Trabajo Fin de Máster
Máster Universitario en Ingeniería de Telecomunicación

Modelos de atención para la transcripción de textos manuscritos históricos

Autor: Luis García González

Tutores: José Carlos Aradillas Jaramillo
Juan José Murillo Fuentes

Dpto. Teoría de la Señal y Comunicaciones
Escuela Técnica Superior de Ingeniería
Universidad de Sevilla

Sevilla, 2020

Trabajo Fin de Máster
Máster Universitario en Ingeniería de Telecomunicación

Modelos de atención para la transcripción de textos manuscritos históricos

Autor:

Luis García González

Tutor:

José Carlos Aradillas Jaramillo

Juan José Murillo Fuentes

Dpto. de Teoría de la Señal y Comunicaciones

Escuela Técnica Superior de Ingeniería

Universidad de Sevilla

Sevilla, 2020

Trabajo Fin de Máster: Modelos de atención para la transcripción de textos manuscritos históricos

Autor: Luis García González

Tutores: José Carlos Aradillas Jaramillo
Juan José Murillo Fuentes

El tribunal nombrado para juzgar el Proyecto arriba indicado, compuesto por los siguientes miembros:

Presidente:

Vocales:

Secretario:

Acuerdan otorgarle la calificación de:

Sevilla, 2020

El secretario del Tribunal

A mi familia

A mis maestros

Agradecimientos

En primer lugar y ante todo me gustaría aprovechar este espacio para agradecer a mis padres y hermano el apoyo constante que me han otorgado y sin el que no hubiese sido posible haber alcanzado el punto en el que me encuentro hoy día.

La realización del Trabajo Fin de Máster marca el punto y final de un estudiante en la carrera universitaria que ha realizado. En este trabajo el estudiante tiene que poner de manifiesto que está preparado para realizar un proyecto digno en el que haya utilizado ampliamente los conocimientos adquiridos durante los años de estudio y en el que vaya más allá introduciendo nuevas herramientas. De esta forma presento a continuación un trabajo que en mi opinión plasma los puntos más importantes que se enseñan durante la carrera. En primer lugar, la base de conocimiento que he recibido de numerosos profesores a través de cada asignatura que me ha sido impartida, a todos estos profesores les doy las gracias. Por otra parte, la necesidad de ser uno mismo el que centre sus objetivos y se enfoque en cumplirlos. Y por último la evidencia de que toda persona debe de buscar más allá cuando no encuentre los resultados obtenidos, teniendo que investigar e instruirse en conocimientos que no le han sido impartidos.

Me gustaría agradecer en especial a mi tutor Don José Carlos Aradillas Jaramillo, el haber mostrado la ilusión constante con la que es necesario afrontar los retos académicos y sin la cuál no sería posible obtener resultados, así como la guía que me ha prestado durante la ejecución de este trabajo.

Por último, me gustaría agradecer a todos mis compañeros, en especial a Paloma, que han compartido clases y prácticas conmigo por haber sido parte de esta parte de mi vida y haberme aportado experiencias que me han ayudado a mejorar.

Luis García González

Sevilla, 2020

Resumen

La transcripción de texto manuscrito es una tarea que cobra importancia a la par de la necesidad de conservar todo el conocimiento e historia que esconden los textos antiguos. La mayoría de los textos importantes de autores reputados llevan tiempo transcritos a texto digital, pero la ingente cantidad de textos menores o específicos sólo son abarcables con procesos industrializados, en este caso a través de software automático.

La automatización que se trata en este proyecto es una automatización a través de modelos de deep learning, los cuales se basan en redes neuronales. Esta automatización se lleva a cabo a partir de una imagen digitalizada del texto manuscrito, realizada manualmente, por lo que esta técnica no puede conocerse como una automatización pura.

Desde la aparición del OCR, son numerosos los autores que han propuesto modelos varios para dar solución a este problema. Estas soluciones abarcan desde reconocimiento de caracteres hasta reconocimiento de frases completas. El objetivo de este proyecto es presentar una solución basada en modelos de atención. De esta manera se propone que la dependencia de la transcripción realizada de los caracteres anteriores y posteriores al que ocupa en cada momento puede aportar claridad y exactitud a la transcripción del mismo, información que va más allá de las propias características que presenta la imagen de cada caracter.

Abstract

The transcription of handwritten text takes on the importance of preserving all the knowledge and history hidden in ancient texts. Most of the known texts or those of reputed authors have been transcribed into digital text for a long time, but the huge amount of minor or specific texts can only be covered by industrialized processes, in this case through automatic software.

The automation dealt with in this project is automation through deep learning models, which are based on neural networks. This automation is carried out from a digitized image of the handwritten text, which must be done by hand, so this technique cannot be known as pure automation.

Since the emergence of OCR, many authors have proposed various models to solve this problem. These solutions range from character recognition to full sentence recognition. The aim of this project is to present a solution based on the attention model. In this way, it is proposed that the dependence of the transcription made of the characters next to the one it occupies at each moment can contribute to the clarity and accuracy of the transcription, thus providing information beyond the own characteristics that it presents in the image of each character.

Índice

Agradecimientos **9**

Resumen **11**

Abstract **13**

Índice **14**

Índice de Tablas **17**

Índice de Figuras **19**

1 Introducción **21**

2 Introducción a la transcripción de manuscritos **23**

2.1. Inteligencia Artificial *23*

2.2. Machine Learning *23*

2.1.1 Tipos de Machine Learning 24

2.3. GPU *24*

2.4. HTR *25*

2.1.2 OCR 25

3 Software **11**

3.1 TensorFlow *11*

3.1.1 Características de Tensorflow 11

3.1.2 Beneficios de Tensorflow 12

3.2 Keras *12*

4 Redes neuronales **13**

4.1 Redes neuronales totalmente conectadas *13*

4.2 Redes convolucionales *14*

4.3 Redes recurrentes y LSTM *14*

4.4 Mecanismos de atención *15*

4.5 Transformación *17*

5 Modelos de atención 19
5.1 Bahdanau 19
5.1.1 Codificador-Decodificador RNN 20
5.1.2 Decodificador 20
5.1.3 Codificador (RNN BIDIRECCIONAL PARA SECUENCIAS) 22
5.2 Luong 22
5.2.1 Atención Global 23
5.2.2 Atención Local 24
5.3 Monotonic 25
6 Aplicación al problema de traducción 28
6.1 Esquema general y modelo del sistema 28
6.2 Dataset 30
6.3 Entrenamiento 31
6.4 Validación 31
6.5 WER 32
6.6 TEST 32
7 Aplicación al problema de transcripción 34
7.1 Esquema general y modelo del sistema 34
7.2 Dataset 36
7.3 Preprocesado de la imagen 36
7.4 Entrenamiento 37
7.5 Validación 38
7.6 Entrenamiento 38
7.6.1 Dense Layer 38
7.6.2 CNN 1 capa 39
7.6.3 CNN de varias capas con dropout 39
7.6.4 Inception_v3 40
7.6.5 Resultados 43
8 Conclusiones 46
Referencias 48
Anexo 52
Glosario 68

ÍNDICE DE TABLAS

Tabla 6–1 Resultados taducción 33
Tabla 7–1 Esquema de inception_v3 41
Tabla 7–2 Resultados de validación 44
Tabla 7–3 Resultados de Test (Comparación) 44
Tabla 7–4 Resultados de validación 45

ÍNDICE DE FIGURAS

Figura 1. Red de perceptrones [23] 13
Figura 4-2. Red convolucional [26] 14
Figura 4-3. Celda individual de LSTM [27] 15
Figura 4-4. Matriz de pesos entre entrada y salida [29] 16
Figura 4-5. Relación entre palabras [29] 17
Figura 5-6. Modelo de predicción de yy [2] 22
Figura 7. Modelo de atención Global [20] 23
Figura 8. Modelo de atención Local [20] 25
Figura 9. Modelo de predicción de enradas atendidas [4] 26
Figura 10 – Esquema problema de traducción 29
Figura 11 – Esquema de modelo de atención [20] 29
Figura 12 – Código de preprocesado de frase 30
Figura 13 – Matriz del modelo de atención [20] 31
Figura 14 – Esquema del problema de transcripción 35
Figura 15 – Código de preprocesado de imagen 37
Figura 16 – Esquema de preprocesado de imágenes 37
Figura 17 – Dense layer [39] 39
Figura 18 – Salida transcripción sílabas 39
Figura 19 – Código capa CNN-POOL-DROPOUT 40
Figura 20 – Salida transcripción palabras 40
Figura 21 – Esquema del modelo de inception_v3 [42] 41
Figura 22 – Esquema de los modelos de inception [43] 42
Figura 23 – Esquema del modelo de transcripción con preprocesado inception_v3 43

1 INTRODUCCIÓN

En los últimos años, gracias al aumento de la capacidad de computación, se ha observado una emergente expansión de la investigación relacionada con la inteligencia artificial, lo cual ha permito poner en práctica multitud de modelos matemáticos que ya estaban enunciados con anterioridad pero que la tecnología de su momento no era capaz de tratar. En palabras sencillas, la inteligencia artificial, no es más que una relación de algoritmos que tienen como finalidad replicar las capacidades intelectuales humanas [1]. En este proyecto, se ha profundizado en especial en una de estas capacidades, el aprendizaje y la toma de decisiones. Estas dos capacidades humanas se intentan modelar dentro de la inteligencia artificial en las conocidas redes neuronales artificiales.

Por otra parte, la transcripción de textos manuscritos (Handwriting Text Recognition (HTR) en inglés), es una tarea que se lleva a cabo desde la antigüedad. El objetivo principal de la realización de esta tarea era preservar el conocimiento o información importante recogido en ciertos documentos, que se perdería en caso de que los documentos en papel desaparecieran por algún motivo. Esta labor se llevaba a cabo mediante los escribas desde tiempos documentados, y la continuaron durante la Edad Media los monjes en los llamados "scriptoriums". La invención de la imprenta tomó de una importancia claramente notoria. Por todo ello, que esta labor no puede quedar apartada tras la aparición de una tecnología que permita facilitar nuevamente la transcripción de obras.

En la actualidad, la gran mayoría, por no decir totalidad de los documentos, tanto de entretenimiento como de ciencias, se elaboran en formato digital, pero la realidad es que no todos los clásicos se encuentran en dicho formato. Es lógico qué a los grandes autores como Cicerón, Voltaire o Sócrates entre otros, les transcriban la mayoría de sus obras a formato digital. Aun así, quedan muchos documentos en papel, no digitalizados, que no tienen la popularidad o peso de estos autores pero que pueden ser muy interesantes a la hora de realizar diversos estudios. Por ejemplo, es posible que un albarán de un astillero del siglo XV no tenga mucho sentido para la gran mayoría de las personas, pero este puede ser de notoriedad para historiadores. Este es el punto por el cual se necesitan transcriptores automáticos que no conlleven un tiempo elevado.

La estructura de este documento se presenta de la siguiente manera. En primer lugar, se realiza una revisión del estado del arte de los modelos de transcripción de escritura manuscrita, presentando la inteligencia artificial, el machine learning y el OCR (Reconocimiento óptico de Caracteres, de sus siglas en inglés), particularizando el HTR (Reconocimiento de textos manuscritos). A continuación, se presenta el software con el que se realiza el proyecto: TensorFlow, y los modelos de machine learning que se utilizarán. Tras ello, se introduce la tecnología GPU, sobre la que nos hemos apoyado a la hora de realizar el proyecto. Más adelante, se explican diversos tipos de redes neuronales hasta llegar a los modelos de atención, que es la tecnología principal utilizada en este proyecto. En el capítulo de los modelos de atención, se presentan los modelos propuestos por Bahdanau [2] y Luong [3], así como la técnica Monotonic [4]. Tras presentar los modelos, se presentan los resultados obtenidos para un problema de traducción español-inglés, en el que se utilizan los nombrados modelos de atención.

También se presenta la forma en la que se verifican los resultados. A continuación, se presenta el problema de HTR y cómo la solución mediante modelos de atención que se ha aplicado al problema de traducción se puede adaptar a dicho problema, teniendo en cuenta sus similitudes y diferencias. Se describen los diversos modelos de CNN, Convolutional Neural Network, que se han aplicado y finalmente los resultados obtenidos.

2 INTRODUCCIÓN A LA TRANSCRIPCIÓN DE MANUSCRITOS

En esta sección se va a hacer un repaso de las tecnologías que permiten que hoy en día se se pueda llevar a cabo la transcripción automática de textos manuscritos. Comenzaremos hablando sobre la inteligencia artificial, para pasar presentar los métodos del machine learning y su implementación software para la ejecución en GPUs. Tras ello, se realizará una breve introducción a la transcripción de textos manuscritos a digitales.

2.1. Inteligencia Artificial

El término inteligencia artificial fue acuñado en 1956 [5], pero no ha sido hasta el siglo XXI cuando se ha hecho popular, gracias al aumento de los volúmenes de datos, capacidad de computación y el avance en algoritmos.

Las primeras investigaciones sobre inteligencia artificial en los años 50 exploraron temas como la resolución de problemas y los métodos simbólicos [6]. En la década de 1960, el Departamento de Defensa de EE.UU. se interesó en este tipo de trabajo y comenzó a entrenar a los ordenadores para imitar el razonamiento humano básico. Por ejemplo, la Agencia de Proyectos de Investigación Avanzada de la Defensa (DARPA) completó proyectos de mapeo de calles en la década de 1970 [7]. Y DARPA produjo asistentes personales inteligentes en 2003 [8], mucho antes de que Siri, Alexa o Cortana fueran nombres muy conocidos. Estos trabajos cimentaron el camino para la automatización y el razonamiento formal que vemos hoy en día, incluyendo sistemas de apoyo a la decisión y sistemas de búsqueda inteligente que pueden ser diseñados para complementar y aumentar las habilidades humanas.

2.2. Machine Learning

El Machine learning nace con el objetivo de solventar problemas en los que se tienen unas entradas y salidas conocidas, pero de los que se desconocen los algoritmos exactos a usar para predecir las salidas a partir de las entradas. La aplicación de esta tecnología se basa en problemas que poseen una cantidad ingente de datos para los que es conocida tanto la entrada como la salida, pero para los cuales no conocemos el proceso por el cual una salida es generada a partir de una entrada [9]. La solución que presenta el machine learning a este problema es la predicción a partir de la extracción de características y patrones comunes que se extraen de los datos conocidos. Esto métodos son conocidos cono *data minning* [10]. A parte de ser una solución a problemas con grandes bases de datos, el machine learning también es parte de la inteligencia artificial. Ser inteligente significa

tener capacidad para aprender y adaptarse a los cambios [11], esto permite que el programador no necesite considerar ni programar todas las posibilidades que presenta cada problema.

El machine learning se basa en la teoría de la probabilidad para crear modelos matemáticos que dan forma a algoritmos predictivos y descriptivos [9].

2.1.1 Tipos de Machine Learning

Los tipos de machine learning pueden categorizarse según la supervisión a la que se encuentran sometido en las siguientes clases [12]:

- **Supervisados**. Los algoritmos supervisados pueden aplicar lo aprendido a unos nuevos datos etiquetados. A partir del análisis de un conjunto de datos de entrenamiento conocido, el algoritmo de aprendizaje produce una función inferida para hacer predicciones sobre los valores de salida. El sistema es capaz de proporcionar predicciones para cualquier nueva entrada después de ser entrenado suficientemente. El sistema tiene que aprender las características claves, etiquetas, de cada entrada en el conjunto de datos para determinar la respuesta [12].
- **No supervisados**. Los algoritmos sin supervisión se utilizan cuando la información utilizada para entrenar no está clasificada ni etiquetada, es decir cuando no se conoce qué características de los datos de entrada son claves. El aprendizaje no supervisado estudia cómo los sistemas pueden inferir una función para describir un modelo implícito a partir de datos no etiquetados. El sistema no encuentra la salida correcta, pero explora los datos y puede sacar inferencias de los conjuntos de datos para describir estructuras ocultas a partir de datos no etiquetados. Podría decirse que extrae su propia salida.
- **Semisupervisados**. Los algoritmos semisupervisados son un punto intermedio entre el aprendizaje supervisado y el no supervisado, ya que utilizan tanto datos etiquetados como no etiquetados para el entrenamiento. Normalmente una pequeña cantidad de datos etiquetados y una gran cantidad de datos no etiquetados. Los sistemas que utilizan este método pueden mejorar considerablemente la precisión del aprendizaje. Por lo general, se elige el aprendizaje semisupervisado cuando los datos etiquetados adquiridos requieren recursos especializados y pertinentes para aprender de ellos. De lo contrario, la adquisición de datos no etiquetados generalmente no requiere recursos adicionales.
- **Aprendizaje por refuerzo**. Estos algoritmos conducen a un método de aprendizaje que interactúa con su entorno produciendo acciones y detectando errores y aciertos. La búsqueda de errores y la rectificación son sus características más relevantes. Este método permite que las máquinas y los agentes de software determinen automáticamente el comportamiento ideal dentro de un contexto específico para maximizar su rendimiento. Se requiere una simple retroalimentación de respuestas para que el agente aprenda cuál es la mejor acción; esto se conoce como la señal de refuerzo.

El machine learning permite el análisis de cantidades masivas de datos. Si bien por lo general ofrece resultados más rápidos y precisos para identificar oportunidades rentables o riesgos peligrosos, también puede requerir tiempo y recursos adicionales para capacitarla adecuadamente.

2.3. GPU

Dentro de los distintos métodos de machine learning, existe un subtipo llamado deep learning. Este subtipo de machine learning necesita de bastante más potencia, refiriéndonos con potencia a más capacidad de almacenamiento y computación, lo que ha llevado al uso de las GPU, para solventar las deficiencias de capacidades que comenzaban a vislumbrarse en las CPU convencionales.

En [13] se explica que, para un sistema orientado a inteligencia operacional, especialmente Deep learning, las CPU (Unidad Central de Procesamiento), por sí solas no son suficientes. Pueden hacer el procesamiento, pero el gran volumen de datos no estructurados que hay que analizar para construir y entrenar modelos de machine learning puede conllevar una ingente cantidad de tiempo que es difícilmente asumible en un proyecto, como es el caso del que se trata en la memoria. Incluso a las CPU de múltiples núcleos tienen capacidad limitada para ciertos modelos de Deep learning. Aquí que es donde entra la GPU (Unidad de Procesamiento Gráfico).

Las GPU son procesadores con núcleos de menor tamaño que las CPUs, pero mucho más lógicos; unidades

aritméticas lógicas (ALU), unidades de control y memoria caché; cuyo diseño básico consiste en procesar un conjunto de cálculos más sencillos e idénticos en paralelo [13].

En un principio su propósito principal era la representación gráfica de los juegos de ordenador. Posteriormente, las GPU fueron mejoradas para acelerar otros cálculos geométricos basados en matrices [14]. Las GPU son particularmente hábiles en los procesamientos de matrices, en los que se requiere la computación de muchos cálculos en paralelo. Las CPUs no tienen esta capacidad para paralelizar las operaciones a gran escala como las GPUs. Es esto lo que las hace ideales para aplicaciones especializadas como el machine learning. Además, se pueden concentrar muchos más núcleos de GPU especializados en la matriz de procesamiento que en una CPU.

Debido a que los núcleos utilizados por las GPU son altamente especializados, no pueden ejecutar un sistema operativo o manejar la lógica de la aplicación del núcleo, por lo que también son necesarias una o más CPU. Sin embargo, lo que estos sistemas sí pueden hacer es acelerar masivamente procesos como la formación de modelos de machine learning, descargando el procesamiento implicado de las CPU a todos esos núcleos del subsistema de la GPU.

2.4. HTR

Handwritten Text Recognition, HTR, es el concepto que engloba al OCR. Se define como la capacidad que posee una máquina para interpretar información manuscrita inteligible de todo tipo de fuentes, como documentos en papel, fotografías, pantallas táctiles y otros dispositivos. En [15], Ward, repasa la historia del reconocimiento de caracteres y categoriza los problemas de HTR de la siguiente forma:

- Online HTR. Con la aparición de los dispositivos táctiles, también han surgido la extracción de características para reconocer las letras que se escriben en tiempo real. Si estás escribiendo una letra A, por ejemplo, la pantalla táctil puede "sentir" que escribes primero una línea en ángulo, luego la otra, y luego la línea horizontal que las une. En otras palabras, el equipo se adelanta en el reconocimiento de las características porque las forma por separado, una tras otra, y eso hace que la extracción de características sea mucho más fácil que tener que elegir las características a partir de la manuscrita.
- Offline HTR. Los datos obtenidos de esta forma se consideran una representación estática de la escritura. Dentro de este tipo de reconocimiento de escritura destacan los motores de OCR [16], que se centran principalmente en el texto impreso a máquina, y en el ICR [17], para el texto manuscrito.

