nuestro objetivo hay categorías que no necesitamos. Unificamos las anotaciones de *formula* y *footnote* en *text*.

3. Como podemos observar en la explicación anterior las celdas que contienen el texto no tienen una categoría asignada. Para solucionar esto hubo que ver a que anotación pertenecía cada celda, ya que sí sabemos las categorías de las anotaciones. Se implementaron dos funciones:

   - La primera calcula si el centro de la *bbox* de la celda esta contenido en la *bbox* de la anotación.
   - La segunda es una función *intersection over union* (IoU), calcula el coeficiente entre el área de la intersección y la unión de las dos *bbox*. Esta función tuvo que ser implementada debido a que se observaron celdas cuyo centro no estaba contenido en ninguna anotación. En total fueron 2214 celdas las que se emparejaron con esta función.

4. A continuación se eliminaron 286 imágenes que no tenían anotaciones. Eran páginas monocromas sin ningún tipo de palabra o dibujo. Pueden ser útiles para entrenar un modelo pero no a la hora del *fine-tuning*.

5. Finalmente se eliminaron las imágenes que no tienen celdas, es decir, en las que el OCR no ha devuelto nada. En total fueron 555.

Una vez hecho esto se reorganizó la información válida. Primero se reindexaron los id's de las imágenes y se renombraron, además se creo un archivo "JSON" homónimo para cada una, con la siguiente estructura:

- *Annotations*: Cada anotación tiene un id único, y una *bbox* y categorías reajustadas.
- *Cells*: Contiene la siguiente información de cada celda: la *bbox*, el texto, el id y la categoría de la anotación a la que pertenece.

Tras este costoso proceso el *dataset* quedo compuesto por:

	Train		Validation		Test	
Categorías	Num.	%	Num.	%	Num.	%
Leyes y regulaciones	10639	15.45	1143	17.25	774	15.58
Patentes	5526	8.02	483	7.48	442	8.89
Artículos científicos	12225	17.75	944	14.62	941	18.94
Informes financieros	22413	32.54	1731	26.80	1739	35.00
Manuales	14332	20.80	1853	28.69	800	16.10
Licitaciones del gobierno	3746	5.44	333	5.16	273	5.49
Total	68881		6458		4969	

En este capítulo se resume la manera elegida para resolver el problema de identificación de entidades. Primero se hace un acercamiento al modelo, explicando su estructura y cómo ha sido entrenado. En segundo lugar se explica el *dataset* con el que se va a realizar el *fine-tuning*, y cómo se ha preprocesado para seleccionar la información necesaria.

Capítulo 4

Entrenamiento y conclusiones

En este último capítulo se explica cómo se ha entrenado el modelo con nuestro *dataset* preprocesado. Seguidamente se hace un análisis de los resultados obtenidos. Para acabar se mencionan posibles mejoras a realizar en el futuro.

4.1. *Fine-tuning*

Como se ha comentado en el capítulo anterior, *fine-tuning* es el proceso de entrenamiento de un modelo con unos datos específicos para que este mejore en una tarea concreta.

Cada modelo se entrena de una manera diferente, adaptar nuestro *dataset* y ajustar los parámetros necesarios es una tarea sensible, ya que un pequeño cambio en la configuración puede conllevar que nuestro modelo de unos resultados no deseados.

El proceso para realizar el *fine-tuning* consta de dos pasos principales, que son los siguientes:

1. El primer paso es *tokenizar* nuestro *dataset*, es decir, adaptar la información para que el modelo sea capaz de leerla. Este proceso es relativamente automático.

   Existe una función propia de la librería Transformers que recibe como *inputs*: la imagen en formato PIL (codifica los píxeles, tamaño...), el texto palabra a palabra, las *bboxs* en formato $[x_1, y_1, x_2, y_2]$ (donde el primer par son las coordenadas de la esquina superior izquierda y el segundo par las de la esquina inferior derecha), y la categoría de cada palabra.

   Dicha función hace internamente el proceso comentado en 3.1.1 y devuelve: una lista con un id para cada palabra, una lista que identifica el tipo de token, la *attenion mask*, es decir, una máscara binaria que identifica los tokens que debe atender el modelo, y la representación correspondiente a las *bbox*, categorías e imagen.

   Esto es así porque en nuestro caso ya sabemos el texto que aparece en la imagen. LayoutLMv2 también tiene la posibilidad de usar Detectron2 [24] para la extracción de texto, esto cambiaría totalmente este paso.

