Trabajo Fin de Máster
Ingeniería de Telecomunicación

Estructura DL para reconocimiento de escritura manuscrita

Autora: Julia Moreno Casanova
Tutores: Juan José Murillo Fuentes
Jose Carlos Aradillas Jaramillo

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

Sevilla, 2021

Trabajo Fin de Máster
Ingeniería de Telecomunicación

Estructura DL para reconocimiento de escritura manuscrita

Autora:
Julia Moreno Casanova

Tutores:
Juan José Murillo Fuentes
Catedrático de Universidad
Jose Carlos Aradillas Jaramillo

Dpto. de Teoría de la Señal y Comunicaciones
Escuela Técnica Superior de Ingeniería
Universidad de Sevilla
Sevilla, 2021

Trabajo Fin de Máster: Estructura DL para reconocimiento de escritura manuscrita

Autora: Julia Moreno Casanova

Tutores: Juan José Murillo Fuentes

Jose Carlos Aradillas Jaramillo

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

Presidente:

Vocales:

Secretario:

Acuerdan otorgarle la calificación de:

Sevilla, 2021

El Secretario del Tribunal

A mi familia

Agradecimientos

En este apartado me gustaría agradecer, en primer lugar, a mi familia y amigos, que han estado apoyándome constantemente a lo largo de estos duros años de Máster, y que no han dejado nunca de hacerlo ahora que finalmente termina.

Por supuesto, agradecer también a Juan José Murillo Fuentes, tutor del proyecto, por haberme introducido a un tema tan interesante como desconocido para mí. Agradecerle además, los materiales que me ha facilitado para el desarrollo del proyecto, así como las reuniones iniciales de control del mismo, en las que, con mucha paciencia, me explicó todos los matices necesarios para que pudiera construir el algoritmo deseado.

En resumen, muchas gracias a todos.

Julia Moreno Casanova

Sevilla, 2021

Resumen

En este Trabajo Fin de Máster, se plantea una estructura de Deep Learning orientada al reconocimiento de escritura manuscrita. Esta estructura, la podemos definir como una especie de sistema encargado de segmentar las líneas de una imagen de texto que le pasemos como entrada. Evidentemente, la salida de este sistema, es decir, la imagen segmentada, podría emplearse como entrada en sistemas de reconocimiento óptico de caracteres, sistemas de transcripción de texto, o cualquier otro que requiera que las líneas de un documento manuscrito deban estar delimitadas. Por tanto, el tipo de algoritmo que planteamos, desarrolla una tarea muy importante, pues ayuda a que un texto manuscrito se pueda digitalizar, y así poder aplicarle los mismos algoritmos de automatización que se emplean para documentos escritos a ordenador.

El código, donde se plasma este sistema, se ha elaborado, principalmente, haciendo uso de las bibliotecas Keras y TensorFlow 2.0 de Python, Numpy, OpenCV2, Scipy, entre otras. Además, se han empleado dos bases de datos para el entrenamiento del algoritmo propuesto: la ICDAR 2013 [14], ya utilizada por [3] (referencia base para el desarrollo de nuestro proyecto), y la DIVA-HisDB [17], para la que es necesario llevar a cabo un proceso de adaptación.

Finalmente, decir que, en este trabajo, se recogen y explican, paso a paso, todos los procesos que realiza el algoritmo propuesto, así como los resultados extraídos del mismo, sus errores y posibles mejoras.

Abstract

In this Master Thesis, we propose a Deep Learning structure oriented to handwriting recognition. This structure can be defined as a kind of system in charge of segmenting the lines of a text image that we pass it as input. Obviously, the output of this system, i.e. the segmented image, could be used as input in optical character recognition systems, text transcription systems, or any other system that requires the lines of a handwritten document to be delimited. Therefore, the type of algorithm we are proposing performs a very important task, as it helps to digitise a handwritten text, so that the same automation algorithms that are used for computer-written documents can be applied to it.

The code, which embodies this system, has been developed mainly using the Python libraries Keras and TensorFlow 2.0, Numpy, OpenCV2, Scipy, among others. In addition, two databases have been used to train the proposed algorithm: ICDAR 2013 [14], already used by [3] (a basic reference for the development of our project), and DIVA-HisDB [17], for which it is necessary to carry out an adaptation process.

Finally, this work includes and explains, step by step, all the processes carried out by the proposed algorithm, as well as the results extracted from it, its errors and possible improvements.

Índice

ÍNDICE DE TABLAS

ÍNDICE DE FIGURAS

1 INTRODUCCIÓN

"La escritura es la pintura de la voz"

-Voltaire -

Los documentos históricos manuscritos son tremendamente importantes para la historia de la humanidad. Tanto es así, que estos son capaces de conectar el presente y el pasado de una forma casi inmediata. Hoy en día solemos convertirlos a su formato digital, de manera que investigadores de todo el mundo puedan acceder a ellos y tratarlos de una forma más sencilla. No obstante, esto supone todo un reto para la persona que consulta el documento, si es que quiere realizar una búsqueda por palabras o identificar de forma automática al escritor del mismo. Acciones que acostumbramos a realizar en textos escritos a ordenador, y que nos simplifican muchísimo cualquier consulta.

De esta forma, para tratar los problemas de automatización inherentes en los documentos manuscritos, la segmentación de las líneas de texto de las imágenes que recogen la versión digital de estos archivos, se convierte en una operación fundamental de preprocesamiento para afrontar muchos de estos problemas de automatización.
A continuación se hace un resumen de los tipos de segmentación existentes, poniendo el foco especialmente en la segmentación de líneas de texto.

1.1 Tipos de segmentación

La segmentación de documentos se puede definir como la técnica necesaria para subdividir un manuscrito en regiones de texto y no texto (gráficos, imágenes, etc…)

En este subapartado se recogen varios tipos de segmentación de textos manuscritos, repasando desde los métodos más sencillos, tales como la detección de bordes de una página o la segmentación por regiones, a los mecanismos más complejos, como pueden ser la segmentación de líneas de texto, palabras y caracteres. Todos ellos se enumerarán y explicarán detenidamente en esta sección.

1.1.1 Detección de bordes en una página

Este tipo de segmentación es la más básica que existe en lo que se refiere a la segmentación de manuscritos. La podemos encontrar en muchas aplicaciones de escaneado de documentos, tanto móviles como las propias ligadas a escáneres de impresoras. Mostramos un ejemplo de la misma en la Figura 1-1, que encontramos a continuación:

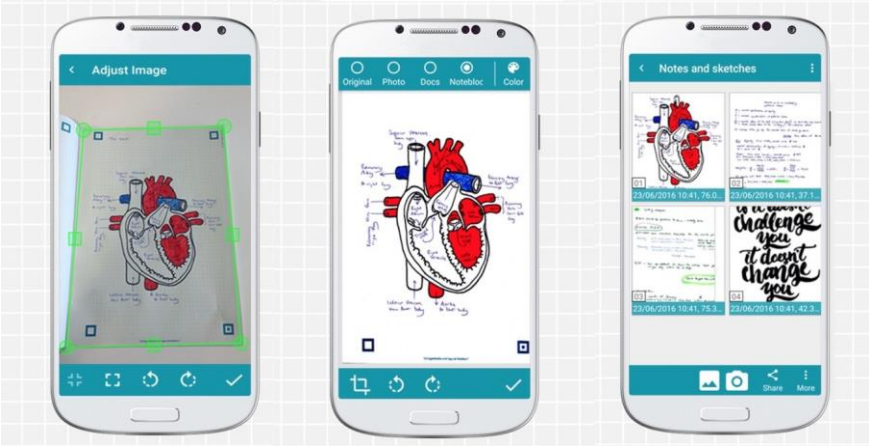


Figura 1-1. Detección de bordes en una página, mediante una aplicación de escaneado de documentos [27]

1.1.2 Delimitación de regiones de texto en documentos manuscritos

En esta parte nos centramos en la delimitación del texto de un documento a través de bloques, que pueden ser: a) rectangulares y b) poligonales. En ambos casos solo se estará delimitando texto puro, es decir, no van a existir ni gráficos, ni figuras dentro de la página que se va a segmentar.