2.1.2 OCR

En [16] Bidyut repasa ampliamente el concepto y evolución del OCR. El OCR se basa en imitar la manera que tenemos las personas en procesar textos a través de las imágenese intentar modelar matemáticamente este proceso. Cuando una persona lee, sus ojos y su cerebro llevan a cabo un reconocimiento óptico de caracteres inconscientemente. Sus ojos están continuamente reconociendo patrones de luz que componen los caracteres; letras, números y otros caracteres especiales como signos de puntuación; y su cerebro los está procesando para averiguar lo que ese texto intenta transmitir. Esto lo realiza "escaneando" palabras enteras o grupos de palabras.

Esta manera de modelar la visión con la lectura y el reconocimiento de lo escrito, la podemos trasladar a los ordenadores, pero eso sí, añadiendo una carga de CPU bastante elevada. El primer problema es que un ordenador no tiene ojos, así que para que lea algo como la página de un libro, se le tiene que presentar una imagen de esa página generada con un escáner óptico o una cámara digital. La página que se crea de esta manera es un archivo gráfico, en formatos como JPG o PNG, y, en lo que respecta a una CPU, no difiere en nada a una fotografía de un paisaje, un edificio o un carnet, básicamente es un conjunto de píxeles completamente sin sentido, somos nosotros quienes debemos darles sentido a esas imágenes. En otras palabras, el ordenador tiene almacenada una imagen de la página en lugar del texto en sí, no es capaz de leer las palabras de la página como nosotros. El OCR es el proceso de convertir una imagen de texto en texto en sí mismo, es decir, producir algo como un archivo TXT o DOC a partir de un JPG escaneado de una página impresa o escrita a mano. Una vez que tenemos una página escrita en formato de imagen, es decir, un formato con el que una CPU puede interactuar, se abren posibilidades sobre el tratamiento de estas imágenes.

Las características más importantes que presenta Bidyut [16] a la hora de tener éxito al transcribir un texto escrito en texto digital se refieren a la propia imagen. Deben ser imágenes con una calidad aceptable; debe ser sencillo separar los caracteres del fondo; y deben evitarse caracteres cursivos e interconectados.

Pero, ¿cómo funciona el OCR? Si nos centramos tan sólo en un carácter, como por ejemplo la A, ya encontramos un problema bastante complejo, ya que cada persona escribe la letra A de una manera ligeramente diferente. Incluso con el texto impreso, hay un problema, porque los libros y otros documentos se imprimen en muchos tipos de fuentes diferentes y la letra A puede ser impresa en muchas formas sutilmente distintas.

En pocas palabras podemos decir que existen dos alternativas a la hora resolver este problema [18]. El primero de ellos, reconociendo los caracteres en su totalidad (reconocimiento de patrones); y el segundo detectando las líneas y trazos individuales de los que están hechos los caracteres (extracción de características) e identificándolos de esa manera:

- **Reconocimiento de patrones**

  Si la letra A tan sólo tuviese una manera única de escribirse, se podría conseguir que un ordenador la reconociera de manera sencilla. Sólo se tendría que comparar la imagen escaneada con una versión almacenada de la letra A y, si las dos coinciden, se catalogaría la entrada cono A. Pero como hemos dicho, la letra A tiene múltiples formas de escribirse. Para solucionar esto, en los años 60, se desarrolló una fuente especial llamada OCR-A [19] que se podía usar en cheques bancarios y demás. Cada letra tenía exactamente el mismo ancho; siendo esta un ejemplo de fuente monoespacio; y los trazos estaban cuidadosamente diseñados para que cada letra pudiera distinguirse fácilmente de todas las demás. Las impresoras de cheques fueron diseñadas para que todas usaran esa fuente, y el equipo de OCR fue diseñado para reconocerla. Al estandarizar una sola fuente, el OCR se convirtió en un problema relativamente fácil de resolver. El único problema es que la mayor parte de lo que se imprime en el mundo no está escrito en OCR-A [19]. Así que el siguiente paso fue enseñar a los programas de OCR a reconocer las letras escritas en una serie de las fuentes más comunes; como por ejemplo Times, Helvética, Courier, etc... Eso significaba que podían reconocer una gran cantidad de texto impreso, pero aun así no había garantía de que se reconociese cada carácter de manera adecuada.

- **Detección de características**

  También conocido como extracción de características o reconocimiento inteligente de caracteres ICR [17]. Esta es la forma más sofisticada de detectar caracteres. Si nos ponemos en la piel de un programa detector de caracteres, ¿en qué características nos centraríamos? Podríamos, por ejemplo, usar una regla como esta: "Si ves dos líneas angulares que se encuentran en un punto en la parte superior, en el centro, y hay una línea horizontal entre ellas a mitad de camino, eso es una letra A". de esta manera aplicaríamos esta regla y seriamos capaces de reconocer la mayoría de las letras mayúsculas sin importar en qué fuente están escritas. En lugar de reconocer el patrón completo de una A, estás detectando las características de los componentes individuales; líneas angulares; líneas cruzadas; zonas sin líneas; de las que está hecho el carácter. La mayoría de los programas modernos de OCR, los que pueden reconocer el texto impreso en cualquier fuente, funcionan mediante la detección de características en lugar del reconocimiento de patrones. Algunos utilizan redes neuronales, que es el ámbito del proyecto que se expone en esta memoria.

Reconocer los caracteres que componen un texto digital impreso con láser es relativamente sencillo en comparación con la decodificación de la escritura manual de una persona. Como hemos dicho anteriormente la escritura digital se rige por unas fuentes, que, aunque sean diversas, están acotadas y reglamentadas, por lo que un comparador sería suficiente para reconocerlas. En cambio, para los textos manuscritos es necesario utilizar una combinación de reconocimiento automático de patrones, extracción de características, y conocimiento sobre el escritor, y en este trabajo que se expone en la memoria el significado de lo que está escrito.

En los casos en que las CPUs tienen que reconocer la escritura manual de manera "industrializada" y en entornos de producción, el problema suele simplificarse. Por ejemplo, las CPUs que clasifican el correo por lo general sólo tienen que reconocer el código postal de un sobre, no la dirección completa. Así pues, sólo tienen que identificar una cantidad relativamente pequeña de texto formado sólo por letras y números básicos. Se solicita explícitamente a las personas a escribir los códigos de manera legible (dejando espacios entre los caracteres,

utilizando todas las letras mayúsculas) y, a veces, los sobres están preimpresos con pequeños cuadrados para que se escriban los caracteres para ayudar a mantenerlos separados.

Los formularios diseñados para ser procesados por OCR a veces tienen recuadros separados para que se escriba cada letra o unas pautas tenues, que animan a las personas a mantener las letras separadas y a escribir de forma legible.

3 SOFTWARE

Ya hemos introducido brevemente la inteligencia artificial, el machine learning y el deep learning, pero todas estas definiciones y postulados necesitan de herramientas para su implementación. Para realizar el proyecto que se expone en esta memoria nos hemos apoyado en el lenguaje Python, que provee un conjunto de librerías bastante completas debido a su actual popularidad.

3.1 TensorFlow

El aprendizaje de las máquinas es una disciplina no trivial y de procesos profundos y complejos. Si se tuviese que codificar desde cero un modelo de machine learning, sería un obstáculo que ralentizaría la evolución en dicho proyecto o investigación. Para solventar este obstáculo Google nos presenta Tensorflow [20], el cual nos facilita la implementación de modelos de machine learning, así como el proceso de adquisición de datos, los modelos de entrenamiento, las predicciones y el perfeccionamiento recursivo de los resultados futuros.

Creado por el equipo de Google Brain, TensorFlow realmente se autodefine como una biblioteca de código abierto para la computación numérica y el aprendizaje de máquinas a gran escala. TensorFlow agrupa una serie de modelos y algoritmos tanto de machine learning como de deep learning y los hace útiles mediante una interfaz común. Utiliza Python para proporcionar una conveniente API de front-end para la construcción de aplicaciones con el framework, mientras ejecuta esas aplicaciones en C++ de alto rendimiento.

TensorFlow puede entrenar y ejecutar redes neuronales profundas para la clasificación de dígitos manuscritos, reconocimiento de imágenes, tratamiento de frases, redes neuronales recurrentes, modelos sequence-to-sequence para la traducción automática, procesamiento de lenguaje natural y simulaciones basadas en PDE (ecuaciones diferenciales parciales). Lo que le hace diferencial es que TensorFlow soporta la predicción de la producción a escala, con los mismos modelos utilizados para la formación.

3.1.1 Características de Tensorflow

TensorFlow permite a los desarrolladores crear modelos de flujo de datos estructurados que describen cómo datos que entran se mueven a través de un modelo, o una serie de nodos de procesamiento. Cada nodo del modelo representa una operación matemática, y cada conexión entre nodos es una matriz de datos multidimensional, también denominada tensor.

Otro de los puntos interesantes es que proporciona todo esto para el programador a través del lenguaje Python. Python es un lenguaje fácil de aprender y trabajar, y proporciona formas convenientes de expresar cómo las abstracciones de alto nivel pueden ser acopladas. Además, Python se está extendiendo de manera bastante rápida lo que proporciona universalidad a los desarrollos de Tensorflow. Los nodos y tensores en TensorFlow son objetos Python, y las aplicaciones de TensorFlow son en sí mismas aplicaciones Python. Por el contrario, las

operaciones matemáticas reales no se realizan en Python. Las librerías de transformaciones que están disponibles a través de TensorFlow están escritas como binarios C++ de alto rendimiento. Python sólo dirige el tráfico entre las piezas, y proporciona abstracciones de programación de alto nivel para unirlas.

Las aplicaciones de TensorFlow pueden ejecutarse en casi cualquier arquitectura, ya sea esta una máquina local, o un potente clúster en la nube, dispositivos iOS y Android, CPUs o GPUs.

3.1.2 Beneficios de Tensorflow

El mayor beneficio que proporciona TensorFlow para el desarrollo del aprendizaje de la máquina es la abstracción. En lugar de tratar los detalles esenciales de la implementación de algoritmos, o de encontrar formas adecuadas de correlar la salida de una función a la entrada de otra, el desarrollador puede centrarse en la lógica general de la aplicación. TensorFlow se ocupa de los detalles internos.

Ahora bien, ¿por qué utilizar Tensorflow frente a sus competidores? Sabemos que TensorFlow compite con otros marcos de aprendizaje máquinas como por ejemplo PyTorch, CNTK, y MXNet, que dan solución a muchas de las mismas necesidades.

En primer lugar, PyTorch, además de ser construido con Python, también comparte otras características con TensorFlow, como los componentes acelerados por hardware. La principal diferencia es que PyTorch no ofrece el mismo nivel de abstracción que TensorFlow, lo que hace que su programación suela conllevar más tiempo. De esta manera, TensorFlow se muestra superior para proyectos más grandes y flujos de trabajo más complejos.

En segundo lugar, CNTK, que es el kit de herramientas cognitivas de Microsoft. Al igual que TensorFlow utiliza una estructura de modelos para describir el flujo de datos. CNTK maneja muchos trabajos de redes neuronales más rápido, y tiene un conjunto más amplio de APIs, Python, C++, C# o Java. Pero no es actualmente tan fácil de aprender o desplegar como TensorFlow que se encuentra en estos momentos bastante más extendido y con más librerías disponibles.

Por último, Apache MXNet, adoptado por Amazon como el principal marco de machine learning en AWS, puede escalar casi linealmente a través de múltiples GPUs y máquinas. También soporta una amplia gama de lenguajes API-Python, C++, Scala, R, JavaScript, Julia, Perl, Go, etc. Pero su desventaja reside en que sus APIs nativas no son tan amables para el usuario como las de TensorFlow.

3.2 Keras

Keras es una API de redes neuronales de alto nivel. Está escrita en Python y tiene la posibilidad de trabajar junto a TensorFlow, CNTK o Theano. El objetivo sobre el que se empezó a desarrollar fue otorgar de agilidad a las experimentaciones con redes neuronales, ya que uno de los puntos clave de una buena investigación es la capacidad de testar las hipótesis que ayuden a confirmarlas o descastarlas lo más rápido posible.

Hoy en día, Keras se utiliza principalmente como biblioteca de deep learning, y para la creación de modelos de manera fácil y rápida de prototipos, permitiendo su ejecución tanto en CPUs como en GPUs, lo que proporciona un plus al proyecto que se expone en esta memoria.

4 REDES NEURONALES

Los modelos de atención han demostrado que producen resultados de vanguardia en los problemas de traducción automática [21], así como en otras tareas de procesamiento del lenguaje. Estos avances aparecen cuando se combinan los modelos de atención con la incorporación de redes neuronales y están estableciendo nuevos y mejores resultados de precisión en la PNL. De esta manera los modelos de atención forman parte de los mejores resultados que se han obtenido a la hora de "programar" una compresión real del lenguaje humano por parte de las máquinas.

4.1 Redes neuronales totalmente conectadas

Las redes neuronales totalmente conectadas son también conocidas como Multi-Layered Perceptrons (MLPs), redes multicapa de perceptrones [22]. Tomemos una de estas redes conectadas, como un perceptrón multicapa con sus capas totalmente conectadas. Una red de alimentación trata todas las características de entrada como únicas e independientes unas de otras, y de manera discreta.

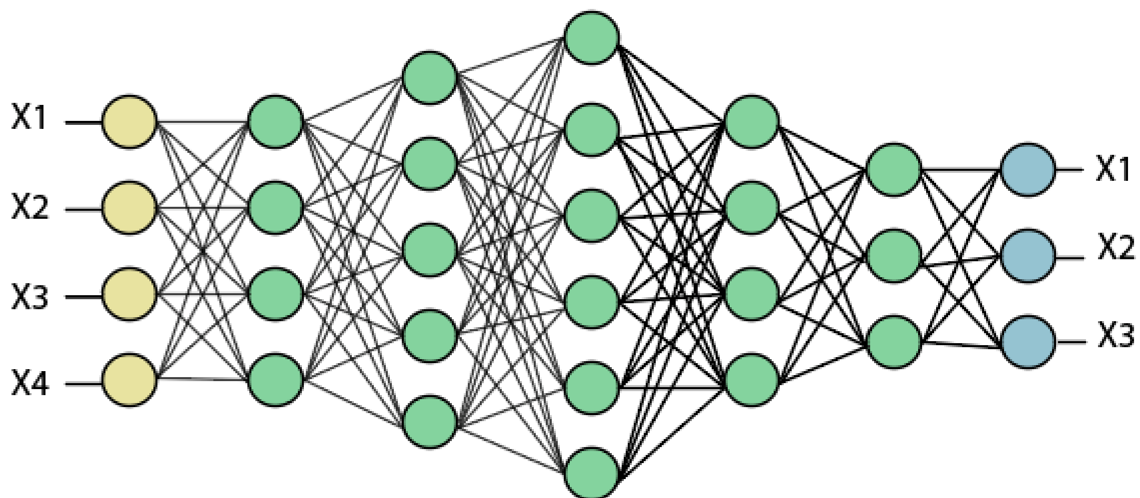

Figura 1. Red de perceptrones [23]

Así pues, si tomamos entonces todas las entradas como independientes, cada característica individual no puede inferir automáticamente sobre otra característica que se sitúe "justo al lado", por tanto, se puede decir que la proximidad no significa mucho, no aporta valor. Por ejemplo, imaginemos que queremos clasificar que tipo de

coche compra una persona según sus características. Que un comprador tenga como característica la profesión de albañil y lugar de residencia una ciudad costera, son entradas que aportan individuamente, y no en conjunción. Este tipo de soluciones son de utilidad para datos como este, pero no tan útiles en los casos en los que hay una estructura subyacente, local a los datos.

4.2 Redes convolucionales

Las redes neuronales convolucionales son redes neuronales utilizadas principalmente para extraer características de imágenes [24]. Las salidas de estas redes suelen conectarse a clasificadores como MLP, que clasifica dichas imágenes, agrupándolas por similitud, o realizando reconocimiento de objetos dentro de las escenas. Por ejemplo, las redes neuronales convolucionales como ConvNets o CNNs se utilizan para identificar rostros, individuos, señales de tráfico, tumores y muchos otros aspectos de los datos visuales.

La eficacia de las redes convolucionales en el reconocimiento de imágenes es una de las principales razones por las que el mundo ha despertado el interés sobre del deep learning. Las CNN están impulsando grandes avances en la visión por ordenador, que tiene aplicaciones obvias para los coches que se conducen solos, la robótica, los aviones no tripulados, la seguridad, los diagnósticos médicos y los tratamientos para los discapacitados visuales.

Las redes convolucionales también pueden realizar tareas más banales, y al mismo tiempo más rentables, y orientadas a los negocios, como el reconocimiento óptico de caracteres (OCR) para digitalizar textos y hacer posible el procesamiento en lenguaje natural en documentos analógicos y escritos a mano, donde las imágenes son símbolos que deben transcribirse, que es el objetivo de este proyecto.

Además, las CNNs también se han aplicado directamente a la analítica de textos. Y se aplican al sonido cuando se representa visualmente como un espectrograma graficando los datos con redes convolucionales de gráficos [25].

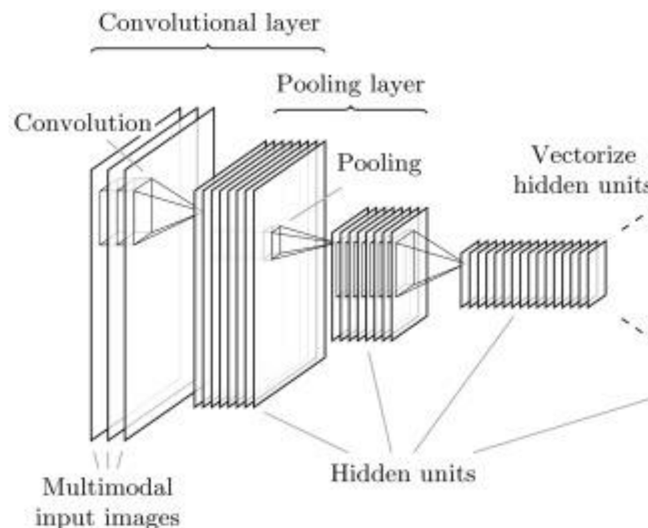

Figura 4-2. Red convolucional [26]

4.3 Redes recurrentes y LSTM

Si quisiéramos parametrizar una conversación de voz, nos percataríamos de que se presenta de manera bidimensional. Siendo una dimensión las palabras de dicha conversación y la segunda el tiempo.

Si comparamos este sistema con las imágenes, vemos que una imagen puede ser vislumbrada en su totalidad en un instante. La mayoría de las imágenes contienen datos tridimensionales, como mínimo. Si se considera que cada color principal es su propia dimensión, y la ilusión de profundidad, entonces la imagen contiene muchos más que dos dimensiones. Y si se trata de vídeo, añadimos también la dimensión del tiempo.

Una red neuronal que se mostró muy prometedora en el procesamiento de palabras es la red neuronal recurrente (RNN), en particular una de sus variantes, la red de memoria a largo plazo (LSTM).