2. El segundo paso es fijar los argumentos del entrenamiento:

   - Número de *epochs*: Es el número de veces que el modelo se entrena con cada dato. Entre una *epoch* y otra los pesos se ajustan con el objetivo de minimizar la función de perdida. En nuestro caso se ha elegido hacer 3 *epochs*, es decir, el modelo se va a entrenar con 36.675 imágenes.
   - Tamaño del lote: Se conoce como lote o *batch* al número de ejemplos que procesa a la vez nuestra CPU o GPU. Cuanto más grande el tiempo de entrenamiento es menor pero el consumo de memoria es mayor. Se ha elegido procesar los ejemplos de uno en uno.

- Optimizador: Algoritmo para ajustar los pesos de nuestro modelo. Se ha elegido AdamW (*adaptive moment estimation with weight decay*), lo resumimos como una evolución del gradiente descendente (1.1.1).
- *Learning rate*: Marca la longitud inicial del paso del algoritmo optimizador. Se ha elegido un valor de 1e-05.

Estos son los argumentos más importantes, en el entrenamiento se han tenido que fijar otros muchos que se escapan del alcance de este trabajo.

El entrenamiento se ha implementado de tal forma que se pueda ejecutar con una llamada API. Esto permite poder ejecutar el código (parte presentado en 5.2) en un ordenador con mayores capacidades, para así poder entrenar con más datos y de forma más rápida. Se contrató un ordenador con GPU, así se acorta el tiempo de entrenamiento en aproximadamente 12 veces.

Por razones de tiempo y de capacidad computacional se ha realizado el *fine-tuning* únicamente con los archivos pertenecientes a la categoría artículos científicos. Se han usado 12225 para el entrenamiento y 944 para la validación, representando estos un 16.4 % del total.

4.2. Evaluación del modelo y limitaciones

Como se ha dicho anteriormente tras cada *epoch* el modelo se evalúa, así se puede saber cuál es el modelo que produce mejores resultados y elegir ese. Para entender las métricas de evaluación debemos introducir unos conceptos básicos primero. Supongamos que estamos analizando la categoría c, entonces:

- Un positivo verdadero (TP) es cuando el *token* pertenezca a c y predecir que sí que pertenece.
- Un positivo falso (FP) es cuando el *token* no pertenezca a c y predecir que sí que pertenece.
- Un negativo verdadero (TN) es cuando el *token* no pertenezca a c y predecir que no pertenece.
- Un negativo falso (FN) es cuando el *token* pertenezca a c y predecir que no pertenece.

Ahora estamos en disposición de entender los datos de evaluación que obtenemos tras cada *epoch*. Esta evaluación se realiza sobre los 944 ejemplos de *validation*. Los datos que obtenemos son los siguientes:

- *Accuracy* (exactitud): Viene dada por $\frac{TP+TN}{TP+FP+TN+FN}$, es decir, los resultados acertados partidos al total. Es deficiente cuando tenemos categorías de tamaños dispares.
- *Precision*: Viene dada por $\frac{TP}{TP+FP}$, es decir, da la probabilidad de que un resultado positivo sea cierto.
- *Recall* (Exhaustividad): Viene dada por $\frac{TP}{TP+FN}$. Si su valor es alto quiere decir que el modelo clasifica correctamente la mayoría de ejemplos positivos.
- F1 *Score*: Combina las medidas de *precision* y *recall*, para facilitar la comparación de resultados, mediante la media armónica. Así se tiene que $F1 = 2\frac{\text{precision} * \text{recall}}{\text{precision} + \text{recall}}$.

Para ver si el proceso ha sido realmente efectivo el modelo se evalúa antes del *fine-tuning* y tras cada *epoch*. Los resultados obtenidos son los siguientes:

	Accuracy	**Precision**	**Recall**	**F1**
LayoutLMv2	0.7934	0.0542	0.0394	0.0456
Epoch 1	0.9871	0.8939	0.9481	0.9202
Epoch 2	0.9902	0.9476	0.9557	0.9516
Epoch 3	0.9933	0.9553	0.9679	0.9626

Los resultados obtenidos son similares a los mejores obtenidos en 3.1.2, con esto nos aseguramos que nuestro modelo es totalmente eficiente para identificar entidades en artículos científicos.

A continuación se analiza el funcionamiento de nuestro modelo a la hora de reconocer entidades en archivos pertenecientes a otra categoría. Para ello evaluamos nuestro modelo sobre todos los documentos del apartado validación de cada una de las categorías.