En el caso de realizar esta *delimitación con bloques rectangulares*, cuando se mezclan líneas cortas y largas en un párrafo de texto, se debe considerar que, a la hora de segmentar, se seleccionaría la longitud de línea más larga, lo que supondría un problema, ya que no sería una segmentación precisa. De la misma manera, hay que tener en cuenta que dicho tipo de delimitación presenta una serie de problemas ante textos cuyas líneas tienen direcciones de escritura diferentes. En la Figura 1-2, podemos observar la *delimitación con bloques rectangulares*.

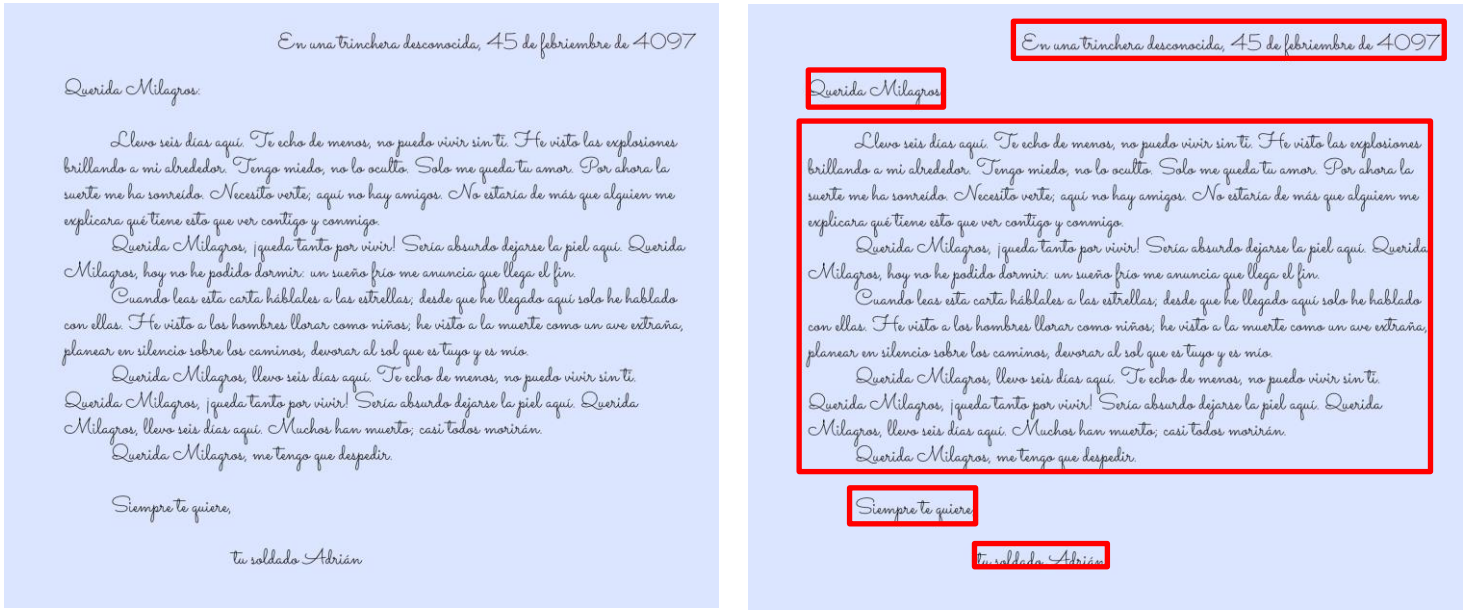


Figura 1-2. Delimitación con bloques rectangulares de regiones de texto en documentos manuscritos [28]

Por otra parte, la *delimitación con bloques poligonales* se presenta como la alternativa a la anterior, resolviendo el problema de la elección de la línea de mayor longitud para llevar a cabo la segmentación. Con esta técnica lo que estaríamos consiguiendo es delimitar el texto rigurosamente, realizando tantos trazos como requiera la forma del párrafo de texto. En la Figura 1-3, se observa esta clase de delimitación.

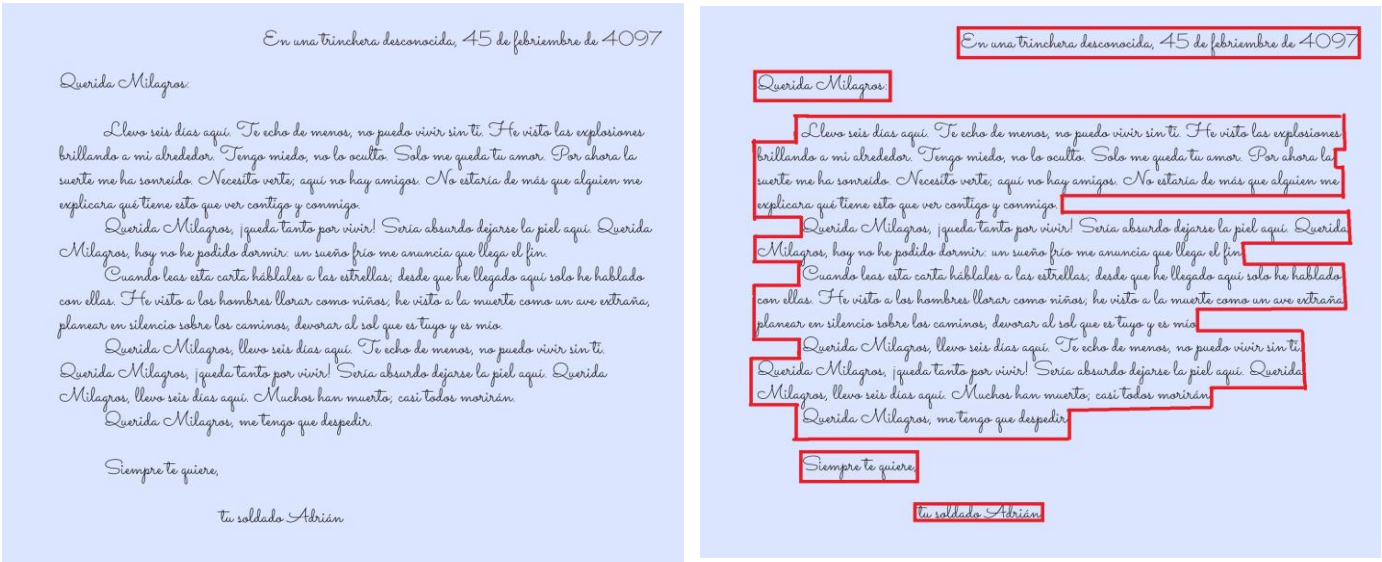


Figura 1-3. Delimitación con bloques poligonales de regiones de texto en documentos manuscritos [28]

1.1.3 Segmentación semántica a nivel de píxel

Este tipo de segmentación [1] tiene como fin el etiquetado de la imagen del documento a nivel de píxel. Esto es, que para cada píxel se asigna una o más etiquetas, que pueden ser: cuerpo de texto principal, decoraciones, anotaciones o comentarios y fondo [2]. Para más información, consultar [3], o seguir el método desarrollado en [2]. En la Figura 1-4, se puede observar un ejemplo de la segmentación semántica nivel de píxel.