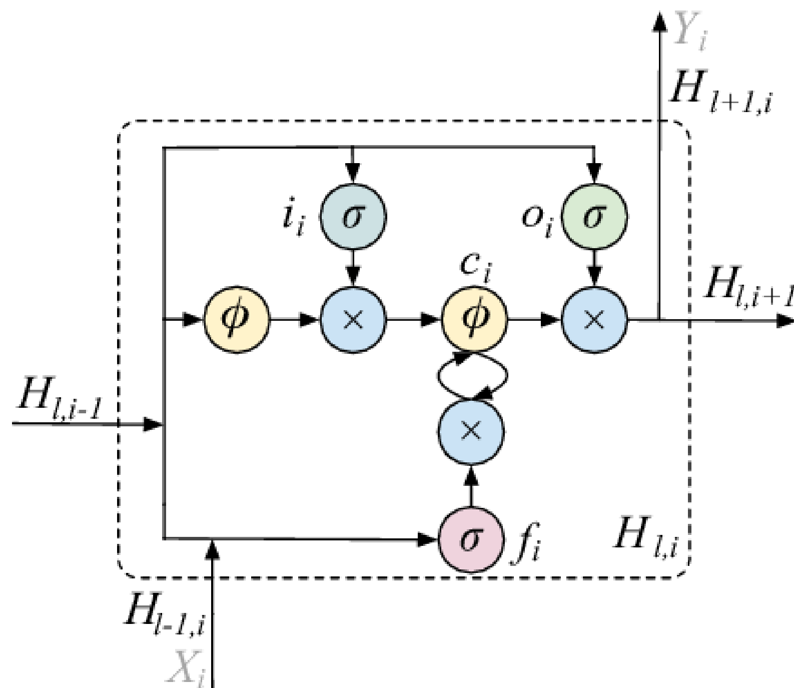

Figura 4-3. Celda individual de LSTM [27]

Para introducir las RNNs, nos vamos a apoyar en un problema concreto, la aplicación a oraciones. De esta manera, las RNN procesan el texto como una máquina quitanieves que va por un camino en una dirección, codificando las palabras en vectores de contexto. Todo lo que saben es el camino que han despejado hasta ahora, solo conocen lo que han visto, lo que se les ha permito ver. El camino que tienen por delante es incierto, es decir, el final de la frase es totalmente confuso y no les da ninguna información adicional. Y el pasado remoto probablemente ya esté un poco nevado; es decir, si se trata de una frase muy larga, probablemente ya han olvidado partes de ella, esto se debe al desvanecimiento del gradiente [28].

En un problema de predicción de frases, las RNNs predicen las palabras que encuentran en función de las palabras que han encontrado antes. Así, las RNNs tienden a tener mucha más información para hacer buenas predicciones cuando llegan al final de una frase que la que tienen al principio. Esto se debe a que se basan en un contexto más amplio a la vez que avanzan en la frase. El problema aparece cuando parte del contexto necesario para predecir la siguiente palabra se encuentra más adelante, como por ejemplo en idiomas en los que las interrogaciones y exclamaciones se marcan al final de las frases. Esto hace que su rendimiento no sea el ideal.

Las RNNs también tienen un problema de memoria. No pueden recordar todo sobre las dependencias de largo alcance. Por tanto, las RNNs ponen demasiado énfasis en que las palabras estén cerca una de la otra, y más énfasis en el contexto ascendente que el contexto descendente.

Podemos aprovechar esta solución y mejorarla con los mecanismos de atención.

4.4 Mecanismos de atención

El mecanismo de atención toma dos frases, o vectores, y las convierte en una matriz donde las palabras de una frase forman las columnas, y las palabras de la otra frase forman las filas. Una vez se tiene esta matriz, se hacen coincidir las palabras, elementos, identificando el contexto relevante y creando los vectores de contexto. Este tipo de modelos se utiliza tradicionalmente para resolver el problema de traducción automática, que es parte del proyecto que se expone en esta memoria [21].

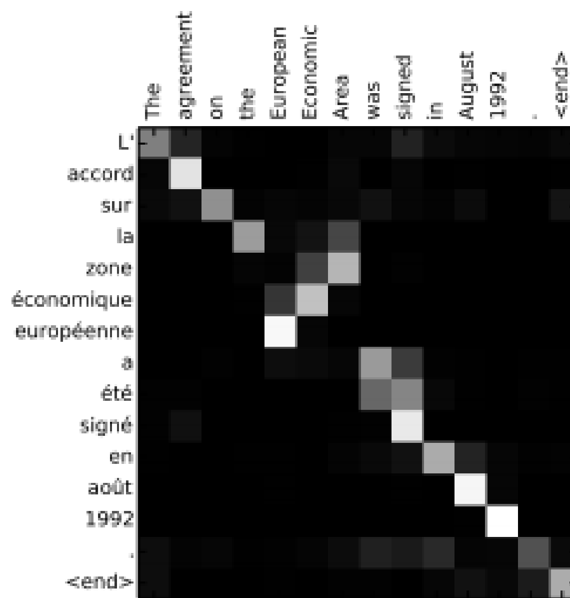

Figura 4-4. Matriz de pesos entre entrada y salida [29]

Una ventaja de los modelos de atención es que podemos ir más allá, es decir, tienen aplicación en diversos problemas. Una de las soluciones que aporta este método no es sólo utilizar la atención para correlacionar el significado de las frases en dos idiomas diferentes y así obtener una traducción con contexto, sino que también se puede comparar una frase con ella misma, para entender cómo algunas partes de esa frase se relacionan con otras. Por ejemplo, podemos utilizar la atención para predecir los antecedentes de los pronombres. Esta solución se denomina "auto-atención" [30], aunque es tan común que también es conocida simplemente como atención.

Una red neuronal armada con un mecanismo de atención puede realmente entender a qué se refiere una oración específicamente. Es decir, sabe cómo ignorar el ruido y centrarse en lo que es relevante, cómo conectar dos palabras relacionadas que en sí mismas no llevan marcadores que se señalen mutuamente.

Así que la atención permite observar la totalidad de una frase, para hacer conexiones entre cualquier palabra en particular y su contexto relevante. Esto aporta una ventaja frente a las RNNs, que tenían memoria corta y con un enfoque en líneas, y también frente a las redes convolucionales enfocadas en la proximidad de las características.

El lenguaje es este conjunto bidimensional que de alguna manera logra expresar las relaciones inherentes a la vida en muchas dimensiones (tiempo, espacio, colores, causalidad), pero sólo puede hacerlo mediante la creación de vínculos sintácticos entre las palabras que no están inmediatamente próximas entre sí en una frase.

La atención permite viajar a través de la sintaxis de una oración para identificar las relaciones con otras palabras que están lejos, mientras que ignora otras palabras que no tienen mucha relación con la palabra sobre la que se intenta hacer una predicción.

Las palabras son construcciones sociales y así, como sus constructores, derivan gran parte de su significado en sus relaciones con otras palabras.

Uno de los límites de los vectores de palabras tradicionales es que presumen que el significado de una palabra es relativamente estable a través de las oraciones. Esta afirmación no es correcta, al menos no del todo. La polisemia se encuentra presente en las oraciones, y debemos tener cuidado con las enormes diferencias de significado de una sola palabra: por ejemplo, lit (un adjetivo que describe algo ardiente) y lit (una abreviatura de literatura) y lit (jerga para intoxicado); o get (un verbo para obtener) y get (la descendencia de un animal).

Aunque proyectos como WordNet [31] y algoritmos como sense2vec [32] son grandes intentos de desentrañar

los significados de todas las palabras en inglés, hay muchos matices en el significado de una palabra dada que sólo se resuelve en relación con su entorno. Es decir, ni siquiera puedes contar todos los sentidos de una palabra en un diccionario. Los matices más finos de significado surgen in situ.

De esta manera, es necesario ampliar el dominio de las entradas que un algoritmo considera, y para ello debemos incluir más información sobre el contexto de una palabra dada. Lo realmente útil del álgebra lineal es que se pueden calcular muchas relaciones a la vez; en este caso, estamos calculando las relaciones de cada palabra de una frase con el resto de palabras de la misma oración, auto-atención, y expresando esas relaciones variables que sugieren el significado de una palabra como un vector. Un vector de contexto. Va más allá de los vectores de palabras y vectores de sentido. Tenemos que vectorizar todas las cosas. Y podemos hacerlo con el mecanismo de atención.

En la auto-atención, se puede hablar de la atención que las palabras se prestan entre sí dentro de una frase. Para cualquier palabra dada, buscamos cuantificar el contexto que la oración provee, e identificar qué otras palabras proveen el mayor contexto con respecto a la palabra en cuestión. Los arcos dirigidos de un gráfico de dependencia semántica pueden darle una intuición de cómo las palabras se conectan entre sí a través de una oración completa, algunas se elevan por encima de otras para ayudar a definir a algunos de sus semejantes.

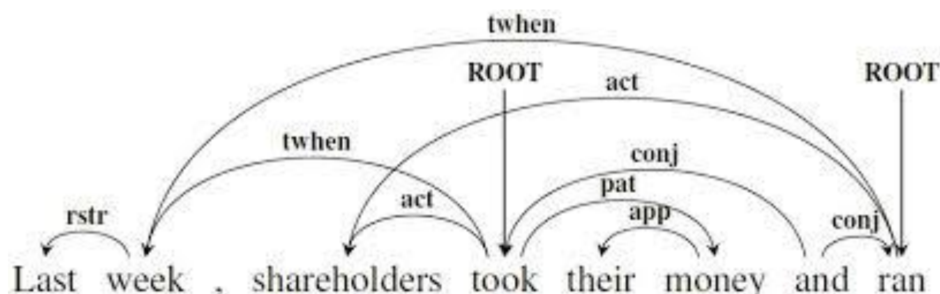

Figura 4-5. Relación entre palabras [29]

Lo que hace más interesante esta solución es cómo las relaciones en un vector de contexto pueden cambiar nuestra comprensión de las frases. Esto

Porque las oraciones no son triviales. Normalmente las oraciones nos han hecho pensar que su significado es lineal, que se despliega como una cinta de impresión a través de la página. Esto no es así. En cualquier frase, algunas palabras tienen una fuerte relación con otras palabras que no se encuentran a su lado. De hecho, las relaciones más fuertes que unen una palabra dada con el resto de la oración pueden ser con palabras bastante distantes de ella.

Así que probablemente sea más adecuado pensar en frases que se pliegan como características en un espacio tridimensional, con una parte de una frase relacionándose con otra parte con la que tiene una afiliación particularmente fuerte.

Cuando consideramos las frases como objetos multidimensionales que se retuercen sobre sí mismos, de repente su capacidad de cartografiar la sintaxis de los objetos reales en el espacio-tiempo (cuyas relaciones causales, correlaciones e influencias también pueden ser remotas, o invisibles) parece más intuitiva, y también se alinea con ciertos enfoques de la neurociencia y la geometría del pensamiento. De hecho, la atención puede ayudarnos a entender las interrelaciones de los objetos en una imagen, así como nos ayuda con el procesamiento del lenguaje natural.

4.5 Transformación

En un principio se utilizó la atención además de otros modelos, como las RNNs o los CNN, se ha comprobado que funciona muy bien por sí solo, sin necesidad de añadidos. Combinadas con capas de alimentación, las

unidades de atención pueden ser simplemente apiladas, para formar codificadores.

Los transformadores utilizan mecanismos de atención para reunir información sobre el contexto pertinente de una palabra dada, y luego codifican ese contexto en el vector que representa la palabra. Así que, en cierto sentido, la atención y los transformadores tienen que ver con representaciones más inteligentes.

Las redes de retroalimentación tratan las características como independientes (género, hermanos); las redes convolucionales se centran en la ubicación relativa y la proximidad; las RNN y las LSTM tienen limitaciones de memoria y tienden a leer en una dirección. En contraste con éstas, la atención y el transformador pueden captar el contexto sobre una palabra de partes distantes de una oración, tanto antes como después de que la palabra aparezca, a fin de codificar la información que nos ayude a comprender la palabra y su función en el sistema llamado oración.

Hay varias arquitecturas para los transformadores. Una de ellas utiliza almacenamientos de valor clave y una forma de memoria.

Si bien la atención se considera típicamente como un mecanismo orientador de la percepción, también puede enfocarse internamente, hacia el contenido de la memoria. Esta idea ha inspirado trabajos en los que se han utilizado algunas arquitecturas de mecanismos de atención para seleccionar la información que se leerá en la memoria interna de la red. Esto ha contribuido a los recientes éxitos en la traducción automática, Bahdanau [2], y ha dado lugar a importantes avances en las tareas de memoria y razonamiento, Graves [33]. Estas arquitecturas ofrecen una novedosa implementación de la recuperación de contenido direccionable [34].

En esta arquitectura, se tiene una clave, un valor y una consulta de búsqueda. La consulta busca sobre las claves de todas las palabras que podrían proporcionarle un contexto. Esas claves están relacionadas con valores que codifican más significado sobre la palabra clave. Cualquier palabra dada puede tener múltiples significados y relacionarse con otras palabras de diferentes maneras, puede tener más de un complejo consulta-valor clave adjunto a ella. Eso es "atención de múltiples cabezas".

Una cosa que hay que tener en cuenta es que la relación de las consultas con las claves y las claves con los valores es diferenciable. Es decir, un mecanismo de atención puede aprender a remodelar la relación entre una palabra de búsqueda y las palabras que proporcionan contexto a medida que la red aprende.

5 MODELOS DE ATENCIÓN

En esta sección se va a profundizar en los diversos modelos de atención que se van a utilizar para dar solución a los problemas tanto de traducción como de transcripción. En total van a ser tres modelos, aunque se realizarán también derivados de ellos mediante la unión de diversos métodos, ya que uno de ellos, el monotonic, no es excluyente, debido a que, como veremos más adelante, tiene una concepción distan a los demás modelos.

5.1 Bahdanau

La traducción automática a través de redes neuronales trata de construir y entrenar una única y gran red neuronal que lea una frase y produzca una traducción correcta.

La mayoría de los modelos propuestos de traducción automática neuronal pertenecen a una familia de codificadores-decodificadores, con un codificador y un decodificador para cada idioma, o implican un codificador específico del idioma aplicado a cada frase cuyas salidas se comparan posteriormente. Una red neural de codificadores lee y codifica una frase fuente en un vector de longitud fija. Por su parte, el decodificador produce una traducción a partir del vector codificado.

En todos los sistemas codificador-decodificador, que consisten en la codificación y descodificación de dos idiomas, se entrenan estos conjuntamente para maximizar la probabilidad de una traducción correcta dada una frase de origen.

Un problema potencial de este enfoque de codificador-decodificador es que una red neuronal debe ser capaz de comprimir toda la información necesaria de una oración de entrada en un vector de longitud fija. Esto puede dificultar que la red neuronal pueda hacer frente a las frases largas. Es fácil comprobar que el rendimiento de un codificador-decodificador básico se deteriora rápidamente a medida que aumenta la longitud de una frase dada como entrada.

Para abordar esta cuestión, el modelo de Bahdanau [2] introduce una ampliación del modelo codificador-decodificador que aprende a alinear y traducir conjuntamente. Cada vez que el modelo propuesto genera una palabra en una traducción, se busca un conjunto de posiciones en una frase donde se concentra la información más relevante. El modelo predice entonces una palabra objetivo basándose en los vectores de contexto asociados con estas posiciones fuente y todas las palabras objetivo generadas anteriormente.

La característica más importante que distingue este enfoque del codificador-decodificador básico es que no intenta codificar una frase de entrada completa en un único vector de longitud fija. En lo que se apoya es en la codificación la oración de entrada en una secuencia de vectores y elige un subconjunto de estos vectores de manera adaptativa mientras decodifica la traducción. Esto libera al modelo de traducción de tener que comprimir

toda la información de una oración de origen, independientemente de su longitud, en un vector de longitud fija.

Bahdanau muestra que el enfoque propuesto de aprender conjuntamente a alinear y traducir logra una mejora significativa en el rendimiento de la traducción con respecto al enfoque básico de codificador-decodificador. La mejora es más evidente con las frases más largas, pero puede observarse con las frases de cualquier longitud.

5.1.1 Codificador-Decodificador RNN

Si nos sumergimos en el modelo Codificador-Decodificador, un codificador lee la frase de entrada, una secuencia de vectores $x = (x_1, \ldots, x_{T_x})$, en un vector c. El enfoque más común es usar un RNN de tal forma que:

$$h_t = f(x_t, h_{t-1}) \quad \text{5-1}$$

$$c = q(\{h_1, \ldots, h_{T_x}\}) \quad \text{5-2}$$

donde $h_t \in R^n$ es un estado oculto en el tiempo t, y c es un vector generado a partir de la secuencia de los estados ocultos. f y q son funciones no lineales. Sutskever [35] usaron una LSTM como f y $q(\{h_1, \ldots, h_T\}) = h_T$, por ejemplo.

El decodificador es normalmente entrenado para predecir la siguiente palabra y_t dado el vector de contexto c y todas las palabras previamente predichas $\{y_1, \ldots, y_{t'-1}\}$. En otras palabras, el decodificador define una probabilidad sobre la traducción y descomponiendo la probabilidad conjunta en los condicionales ordenados:

$$p(y) = \prod_{t=1}^{T} p(y_t | \{y_1, \ldots, y_{t-1}\}, c) \quad \text{5-3}$$

donde $y = y_1, \ldots, y_{T_y}$. Con una RNN, cada probabilidad condicional se modela como

$$p(y_t | \{y_1, \ldots, t_{t-1}\}, c) = g(y_{t-1}, s_t, c) \quad \text{5-3}$$

donde g es una función no lineal, potencialmente de varias capas, que produce la probabilidad de y_t, y s_t es el estado oculto del RNN. Cabe señalar que se pueden utilizar otras arquitecturas como un híbrido de un RNN y una red neural de convolucional.

5.1.2 Decodificador

En la arquitectura que propone Bahdanau, se define cada probabilidad condicional como:

$$p(y_i | y_1, \ldots, y_{i-1}, x) = g(y_{i-1}, s_i, c_i) \quad \text{5-4}$$

donde s_i es un estado oculto de RNN para el tiempo i, calculado a partir de:

$$s_i = f(s_{i-1}, y_{i-1}, c_i) \quad \text{5-5}$$

Se observa que, a diferencia del enfoque codificador-decodificador, en esta probabilidad está condicionada a una vector de contexto c_i para cada palabra de destino y_i .

El vector de contexto c_i depende de una secuencia de pesos $(h_i, \dots, h_{T_x})$ a la que un codificador mapea la frase de entrada. Cada peso h_i contiene información sobre toda la secuencia de entrada con un enfoque en las partes que rodean la $i - ésima$ palabra de la secuencia de entrada.

El vector de contexto c_i se calcula como una suma ponderada de estos pesos h_i:

$$\alpha_{ij} = \frac{\exp\,(e_{ij})}{\sum_{k=1}^{T_x} \exp\,(e_{ik})} \quad \text{5-6}$$

Donde:

$$e_{ij} = a(s_{i-1}, h_j) \quad \text{5-7}$$

es un modelo de alineación que puntúa como correctas la coincidencia de las entradas alrededor de la posición j y la salida en la posición i. La puntuación se basa en el estado oculto de RNN s_{i-1} (justo antes de emitir y_i, (5-4) y $la\ j - ésima$ anotación $h_j$de la frase de entrada.

Se parametriza el modelo de alineación como una red neuronal de avance que se entrena conjuntamente con todos los demás componentes del sistema propuesto. A diferencia de la traducción automática tradicional, la alineación no se considera una variable latente. En su lugar, el modelo de alineación calcula directamente una alineación suave, lo que permite que el gradiente de la función de costo sea retro propagado. Este gradiente puede ser usado para entrenar el modelo de alineación, así como todo el modelo de traducción conjuntamente.

Podemos entender el enfoque de tomar una suma ponderada de todos los pesos como la computación de un peso esperado, donde la expectativa está por encima de las posibles alineaciones. Se deja que α_{ij} sea la probabilidad de que la palabra objetivo y_i está alineada, o traducida, con una palabra de origen x_j . Entonces, el vector de contexto $i - ésimo$ c_i es la anotación esperada sobre todas las anotaciones con probabilidades α_{ij}.

La probabilidad α_{ij}, o su energía asociada e_{ij}, refleja la importancia de un peso h_j con respecto al anterior estado oculto s_{i-1} al decidir el siguiente estado s_i y generar y_i. Intuitivamente, esto implementa un mecanismo de atención en el decodificador. El decodificador decide a que partes de la oración de entrada hay que prestar atención. Al dejar que el decodificador tenga un mecanismo de atención, aliviamos la de la carga de tener que codificar toda la información de la frase fuente en un vector de longitud fija. Con este nuevo enfoque la información puede ser difundida a lo largo de la secuencia de pesos, que pueden ser recuperadas selectivamente por el decodificador en consecuencia.