	Accuracy	**Precision**	**Recall**	**F1**
Leyes y regulaciones	0.8444	0.5339	0.4748	0.5026
Patentes	0.7958	0.5643	0.4864	0.5225
Informes financieros	0.7264	0.4560	0.3512	0.3968
Manuales	0.8311	0.4608	0.3391	0.3907
Licitaciones del gobierno	0.8017	0.5831	0.4976	0.5370

Cuando analizamos en profundidad los resultados observamos las siguientes limitaciones:

- Se obtienen unos resultados menos eficientes a la hora de predecir las categorías que aparecen menos veces, entre estas destacamos las categorías *title* y *caption*.
- Obtenemos peores resultados a la hora de predecir la categoría *page-footer*. Esto es debido a que en el preprocesado de nuestros datos nos quedamos solamente con un número fijo de *tokens*, lo que implica que cuando hay muchos los del final de la hoja no se incluyan.
- Cuando extrapolamos nuestro modelo a documentos de otra categoría los resultados empeoran. Si observamos la tabla 3.2.2 vemos que se obtienen peores valores cuanto más grande es el conjunto de datos de validación, achacamos esto a que cuantos más datos hay mayor variedad.

En el Anexo C (5.3) se puede ver un ejemplo de las categorías predichas por el modelo.

4.3. Desafíos futuros

Haber realizado el entrenamiento únicamente con artículos científicos hace que este modelo en la práctica solo se vaya a usar sobre este tipo de documentos. Esto deja una puerta abierta para seguir mejorándolo y poder hacerlo más completo.
Se proponen los siguientes desafíos para el futuro:

- La continuación del entrenamiento con documentos pertenecientes a otra categoría y ver cómo cambian los resultados a la hora de extrapolarlo a otro tipo de documentos.
- Investigar sobre cómo optimizar los tiempos de entrenamiento.
- Entrenar el modelo con los mismos documentos girados, esto puede solucionar problemas como el mencionado anteriormente con la categoría *page-footer*.
- Probar a dividir el entrenamiento en distintos grupos par así exigir menos capacidad de almacenamiento y de memoria al ordenador.

Aunque este capítulo es el más corto en extensión ha resultado el más largo en tiempo de realización. Aquí se explica el proceso seguido para poder realizar correctamente el entrenamiento y un pequeño análisis de los resultados obtenidos. En el anexo B (5.2) se muestra parte del código usado en la realización de este capítulo.

Bibliografía

[1] CRESPÍ, A.(2018), *Modelos ocultos de Markov para el etiquetado de texto*, Universidad de las Islas Baleares, disponible en https://dspace.uib.es/xmlui/bitstream/handle/11201/150347/tfm_2018-19_MADM_act613_2116.pdf?sequence=1&isAllowed=y.

[2] MIKOLOV, T., CHEN, K., CORRADO, G., DEAN, J., (2013), *Efficient Estimation of Word Representations in Vector Space*, https://arxiv.org/pdf/1301.3781v3.pdf.

[3] MIKOLOV, T., SUTSKEVER, I., CHEN, K., CORRADO, G., DEAN, J., (2013), *Distributed Representations of Words and Phrases and their Compositionality*, https://arxiv.org/pdf/1310.4546.pdf.

[4] HUGGINGFACE, https://huggingface.co/learn/nlp-course/chapter6/6?fw=pt#wordpiece-tokenization

[5] SONG, X., ZHOU, D. (2021), *A Fast WordPiece Tokenization System,* https://ai.googleblog.com/2021/12/a-fast-wordpiece-tokenization-system.html

[6] VASWANI, A., SHAZEER, N., PARMAR, N., USZKOREIT, J., JONES, L., GOMEZ, A., KAISER, L., POLOSUKHIN, I. (2017), *Atention Is All You Nedd*, disponible en https://arxiv.org/pdf/1706.03762.pdf

[7] DEVLIN, J., CHANG, M., LEE, K., TOUTANOVA, K. (2019), *BERT: Pre-training of Deep Bidirectional Transformers for Language Understanding*, disponible en https://arxiv.org/pdf/1810.04805.pdf

[8] RADFORD, A., NARASIMHAM, K., SALIMANS, T., SUTSKEVER, I. (2018), *Improving Language Understanding by Generative Pre-Training*, disponible en https://s3-us-west-2.amazonaws.com/openai-assets/research-covers/language-unsupervised/language_understanding_paper.pdf

[9] XU, Y., LI, M., CUI, L., HUANG, S., WEI, F., ZHOU, M. (2020), *LayoutLM: Pre-training of Text and Layout for Document Image Understanding*, disponible en https://arxiv.org/pdf/1912.13318.pdf

[10] XU, Y., LV, T., CUI, L., WANG, G., LU, Y., FLORENCIO, D., ZHANG, C., WEI, F. (2021), *LayoutXLM: Multimodal Pre-training for Multilingual Visually-rich Document Understanding*, disponible en https://arxiv.org/pdf/2104.08836.pdf

[11] YANG XU, YIHENG XU,LV, T., CUI, L., WEI, F., WANG, G., LU, Y., FLORENCIO, D., ZHANG, C., CHE, W., ZHANG, M., ZHOU, L. (Enero 2022), *LayoutLMv2: Multi-modal Pre-training for Visually-rich Document Understanding*, disponible en https://arxiv.org/pdf/2012.14740v4.pdf