(a) (b)

Figura 1-4. (a) Imagen en el dominio RGB; y (b) Segmentación semántica a nivel de píxel [2]

1.1.4 Segmentación de líneas de texto

La segmentación de líneas de texto en documentos manuscritos es un paso muy importante en lo que se refiere al procesamiento de los mismos. De hecho, en este tipo de textos la detección de las líneas que los componen es un problema cuya resolución no es sencilla. Actualmente, se suele tratar dicho problema en base a dos asunciones: a) que el espacio entre líneas vecinas es significativa y b) que las líneas son más o menos rectas. No obstante, para los textos manuscritos estas asunciones pueden no ser válidas.

En general la técnica consiste, como su propio nombre indica, en la segmentación del texto en líneas. Cabe destacar que existen infinidad de algoritmos para llevar a cabo esta tarea. Como ejemplo, se hace referencia al método basado en el modelo de Mumford-Shah (MS) que encontramos en [4]. En este algoritmo la imagen de texto viene representada por dos regiones uniformes: la región de líneas de texto y la región de fondo. De esta forma, el modelo consigue hacer una segmentación bastante buena de las líneas.

También podemos encontrar algoritmos, como el presentado en [2], que a través de una fase previa de segmentación a nivel de pixel, consigue obtener polígonos precisos y ajustados en torno a las líneas de texto, consiguiéndose así una extracción precisa del conjunto del texto. En la Figura 1-5 se muestra un ejemplo de segmentación de líneas de texto.

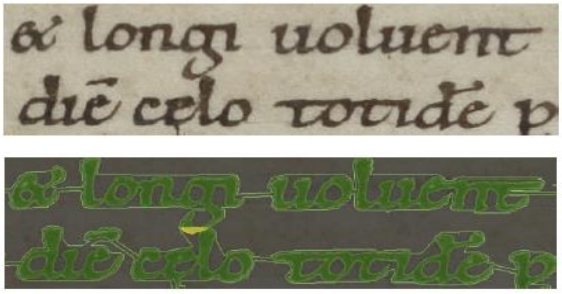


Figura 1-5.Segmentación de dos líneas de texto [2]

Generalmente todos estos métodos de segmentación de líneas de texto, pueden ser clasificados en tres grupos bien diferenciados: métodos globales, encargados de estimar en primer lugar la localización de las líneas de texto; métodos locales, encargados de encontrar primero las unidades locales; y métodos híbridos, los cuales intentan incorporar tanto la información global como la local, para formar las líneas de texto [5].

1.1.5 Segmentación de palabras y de caracteres

En este subapartado se pone el foco en la segmentación de palabras y caracteres [6].

- **Segmentación de palabras**: La idea general de este tipo de segmentación se basa en detectar todos los espacios entre palabras dentro de una misma línea de texto (directamente en la imagen del documento manuscrito). Normalmente, dentro de un texto hay muchos caracteres que se superponen, de manera que esa superposición se puede considerar como un componente conectado (CC), existiendo muchos de ellos en una línea de texto. En la Figura 1-6 vemos un ejemplo de este proceso.

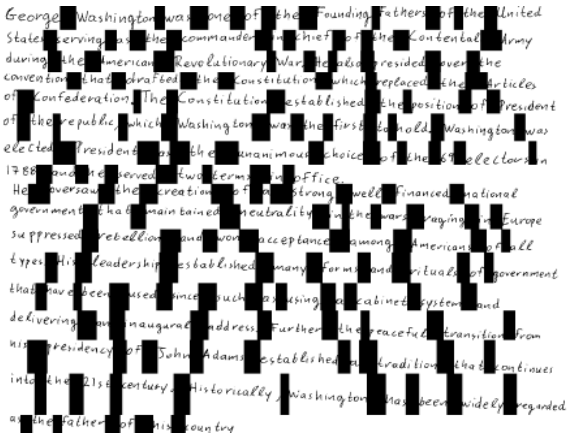


Figura 1-6. Segmentación de palabras [6]

- **Segmentación de caracteres**: Aquí nos encontramos con una ampliación de la técnica anterior, donde el análisis y el refinamiento de los componentes conectados dentro de una misma línea de texto, toma protagonismo. Con ello, conseguimos localizar el espacio entre caracteres, enfoque que ya se determinaba en la segmentación de palabras. En la Figura 1-7, se puede apreciar un ejemplo de este tipo de segmentación.

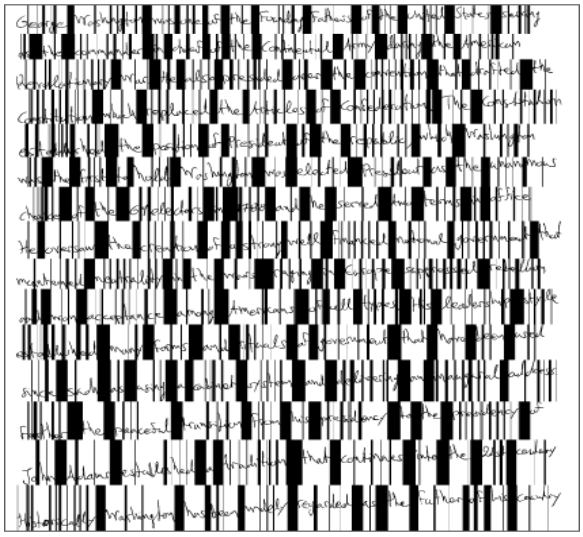


Figura 1-7. Segmentación de caracteres [6]

1.1.6 Detección de la espina dorsal de cada línea del texto

Este tipo de segmentación introduce un nuevo concepto: la *espina dorsal*. La espina dorsal de una línea de texto, se podría definir como la recta que cruza dicha línea y determina la dirección principal de esta [3] (técnica muy parecida al caso de los bloques poligonales). Para más información, consultar [3].

Además, cabe resaltar que este método se presenta por primera vez en [5], documento en el que profundizaremos más adelante. Podemos ver un ejemplo de la detección de la espina dorsal en la Figura 1-8, que se muestra a continuación.