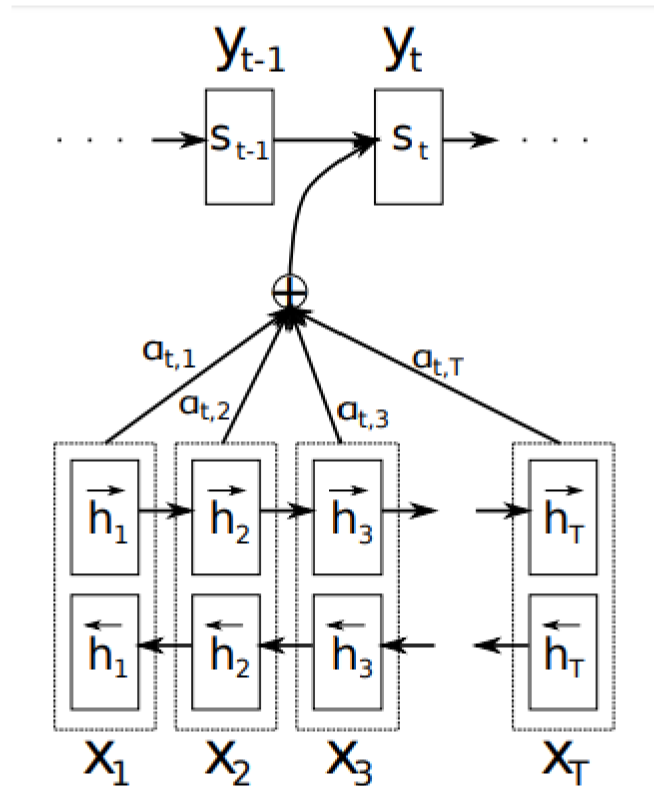

Figura 5-6. Modelo de predicción de y_y [2]

5.1.3 Codificador (RNN BIDIRECCIONAL PARA SECUENCIAS)

Normalmente el codificador RNN lee una secuencia de entrada x en orden desde el primer símbolo x_1 hasta el último x_{T_x}. Sin embargo, en el esquema que nos propone el modelo de Bahdanau, aparece que el peso de cada palabra esconde no sólo las palabras precedentes, sino también las siguientes. Por lo tanto, propone utilizar un RNN bidireccional que se ha utilizado normalmente con problemas de reconocimiento de voz.

Un BiRNN consiste en varios RNNs hacia adelante y hacia atrás. El RNN $\vec{f}$ hacia adelante lee la secuencia de entrada como se ordena (de x_1 a x_{T_x}) y calcula una secuencia de estados ocultos hacia adelante ($\vec{h_1}$, …, $\vec{h_{T_x}}$). El RNN $\overleftarrow{f}$ hacia atrás lee la secuencia en el orden inverso (de x_{T_x} a x_1), resultando en una secuencia de estados ocultos hacia atrás ($\overleftarrow{h_1}$, …, $\overleftarrow{h_{T_x}}$).

De esta forma se obtiene un peso para cada palabra $x_j$concatenando el estado oculto hacia adelante $\vec{h_j}$ y hacia atrás $\overleftarrow{h_j}$, es decir, h_j, i.e., $h_j = [\vec{h_j^T}, , \overleftarrow{h_j^T}]$. De esta manera, la anotación h_j contiene la codificación tanto de las palabras precedentes como de las siguientes. Debido a la tendencia de los RNN a representar mejor las entradas recientes, la anotación h_j se centran en las palabras alrededor de x_j.

5.2 Luong

Por su parte, el modelo de Luong [3] nos presenta dos maneras simples y efectivas del modelo de atención, la primera de ellas con un enfoque global que siempre atiende a todas las palabras de origen y la segunda con un enfoque local que sólo mira un subconjunto de palabras de origen por cada iteración. Luong demuestra la efectividad de ambos enfoques en las tareas de traducción de en ambas direcciones. Con estos métodos. Luong consigue una ganancia significativa de 5,0 puntos BLEU sobre los sistemas que no implementan modelos de atención.

5.2.1 Atención Global

La idea sobre la que pivota un modelo de atención global es la de considerar todos los estados ocultos del codificador. Esto se consigue al derivar el vector de texto c_t. En este tipo de modelo, un vector de alineación de longitud variable a, cuyo tamaño es igual al número de pasos de tiempo en el lado de la fuente, se deriva comparando el estado oculto actual del objetivo h_t con cada estado oculto de la fuente $\overline{h_s}$:

$$a_t(s) = align(h_t, \overline{h_s}) = \frac{\exp\left(score(h_t, \overline{h_s})\right)}{\sum_{s'} \exp\left(score(h_t, \overline{h_{s'}})\right)} \tag{5-8}$$

Aquí, los pesos se refieren a una función basada en el contenido para la cual consideramos tres entornos diferentes, como se muestran en la siguiente ecuación:

$$score(h_t, \overline{h_s}) = \begin{cases} h_t^T, \overline{h_s} & dot \\ h_t^T W_a h_s & general \\ v_a^T \tanh\left(W_a\left[h_t, \overline{h_s}\right]\right) & concat \end{cases} \tag{5-9}$$

Además, en este primer modelo Luong [3] usa una función basada en la localización en la que los pesos de alineación se calculan únicamente a partir del estado oculto del objetivo h_t como aparece en la ecuación 5-10.

$$a_t = softmax(W_{ah_t}) \tag{5-10}$$

Dado el vector de alineación como pesos, el vector c_t se calcula como el promedio ponderado sobre todos los estados de origen ocultos. En comparación con Bahdanau [2] el enfoque de Luong [3] en la atención global hay varias diferencias clave que reflejan cómo simplifican y generalizan desde el modelo original. En primer lugar, simplemente usan estados ocultos en las capas superiores del LSTM tanto en el codificador como en el decodificador como se ilustra en la figura 7.

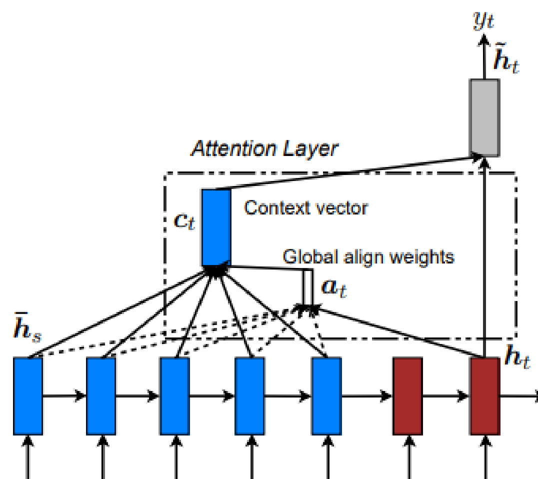

Figura 7. Modelo de atención Global [20]

En el modelo de atención propuesto por Bahdanau [2], sin embargo, se utiliza la concatenación hacia delante y hacia atrás de los estados ocultos de la fuente en el codificador bidireccional y apuntar a estados ocultos en su decodificador unidireccional no apilable. En segundo lugar, las rutas de cálculo propuestas por Luong [3] son más simples; se va desde $h_t \rightarrow a \rightarrow c_t \overline{h_t}$ y luego se realiza una predicción como la detallada en la figura 7.

5.2.2 Atención Local

La atención global presenta el inconveniente de que tiene que atender a todas las palabras de la oración de entrada para cada palabra de la oración de salida, lo cual es costoso en tiempo y computación y puede hacer que sea poco práctico traducir secuencias largas, por ejemplo, párrafos o documentos. Para subsanar esta deficiencia, Luong propone un mecanismo de atención local que opta por centrarse sólo en un pequeño subconjunto de las posiciones de origen por cada palabra de destino.

Este modelo se inspira en el equilibrio entre los modelos de atención suave y dura propuestos por Xu [4] para abordar la tarea de generación de pies de foto. En su trabajo, la atención blanda se refiere al enfoque de atención global en el que se colocan pesos "blandos" sobre todos los partes de la imagen origen. La atención dura, por otro lado, selecciona una zona de la imagen para computarla a la vez. Aunque es menos costoso en el momento de la inferencia, el modelo de atención dura requiere técnicas más complicadas, como la reducción de la varianza o el refuerzo de aprender a entrenar.

El mecanismo de atención local se centra selectivamente en una pequeña ventana de contexto y es diferenciable. Este enfoque tiene la ventaja de evitar el costoso cálculo que se realiza en el modelo de atención blanda y al mismo tiempo es más fácil de entrenar que el enfoque de la atención dura.

Concretando, el modelo genera primero una posición alineada p_t para cada palabra objetivo en el tiempo t. El vector de contexto c_t se deriva entonces como un promedio ponderado sobre el conjunto de estados de origen ocultos dentro de la ventana $[p_{t-D}, p_{t+D}]$; D es seleccionado empíricamente.8

A diferencia del enfoque global, el vector de alineación local es ahora de dimensiones fijas, es decir, $\in R^{2D+1}$. Se consideran dos variantes del modelo.

- Alineación monótona (local-m) - simplemente se establece $p_t = t$ asumiendo que las secuencias de origen y destino están aproximadamente alineadas monótonamente. La alineación vector en se define según la ecuación (5-8).
- Alineamiento predictivo (local-p) - en lugar de asumir alineamientos monótonos, el modelo predice una posición alineada de la siguiente manera:

$$a_t(s) = align(h_t, \overline{h_s}) \exp\left(-\frac{(s - p_t)^2}{2\sigma^2}\right) \quad \text{5-11}$$

Se utiliza la misma función de alineación que en la ecuación (5-8) y la desviación estándar se establece empíricamente como $\sigma = D/2$.

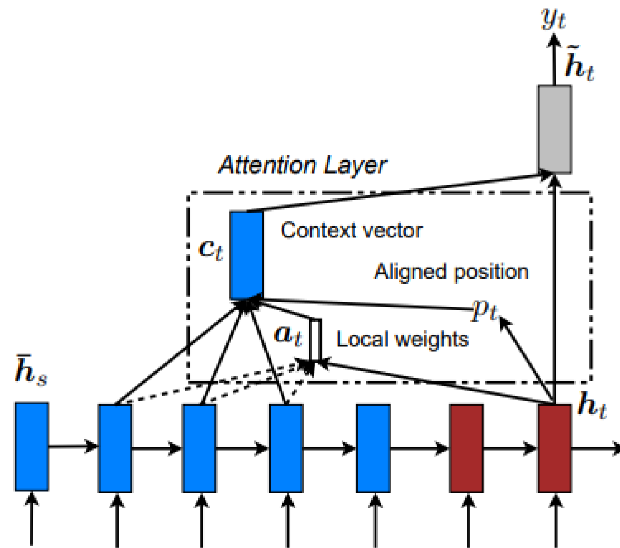

Figura 8. Modelo de atención Local [20]

5.3 Monotonic

Los modelos de atención como los de Bahdanau [2] y Luong [3] procesan, habitualmente, una secuencia de entrada con un codificador de red neural recurrente, RNN, para producir una secuencia de estados ocultos, denominada memoria. Tras ello, un decodificador RNN produce autorregresivamente la secuencia de salida. En cada paso temporal de salida, el decodificador está directamente condicionado por el mecanismo de atención, que permite al decodificador remitirse a las entradas de la secuencia de estados ocultos del codificador. Esta utilización de los estados ocultos del codificador como memoria da al modelo la capacidad de salvar largos desfases de tiempo entre entrada y salida, que se resume en los hablado en apartados anteriores de la facilidad con la que se traducen las oraciones largas. Este punto supone una clara ventaja sobre los modelos sequence to sequence que carecen de un mecanismo de atención.

La atención suave inspecciona cada entrada de la memoria en cada paso temporal de salida, permitiendo efectivamente que el modelo condicione cualquier entrada de la secuencia de entrada arbitrariamente. Esto deteriora su respuesta a secuencias muy largas, como, por ejemplo, pueden ser los documentos completos. Además, como la atención suave posibilita atender cada entrada en la memoria en cada slot de tiempo de salida, se debe esperar hasta que la secuencia de entrada haya sido procesada completamente antes de producir la salida. Este déficit es el principal por el cual los mecanismos de atención convencionales no se consideren aplicables a problemas de traducción en tiempo real. Raffel y Ellis [36] señalaron recientemente que estos problemas pueden mitigarse cuando la alineación entre la entrada y la salida es monótona, es decir, la correspondencia entre los elementos de la secuencia de entrada y salida no implica un reordenamiento. Esta propiedad está presente en diversos problemas del mundo real, como el reconocimiento y la síntesis del habla, en los que la entrada y la salida comparten un orden temporal natural alineado. En otros contextos, la alineación sólo implica reordenamientos locales, por ejemplo, la traducción automática para determinados pares de idiomas.

Sobre la base de esta observación, Raffel y Ellis [36] introdujeron un mecanismo de atención que impone explícitamente una alineación monótona dura de entrada y salida, que permite la decodificación en línea. Sin embargo, la limitación de la monotonía dura también limita la expresividad del modelo en comparación con la atención blanda, que puede inducir una alineación blanda arbitraria.

Para abordar un sistema de este tipo con la mencionada atención suave, Raffel y Ellis [36] propusieron un mecanismo de atención monótona dura, cuyo proceso de atención puede describirse de la siguiente manera. En el momento temporal i de salida, el mecanismo de atención comienza a inspeccionar las entradas de memoria a partir del índice de memoria que atendió en el paso temporal de salida anterior, denominado t_{i-1}. Luego calcula un escalar de energía no normalizada $e_{i,j}$ para $j = t_{i-1}, t_{i-1} + 1$,... para pasar luego estos valores de energía a una función sigmoide logística $\sigma(\cdot)$ que produce "probabilidades de selección" $p_{i,j}$. Entonces, una decisión

discretizada definida como “atender/no atender” $z_{i,j}$ es muestreada de una variable aleatoria de Bernoulli parametrizada por $p_{i,j}$. Si recapitulamos en este punto tenemos:

$$a_t(s) = align(h_t, \overline{h_s})\exp\left(-\frac{(s-p_t)^2}{2\sigma^2}\right) \quad \text{5-12}$$

$$p_{i,j} = \sigma(e_{i,j}) \quad \text{5-13}$$

$$z_{i,j} \sim Bernoulli(p_{i,j}) \quad \text{5-14}$$

En el momento en el que se cumple $z_{i,j} = 1$ para alguna j, el modelo se detiene y establece $t_i = j$ y $c_i = h_{t_i}$. Este proceso se visualiza en la figura 9. Se puede observar que debido a que el mecanismo de atención sólo hace una pasada por la memoria, tiene un coste $O(max(T, U))$ (lineal). Además, para atender la entrada de memoria h_j, el codificador RNN sólo necesita tener procesadas las entradas de la secuencia de entrada $x_1, \ldots, x_j$. Por último, se observa que si $p_{i,j} \in \{0,1\}$ entonces la asignación de $c_i = h_{t_i}$ es equivalente a la marginación sobre las posibles trayectorias de alineación.

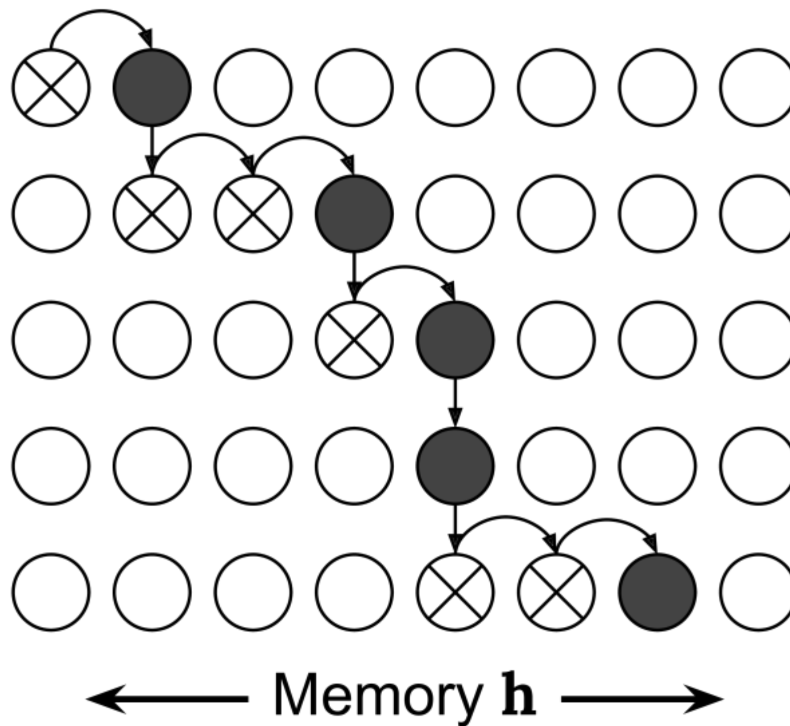

Figura 9. Modelo de predicción de enradas atendidas [4]

Debido a que este proceso de atención involucra el muestreo y la asignación dura, los modelos que utilizan la atención monótona dura no pueden ser entrenados con la backpropagation.

6 APLICACIÓN AL PROBLEMA DE TRADUCCIÓN

En la primera fase de este proyecto se ha realizado la ejecución y comparación de los distintos modelos de atención que se han explicado hasta este momento. Esta comparación se va a realizar a través de un modelo de traducción español-inglés. El sistema se ha basado en TensorFlow y se han utilizado modelos dados por la API de Keras, realizándose la ejecución en un entorno de GPU.

Para realizar estas ejecuciones se ha partido de un modelo de traducción sequence to sequence al que se la ha añadido un modelo de atención entre el codificador y el decodificador. El comportamiento final de este modelo son salidas de frases completas en inglés a entradas de frases completas en español.

6.1 Esquema general y modelo del sistema

El esquema general de este modelo es el que se puede observar en la figura 10. En la que como se observa aparecen los siguientes pasos:

1. La entrada al sistema es una frase cualquiera en español, como, por ejemplo, “Este es un ejemplo del modelo”
2. El primer paso del modelo es procesar esta frase. Para lo cual se introduce dos tokens, <start> y <end> al principio y final de la oración respectivamente.
3. El siguiente paso es codificar esta oración en vectores de longitud fija. De esta manera cada palabra, incluido los tokens, se convierten en un vector unívoco.
4. A continuación, estos vectores pasan al modelo de atención, el cual se encarga de otorgar a la secuencia de vectores de entrada, otra secuencia de vectores de salida.
5. Finalmente, teniendo ya los vectores de salida, cada uno de ellos corresponde unívocamente a un vocablo en inglés por lo que ya se tendrían la salida del sistema.

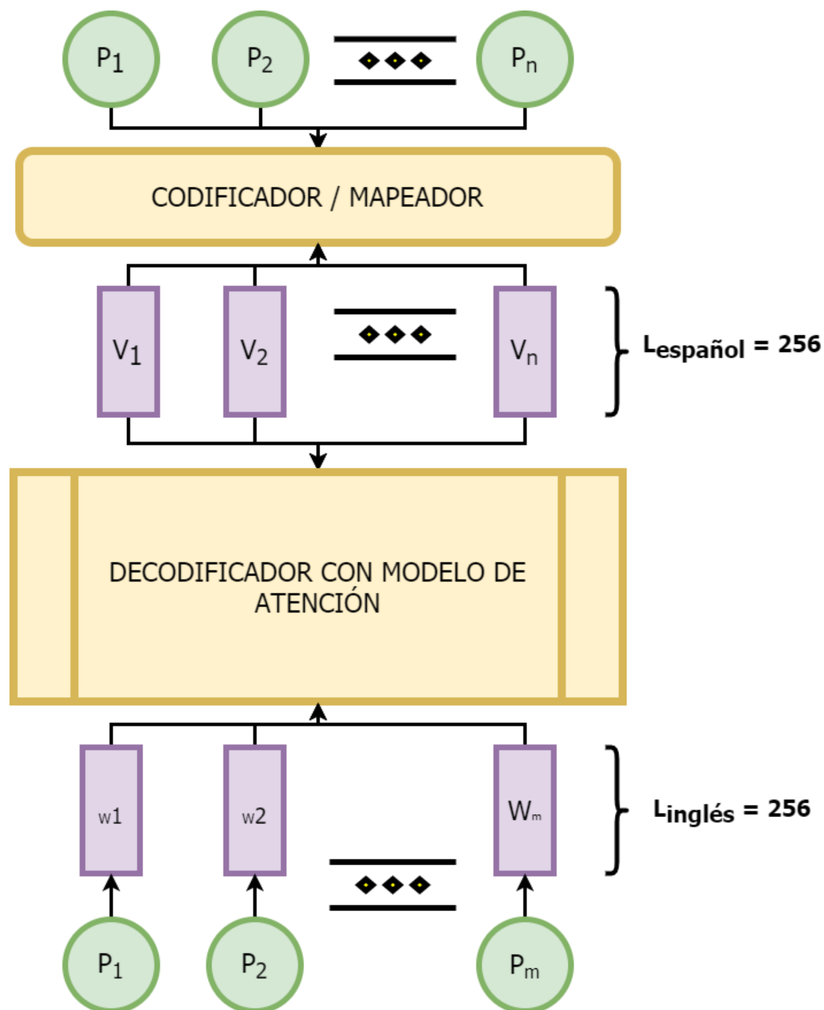

Figura 10 – Esquema problema de traducción

En el sistema propiamente dicho, se implementa un modelo de codificador-decodificador con atención. En la figura 11 se muestra como a cada palabra de entrada ya codificada, o mapeada, se le asigna un peso por el mecanismo de atención que luego es utilizado por el decodificador para predecir la siguiente palabra de la oración.