[12] HUANG, Y., LV, T., CUI, L., LU, Y., WEI, F. (Julio 2022), *LayoutLMv3: Pre-training for Document AI with Unified Text and Image Masking*, disponible en https://arxiv.org/pdf/2204.08387.pdf

[13] LEWIS, D., AGAM, G., ARGAMON, S., FRIEDER, O., GROSSMAN, D., HEARD, J.(2006) *Building a Test Collection for Complex Document Information Processing*, disponible en https://dl.acm.org/doi/pdf/10.1145/1148170.1148307?casa_token=13Jb96b8QvUAAAAA:S--DsI4yTmj4hRj-rHlmzTY1e3HeIgSxcTge_Q6JKbljNYp801MVuP3ljTZyoetj_kZv38gaIuKy

[14] JAUME, G., KEMAL, H., THIRAN, J-P.(2019) *FUNSD: A Dataset for Form Understanding in Noisy Scanned Documents*, disponible en https://ieeexplore.ieee.org/stamp/stamp.jsp?tp=&arnumber=8892998

[15] PARK, S., SHIN, S., LEE, B., LEE, J., SURH, J., SEO, M., LEE, H.(2019) *A consolidated receipt dataset for postocr parsing.*

[16] HUANG, Z., CHEN, K., HE, J., BAI, X., KARATZAS, D., LU, S., JAWAHAR, C.(2019) *ICDAR2019 Competition on Scanned Receipt OCR and Information Extraction*, disponible en https://ieeexplore.ieee.org/stamp/stamp.jsp?tp=&arnumber=8977955

[17] GRALINSKI, F., STANISLAWEK, T., WRÓBLEWSKA, A., LIPINSKI, D., KALISKA, A., ROSALSKA, P., TOPOLSKI, B., BIECEK, P.(2020) *Kleister: A novel task for Information Extraction involving Long Documents with Complex Layout*, disponible en https://arxiv.org/pdf/2003.02356.pdf

[18] HARLEY, A., UFKES, A., DERPANIS, K.(2015) *Evaluation of deep convolutional nets for document image classification and retrieval*

[19] MATHEW, M., KARATZAS, D., JAWAHAR, C.(2021) *Docvqa: A dataset for vqa on document images.*

[20] PFITZMANN, B., AUER, C., DOLFI, M., NASSAR, A., STAAR, P. (2022), *DocLayNet: A Large Human-Annotated Dataset for Document-Layout Analysis* , disponible en https://arxiv.org/pdf/2206.01062.pdf

[21] ZHONG, X., TANG, J., JIMENO, A. (2019), *PubLayNet: largest dataset ever for document layout analysis* , disponible en https://arxiv.org/pdf/1908.07836.pdf

[22] LI, M., XU, Y., CUI, L., HUANG, S., WEI, F., LI, Z., ZHOU, M. (2020), *DocBank: A Benchmark Dataset for Document Layout Analysis* , disponible en https://arxiv.org/pdf/2006.01038.pdf

[23] PFITZMANN, B., AUER, C., DOLFI, M., NASSAR, A., STAAR, P. (2022), *IBM DocLayNet Labeling Guide* , disponible en https://raw.githubusercontent.com/DS4SD/DocLayNet/main/assets/DocLayNet_Labeling_Guide_Public.pdf

[24] WU, Y., KIRILLOV, A., MASSA, F., LO, W., GIRSHICK, R. (2019), *Detectron2* , disponible en https://github.com/facebookresearch/detectron2