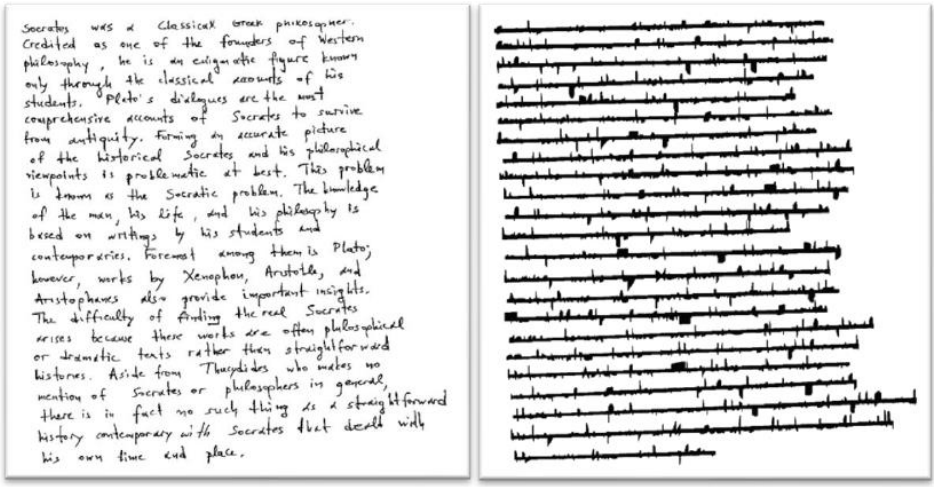


Figura 1-8. Detección de la espina dorsal del texto

1.1.7 Detección de la línea base (Base Line)

Aquí encontramos una técnica que cada vez está más en tendencia, la detección de la línea base o en inglés *baseline*. Una línea base se puede definir como aquella línea "virtual" en la que descansan la mayoría de los caracteres.

Las líneas de texto se anotan con una sola línea de base, de manera que los objetos no textuales (dibujos, gráficos etc…) no son anotados. Este tipo de técnica se ha empleado en numerosas tareas, como la de que se presenta en [7]. En la Figura 1-9, se puede observar el funcionamiento de este método.

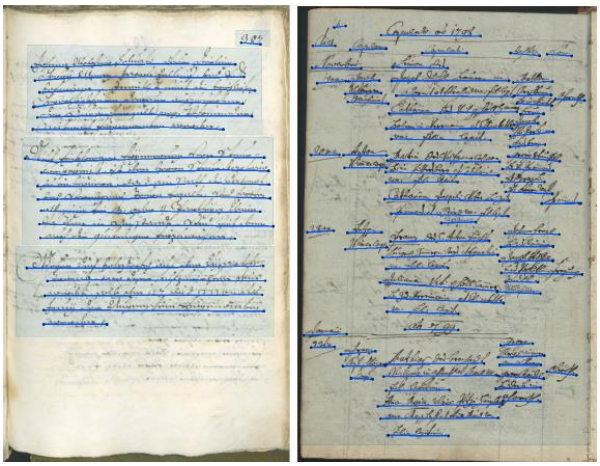


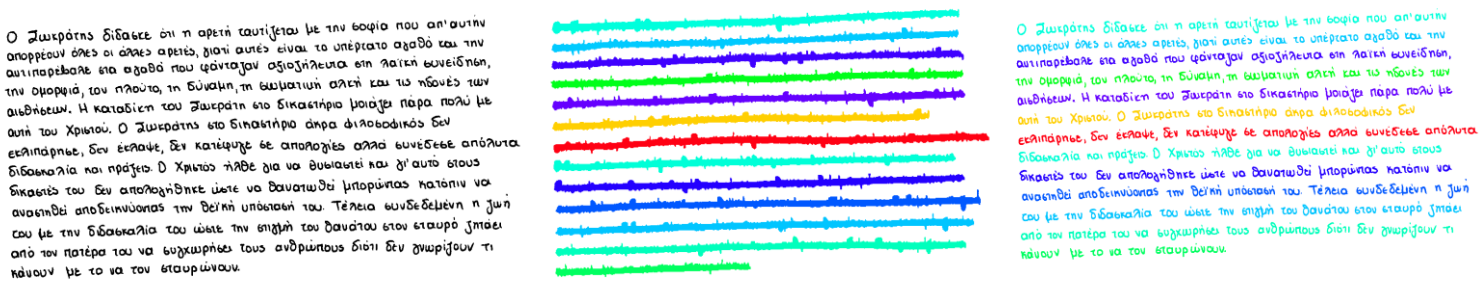
Figura 1-9. Detección de la línea base (Base Line) [7]

1.2 Objetivos del Proyecto

Una vez que hemos entrado en materia, habiendo descrito varios tipos de segmentación, estamos en disposición de describir los objetivos del proyecto.

En primer lugar, el objetivo principal del trabajo es la *revisión y optimización* de una estructura de Deep Learning encargada de segmentar las líneas de texto de imágenes de documentos manuscritos, empleando pues el lenguaje de programación Python.

Para desempeñar dicho objetivo, partimos del modelo construido en [3], el cual define un algoritmo capaz de crear una *máscara de segmentación* donde las líneas vengan diferenciadas por los grosores de las mismas. Una vez que se tiene la máscara anterior, se emplea para compararla con la imagen del texto original píxel a píxel, obteniendo la segmentación de dicho texto manuscrito. En la Figura 1-10, vemos el proceso de segmentación descrito.

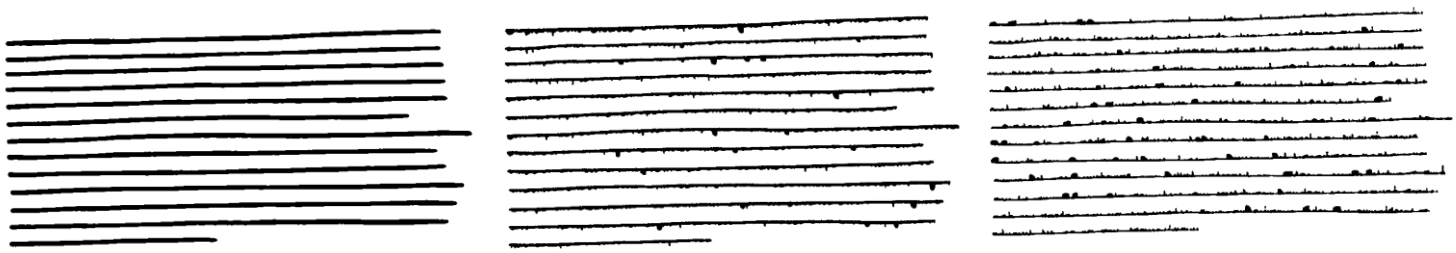


(a) (b) (c)

Figura 1-10. (a) Imagen 0001 original del dataset ICDAR 2013; (b) Máscara de segmentación de (a); (c) Segmentación de (a)

No obstante, en [3] se señala que existen varios obstáculos para crear la máscara anterior. El más habitual, en los textos manuscritos, es la existencia de los llamados *caracteres conflictivos*, que no son más que caracteres pertenecientes a líneas distintas pero que se tocan entre sí. Para poder resolver este problema, y por tanto, evitar que dichos caracteres se toquen, en [3] se decidió emplear 3 modelos distintos de redes neurales: uno para extraer los esqueletos de las líneas, otro para extraer las partes inferiores y un último para extraer las superiores. Sin embargo, en nuestro trabajo se propone optimizar esta tarea, y en vez de usar tres modelos de redes neurales, emplear solo dos, constituyendo un sistema más eficiente y nada redundante. Así pues, este proyecto se caracteriza por exponer dos modelos de redes neurales: uno encargado de extraer los esqueletos de las líneas, tarea que englobamos bajo el nombre de problema de Thick Backbone, como ya se hacía antes en [3]; y otro encargado de extraer tanto las partes superiores de las líneas como las inferiores, con una sola red, tarea que denominamos problema de ZigZag Backbone al igual que en [3].