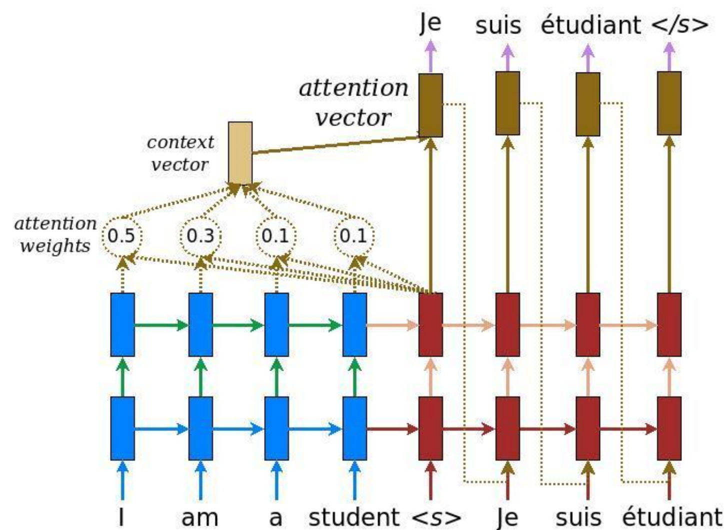

Figura 11 – Esquema de modelo de atención [20]

La entrada se hace pasar por un modelo de codificación que nos da una salida codificada, por la cual a cada palabra de entra le corresponde un vector de longitud fija. A continuación, se muestran las ecuaciones que sigue este modelo de codificador→modelo de atención→decodificador:

$$\alpha_{ts} = \frac{\exp\,(score(h_t, \overline{h_s}))}{\sum_{s'=1}^{S} \exp\,(h_t, \overline{h_{s'}})} \qquad Attention\ weights \tag{6-1}$$

$$c_t = \sum_s \alpha_{ts} \overline{h_s} \qquad Context\ vector \tag{6-2}$$

$$a_t = f(c_t, h_t) = \tanh(W_c[c_t; h_t]) \qquad Attention\ vector \tag{6-3}$$

6.2 Dataset

Para este proyecto se utiliza el dataset que proporcionado en [37]. Dentro de todas las posibilidades que nos otorga este dataset, se ha utilizado el referente a la traducción entre español e inglés.

Como hemos nombrado anteriormente, el primer paso, es preprocesar el dataset. De esta manera a las frases otorgadas se le eliminan los caracteres especiales, es decir, tildes, barras, puntos especiales, etc. En el código siguiente se muestra esta acción de preprocesado:

```
def preprocess_sentence(w):
    w = unicode_to_ascii(w.lower().strip())

    # Se separan los caracteres especiales de las palabras
    w = re.sub(r"([?.!,¿])", r" \1 ", w)
    w = re.sub(r'[" "]+', " ", w)

    # Se elimina todos lo caracteres excepto (a-z, A-Z, ".", "?", "!", ",")
    w = re.sub(r"[^a-zA-Z?.!,¿]+", " ", w)

    w = w.rstrip().strip()

    # Se añaden los tokens
    w = '<start> ' + w + ' <end>'

    return w
```

Figura 12 – Código de preprocesado de frase

A continuación, a cada entrada se le asigna su salida correspondiente en una lista de elementos string, creando de esta manera pares de entrenamiento y validación.

El siguiente paso es crear un índice unívoco para cada palabra, en cada uno de los idiomas, creando de esta manera un diccionario vectorial. Una vez se tiene el número total de palabras se procede a otorgar de una longitud fija a los vectores que contienen la correspondencia con las palabras.

En estas ejecuciones se toman 200000 muestras, es decir, 200000 pares de frases español-inglés para entrenar y

validar el modelo.

Finalmente, se realiza la separación aleatoria de los datos entre datos de entrenamiento y datos de validación con un porcentaje 80%-20% respectivamente.

6.3 Entrenamiento

El entrenamiento de este modelo se apoya en la backpropagation y sigue lo siguientes pasos:

1. Se pasa la entrada a través del codificador que devuelve la salida del codificador y el estado oculto del codificador.
2. Se le pasa al decodificador la salida del codificador, el estado oculto del codificador y la entrada del decodificador (que es la señal de inicio).
3. El decodificador devuelve las predicciones que ha realizado y el estado oculto del decodificador.
4. El estado oculto del decodificador se pasa de nuevo al modelo de atención y las predicciones se utilizan para calcular la pérdida.
5. Se utiliza un forzado "teacher" para a toma de decisión de la siguiente entrada al decodificador.
6. El forzado "teacher" es la técnica en la que la palabra objetivo se pasa como la siguiente entrada al decodificador.
7. Como último paso se calculan los gradientes y se aplican al optimizador y a la backpropagation.

6.4 Validación

La validación se realiza de manera muy similar al entrenamiento, excepto que en esta ocasión no se usa el forzado "teacher". La entrada del decodificador en cada slot de tiempo son sus predicciones previas junto con el estado oculto y la salida del codificador. De esta manera el modelo va encadenando vectores hasta que llega al vector que mapea el vocablo <end>, el cuál es el token de finalización de la entrada. En ese punto termina de predecir y guarda los pesos de atención para cada paso de tiempo.

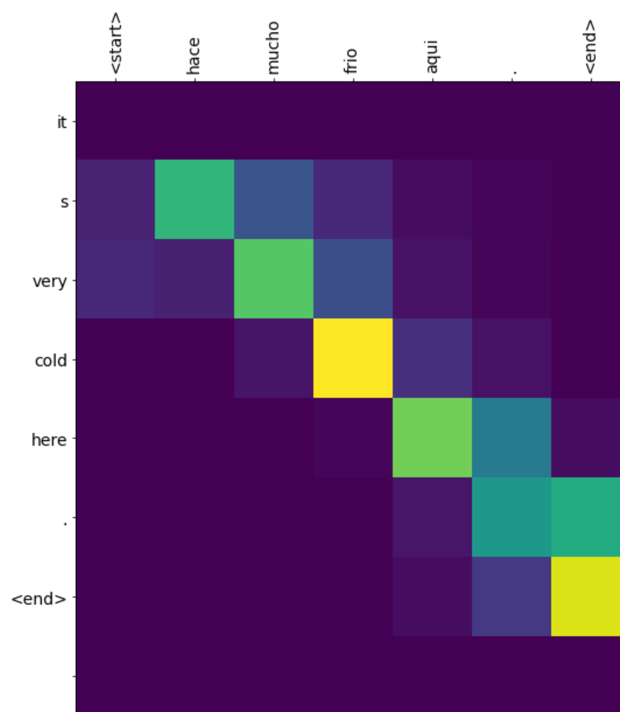

Figura 13 – Matriz del modelo de atención [20]

Finalmente pinta la matriz de pesos, en la cual se observan cuáles son los vocablos o palabras más influyentes de la oración de entrad que hace conformar o predecir la oración, traducida al inglés, de salida. Como se puede observar en la figura 13.

6.5 WER

Finalmente, para comprobar el buen funcionamiento del modelo nos apoyamos en el conocido como WER, (Word Error Rate). El WER es el número de palabras erróneas que encontramos en una secuencia dividido por el total de palabras de dicha secuencia. El WER sigue la ecuación 6-4.

$$WER = (sustituciones + inserciones + supresiones) \,/\, número\ de\ palabras \qquad 6\text{-}4$$

- **Sustitución,** ocurre cuando una palabra es reemplazada por otra. Aplicado a nuestro proyecto, sería cuando una palabra, por ejemplo, “hola” en lugar de ser traducida como “hello” se traduce como “yellow”.
- **Inserción,** es cuando aparece una palabra que no está en la oración de entrada. Por ejemplo, en nuestro modelo, cuando la pablar “vete” se traduce por la secuencia de palabras “get out”.
- **Supresión,** se produce cuando una palabra que aparece en la oración de entrada se deja fuera de la oración traducida. Por ejemplo, en nuestro sistema, cuando la secuencia de palabras “a mí me gusta” se traduce por la secuencia “I like to”

De esta forma nos podríamos encontrar una traducción en la que una oración contiene 20 palabras en total y la oración traducida incluya 17 palabras habiendo participado todos los efectos explicados en los párrafos anteriores.

El cálculo de la WER se basa en una medida llamada "distancia de Levenshtein". La distancia de Levenshtein es una medida de las diferencias entre dos secuencias. En este caso, las secuencias son dos oraciones, una en cada idioma.

6.6 TEST

Como se ha explicado en apartados anteriores, este análisis se va a realizar para diversos modelos de atención, entre ellos los propuestos por Bahdanau, por Luong y la aplicación monotónica a cada uno de ellos. Los testeos se realizarán con los mismos parámetros, que son los que se muestran a continuación:

- Número de pares de oraciones = 200000
- Porcentaje de pares de oraciones para entrenamiento y validación: 80%-20%
- Número de épocas de entrenamiento = 50
- Longitud de vectores correspondientes a los vocablos = 256
- BATCH_SIZE = 64

Para estos parámetros y con el modelo que se ha explicado en este apartado se consiguen los resultados de la tabla siguiente:

Tabla 6–1 Resultados taducción

Modelo de atención	Loss	WER (%)
Bahdanau	1,02	0,82
Luong	1,03	0,90
Monotonic	0,99	0,84
Bahdanau-monotonic	0,99	1,00
Luong-monotonic	1,02	0.78

Como se puede observar en la tabla, todos los modelos obtienen resultados de la WER muy parejos. Como puntos interesantes destacar que los modelos de atención que no conllevan monotonic contienen unos mejores resultados. Esto es debido a que tienen la vista completa de las secuencias de entradas. Por tanto, se puede decir que, si es asumible el tiempo de ejecución de un modelo sin monotonic, este se hace completamente prescindible.

También se observa las perdidas en los modelos a los que se les aplica la atención monotonic son menores que a los que no, esto es debido a la propia naturaleza del monotonic, ya que no utiliza retroalimentación.

7 APLICACIÓN AL PROBLEMA DE TRANSCRIPCIÓN

Tal y como se ha explicado en la introducción de esta memoria, la transcripción de textos manuscritos se ha convertido en una tarea realmente necesaria en los momentos actuales. En un mundo que tiende a la globalización, al open source y al acceso inmediato de la información, se hace necesario que esta información sea provista.

La idea de la transcripción, o reconocimiento de texto, es bastante sencilla conceptualmente. A partir de una imagen se extraen una secuencia de características de los datos, que más tarde se comparan con una secuencia de etiquetas usando un modelo de machine learning.

Al igual que en el problema de traducción, el sistema estará implementado en TensorFlow y se han utilizado modelos proporcionados por la API de Keras, realizándose la ejecución en un entorno de GPU. Al mismo tiempo, se expone el modelo de transcripción de la manera más parecida posible al modelo de traducción, para que de esta manera puedan ser fácilmente comparados.

7.1 Esquema general y modelo del sistema

En este proceso podemos presentar una analogía clara al problema de traducción. En el problema de traducción descrito en la memoria tenemos un sistema que tiene como entradas una secuencia de palabras en español a la que le corresponde una secuencia de salida en inglés. De la misma forma en el problema de transcripción se modela un sistema que tiene como entrada una imagen, que traducimos a una secuencia de entrada, y el sistema nos da como salida una secuencia de caracteres. De esta manera podemos decir que la secuencia de vectores de características que extraemos de las imágenes se corresponde a la secuencia de vectores del idioma origen y la secuencia de salida de vectores que corresponden a caracteres sería análoga a la secuencia de vectores de palabras en el idioma destino.

El esquema general de este modelo es muy parecido al del modelo de traducción, como se puede observar en la figura 14. De esta manera podemos sacar los siguientes subconjuntos:

1. La entrada en esta ocasión varía, y en lugar de ser una frase cualquiera en español, es una imagen en la que aparece un texto fotografiado. Las imágenes son de tamaños diversos y se les realizará un preprocesado que se explicará en posteriores apartados.

2. En este caso, el primer paso del sistema es procesar la imagen ya preprocesada. Esta acción se realiza a través de una red CNN, que tomará como entrada las imágenes y tendrá como salida uno vectores de longitud definida, que contendrán las características de la imagen procesada, de longitud fija.

   Estos vectores son análogos a los expuestos en el modelo de traducción que básicamente eran una traducción o mapeo de las palabras en español a un vector de longitud fija. Por el contrario, en el problema de transcripción estos vectores no tienen una correspondencia unívoca, sino que son el resultado de un modelo de red CNN que es necesario entrenar y que de manera pragmática se podría decir que son las características de las imágenes.

3. A continuación, estos vectores pasan al modelo de atención, el cual se encarga de otorgar a la secuencia de vectores de entrada, otra secuencia de vectores de salida.

   Este caso es totalmente análogo al problema de traducción, ya que, si nos abstraemos del significado de los vectores, tan solo tenemos una asociación de vectores de entrada que generan otra asociación de vectores de salida. Siendo los vectores de entrada al modelo de atención la salida de la red CNN y los vectores de salida del modelo de atención los vectores unívocos asociados a cada carácter.

4. Finalmente, teniendo ya los vectores de salida, cada uno de ellos corresponde unívocamente a un carácter por lo que ya se traduce, o mapea, esta secuencia de vectores a los caracteres correspondientes y se obtiene el texto predicho.

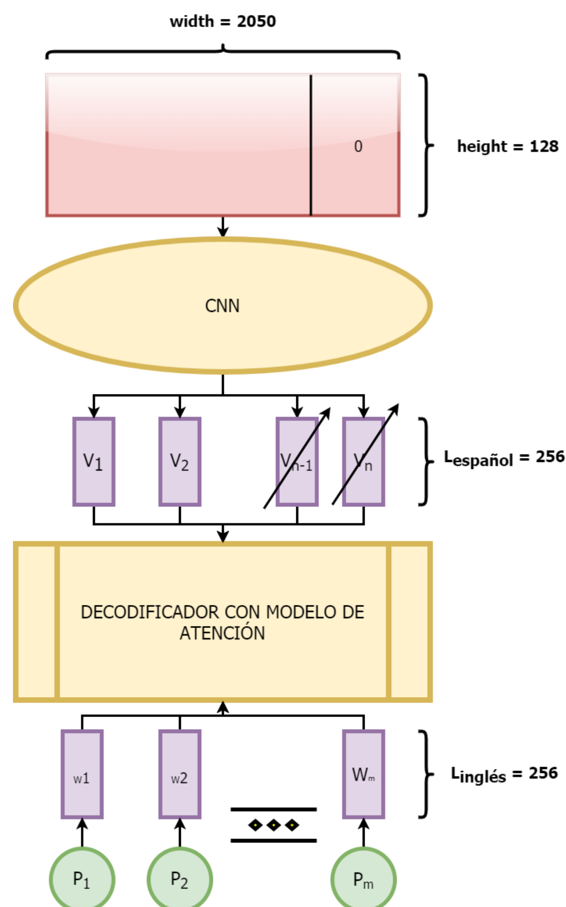

Figura 14 – Esquema del problema de transcripción

Al igual que en el modelo del problema de traducción, en el modelo del problema de transcripción, se implementa un modelo de codificador-decodificador con atención. En la figura 11 se muestra que a cada palabra de entrada ya codificada; es decir, cada vector de característica de la imagen; se le asigna un peso por el mecanismo de atención que luego es utilizado por el decodificador para predecir el siguiente carácter de la oración.

La entrada se hace pasar por un modelo de codificador que nos da la salida de forma ya codificada, que básicamente como hemos dicho anteriormente es un vector de longitud fija. Las ecuaciones que siguen este modelo de codificador-modelo de atención-codificador son las mostradas en (6-1), (6-2) y (6-3):

$$\alpha_{ts} = \frac{\exp\left(score(h_t, \overline{h_s})\right)}{\sum_{s'=1}^{S} \exp\left(h_t, \overline{h_{s'}}\right)} \qquad Attention\ weights$$

$$c_t = \sum_{s} \alpha_{ts} \overline{h_s} \qquad Context\ vector$$

$$a_t = f(c_t, h_t) = \tanh(W_c[c_t; h_t]) \qquad Attention\ vector$$

7.2 Dataset

El set de datos que se ha utilizado en este proyecto es el set de datos propuesto por ICFHR 2018 Competition over READ dataset [38]. El cual, provee de imágenes en escala de grises ya segmentadas a nivel de línea de diversos tamaños. A su vez nos provee de tres ficheros con extensión ".csv". Cada uno de ellos contiene información sobre los sets de test, train y validación. Cada fichero ".csv" contienen una tabla con información importante para cada imagen. En cada fila de la tabla aparecen, el nombre de la imagen, el texto que hay escrito en ellas, una secuencia de números que equivale al texto, el tamaño que tiene la imagen, y en la última columna aparece la longitud del texto en la imagen (número de caracteres).

A partir de este set de datos, se crea una estructura de dataset en la cual aparecen pares de información. Cada par de datos contiene la entrada, una imagen, y la salida, el texto al que corresponde cada imagen.

Como se ha comentado el set de datos se encuentra ya dividido entre entrenamiento, validación y test, por lo que no es necesario segmentarlo en esta ocasión.

- El set de entrenamiento está compuesto por 11425 líneas.
- El set para validación está compuesto por 500 líneas.
- El set de test está compuesto por 2878 líneas. A su vez está desglosado en 5 test distintos, que se corresponden con 5 documentos distintos en los que se pretende evaluar individualmente el rendimiento de la red según escritores distintos.

Como se ha comentado las imágenes proporcionadas no están normalizadas, es decir, no tienen tamaños iguales. Por lo tanto, es necesario realizarle un preprocesado a la imagen para homogeneizar sus características y que de esta forma sea el DATASET válido para tomarse como entrada de una red convolucional CNN.

7.3 Preprocesado de la imagen

La idea principal del preprocesado es obtener un set de imágenes con las mismas dimensiones sin que esta

transformación afecte considerablemente a sus características más visuales, llanamente, que a nivel cognitivo humano en la imagen preprocesada se pueda leer el mismo texto que en la imagen original sin demasiado esfuerzo.