Capítulo 5

Anexos

5.1. Anexo A: Ejemplo DocLayNet

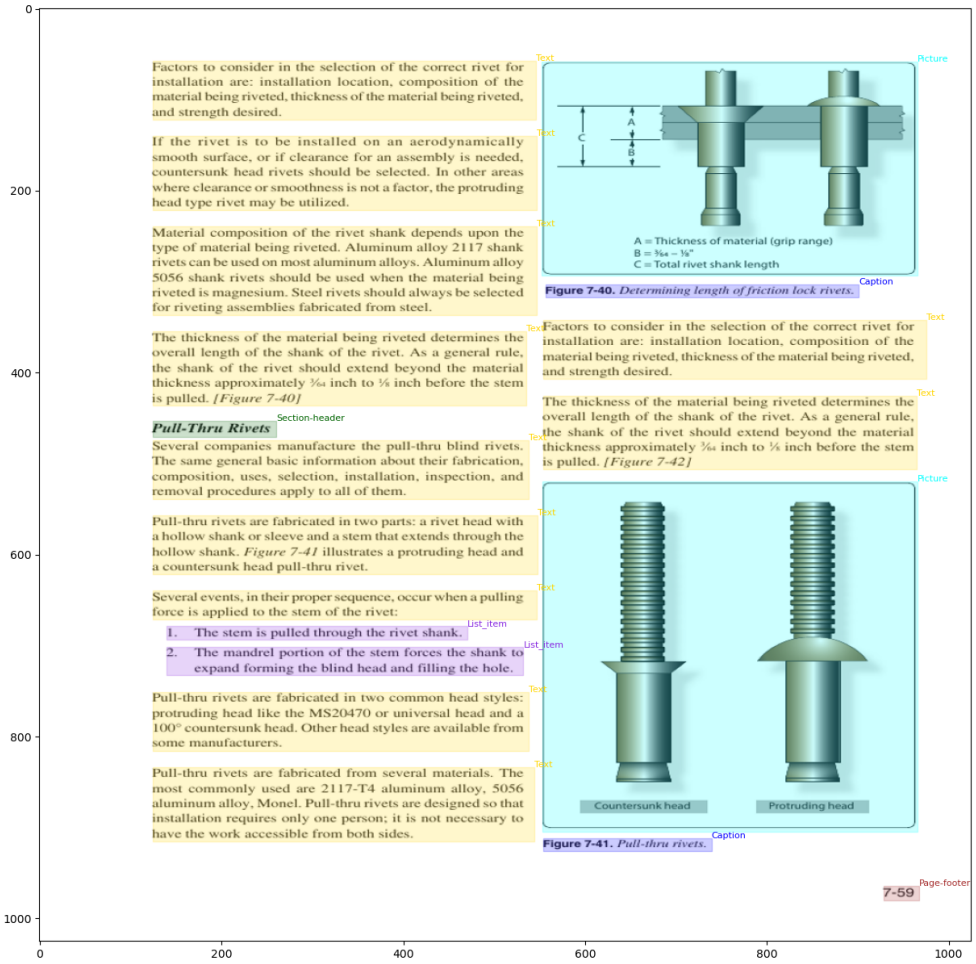


Figura 5.1: Representación de las *bboxs* de las anotaciones

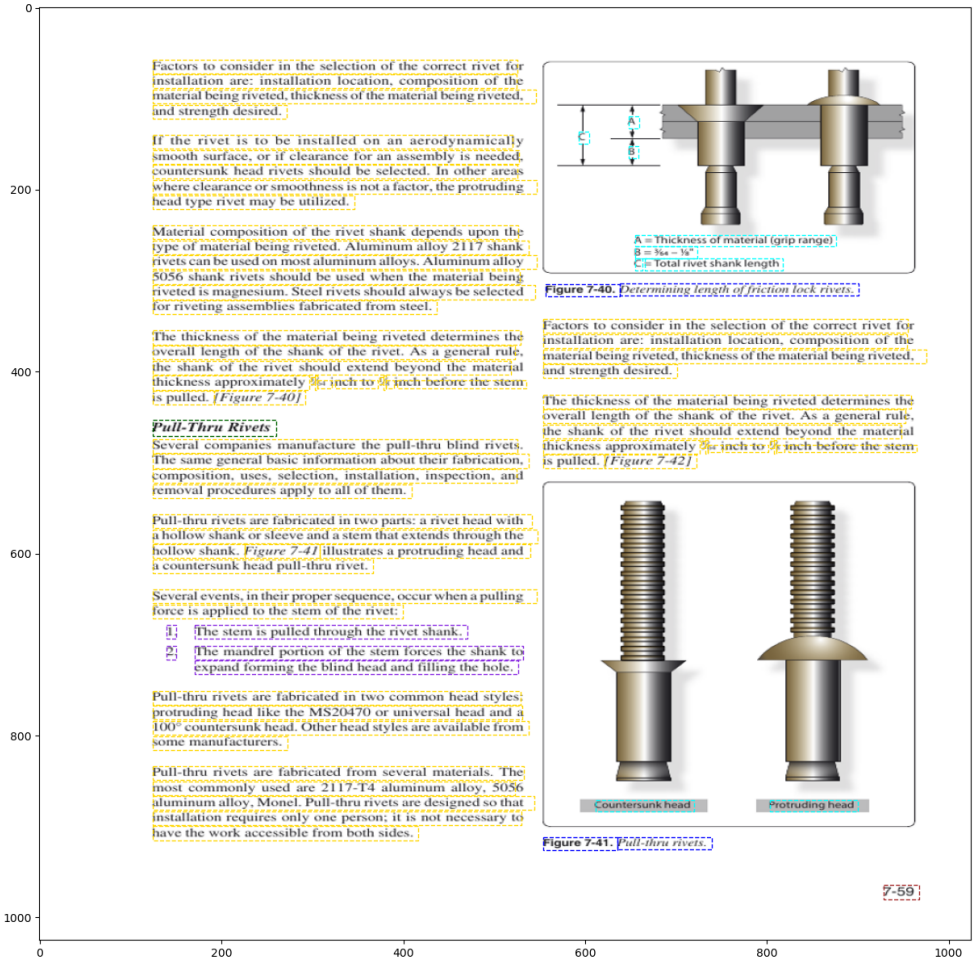


Figura 5.2: Representación de las *bboxs* de las celdas

5.2. Anexo B: Código

- Código de modificación del *dataset*:

Listing 5.1: Esta función ajusta los bordes de las *bbox* a los márgenes de las hojas.

```
def ajuste_bordes(bbox):
    x = bbox[0] if bbox[0] > 0 else 0
    y = bbox[1] if bbox[1] > 0 else 0
    w = bbox[2] if bbox[0] + bbox[2] < 1025 else 1025 - bbox[0]
    h = bbox[3] if bbox[1] + bbox[3] < 1025 else 1025 - bbox[1]
    return [x, y, w, h]
```

Listing 5.2: Esta función unifica las categorías *formula* y *footnote* en text.

```
def ajuste_categorias(n):
    if n == 1:
        n = 0
    elif n == 2 or n == 3 or n == 10:
        n = 7
    elif n == 4:
        n = 1
    elif n == 5:
        n = 2
    elif n == 6:
        n = 3
    elif n == 7:
        n = 4
    elif n == 8:
        n = 5
    elif n == 9:
        n = 6
    elif n == 11:
        n = 8
    return n
```

Listing 5.3: Esta función devuelve el coeficiente entre la intersección de dos celdas y el área de la primera.

```
def IoU(cell, annotation):
    x, y, w, h = cell
    x1, y1, w1, h1 = annotation
    supx = max(x, x1)
    supy = max(y, y1)
    infx = min(x + w, x1 + w1)
    infy = min(y + h, y1 + h1)
    in_h = max(infy - supy, 0)
    in_w = max(infx - supx, 0)
    areaIn = in_h * in_w
    areaCell = w * h
    iou = areaIn / areaCell
    return iou
```

Listing 5.4: Esta función comprueba si el centro de una *bbox* está contenido en otra.

```
def ApertenecaB(A, B):
    if B[0] < (A[0] + A[0] + A[2]) / 2 < B[0] + B[2]
        and B[1] < (A[1] + A[1] + A[3]) / 2 < B[1] + B[3]:
    return True
```

Listing 5.5: Esta función carga el core del *dataset*.

```
def cargaCore(old_path, TTV):
    images1 = load_dataset('json',data_files=fr"{old_path
        }/DocLayNet_core/COCO/{TTV}.json", field='images')
    images = images1["train"]
    annotations1 = load_dataset('json', data_files=fr"{
        old_path}/DocLayNet_core/COCO/{TTV}.json", field='
        annotations')
    annotations = annotations1["train"]
    return images, annotations