Finalmente, con un algoritmo diseñado para esta tarea final [3], se conseguirá una relación entre los tres tipos de partes extraídas, de manera que formen la máscara deseada. Así pues, aunque el texto presente *caracteres conflictivos*, como se han separado las líneas en tres partes es muy complicado que la segmentación del mismo presente este defecto. Esto es, que la parte inferior (superior) de la línea, solape con la parte superior (inferior), de la línea inferior (superior). En la siguiente Figura 1-11, se pueden apreciar las tres partes que se extraen de un texto manuscrito, sin ser previamente procesadas (lo que llamaríamos en primera instancia "pre-segmentación").



(a) (b) (c)

Figura 1-11. (a) Esqueleto de la imagen 001 del dataset ICDAR2013; (b) Parte inferior de las líneas de (a); (c) Parte superior de la las líneas de (a)

Otro punto de la lista de objetivos del trabajo es el de detectar y corregir el máximo número de casos de error detectados posible. De manera que se mejore la robustez del sistema propuesto.

Otro objetivo importante a destacar en este proyecto es el de adaptar otra base de datos, distinta en gran medida a la empleada en [3], al algoritmo desarrollado. De esta manera, una de las metas finales del trabajo será la evaluación posterior del método propuesto con ambas bases de datos, de forma que se puedan realizar varias comparaciones entre las mismas incluyendo en estas el algoritmo sin mejorar, es decir, el presentado en [3].

1.3 Estructura de la memoria

Esta memoria consta de un total de ocho capítulos y un anexo, siendo el primero éste que estamos desarrollando. En cuanto al siguiente capítulo, o sea el *capítulo dos,* se puede decir que hacemos una definición de los tipos de arquitectura de red más empleados en la actualidad, repasando también varios algoritmos previos al nuestro, que destacan por sus métodos de segmentación de líneas de textos manuscritos.

Posteriormente, en el *capítulo tres,* se enumeran una serie de bases de datos de textos manuscritos, interesantes para aplicar el proceso de segmentación de líneas de texto (se terminan eligiendo dos de ellas para trabajarlas en nuestro proyecto).

Seguidamente, en el *capítulo cuatro*, se hace un estudio de la estructura de las bases de datos seleccionadas en el capítulo dos, así como la aportación de una explicación detallada de la generación de los archivos de ground-truth y de los tensores. Dichos archivos serán necesarios para entrenar y probar el algoritmo propuesto.

Así pues, es en el *capítulo cinco*, donde se empieza a abordar el tema de la "pre-segmentación". Aquí, antes que nada, se presenta la propuesta de nueva red neuronal del proyecto. Después se recogen los resultados del entrenamiento y evaluación del algoritmo para ambas bases de datos y se comparan.

Más tarde, en el *capítulo seis*, se describe todo lo referente al proceso real de segmentación. Además, se plantean y explican los procesos de detección y corrección de los diferentes errores pertinentes. Finalmente, se detallan los resultados finales de segmentación del proyecto para ambas bases de datos, y se comparan con otros algoritmos.

Como en todo trabajo, en el *capítulo siete*, se enumeran las conclusiones extraídas del mismo, para después, en el *capítulo ocho*, presentar varias mejoras posibles del algoritmo propuesto.

Con objeto de ilustrar mejor todo este proceso, se muestra a continuación, en la Figura 1-12, un esquema que recoge la estructura "*pipeline*" del proyecto. Cada bloque que compone el diagrama se subdividirá a su vez en otros bloques, empleados para describir la función del bloque raíz, cosa que se hará en el capítulo correspondiente.

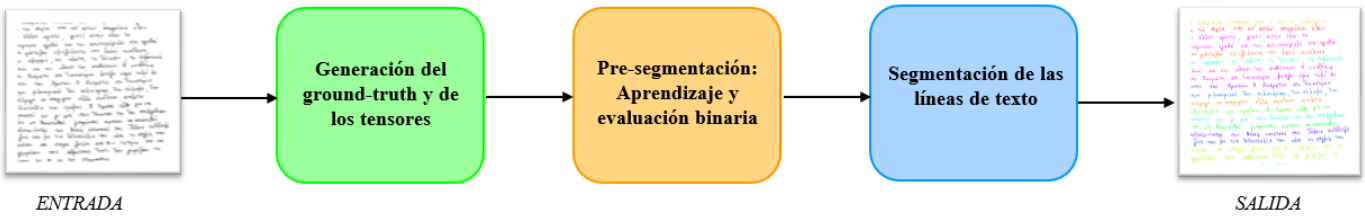


Figura 1-12. Estructura "*pipeline*" del proyecto, compuesta por: la entrada del sistema, que se trata de las imágenes de la base de datos seleccionada; el bloque de generación de ground-truth y tensores; el bloque de pre-segmentación; el bloque de segmentación; y por último, la salida del sistema, que se trata de las segmentaciones de las imágenes de la base de datos seleccionada.

2 ESTADO DEL ARTE

En esta sección se van a tratar varios conceptos con objeto de que el lector conozca las diferentes posibilidades, instrumentos y métodos que existen en el análisis y tratamiento de textos históricos. Estos conceptos vienen recogidos en dos subapartados, en los que se hace referencia a las distintas arquitecturas de redes neurales disponibles en los artículos que trataremos posteriormente, así como al conjunto de métodos o herramientas existentes empleadas para la segmentación de documentos manuscritos.

2.1 Arquitecturas de red

En este subapartado se encuentran varios tipos de arquitecturas de redes neurales. Evidentemente, hay muchas clases de arquitecturas disponibles que no dejan de ser importantes y completas. No obstante, en este proyecto solo trataremos aquellas pertenecientes a los artículos que revisaremos posteriormente en esta sección.

2.1.1 Red neural convolucional (CNN)

Antes de poder hablar del concepto de red neural convolucional, también conocido por redes convolucionales, hay que hacer referencia a las redes neurales artificiales (ANN). Las ANN son sistemas de procesamiento computacional que se inspiran en la manera en que funciona el sistema nervioso de un ser humano. Este tipo de redes, se compone de una serie de nodos computacionales (neuronas) que trabajan interconectados, de forma que puedan aprender de manera colectiva, optimizando su salida final [8].

De esta manera, las CNNs, son análogas a las ANNs tradicionales [8], ya que están compuestas por neuronas que se van a auto-optimizar a través del aprendizaje. Las CNNs por su parte, se consideran un tipo especial de red neural empleado para el procesamiento computacional de datos. Una diferencia importante entre las ANNs tradicionales y CNNs, es que estas últimas, se emplean principalmente para lo referido al reconocimiento de patrones dentro de imágenes [8], de manera que la entrada a estas redes estará compuesta por imágenes.