Las medidas que se van a tomar deben de ser uniformes por lo que tienen que quedar claras antes de realizar el escalado. La propia base de datos propone altura de escalado de 128 píxeles, que es la que vamos a seleccionar. Al tener la altura definida debemos de tener cuidado con el ancho de la imagen para no modificar la relación de aspecto

las imágenes y que los textos se vuelvan ilegibles. De esta manera, en primer lugar, realizamos un escalado, el conocido como thumbnail, que redimensiona la imagen según un alto o ancho definido respetando a relación de aspecto, especificando el valor de la altura a 128 píxeles, y una vez tenemos todas las imágenes escaladas a una altura fija, observamos la máxima anchura de todas las imágenes, en nuestro caso 2050 píxeles, que será el valor de ancho que tomaremos para todas las imágenes. Como cada imagen tiene una anchura distinta, y menor, de 2050 píxeles, se le añade un pad, en este caso apoyándonos en el escalado crop, que añade píxeles negros hasta completar el tamaño requerido para la imagen.

```
im = Image.open(h['imgName'][i] + ".jpg")
im.thumbnail([im.size[0], 128], Image.ANTIALIAS)
width, height = im.size
left = 0
top = 0
right = 2050
bottom = height
im = im.crop((left, top, right, bottom))
im.save(h['imgName'][i] + ".jpg")
```

Figura 15 – Código de preprocesado de imagen

De esta manera se transformas las imágenes tal y como aparece en el siguiente diagrama:

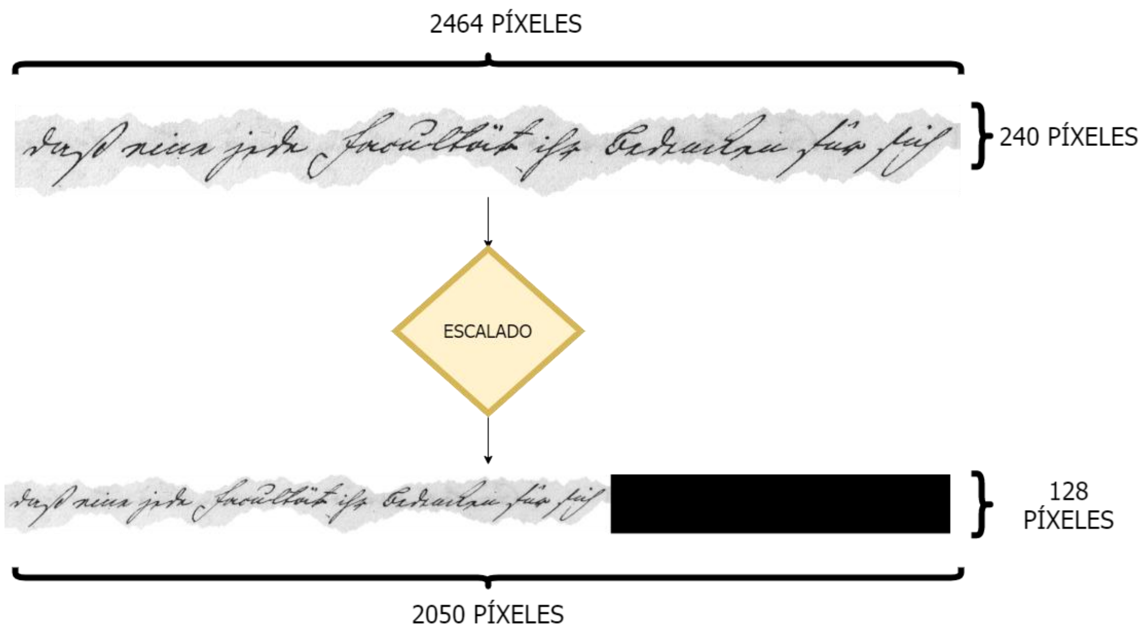

Figura 16 – Esquema de preprocesado de imágenes

7.4 Entrenamiento

Nuevamente, el entrenamiento vuelve a ser muy parecido al del problema de traducción, solo que en este también

es necesario entrenar la CNN de reducción de las imágenes. De esta forma este modelo también se apoya en la backpropagation y sigue lo siguientes pasos:

1. Se pasa la entrada a través del codificador, en esta ocasión con la CNN incorporada, que devuelve la salida del codificador y su estado oculto.
2. Tras ello, se pasa al decodificador la salida del codificador, su estado oculto y la entrada del decodificador (que es la señal de inicio).
3. El decodificador devuelve las predicciones que ha realizado y el estado oculto del decodificador.
4. El estado oculto del decodificador se pasa de nuevo al modelo de atención y las predicciones se utilizan para calcular la pérdida.
5. Se utiliza un forzado “teacher” para a toma de decisión de la siguiente entrada al decodificador. El forzado “teacher” es la técnica en la que la palabra objetivo se pasa como la siguiente entrada al decodificador.
6. Como último paso se calculan los gradientes y se aplican al optimizador y a la backpropagation.

7.5 Validación

La validación se realiza de manera muy similar al entrenamiento, excepto que en esta ocasión no se usa el forzado “teacher”. La entrada del decodificador en cada slot de tiempo son sus predicciones previas junto con el estado oculto y la salida del codificador. De esta manera el modelo va encadenando vectores hasta que llega al vector de final de la imagen, el cuál es el token de finalización de la entrada. En ese punto termina de predecir y guarda los pesos de atención para cada paso de tiempo. En este caso de validación, las imágenes proporcionadas son textos manuscritos pertenecientes a autores con los que se h realizado el entrenamiento.

7.6 Entrenamiento

Este apartado se podría considerar como el resultado del proyecto, ya que vamos a analizar los resultados obtenidos para distintos modelos de red convolucional, CNN, aplicando los diversos test que nos ofrece el set de datos y comparándolos con otros estudios.

7.6.1 Dense Layer

La primera CNN que se abarca en el proyecto no es realmente una CNN sino una dense layer, o full-connected layer. En este primer procesado de la imagen básicamente tomamos cada columna de una imagen y la traducimos a una palabra del idioma imagen. Cada palabra tiene 128 píxeles y cada píxel 8 bits, por lo que en total estamos inventando un lenguaje con 8128 = 3,94e+115, lo que crearía un modelo totalmente desproporcionado teniendo en cuenta que la RAE contiene 88.000. Aun así, entrenamos el modelo obteniendo como es lógico un resultado no admisible, ya que las pérdidas, el conocido como batch loss es superior a 1, lo cual no es un resultado deseable.

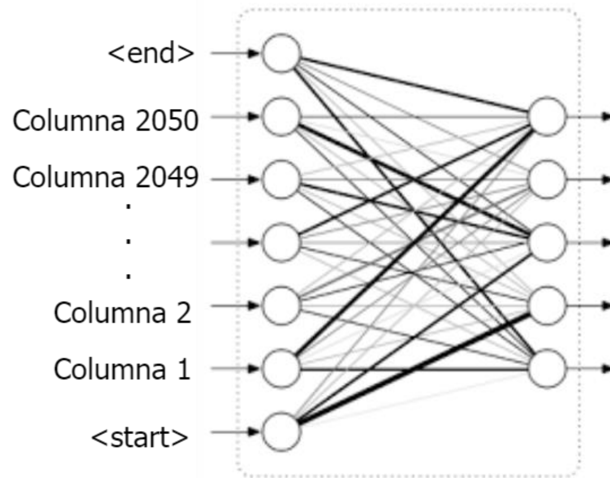

Figura 17 – Dense layer [39]

7.6.2 CNN 1 capa

El segundo procesado que se plantea es el de reducción de la matriz de imagen a través de una sola capa de CNN. De esta manera se modela una CNN de una sola capa con 256 filtros y una reducción de la imagen de unas dimensiones de [128, 2050] a [1, 1000]. De esta manera convertimos cada imagen de entrada en una matriz de tamaño [1, 1000, 256], imitando de esta manera una frase de entrada del modelo de traducción, que contaría con 1000 palabras codificadas en vectores de tamaño 256. Tras entrenar este modelo con 200 epochs y llegar a unas pérdidas de entrenamiento en torno al 3%, se obtienen unos datos de validación malos, con un CER del 68%. Aunque obviamente este resultado no es positivo, sí se pueden extraer algunas conclusiones, ya que, si observamos algunas de las frases transcritas, se puede observar que la transcripción no es tan aleatoria como pudiera parecer en un primer momento. En la transcripción mostrada a continuación, no se encuentran demasiadas consonantes o vocales juntas, sino que estás se distribuyen, en lo que parece ser ciertas sílabas. Por lo que podemos decir que con esta CNN de una sola capa ya comienza a influir el modelo de atención, ya que, si este no influyese, la transcripción sería completamente aleatoria pudiendo encontrarse predicciones del tipo "fjsrtjrtj" o "ferteoaoe".

Imagen: 438,30887_0234_1068781_region_1495472894328_1604_r1027

Predicted translation: [‘t’ ‘e’ ‘r’ ‘r’ ‘u’ ‘a’ ‘s’ ‘s’ ‘e’ ‘r’ ‘a’ ‘b’ ‘t’ ‘<end>’]

La corrección: how modest a man

Figura 18 – Salida transcripción sílabas

7.6.3 CNN de varias capas con dropout

En este modelo nos basamos en el beneficio que aporta el dropout a la reducción de imágenes utilizadas en modelos de transcripción. Esta teoría está respaldada en [40], en los que se aportan resultados de transcripción relevantes con la aplicación del dropout. El modelo que se presenta consta de seis fases. Las seis fases son repeticiones de una capa convolucional seguida de una capa pool y otra de dropout, definida de la siguiente forma:

```
#Variables para dropout
conv_keep_prob = 0.8

###########Layer 1
self.CNN = tf.keras.layers.Conv2D(filters = 16, kernel_size = [3,3], strides=[1,1], activation=tf
.nn.leaky_relu)
self.pool = tf.keras.layers.MaxPool2D(pool_size=(4, 2), strides=None)
self.drop = tf.keras.layers.Dropout(rate = 0.0)
```

Figura 19 – Código capa CNN-POOL-DROPOUT

De esta manera, se consigue transformar la imagen de entrada de tamaño [128, 2050] en un matriz de tamaño [1, 250, 1024]. Al igual que en el apartado anterior, si realizamos una analogía con el problema de traducción, se podría decir que hemos transformado la imagen de entrada en una oración de 250 caracteres, en las que cada palabra está codificada en un vector de longitud fija de tamaño 1024.

Al igual que en el entrenamiento del apartado anterior, se realiza un entrenamiento de 200 epochs, llegando el modelo a una tasa de pérdidas de 1,4%. Al realizar las pruebas de validación encontramos nuevamente un mal resultado, aunque mejor que el del modelo de una sola capa, obteniéndose un valor del CER del 41%. Además, al igual que en el apartado anterior se vislumbraban ciertas sílabas, en este modelo se comienzan a observar la aparición de ciertas palabras, aunque estas no sean la transcripción real de la imagen del texto manuscrito. Se muestra a continuación un ejemplo de ello:

Imagen: 30884_0026_1066308_region_1485959662821_22_r1011.jpg

Predicted translation: [‘t’, ‘h’, ‘i’, ‘n’, ‘space’, ‘o’, ‘f’, ‘space’, ‘t’, ‘h’, ‘i’, ‘s’, ‘<end>’]

La corrección: When a quality is thought about as a distinct object, it is said to be

Figura 20 – Salida transcripción palabras

Como se puede observar, claramente la predicción es completamente errónea, pero es de interés destacar que la salida de un modelo que predice caracteres de como resultado palabras completas y bien delimitadas. Obviamente se puede decir que esto es resultado del modelo de atención, ya que si este no existiese no tendría ningún sentido que apareciesen estos vocablos.

7.6.4 Inception_v3

El modelo final de este proyecto es el inception_v3. Este, es un modelo pre-entrenado, desarrollado por GoogleNet [41], altamente eficiente que posibilita de manera sencilla la adaptación de un dataset de entrada que producen unas salidas deseadas. De esta manera, en el proyecto que nos ocupa se utiliza de forma que las imágenes que contienen textos manuscritos puedan reducirse a una matriz de dimensiones fijas que sean la entrada de los distintos modelos de atención.

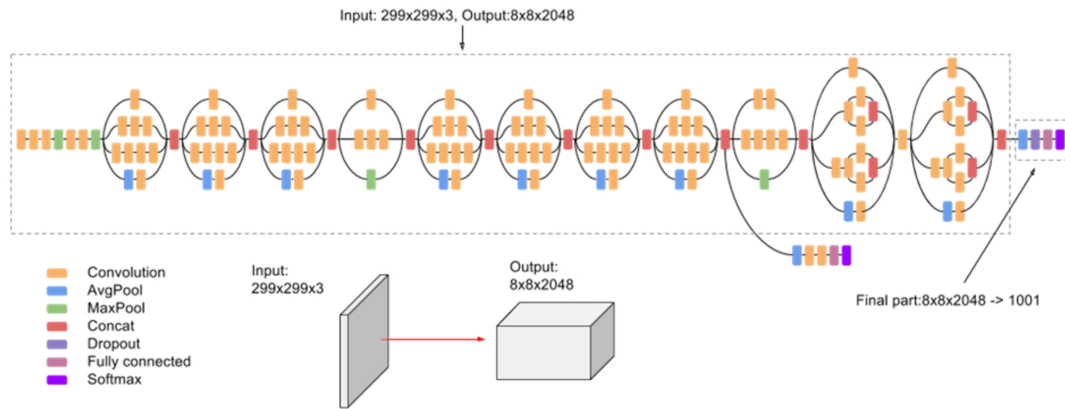

Figura 21 – Esquema del modelo de inception_v3 [42]

El modelo está preparado para imágenes de entrada con tamaño [299, 299, 3] aunque es capaz de aceptar diversos tamaños, con la única condición de que las imágenes de entrada sean RGB, es decir, que tengan 3 canales de profundidad.

Como aparece en [43], el modelo de inception_v3 contiene las etapas descritas en la siguiente tabla.

Tabla 7–1 Esquema de inception_v3

type	patch size/stride or remarks	input size
conv	3×3/2	299×299×3
conv	3×3/1	149×149×32
conv padded	3×3/1	147×147×32
pool	3×3/2	147×147×64
conv	3×3/1	73×73×64
conv	3×3/2	71×71×80
conv	3×3/1	35×35×192
3×Inception	As in figure 22 A	35×35×288
5×Inception	As in figure 22 B	17×17×768
2×Inception	As in figure 22 C	8×8×1280
pool	8 × 8	8 × 8 × 2048

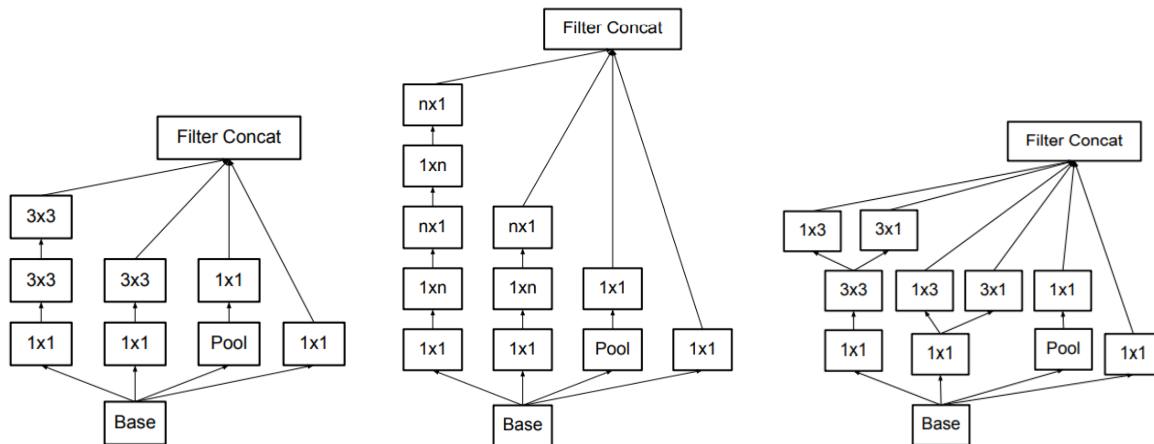

Figura 22 – Esquema de los modelos de inception [43]

En un primer momento, se observa que este modelo no encaja con el objetivo de este proyecto, que no es otro que adaptar un modelo de atención para traducción a un problema de transcripción. Para ello el planteamiento del proyecto es tratar las imágenes de textos manuscritos como si se tratasen de oraciones. De esta forma se observa qué realizando la misma analogía que en los apartados anteriores, con el modelo de traducción, encontraríamos que cada imagen de entrada constaría de 8 frases, con 8 palabras, y cada palabra estaría codificada en un vector de longitud fija de tamaño 2048. Así pues, vamos a tener que adaptar el modelo provisto por Google a nuestras necesidades.

Para poder utilizar este modelo es necesario que la imagen de entrada posea 3 capas de profundidad, por lo que, en primer lugar, se ha realizado una copia de la imagen de tamaño [128, 2050] para tener una imagen con tres capas, siendo las tres capas iguales. De esta manera se obtienen unas imágenes con dimensiones [128, 2050, 3]. Así pues, haciendo uso del modelo de procesado inception_v3, pasamos de tener una entrada de [128, 2050, 3], a una entrada de [8, 8, 2048], que como se ve, reduce considerablemente el ancho y alto de las imágenes añadiendo toda la carga de la información a la dimensión de profundidad. De esta manera resulta más sencillo y ágil proceder a codificar las columnas de estas imágenes como vectores de entrada con longitud fija.

Aun así, siguen apareciendo discordancias con la idea principal del proyecto, ya que la idea es reducir la imagen de entrada a un vector con una única fila y un número de columnas mayor, en casi todos los casos, al número de caracteres de la entrada, puesto que el problema realiza la transcripción de caracteres, y no tendría sentido que cada vector de entrada tuviese información de más de un caracter. Para conseguir este objetivo, se han adaptado las etapas de reducción, para obtener una transformación de la imagen de entrada a una matriz de dimensiones [1, 260, 2048]. Así, al realizar nuevamente la analogía con el problema de traducción, cada imagen se transforma en una oración con 260 palabras, cuyas palabras están codificadas en vectores de longitud fija de tamaño 2048. Adaptando la entrada del modelo al esquema mostrado en la figura 21. Además, en el anexo, se muestra la adaptación que se ha realizado del inception_v3 al problema de transcripción.

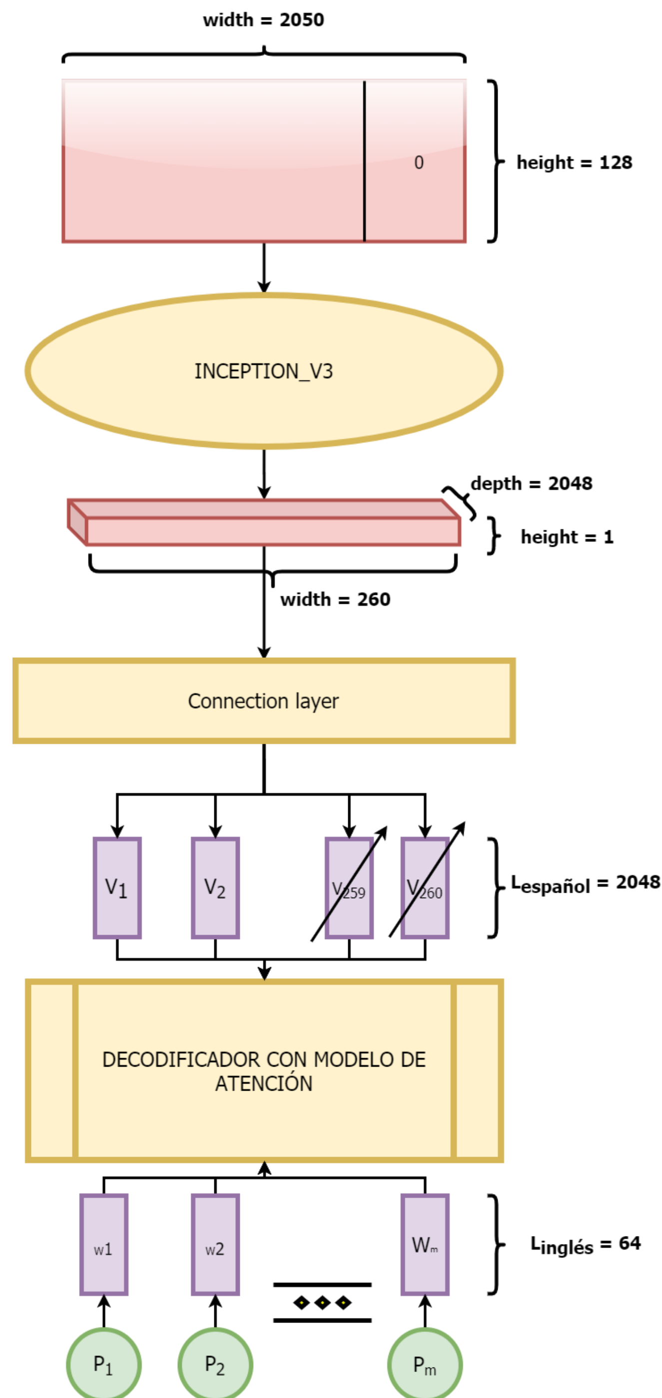

Figura 23 – Esquema del modelo de transcripción con preprocesado inception_v3

7.6.5 Resultados

Con la adaptación presentada del módulo de procesamiento de imágenes inception_v3 al problema de transcripción, se han realizado diversas simulaciones. Estas simulaciones se dividen por modelos de atención, habiéndose implementado al igual que en el problema de traducción, los modelos de atención de Bahdanau [2], Luong [3], Monotonic [4], y las adaptaciones del monotonic a los modelos de Bahdanau [2] y Luong [3].