```

Listing 5.6: Esta función es utilizada para realizar la reindexación, además devuelve la cantidad de anotaciones de cada documento y los documentos sin anotaciones.

```
def annINim(images, annotations):
    relation = {}
    cantidadAnn = {}
    no_annotations = []
    k = 0
    for im in images:
        list_ann = []
        a = k
        im_id = im['id']
        for i in range(k, len(annotations)):
            if im_id == annotations[i]['image_id']:
                list_ann.append(annotations[i])
                k = k + 1
            elif a != k:
                break
        if a == k:
            no_annotations.append(im['file_name'].replace
                ('.png', ''))
        relation[im['file_name'].replace('.png', '')] =
            list_ann
        cantidadAnn[im['file_name'].replace('.png', '')]
            = k - a
    return relation, cantidadAnn, no_annotations
```

Listing 5.7: Esta función clasifica los documentos por categorías.

```
def tipoArchivo(images, lista):
    categorias = {}
    l1, l2, l3, l4, l5, l6 = [], [], [], [], [], []
    for i in images:
        if i['file_name'].replace('.png', '') in lista:
            if i['doc_category']=='laws_and_regulations':
                l1.append(i['file_name'].replace('.png',
                    ''))
            elif i['doc_category']=='patents':
                l2.append(i['file_name'].replace('.png',
                    ''))
            elif i['doc_category'] == '
                scientific_articles':
                l3.append(i['file_name'].replace('.png',
                    ''))
            elif i['doc_category']=='financial_reports':
                l4.append(i['file_name'].replace('.png',
                    ''))
            elif i['doc_category']=='manuals':
                l5.append(i['file_name'].replace('.png',
                    ''))
            elif i['doc_category']=='government_tenders':
                l6.append(i['file_name'].replace('.png',
                    ''))
    categorias['laws_and_regulations'] = l1
    categorias['patents'] = l2
    categorias['scientific_articles'] = l3
    categorias['financial_reports'] = l4
    categorias['manuals'] = l5
    categorias['government_tenders'] = l6
    return categorias
```