Otra de las características de esta clase de redes, es que utilizan la operación de convolución en al menos una de sus capas. Así, se puede decir que las CNNs están compuestas por tres tipos de capas: las *convolucionales*, que juegan un papel sumamente importante en el mecanismo de operación de las CNNs; las *pooling,* que se encargan de reducir, entre otros, la dimensionalidad de la red, el número de parámetros y la complejidad del modelo [8]; y las capas *completamente conectadas*, que contienen neuronas que están conectadas de forma directa a los dos capas inmediatamente adyacentes. Podemos observar esta clase de arquitectura en la Figura 2-1.

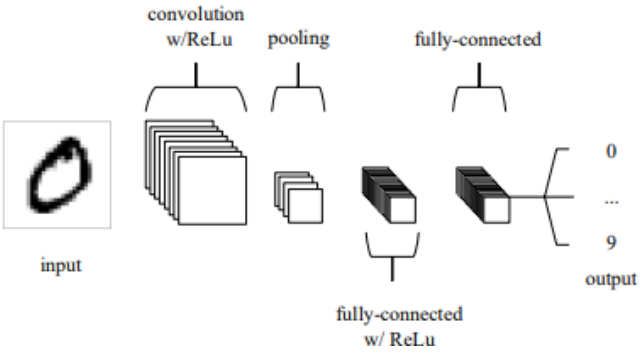


Figura 2-1. Red neural convolucional (CNN) [8]

2.1.2 U-Net

La U-Net, es un tipo de CNN propuesto por primera vez por Olaf Ronneberger, Phillip Fischer y Thomas Brox en 2015 con la idea en mente de una mejor segmentación para imágenes biomédicas [9]. Es necesario señalar que debe su nombre a su propia arquitectura, ya que cuando es representada, nos recuerda a la letra U, como podemos ver en la Figura 2-2. Además, como se puede observar en dicha figura, la U-Net básicamente consiste en un camino contractor, que sería la parte izquierda de la imagen, y en un camino expansivo, que sería la parte derecha de la figura. Ambos caminos quedan unidos por una especie de cuello de botella, situado en la parte baja de la imagen.

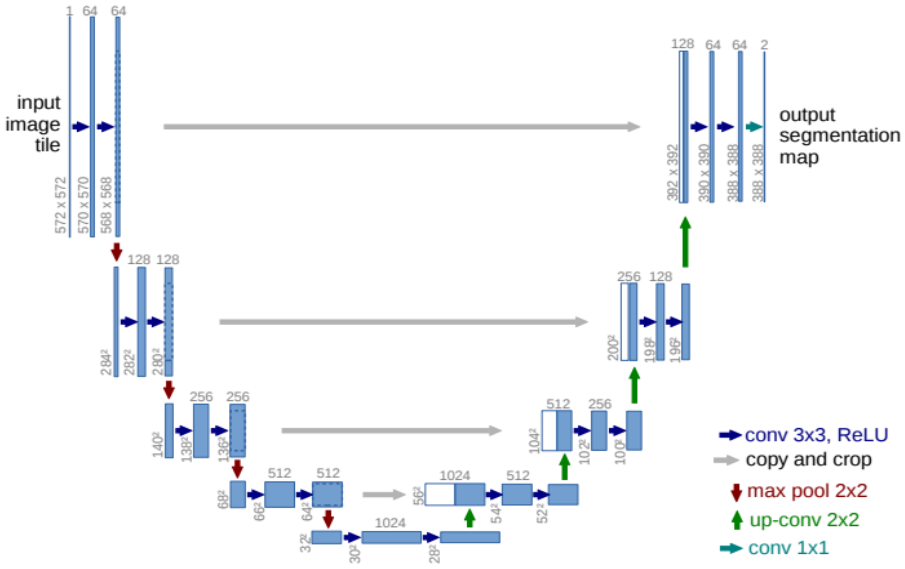


Figura 2-2. Arquitectura U-Net [9]

2.1.3 Red neural residual (ResNet)

Las ResNet han destacado siempre por su rendimiento y habilidad para construir redes profundas, sin tener que enfrentarse al problema de los gradientes que se desvanecen o explotan. Además, según referencias empíricas, este tipo de redes son mucho más sencillas de optimizar que otras que se puedan encontrar en la actualidad. En realidad, las redes residuales se parecen más a mecanismos de atención, ya que pueden modelar el estado interno de la red, opuesto a las entradas [10].

Por otra parte, resaltar que su arquitectura está compuesta normalmente de entidades apiladas, conocidas por el nombre de bloques residuales, capaces de contener un módulo y una identidad de bucle [11], y que están conectados entre ellos. Además, entre los ejemplos de red ResNet conocidos hasta la fecha, se pueden encontrar ResNets capaces de contener hasta 152 capas [12], incluyendo los tres tipos de capas vistas en el apartado de CNN. En la siguiente figura, Figura 2-3, mostramos un ejemplo de esta red.

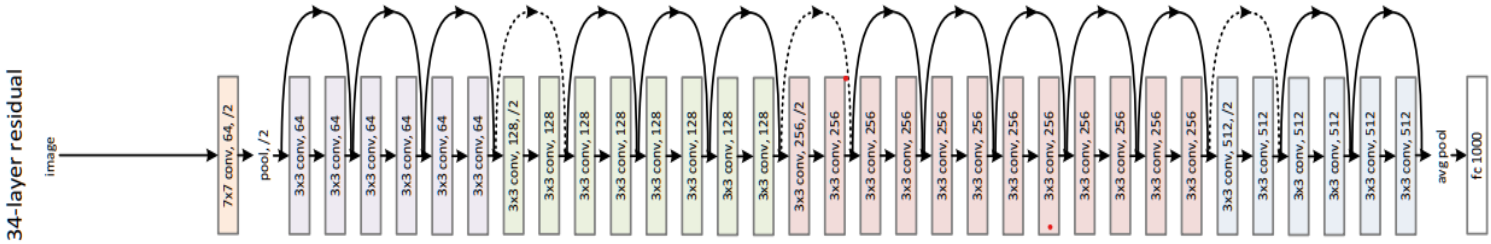


Figura 2-3. Red neural residual (ResNet) [10]

2.1.4 Dilated or atrous convolution

Esta clase de arquitectura debe su nombre a la expresión francesa *algorithm à trous*, que en su traducción al español queda como, algoritmo con agujeros. De esta manera, la idea que plantea es la de introducir una serie de agujeros, que en realidad serían ceros, en la entrada de la red mientras que se ejecuta la convolución atrous Figura 2-4. Así, el mapa de características obtenido en la salida es más grande [13].