Las pruebas realizadas cuentan con los siguientes parámetros. En primer lugar, se ha entrenado el sistema para cada modelo de atención un total de 250 epochs con el set de entrenamiento que como aparece en el apartado de dataset contiene 11425 entradas distintas. Una vez se encuentra entrenado el modelo, se valida con el set de validación, textos provenientes de los mimos autores con los que se ha entrenado, y finalmente se realizan los

test, con texto de autores con los que no se ha entrenado el modelo, es decir, son totalmente desconocidos por los sistemas. Estos autores con los que se realizan los test son Konzil, Schiller, Ricordi, Patzig y Schwerin. Además, los resultados obtenidos serán comparados con los de otras soluciones aportadas al mismo problema y que se extraen de [44].

A continuación, se muestran en la tabla 7-2, los resultados obtenidos para la validación de cada modelo de atención.

Tabla 7–2 Resultados de validación

Modelo de atención	Loss	CER (%)
Bahdanau	3,33	13,58
Luong	4,78	14,37
Monotonic	9,02	18,12
Bahdanau-monotonic	2,83	17,35
Luong-monotonic	3,66	20,89

Una vez entrenados y validados los modelos los sometemos a las entradas de test, comparando los resultados en la siguiente tabla con los otros modelos aportados en [44].

Tabla 7–3 Resultados de Test (Comparación)

Modelos	CER (%)
Yousef, Hussain, Mohammed - 16(128,512)	25,34
Yousef, Hussain, Mohammed - 8(128,512)	25,90
OSU	31,39
ParisTech	32,25
LITIS	35,29
PRHLT	32,79
RPPDI	30,80
(Ours) Bahdanau	**34,29**
(Ours) Luong	**33,84**
(Ours) Monotonic	**38,12**
(Ours) Bahdanau-monotonic	**36,96**
(Ours) Luong-monotonic	**41,14**

En la tabla 7-3, en la que se comparan los modelos probados en este proyecto con otros estudios, se puede observar que los CER obtenidos en este proyecto no llegan al nivel de otros desarrollados, pero no quedan tampoco muy lejanos a ellos. A continuación, en la tabla 7-3, se desgranan los CER obtenidos para cada modelo según el escritor al que pertenece los textos manuscritos.

Tabla 7–4 Resultados de validación

Modelo de atención	Konzil (CER (%))	Schiller (CER(%))	Ricordi (CER(%))	Patzig (CER(%))	Schwerin (CER(%))
Bahdanau	33,13	35,03	37,83	31,00	35,29
Luong	30,51	34,66	37,11	31,01	35,61
Monotonic	36,28	39,50	37,99	37,73	38,80
Bahdanau-monotonic	36,28	34,50	36,92	34,28	38,01
Luong-monotonic	42,90	43,35	41,27	39,11	40,08

Comparando los resultados de CER obtenidos por autor se observa como todos esto son cercanos al valor del total, es decir, no se presentan grandes dispersiones de resultados según el autor. Esto puede ser debido a que al no estar ninguno de los modelos entrenados con textos manuscritos de estos autores, el modelo no conoce las características individuales de cada uno de ellos, lo que hace que su transcripción sea genérica.

8 CONCLUSIONES

En el proyecto que se plasma en esta memoria se han abordado la solución de dos tipos de problemas mediante el modelo de atención. En primer lugar, se ha aplicado el modelo de atención al problema de traducción, extrapolando esta solución posteriormente al problema de transcripción.

En la actualidad el problema de traducción presenta diversas soluciones, y se encuentra en unas cotas de WER bastante bajas. Además, aquellas predicciones que presentan fallos, no significan que sean malas traducciones, sino que no coinciden con la frase que se presenta como solución. Esto se debe a que en los idiomas se pueden transmitir la misma información de varias formas, sin que existan realmente unas traducciones unívocas. Por lo que, los resultados obtenidos realmente son una medición conservadora de la idoneidad del modelo, que en la práctica es superior. Esta suposición no es válida para el problema de transcripción, en el que sí se pretende que el texto digitalizado coincida plenamente con el manuscrito en la imagen.

Respecto a los distintos modelos de atención probados en la traducción, no se observan grandes diferencias, pero sí es cierto que los resultados de aquellos modelos que comprueban la entrada en su totalidad, Bahdanau [2] y Luong [3], en su atención global, obtienen mejores resultados. Al no ser la diferencia entre modelos tan elevada, se puede concluir que en casos en los que sea necesario inmediatez en la traducción, el uso de modelos con ventanas, los que solo usan parte de la entrada para predecir la salida, dan uno resultados bastante válidos.

Respecto al problema de transcripción es posible extraer bastantes conclusiones e indicios. En el proyecto reflejado en esta memoria se ha demostrado que la adaptación del problema de traducción al de transcripción es plausible. Aun así, esta adaptación tiene ciertas diferencias que condicionan al modelo, al menos con la solución dada en este proyecto. En primer lugar, la "transformación" que se realiza a una imagen para convertirla es una oración de entrada origina que los vectores que consideramos caracteres no tengan la información completa de dicho carácter, incluso es posible que la información que posea concierna a dos o más caracteres. Esta distinción hace que, a diferencia de la traducción, las entradas no tengan una relación unívoca con las palabras de salida, es decir, muchos vectores pueden contener una letra "A", parte de ella, o ella misma conjuntamente con otros caracteres.

Al igual que en el problema de traducción, en el de transcripción tampoco se aprecian grandes diferencias entre los distintos modelos de atención probados. En este caso, puede ser debido al dataset de entrada, en el cual no se observaban oraciones o párrafos largos, sino más bien cortos. Así pues, es posible que con dataset de textos manuscritos con textos más largos, estos resultados se acrecienten.

Otras de las claves a extraer de este proyecto es la clara aportación del modelo de atención a la transcripción. Esto se puede apreciar más en los fallos que en los aciertos del algoritmo, ya que se presentan bastantes errores en los que un modelo que predice caracteres obtiene fallos que en lugar de ser de caracteres son de palabras completas y reales como las mostradas en esta memoria. Este punto tiene su parte positiva pero también negativa, ya que en lugar de tener fallos de un solo carácter que siga haciendo comprensible el texto como si se tratase de una simple errata, en su lugar aparece otra palabra similar que puede cambiar el contexto. Por otra parte, la clara

búsqueda de palabras completas por parte del modelo también hace que las erratas se minimicen.

Como trabajo futuro aparece una clara dirección de investigación, que sería la unión del modelo de atención con modelos de transcripción exitosos, en aras de comprobar si estas dos soluciones son compatibles y mejoran los resultados obtenidos.

REFERENCIAS

[1] S. J. Russell y P. Norvig, Artificial Intelligence: A Modern Approach, 2009.

[2] B. Dzmitry, C. Kyunghyun y B. Yoshua, Neural Machine Translation by Jointly Learning to Align and Translate, vol. 2, 2014.

[3] L. Minh-Thang, P. Hieu y D. M. Christopher, Effective Approaches to Attention-based Neural Machine Translation, 2015.

[4] J. K. Kelvin, Show, attend and tell: Neural image caption generation with visual attention., 2015.

[5] J. McCarthy, What Is Artificial Intelligence, 2011.

[6] J. McCarthy, M. Minsky, N. Rochester y C. Shannon, A Proposal for the Dartmouth Summer Research Project on Artificial Intelligence, 1955.

[7] N. (. S. N. R. Council), Developments in Artificial Intelligence, 1999.

[8] P. McCorduck, Machines Who Think, 2004.

[9] F. B. Ethem Alpaydin, Introduction to Machine Learning, MIT Press, 2014.

[10] Rokach y M. Oded, Data Mining and Knowledge Discovery Handbook, 2010.

[11] U. Neisser, G. Boodoo, T. J. Bouchard, A. W. Boykin, N. Brody, S. J. Ceci, D. F. Halpern, J. C. Loehlin, R. Perloff, R. J. Sternberg y S. Urbina, Intelligence: Knowns and unknowns, American Psych, 1996.

[12] G. Churiwala y A. R. Rebala, An Introduction to Machine Learning, 2019.

[13] G. Baltaza, «https://blogs.oracle.com/datascience/cpu-vs-gpu-in-machine-learning,» 2018. [En línea].

[14] Y. Li, Feng y C. Zhou, «Optimizing Memory Efficiency for Deep Convolutional Neural Networks on GPU,» 2016. [En línea].

[15] R. Ward, «http://ruetersward.com/biblio.html,» 2020. [En línea].

[16] B. B. Chaudhuri, «Digital Document Processing,» [En línea].

[17] «https://web.archive.org/web/20110802175557/https://observatorio.iti.upv.es/resources/project/58,» 2011. [En línea].

[18] C. T. Charles, S. Ching y W. Toru, «The State of the Art in On-Line Handwriting,» *EEE Trans. Pattern Anal,* 1990.

[19] *ISO/IEC JTC 1/SC 2/WG 3,* 2000.

[20] «https://www.tensorflow.org/,» [En línea].

[21] M. Allard, «https://medium.com/inside-machine-learning/what-is-a-transformer-d07dd1fbec04,» 2019. [En línea].

[22] T. Hastie, R. Tibshirani y J. Friedman, «The Elements of Statistical Learning: Data Mining, Inference, and Prediction.,» 2009.

[23] «https://www.javatpoint.com/pytorch-convolutional-neural-network,» [En línea].

[24] A. Krizhevsky, I. Sutskever y G. E. Hinton, Imagenet classification with deep convolutional neural networks, 2012.

[25] C. Burges, L. Bottou, M. Welling, Z. Ghahramani y K. Q. Weinberger, Deep content-based music recommendation, 2013.

[26] «https://www.sciencedirect.com/topics/computer-science/convolutional-layer,» [En línea].

[27] «https://colah.github.io/posts/2015-08-Understanding-LSTMs/,» [En línea].

[28] S. Hochreiter, Untersuchungen zu dynamischen neuronalen Netzen, Institut f. Informatik, Technische Univ. Munich, 1991.

[29] «https://towardsdatascience.com/day-1-2-attention-seq2seq-models-65df3f49e263,» [En línea].

[30] A. L. Martin Längkvist, A Deep Learning Approach with an Attention Mechanism for Automatic Sleep Stage Classification, 2018.

[31] «https://wordnet.princeton.edu/,» [En línea].

[32] T. Andrew, M. Phil y L. John, sense2vecA Fast and Accurate Method for Word Sense Disambiguation In Neural Word Embeddings, 2015.

[33] G. Alex, W. Greg y D. Ivo, Neural turing machines, 2014.

[34] L. Weng, «https://lilianweng.github.io/lil-log/2018/06/24/attention-attention.html,» 2018. [En línea].

[35] I. Sutskever, O. Vinyals y Q. Le, Sequence to sequence learning with neural networks., 2014, p. 12.

[36] R. Colin y E. Daniel, Pruning subsequence search with attention-based embedding. In Proceedings of the 41st IEEE International Conference on Acoustics, Speech, and Signal Processing, 2016.

[37] «www.manythings.org/anki/,» [En línea].

[38] «http://icfhr2018.org/,» 2018. [En línea].

[39] «https://www.cloudera.com/tutorials/introduction-to-convolutional-neural-networks.html,» [En línea].

[40] Pham, Bluche, Kermorvant y Louradour, «Dropout Improves Recurrent Neural Networks for Handwriting

Recognition,» de *International Conference on Frontiers in Handwriting*, 2014.

[41] S. Christian, V. Vincent, I. Sergey, S. Jonathon y W. Zbigniew, «Rethinking the Inception Architecture for Computer Vision,» 2015.

[42] «https://cloud.google.com/tpu/docs/inception-v3-advanced?hl=es-419,» [En línea].

[43] C. Szegedy, V. Vanhoucke y Z. Wojna, Rethinking the Inception Architecture for Computer Vision, 2014.

[44] Y. Mohamed, F. H. Khaled y S. M. Usama, «Accurate, Data-Efficient, Unconstrained Text Recognition with Convolutional Neural Networks,» 2015.

[45] A. Rehman, D. Mohammad, G. Sulong y T. Saba, «Simple and effective techniques for core-region detection and slant correction in offline script recognition,» 2009.

[46] «https://wordnet.princeton.edu/,» [En línea].