Listing 5.8: Esta función reindexa y crea las anotaciones con la información necesaria.

```
def creaAnotaciones(relation, fn, imid):
    annotations = []
    for j, i in enumerate(relation[fn]):
        annotations.append({
        'id': fr"{imid}_{j}",
        'bbox':     f.ajuste_bordes(i['bbox']),
        'category': f.ajuste_categorias(i['category_id'])
            })
    final_json['annotations'] = annotations
    return annotations
```

Listing 5.9: Esta función reindexa y crea las celdas con la información necesaria.

```
def creaCeldas(annotations, aux,
    ContadorCeldasSinAnotacion, valoresIoU):
    final_json = {'annotations': annotations}; cel = []
    for cell in aux:
        dic = {'bbox': f.ajuste_bordes(cell['bbox']),'
            text': cell['text']}; bbox = dic['bbox']
        for ann in annotations:
            if f.AperteneceaB(bbox, ann['bbox']):
                dic['annotation'] = ann['id']
                dic['category'] = ann['category']
                cel.append(dic)
                break
        else:
            try:
                maxiou = 0
                for ann in annotations:
                    fiou = f.IoU(bbox, ann['bbox'])
                    if fiou > maxiou:
                        maxiou = fiou
                        indice = ann
                if maxiou:
                    valoresIoU.append({'id': indice['id'
                        ], 'IoU': maxiou})
                    dic['annotation'] = indice['id']
                    dic['category'] = indice['category']
                    cel.append(dic)
                if maxiuou == 0:
                    ContadorCeldasSinAnotacion =
                        ContadorCeldasSinAnotacion + 1
            except:
                pass
    final_json['cells'] = cel
    return final_json, valoresIoU,
        ContadorCeldasSinAnotacion
```

- Código de preprocesado del *dataset*:

Listing 5.10: Esta función preprocesa los datos de nuestro *dataset*.

```
def preprocess_data(ds,modelo,OCR):
    image=ds['image']
    words=ds['words']
    boxes=ds['boxes']
    word_labels=ds['word_labels']
    processor = LayoutLMv2Processor.from_pretrained
                (modelo, revision= OCR)
    encoded_inputs = processor(image, words, boxes=boxes,
            word_labels=word_labels,
                padding="max_length", truncation=True)
    return encoded_inputs
```

Listing 5.11: Esta función crea el *dataset* procesado.

```
def createDS(ds,labels,modelo,OCR):
    features = Features({
        'image': Array3D(dtype="int64", shape=(3, 224,
           224)),
        'input_ids': Sequence(feature=Value(dtype='int64'
           )),
        'attention_mask': Sequence(Value(dtype='int64')),
        'token_type_ids': Sequence(Value(dtype='int64')),
        'bbox': Array2D(dtype="int64", shape=(512, 4)),
        'labels': Sequence(ClassLabel(names=labels)),
    })
    dataset=load_from_disk(ds)
    train_dataset = dataset.map(preprocess_data,batched=
       True, remove_columns=dataset.column_names,features
       =features,fn_kwargs={'modelo':modelo,'OCR':OCR})
    train_dataset.set_format(type="torch")
    return train_dataset
```

- Código del entrenamiento:

Listing 5.12: Esta función fija los argumentos para el entrenamiento.