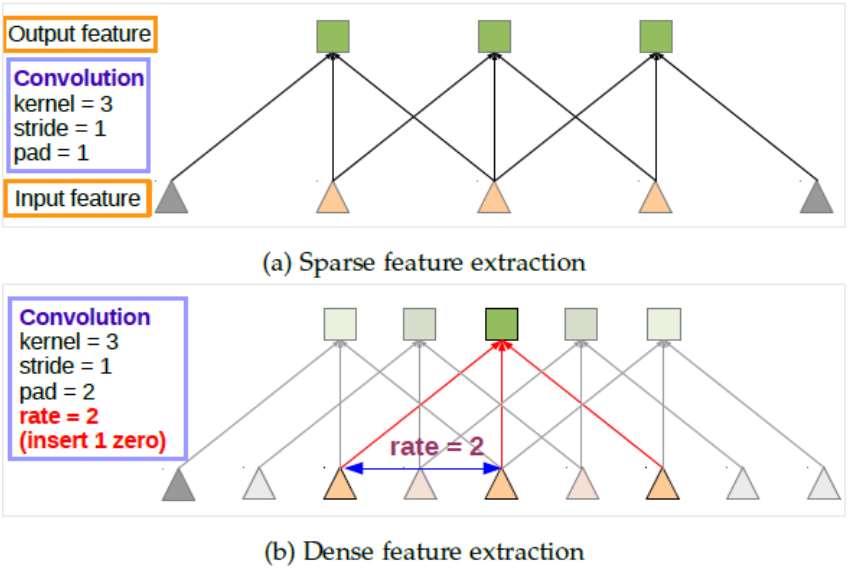


Figura 2-4. Dilated or atrous convolution [29]

2.2 Métodos disponibles

Esta parte del trabajo se centrará en la descripción de una serie de artículos en la que se enumerarán sus ventajas y desventajas. Además, decir que algunos de los algoritmos presentes en estos artículos, se emplearán como referencia para seguir desarrollando este proyecto.

2.2.1 Artículo 1: Text line segmentation using a fully convolutional network in handwritten document images

En primer lugar, decir que el artículo, *Vo, Q. N., Kim, S. H., Yang, H. J., & Lee, G. S. (2018). "Text line segmentation using a fully convolutional network in handwritten document images". IET Image Processing, 12(3), 438–446* [5], tiene como objetivo plantear una solución para la detección de líneas de texto en documentos manuscritos, que se base principalmente en una red totalmente convolucional, o como se conoce por sus siglas en inglés, FNC (Fully Convolutional Network).

Para poder entender la solución que aquí se expone, es necesario abordar el problema que se trata de resolver: la *segmentación de líneas de texto*. Para empezar, hay que decir que el objetivo principal de este tipo de segmentación es el de agrupar y alinear los llamados "componentes conectados" o CC en una línea de texto, de manera que se forme una secuencia de palabras. Este proceso, así descrito, puede parecer algo bastante sencillo si hablamos de documentos digitales escritos a máquina, pero no lo es cuando se trata de documentos manuscritos. En este último caso, hay que tener en cuenta muchísimos aspectos a la hora de agrupar los CC, como los diferentes estilos de escritura o que las líneas se toquen unas con otras, que claramente hacen que el rendimiento del algoritmo de segmentación construido se vea mermado.

Siguiendo el hilo de la segmentación de líneas de texto, cabe destacar que, como ya se comentaba en la introducción, los métodos existentes se pueden clasificar en tres categorías: *métodos globales*, caracterizados por centrarse primero en la estimación de la localización de las líneas de texto, para después formar las secuencias de componentes asignadas a líneas de texto concretas y separar los componentes pertenecientes a muchas líneas de texto; *métodos locales*, que a diferencia de los anteriores, tratan de encontrar las unidades

locales primero, que después se agruparán en líneas de texto separadas ; y *métodos híbridos,* encargados de incorporar información global y local de manera que se puedan construir las líneas de texto. Sin embargo, hay que señalar que la separación de caracteres, sobre todo aquellos que se tocan y pertenecen a diferentes líneas, es un problema bastante complejo de solucionar, y de hecho es uno de los mayores problemas a los que se enfrentan los autores de este artículo.

Por otra parte, como ya se introducía en esta sección, la mayor contribución que aporta el artículo es la aplicación de la FCN a la detección de líneas de texto en imágenes de documentos manuscritos. La red que ellos construyen es capaz de aprender a generar un mapa de energía, para poder extraer la localización de las líneas que forman el texto. Esto lo hacen a partir de un set de datos de "ground-truth", formado por mapas binarios donde se encuentra la estimación del lugar de las líneas en el texto.

Según los autores del artículo, la FCN que ellos desarrollan, presenta un mapa de energía con una mejor conexión de los componentes del texto que otros mapas de energía generados por diferentes métodos globales. Además, presenta la ventaja de que no se necesitan reglas heurísticas ni diseños manuales. De esta manera, el sistema aquí diseñado, presenta la mejor adaptabilidad para diferentes clases de imágenes de documentos.

En lo referido a la arquitectura empleada en este sistema, decir que se trata de la Convolutional Neural Network o CNN. Concretamente, varias capas de esta arquitectura han sido empleadas consiguiendo lo que se conoce como FCN, de forma que así se pueda tener en cuenta las coordenadas de lugar en una imagen, cosa que no se conseguía con la tradicional CNN.

Otro de los objetivos o puntos importantes de este proyecto (para lograr una buena segmentación) es el de construir un sistema que se base en una red FCN como módulo central. Dicha solución se presenta en la Figura 2-5. Como se puede observar en la misma, la red coge como entrada un trocito de una imagen en la que distinguen varias líneas de texto, para obtener a la salida dos mapas de calor, uno para las líneas de texto y otro para el fondo de la imagen.

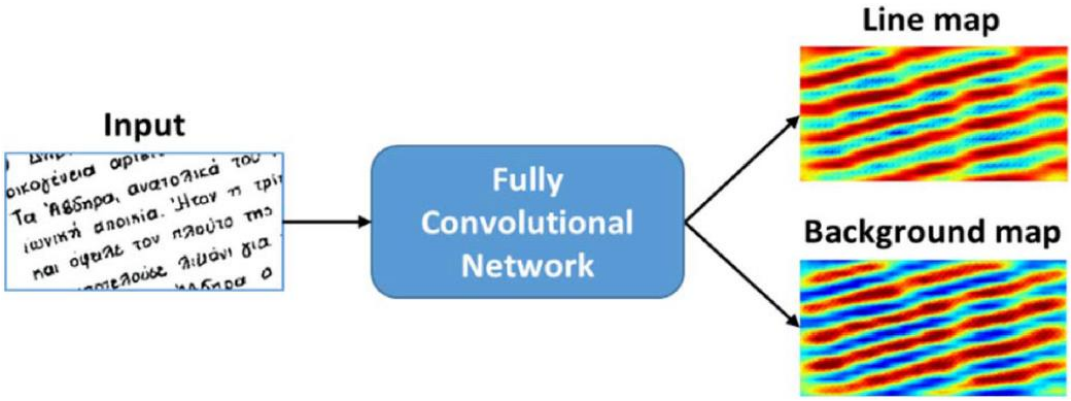


Figura 2-5. Representación del resumen de la propuesta de extracción de cadenas de texto [5]

En la fase de entrenamiento, la "ground-truth" es, como se comentaba anteriormente, un mapa binario cuya función es la de contar con una etiqueta para cada pixel que forme la imagen de entrada, esto es, una etiqueta para las líneas de texto y otra para el fondo de la imagen.

Además, después de analizar con detenimiento la generación de ambos mapas de calor, los autores consideran que es redundante estimar ambos, ya que realmente son salidas equivalentes. Por tanto, dejan muy claro que con el mapa de línea sería suficiente para llevar a cabo la detección de las líneas de un texto.