ANEXO

Anexo I: Código del modelo de transcripción

```
# -*- coding: utf-8 -*-
"""
@author: Luis
"""

from __future__ import absolute_import, division, print_function, unicode_literals

import tensorflow as tf

# You'll generate plots of attention in order to see which parts of an image
# our model focuses on during captioning
import matplotlib.pyplot as plt

# Scikit-learn includes many helpful utilities
from sklearn.model_selection import train_test_split
from sklearn.utils import shuffle

import sys
import unicodedata
import re
import numpy as np
import os
import time
import json
from glob import glob
from PIL import Image
import pickle
import pandas as pd
import csv


# Converts the unicode file to ascii
def unicode_to_ascii(s):
    return ''.join(c for c in unicodedata.normalize('NFD', s)
        if unicodedata.category(c) != 'Mn')

"""Inception V3 model for Keras.
"""
backend = None
layers = None
models = None
keras_utils = None
```

```python
def conv2d_bn(x,
              filters,
              num_row,
              num_col,
              padding='same',
              strides=(1, 1),
              name=None):
    """Utility function to apply conv + BN.
    # Arguments
        x: input tensor.
        filters: filters in `Conv2D`.
        num_row: height of the convolution kernel.
        num_col: width of the convolution kernel.
        padding: padding mode in `Conv2D`.
        strides: strides in `Conv2D`.
        name: name of the ops; will become `name + '_conv'`
            for the convolution and `name + '_bn'` for the
            batch norm layer.
    # Returns
        Output tensor after applying `Conv2D` and `BatchNormalization`.
    """
    if name is not None:
        bn_name = name + '_bn'
        conv_name = name + '_conv'
    else:
        bn_name = None
        conv_name = None
    if tf.keras.backend.image_data_format() == 'channels_first':
        bn_axis = 1
    else:
        bn_axis = 3
    x = tf.keras.layers.Conv2D(
        filters, (num_row, num_col),
        strides=strides,
        padding=padding,
        use_bias=False,
        name=conv_name)(x)
    x = tf.keras.layers.BatchNormalization(axis=bn_axis, scale=False, name=bn_name)(x)
    x = tf.keras.layers.Activation('relu', name=name)(x)
    return x


def InceptionV3(include_top=True,
                weights='imagenet',
                input_tensor=None,
                input_shape=None,
                pooling=None,
                classes=1000,
                **kwargs):
    global backend, layers, models, keras_utils
```

```
backend, layers, models, keras_utils = get_submodules_from_kwargs(kwargs)

if not (weights in {'imagenet', None} or os.path.exists(weights)):
    raise ValueError('The `weights` argument should be either '
                     '`None` (random initialization), `imagenet` '
                     '(pre-training on ImageNet), '
                     'or the path to the weights file to be loaded.')

if weights == 'imagenet' and include_top and classes != 1000:
    raise ValueError('If using `weights` as `"imagenet"` with `include_top`'
                     ' as true, `classes` should be 1000')

# Determine proper input shape
input_shape = _obtain_input_shape(
    input_shape,
    default_size=299,
    min_size=75,
    data_format=tf.keras.backend.image_data_format(),
    require_flatten=include_top,
    weights=weights)

if input_tensor is None:
    img_input = tf.keras.layers.Input(shape=input_shape)
else:
    if not tf.keras.backend.is_keras_tensor(input_tensor):
        img_input = tf.keras.layers.Input(tensor=input_tensor, shape=input_shape)
    else:
        img_input = input_tensor

if tf.keras.backend.image_data_format() == 'channels_first':
    channel_axis = 1
else:
    channel_axis = 3

x = conv2d_bn(img_input, 32, 3, 3, strides=(2, 2), padding='valid')
x = conv2d_bn(x, 32, 3, 3, padding='valid')
x = conv2d_bn(x, 64, 3, 3)
x = tf.keras.layers.MaxPooling2D((3, 3), strides=(2, 2))(x)

x = conv2d_bn(x, 80, 1, 1, padding='valid')
x = conv2d_bn(x, 192, 3, 3, padding='valid')
x = tf.keras.layers.MaxPooling2D((3, 3), strides=(2, 2))(x)

# mixed 0: 35 x 35 x 256
branch1x1 = conv2d_bn(x, 64, 1, 1)

branch5x5 = conv2d_bn(x, 48, 1, 1)
branch5x5 = conv2d_bn(branch5x5, 64, 5, 5)
```

```
branch3x3dbl = conv2d_bn(x, 64, 1, 1)
branch3x3dbl = conv2d_bn(branch3x3dbl, 96, 3, 3)
branch3x3dbl = conv2d_bn(branch3x3dbl, 96, 3, 3)

branch_pool = tf.keras.layers.AveragePooling2D((3, 3),
                                      strides=(1, 1),
                                      padding='same')(x)
branch_pool = conv2d_bn(branch_pool, 32, 1, 1)
x = tf.keras.layers.concatenate(
    [branch1x1, branch5x5, branch3x3dbl, branch_pool],
    axis=channel_axis,
    name='mixed0')

# mixed 1: 35 x 35 x 288
branch1x1 = conv2d_bn(x, 64, 1, 1)

branch5x5 = conv2d_bn(x, 48, 1, 1)
branch5x5 = conv2d_bn(branch5x5, 64, 5, 5)

branch3x3dbl = conv2d_bn(x, 64, 1, 1)
branch3x3dbl = conv2d_bn(branch3x3dbl, 96, 3, 3)
branch3x3dbl = conv2d_bn(branch3x3dbl, 96, 3, 3)

branch_pool = tf.keras.layers.AveragePooling2D((3, 3),
                                      strides=(1, 1),
                                      padding='same')(x)
branch_pool = conv2d_bn(branch_pool, 64, 1, 1)
x = tf.keras.layers.concatenate(
    [branch1x1, branch5x5, branch3x3dbl, branch_pool],
    axis=channel_axis,
    name='mixed1')

# mixed 2: 35 x 35 x 288
branch1x1 = conv2d_bn(x, 64, 1, 1)

branch5x5 = conv2d_bn(x, 48, 1, 1)
branch5x5 = conv2d_bn(branch5x5, 64, 5, 5)

branch3x3dbl = conv2d_bn(x, 64, 1, 1)
branch3x3dbl = conv2d_bn(branch3x3dbl, 96, 3, 3)
branch3x3dbl = conv2d_bn(branch3x3dbl, 96, 3, 3)

branch_pool = tf.keras.layers.AveragePooling2D((3, 3),
                                      strides=(1, 1),
                                      padding='same')(x)
branch_pool = conv2d_bn(branch_pool, 64, 1, 1)
x = tf.keras.layers.concatenate(
    [branch1x1, branch5x5, branch3x3dbl, branch_pool],
    axis=channel_axis,
```

```
    name='mixed2')

# mixed 3: 17 x 17 x 768
branch3x3 = conv2d_bn(x, 384, 3, 3, strides=(2, 2), padding='valid')

branch3x3dbl = conv2d_bn(x, 64, 1, 1)
branch3x3dbl = conv2d_bn(branch3x3dbl, 96, 3, 3)
branch3x3dbl = conv2d_bn(
    branch3x3dbl, 96, 3, 3, strides=(2, 2), padding='valid')

branch_pool = tf.keras.layers.MaxPooling2D((3, 3), strides=(2, 2))(x)
x = tf.keras.layers.concatenate(
    [branch3x3, branch3x3dbl, branch_pool],
    axis=channel_axis,
    name='mixed3')

# mixed 4: 17 x 17 x 768
branch1x1 = conv2d_bn(x, 192, 1, 1)

branch7x7 = conv2d_bn(x, 128, 1, 1)
branch7x7 = conv2d_bn(branch7x7, 128, 1, 7)
branch7x7 = conv2d_bn(branch7x7, 192, 7, 1)

branch7x7dbl = conv2d_bn(x, 128, 1, 1)
branch7x7dbl = conv2d_bn(branch7x7dbl, 128, 7, 1)
branch7x7dbl = conv2d_bn(branch7x7dbl, 128, 1, 7)
branch7x7dbl = conv2d_bn(branch7x7dbl, 128, 7, 1)
branch7x7dbl = conv2d_bn(branch7x7dbl, 192, 1, 7)

branch_pool = tf.keras.layers.AveragePooling2D((3, 3),
                                      strides=(1, 1),
                                      padding='same')(x)
branch_pool = conv2d_bn(branch_pool, 192, 1, 1)
x = tf.keras.layers.concatenate(
    [branch1x1, branch7x7, branch7x7dbl, branch_pool],
    axis=channel_axis,
    name='mixed4')

# mixed 5, 6: 17 x 17 x 768
for i in range(2):
    branch1x1 = conv2d_bn(x, 192, 1, 1)

    branch7x7 = conv2d_bn(x, 160, 1, 1)
    branch7x7 = conv2d_bn(branch7x7, 160, 1, 7)
    branch7x7 = conv2d_bn(branch7x7, 192, 7, 1)

    branch7x7dbl = conv2d_bn(x, 160, 1, 1)
    branch7x7dbl = conv2d_bn(branch7x7dbl, 160, 7, 1)
    branch7x7dbl = conv2d_bn(branch7x7dbl, 160, 1, 7)
```

```
    branch7x7dbl = conv2d_bn(branch7x7dbl, 160, 7, 1)
    branch7x7dbl = conv2d_bn(branch7x7dbl, 192, 1, 7)

    branch_pool = tf.keras.layers.AveragePooling2D(
        (3, 3), strides=(1, 1), padding='same')(x)
    branch_pool = conv2d_bn(branch_pool, 192, 1, 1)
    x = tf.keras.layers.concatenate(
        [branch1x1, branch7x7, branch7x7dbl, branch_pool],
        axis=channel_axis,
        name='mixed' + str(5 + i))

# mixed 7: 17 x 17 x 768
branch1x1 = conv2d_bn(x, 192, 1, 1)

branch7x7 = conv2d_bn(x, 192, 1, 1)
branch7x7 = conv2d_bn(branch7x7, 192, 1, 7)
branch7x7 = conv2d_bn(branch7x7, 192, 7, 1)

branch7x7dbl = conv2d_bn(x, 192, 1, 1)
branch7x7dbl = conv2d_bn(branch7x7dbl, 192, 7, 1)
branch7x7dbl = conv2d_bn(branch7x7dbl, 192, 1, 7)
branch7x7dbl = conv2d_bn(branch7x7dbl, 192, 7, 1)
branch7x7dbl = conv2d_bn(branch7x7dbl, 192, 1, 7)

branch_pool = tf.keras.layers.AveragePooling2D((3, 3),
                                      strides=(1, 1),
                                      padding='same')(x)
branch_pool = conv2d_bn(branch_pool, 192, 1, 1)
x = tf.keras.layers.concatenate(
    [branch1x1, branch7x7, branch7x7dbl, branch_pool],
    axis=channel_axis,
    name='mixed7')

# mixed 8: 8 x 8 x 1280
branch3x3 = conv2d_bn(x, 192, 1, 1)
branch3x3 = conv2d_bn(branch3x3, 320, 3, 3,
                      strides=(2, 2), padding='valid')

branch7x7x3 = conv2d_bn(x, 192, 1, 1)
branch7x7x3 = conv2d_bn(branch7x7x3, 192, 1, 7)
branch7x7x3 = conv2d_bn(branch7x7x3, 192, 7, 1)
branch7x7x3 = conv2d_bn(
    branch7x7x3, 192, 3, 3, strides=(2, 2), padding='valid')

branch_pool = tf.keras.layers.MaxPooling2D((3, 3), strides=(2, 2))(x)
x = tf.keras.layers.concatenate(
    [branch3x3, branch7x7x3, branch_pool],
    axis=channel_axis,
    name='mixed8')
```

```
# mixed 9: 8 x 8 x 2048
for i in range(2):
    branch1x1 = conv2d_bn(x, 320, 1, 1)

    branch3x3 = conv2d_bn(x, 384, 1, 1)
    branch3x3_1 = conv2d_bn(branch3x3, 384, 1, 3)
    branch3x3_2 = conv2d_bn(branch3x3, 384, 3, 1)
    branch3x3 = tf.keras.layers.concatenate(
        [branch3x3_1, branch3x3_2],
        axis=channel_axis,
        name='mixed9_' + str(i))

    branch3x3dbl = conv2d_bn(x, 448, 1, 1)
    branch3x3dbl = conv2d_bn(branch3x3dbl, 384, 3, 3)
    branch3x3dbl_1 = conv2d_bn(branch3x3dbl, 384, 1, 3)
    branch3x3dbl_2 = conv2d_bn(branch3x3dbl, 384, 3, 1)
    branch3x3dbl = tf.keras.layers.concatenate(
        [branch3x3dbl_1, branch3x3dbl_2], axis=channel_axis)

    branch_pool = tf.keras.layers.AveragePooling2D(
        (3, 3), strides=(1, 1), padding='same')(x)
    branch_pool = conv2d_bn(branch_pool, 192, 1, 1)
    x = tf.keras.layers.concatenate(
        [branch1x1, branch3x3, branch3x3dbl, branch_pool],
        axis=channel_axis,
        name='mixed' + str(9 + i))

if include_top:
    # Classification block
    x = tf.keras.layers.GlobalAveragePooling2D(name='avg_pool')(x)
    x = tf.keras.layers.Dense(classes, activation='softmax', name='predictions')(x)
else:
    if pooling == 'avg':
        x = tf.keras.layers.GlobalAveragePooling2D()(x)
    elif pooling == 'max':
        x = tf.keras.layers.GlobalMaxPooling2D()(x)

# Ensure that the model takes into account
# any potential predecessors of `input_tensor`.
if input_tensor is not None:
    inputs = keras_utils.get_source_inputs(input_tensor)
else:
    inputs = img_input
# Create model.
model = tf.keras.models.Model(inputs, x, name='inception_v3')

# Load weights.
if weights == 'imagenet':
```

```
        if include_top:
            weights_path = tf.keras.utils.get_file(
                'inception_v3_weights_tf_dim_ordering_tf_kernels.h5',
                WEIGHTS_PATH,
                cache_subdir='models',
                file_hash='9a0d58056eeedaa3f26cb7ebd46da564')
        else:
            weights_path = tf.keras.utils.get_file(
                'inception_v3_weights_tf_dim_ordering_tf_kernels_notop.h5',
                WEIGHTS_PATH_NO_TOP,
                cache_subdir='models',
                file_hash='bcbd6486424b2319ff4ef7d526e38f63')
        model.load_weights(weights_path)
    elif weights is not None:
        model.load_weights(weights)

    return model


def preprocess_input(x, **kwargs):
    """Preprocesses a numpy array encoding a batch of images.
    # Arguments
        x: a 4D numpy array consists of RGB values within [0, 255].
    # Returns
        Preprocessed array.
    """
    return preprocess_input_2(x, mode='tf', **kwargs)



def preprocess_sentence(w):
    w = unicode_to_ascii(w.lower().strip())

    w = " ".join(w)
    # creating a space between a word and the punctuation following it
    # eg: "he is a boy." => "he is a boy ."
    # Reference:- https://stackoverflow.com/questions/3645931/python-padding-punctuation-with-
white-spaces-keeping-punctuation
    w = re.sub(r"([?.!,¿])", r" \1 ", w)
    w = re.sub(r'[" "]+', " ", w)

    # replacing everything with space except (a-z, A-Z, ".", "?", "!", ",")
    w = re.sub(r"[?.!,¿]+", " ", w)

    # adding a start and an end token to the sentence
    # so that the model know when to start and stop predicting.
    w = '<start> ' + w + ' <end>'
    print(w)
    return w

# Store captions and image names in vectors
```

```
all_captions = []
all_img_name_vector = []



#for annot in annotations['annotations']:
#    caption = '<start> ' + annot['caption'] + ' <end>'
#    image_id = annot['image_id']
#    full_coco_image_path = PATH + 'COCO_train2014_' + '%012d.jpg' % (image_id)

#    all_img_name_vector.append(full_coco_image_path)
#    all_captions.append(caption)


h = pd.read_csv('train.csv') # cargo la tabla del excel

word_pairs = []
for i in h.transpose():
     word_pairs[len(word_pairs):len(word_pairs)] = [h['imgName'][i] + ".jpg"] # se procesa imagen
     all_captions[len(all_captions):len(all_captions)] = [preprocess_sentence(h['transcription'][
i])]

print(all_captions)

all_img_name_vector = word_pairs
# Shuffle captions and image_names together
# Set a random state
train_captions, img_name_vector = shuffle(all_captions,
                                          all_img_name_vector,
                                          random_state=1)



num_examples = 11425
train_captions = train_captions[:num_examples]
img_name_vector = img_name_vector[:num_examples]


len(train_captions), len(all_captions)


def load_image(image_path):
    img = tf.io.read_file(image_path)
    img = tf.image.decode_jpeg(img, channels=1)
    img = tf.image.grayscale_to_rgb(img)
    img = tf.image.resize(img, (128, 2050))
    img = preprocess_input(img)
    return img, image_path

image_model = InceptionV3(include_top=False, weights='imagenet')
new_input = image_model.input
```

```
hidden_layer = image_model.layers[-1].output

image_features_extract_model = tf.keras.Model(new_input, hidden_layer)


# Get unique images
encode_train = sorted(set(img_name_vector))

# Feel free to change batch_size according to your system configuration
image_dataset = tf.data.Dataset.from_tensor_slices(encode_train)
image_dataset = image_dataset.map(load_image, num_parallel_calls=tf.data.experimental.AUTOTUNE).b
atch(16)

for img, path in image_dataset:
  batch_features = image_features_extract_model(img)
  batch_features = tf.reshape(batch_features, (batch_features.shape[0], -
1, batch_features.shape[3]))

  for bf, p in zip(batch_features, path):
    path_of_feature = p.numpy().decode("utf-8")
    np.save(path_of_feature, bf.numpy())


# Find the maximum length of any caption in our dataset
def calc_max_length(tensor):
    return max(len(t) for t in tensor)

# Choose the top 5000 words from the vocabulary
top_k = 5000
tokenizer = tf.keras.preprocessing.text.Tokenizer(num_words=top_k,
                                                  oov_token="<unk>",
                                                  filters='!"#$%&()*+.,-/:;=?@[\]^_`{|}~ ')
tokenizer.fit_on_texts(train_captions)
train_seqs = tokenizer.texts_to_sequences(train_captions)

tokenizer.word_index['<pad>'] = 0
tokenizer.index_word[0] = '<pad>'

# Create the tokenized vectors
train_seqs = tokenizer.texts_to_sequences(train_captions)


# Pad each vector to the max_length of the captions
# If you do not provide a max_length value, pad_sequences calculates it automatically
cap_vector = tf.keras.preprocessing.sequence.pad_sequences(train_seqs, padding='post')

# Calculates the max_length, which is used to store the attention weights
max_length = calc_max_length(train_seqs)

# Create training and validation sets using an 80-20 split
```

```
img_name_train, img_name_val, cap_train, cap_val = train_test_split(img_name_vector,
                                                                    cap_vector,
                                                                    test_size=0,
                                                                    random_state=0)



len(img_name_train), len(cap_train), len(img_name_val), len(cap_val)


# Feel free to change these parameters according to your system's configuration

BATCH_SIZE = 64
BUFFER_SIZE = 1000
embedding_dim = 256
units = 512
vocab_size = top_k + 1
num_steps = len(img_name_train) // BATCH_SIZE

# Load the numpy files
def map_func(img_name, cap):
  img_tensor = np.load(img_name.decode('utf-8')+'.npy')
  return img_tensor, cap


dataset = tf.data.Dataset.from_tensor_slices((img_name_train, cap_train))

# Use map to load the numpy files in parallel
dataset = dataset.map(lambda item1, item2: tf.numpy_function(
          map_func, [item1, item2], [tf.float32, tf.int32]),
          num_parallel_calls=tf.data.experimental.AUTOTUNE)

# Shuffle and batch
dataset = dataset.shuffle(BUFFER_SIZE).batch(BATCH_SIZE)
dataset = dataset.prefetch(buffer_size=tf.data.experimental.AUTOTUNE)



class LuongAttention(tf.keras.Model):
  def __init__(self, units):
    super(LuongAttention, self).__init__()
    self.W1 = tf.keras.layers.Dense(units)
    self.W2 = tf.keras.layers.Dense(units)
    self.V = tf.keras.layers.Dense(1)

  def call(self, features, hidden):
    # features(CNN_encoder output) shape == (batch_size, 64, embedding_dim)

    # hidden shape == (batch_size, hidden_size)
    # hidden_with_time_axis shape == (batch_size, 1, hidden_size)
```

```
    hidden_with_time_axis = tf.expand_dims(hidden, 1)

    # score shape == (batch_size, 64, hidden_size)
    score = tf.nn.tanh(self.W1(features) + self.W2(hidden_with_time_axis))

    # attention_weights shape == (batch_size, 64, 1)
    # you get 1 at the last axis because you are applying score to self.V
    attention_weights = tf.nn.softmax(self.V(score), axis=1)

    # context_vector shape after sum == (batch_size, hidden_size)
    context_vector = attention_weights * features
    context_vector = tf.reduce_sum(context_vector, axis=1)

    return context_vector, attention_weights


class CNN_Encoder(tf.keras.Model):
    # Since you have already extracted the features and dumped it using pickle
    # This encoder passes those features through a Fully connected layer
    def __init__(self, embedding_dim):
        super(CNN_Encoder, self).__init__()
        self.fc = tf.keras.layers.Dense(embedding_dim)
    def call(self, x):
        x = self.fc(x)
        x = tf.nn.relu(x)
        return x


class RNN_Decoder(tf.keras.Model):
  def __init__(self, embedding_dim, units, vocab_size):
    super(RNN_Decoder, self).__init__()
    self.units = units

    self.embedding = tf.keras.layers.Embedding(vocab_size, embedding_dim)
    self.gru = tf.keras.layers.GRU(self.units,
                                   return_sequences=True,
                                   return_state=True,
                                   recurrent_initializer='glorot_uniform')
    self.fc1 = tf.keras.layers.Dense(self.units)
    self.fc2 = tf.keras.layers.Dense(vocab_size)

    self.attention = LuongAttention(self.units)

  def call(self, x, features, hidden):
    # defining attention as a separate model
    context_vector, attention_weights = self.attention(features, hidden)

    # x shape after passing through embedding == (batch_size, 1, embedding_dim)
    x = self.embedding(x)
```

```
    # x shape after concatenation == (batch_size, 1, embedding_dim + hidden_size)
    x = tf.concat([tf.expand_dims(context_vector, 1), x], axis=-1)

    # passing the concatenated vector to the GRU
    output, state = self.gru(x)

    # shape == (batch_size, max_length, hidden_size)
    x = self.fc1(output)

    # x shape == (batch_size * max_length, hidden_size)
    x = tf.reshape(x, (-1, x.shape[2]))

    # output shape == (batch_size * max_length, vocab)
    x = self.fc2(x)

    return x, state, attention_weights

  def reset_state(self, batch_size):
    return tf.zeros((batch_size, self.units))


encoder = CNN_Encoder(embedding_dim)
decoder = RNN_Decoder(embedding_dim, units, vocab_size)


optimizer = tf.keras.optimizers.Adam()
loss_object = tf.keras.losses.SparseCategoricalCrossentropy(
    from_logits=True, reduction='none')

def loss_function(real, pred):
  mask = tf.math.logical_not(tf.math.equal(real, 0))
  loss_ = loss_object(real, pred)

  mask = tf.cast(mask, dtype=loss_.dtype)
  loss_ *= mask

  return tf.reduce_mean(loss_)


checkpoint_path = "./checkpoints/train"
ckpt = tf.train.Checkpoint(encoder=encoder,
                           decoder=decoder,
                           optimizer = optimizer)
ckpt_manager = tf.train.CheckpointManager(ckpt, checkpoint_path, max_to_keep=5)



start_epoch = 0
```

```python
if ckpt_manager.latest_checkpoint:
  start_epoch = int(ckpt_manager.latest_checkpoint.split('-')[-1])
  # restoring the latest checkpoint in checkpoint_path
  ckpt.restore(ckpt_manager.latest_checkpoint)
```

```python
# adding this in a separate cell because if you run the training cell
#@tf.function
def train_step(img_tensor, target):
  loss = 0

  # initializing the hidden state for each batch
  # because the captions are not related from image to image
  hidden = decoder.reset_state(batch_size=target.shape[0])

  dec_input = tf.expand_dims([tokenizer.word_index['<start>']] * target.shape[0], 1)

  with tf.GradientTape() as tape:
      features = encoder(img_tensor)

      for i in range(1, target.shape[1]):
          # passing the features through the decoder
          predictions, hidden, _ = decoder(dec_input, features, hidden)

          loss += loss_function(target[:, i], predictions)

          # using teacher forcing
          dec_input = tf.expand_dims(target[:, i], 1)

  total_loss = (loss / int(target.shape[1]))

  trainable_variables = encoder.trainable_variables + decoder.trainable_variables

  gradients = tape.gradient(loss, trainable_variables)

  optimizer.apply_gradients(zip(gradients, trainable_variables))

  return loss, total_loss

EPOCHS = 250

for epoch in range(start_epoch, EPOCHS):
    start = time.time()
    total_loss = 0

    for (batch, (img_tensor, target)) in enumerate(dataset):
        batch_loss, t_loss = train_step(img_tensor, target)
        total_loss += t_loss
```

```
        if batch % 100 == 0:
            print ('Epoch {} Batch {} Loss {:.4f}'.format(
              epoch + 1, batch, batch_loss.numpy() / int(target.shape[1])))
    # storing the epoch end loss value to plot later
    #loss_plot.append(total_loss / num_steps)

    if epoch % 5 == 0:
      ckpt_manager.save()

    print ('Epoch {} Loss {:.6f}'.format(epoch + 1,
                                         total_loss/num_steps))
    print ('Time taken for 1 epoch {} sec\n'.format(time.time() - start))



def evaluate(image):
    #attention_plot = np.zeros((max_length, attention_features_shape))

    hidden = decoder.reset_state(batch_size=1)

    temp_input = tf.expand_dims(load_image(image)[0], 0)
    img_tensor_val = image_features_extract_model(temp_input)
    img_tensor_val = tf.reshape(img_tensor_val, (img_tensor_val.shape[0], -
1, img_tensor_val.shape[3]))

    features = encoder(img_tensor_val)


    dec_input = tf.expand_dims([tokenizer.word_index['<start>']], 0)
    result = []

    for i in range(max_length):
        predictions, hidden, attention_weights = decoder(dec_input, features, hidden)

        predicted_id = tf.random.categorical(predictions, 1)[0][0].numpy()
        result.append(tokenizer.index_word[predicted_id])

        if tokenizer.index_word[predicted_id] == '<end>':
            return result

        dec_input = tf.expand_dims([predicted_id], 0)

    return result

##################################CER################################################
def levenshtein(s1, s2, cost={}):
        # based on Wikipedia/Levenshtein_distance#Python
        if len(s1) < len(s2):
            return levenshtein(s2, s1)
```

```
        if len(s2) == 0:
            return len(s1)

        def get_cost(c1, c2, cost):
            return 0 if (c1 == c2) else cost.get((c1, c2), 1)

        previous_row = range(len(s2) + 1)
        for i, c1 in enumerate(s1):
            current_row = [i + 1]
            for j, c2 in enumerate(s2):
                insertions = previous_row[j + 1] + 1 # j+1 instead of j since previous_row and cu
rrent_row are one character longer
                deletions = current_row[j] + 1       # than s2
                substitutions = previous_row[j] + get_cost(c1, c2, cost)
                current_row.append(min(insertions, deletions, substitutions))
            previous_row = current_row
        previous_row[-1] = previous_row[-1]
        porcentaje = previous_row[-1]/len(s1)
        return previous_row[-1], porcentaje


################################CER#########################################################


CER = 0

for lin in h
      lin = lin + 2
      result = evaluate(h['imgName'][lin] + ".jpg")
      num_errores, porcentaje_errores = levenshtein(result, h['transcription'][lin])
      CER = (porcentaje_errores + CER*(lin - 2))/(lin - 1); print(a)
      print("CER")
      print(CER)
```

GLOSARIO

CNN	convolutional neural network
OCR	optical character recognitio
ICR	intelligent character recognition
RNN	recurrent neural network
LSTM	long short-term memory
MLP	multilayer perceptron
PNL	procesamiento del lenguaje natural
GPU	Graphics Processor Unit
CPU	Central Processing Unit