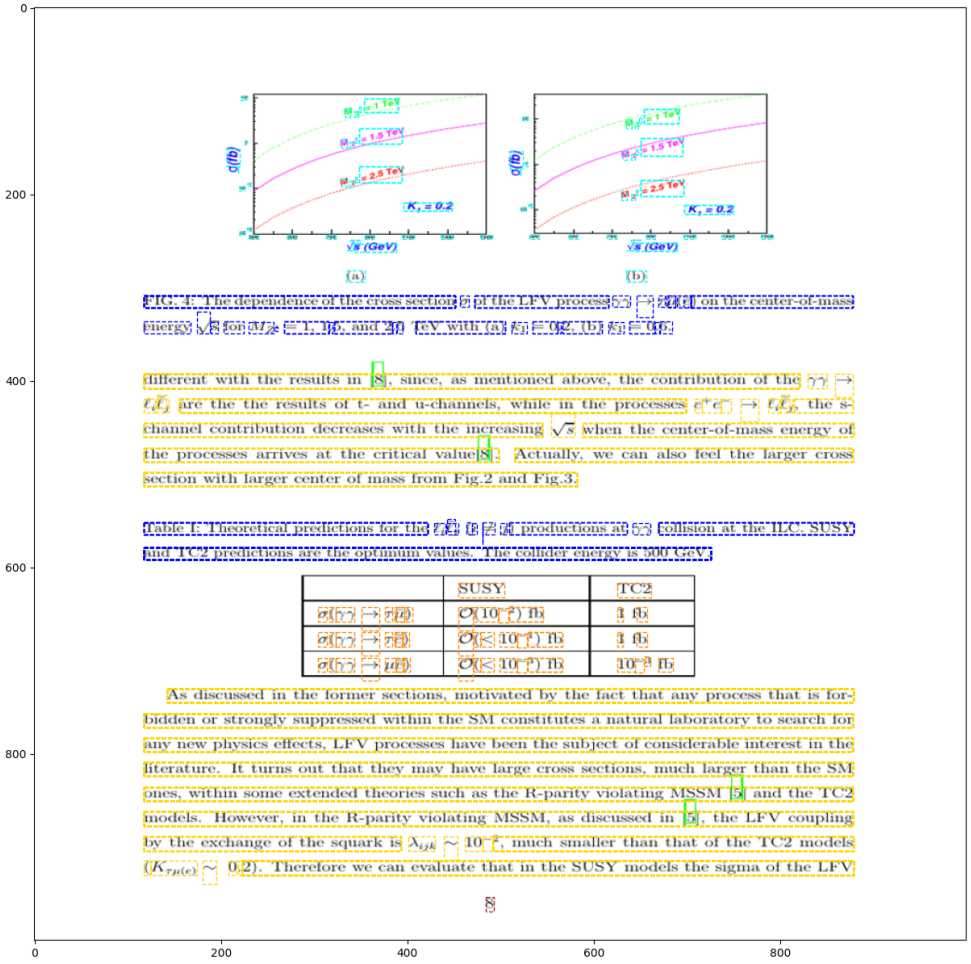
```
def trArgs(output_path,epochs,seed,train_batch_size,
   eval_batch_size,l_rate,log_strategy,eval_strategy,
   save_strategy, eval_steps,best_model,metric):
training_args = TrainingArguments(output_dir=output_path,
            num_train_epochs=epochs, seed=seed,
            per_device_train_batch_size=train_batch_size,
            per_device_eval_batch_size=eval_batch_size,
            learning_rate=l_rate,
            logging_strategy=log_strategy,
            evaluation_strategy=eval_strategy,
            save_strategy=save_strategy,
            eval_steps=eval_steps,
            load_best_model_at_end=best_model,
            metric_for_best_model=metric)
return training_args
```

Listing 5.13: Esta función crea el *trainer* a partir del dataset procesado y los argumentos del entrenamiento.

```
def createTrainer(train_dataset,training_args,
    eval_dataset,modelo,OCR,labels):
     processor = LayoutLMv2Processor.from_pretrained(
         modelo, revision=OCR)
     train_dataloader = DataLoader(train_dataset)
     batch = next(iter(train_dataloader))
     model = LayoutLMv2ForTokenClassification.
         from_pretrained(modelo,num_labels=len(labels))
     trainer = Trainer(
          model=model,
          args=training_args,
          train_dataset=train_dataset,
          eval_dataset=eval_dataset,
          tokenizer=processor,
          data_collator=default_data_collator,
          compute_metrics=compute_metrics)
     return trainer
```

5.3. Anexo C: Ejemplo de aplicación del modelo

Nuestro modelo únicamente devuelve la categoría a la que pertenece cada *bbox*, aquí las hemos dibujado diferenciando cada categoría con un color diferente.



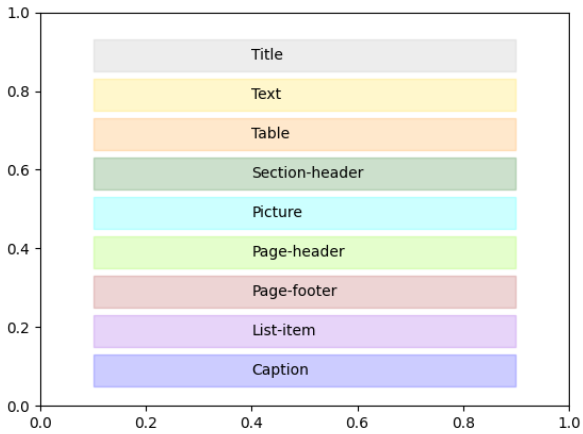


Figura 5.3: Índice de los colores usados en el Anexo A y en el Anexo C.