Volviendo al módulo central FCN, decir que los autores del artículo probaron tres estructuras diferentes: FCN-pool2, FCN-pool3 y FCN-pool4. Estas se pueden observar en la Figura 2-6, y cada una contiene un flujo de convolución, capas de pooling (señaladas en la Figura 2-6 con un color anaranjado) o traducido al español, capas de agrupamiento, y finalmente una capa de deconvolución. Aunque se usan diferentes cantidades de capas dependiendo el tipo de FCN que sea.

Para poder determinar qué estructura FCN, de las tres expuestas, es más adecuada para la tarea de extracción de líneas de un texto manuscrito, los autores las entrenan con el mismo conjunto de entrenamiento, de forma que se obtiene, a la salida respectiva de cada modelo, un mapa de línea. El resultado para cada estructura puede apreciarse en la Figura 2-7, donde aparecen los mapas de línea correspondientes.

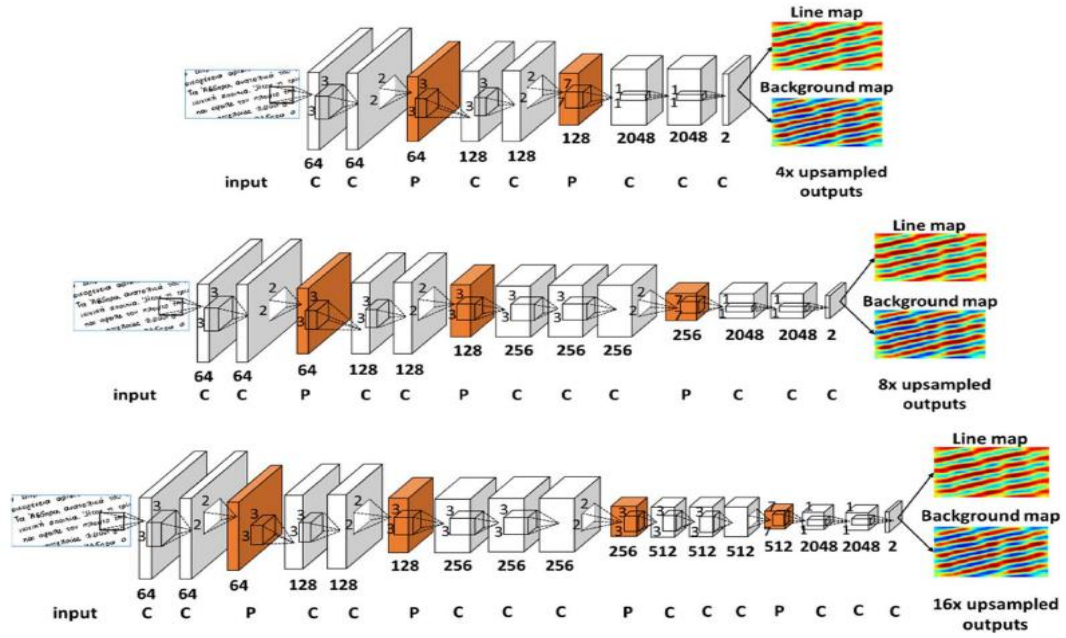


Figura 2-6. Estructuras de red probadas para el etiquetado de líneas de texto. De arriba a abajo: FCN-pool2, FCN-pool3 y FCN-pool4 [5].

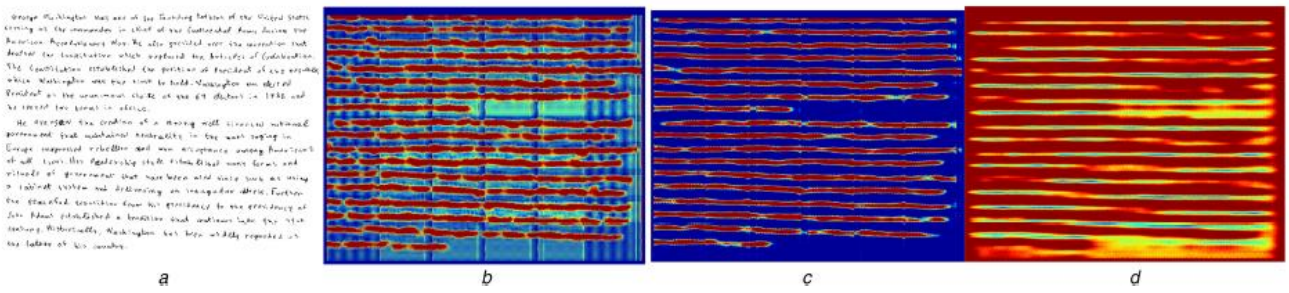


Figura 2-7. Mapas de líneas de una imagen de documento generada por las arquitecturas de aprendizaje: (a) Imagen de entrada, (b) Salida de FCN-pool2, (c) Salida de FCN-pool3, (d) Salida de FCN-pool4 [5].

Al examinar los mapas de línea obtenidos para cada tipo de red, se puede apreciar que la salida de FCN-pool2 está compuesta por la salida de la segunda capa de pooling, que tiene una resolución mayor que las siguientes capas de agrupación. Se puede observar además, que el mapa de línea de FNC-pool2 presenta unas líneas de baja energía, representadadas en la Figura 2-7 (b) por unas líneas verticales azules, que pueden indicar que cuando exite un espacio grande entre dos palabras consecutivas de un línea y las regiones rojas de la imagen, ambas no son capaces de conectarse completamente [13], apareciendo estas líneas verticales. Además de estas líneas azules, también se pueden apreciar en la Figura 2-7 (b) unas regiones verdes, que denotan una energía media, y que parecen venir de líneas de texto que la red piensa que deberían ser como las demás, más largas [13]. Aunque también existen otras regiones verdes, que lo dejan entrever es que hay una distancia relativamente corta entre dos líneas. Por otra parte, en lo que se refiere al mapa de línea de la FNC-pool4, se ha generado a partir de la cuarta capa de pooling, y como se puede apreciar en Figura 2-7 (d), el resultado no es muy preciso, más bien tiene una resolución baja. Por tanto, la única estructura que destaca por encima de las demás es la FNC-pool3, que consigue la mejor calidad de mapa de línea, ofreciendo los resultados más precisos. Así pues, esta opción es la mejor, y es la estructura que se elige finalmente.

Un vez que ya se ha conseguido generar el mapa de línea que se quería, se discute el problema de la extracción

de las líneas de un texto manuscrito. Este se basa en la extracción de una cadena de texto (que básicamente es una cadena de píxeles) del mapa de línea, que sería en comparación con esa cadena de píxeles, un grupo de píxeles que corresponde a un área de texto. Es decir, una cadena de texto se corresponde con una línea de texto, mientras que el mapa de línea sería la región del texto donde se incluye esa línea.

Para realizar esta extracción es necesario realizar tres pasos previos: *binarización del mapa de línea*, que consiste en etiquetar las regiones de texto, que serían aquellas dotadas de mayor energía; *extracción del esqueleto de la línea*, que trabaja con la posibilidad de que existan varias regiones de texto en la misma línea; y por último la *fusión del esqueleto de la línea,* paso encargado de la construcción de la línea a partir de todos los esqueletos posibles que pueden pertenecer a ésta. Con estos tres pasos, se consigue ensamblar la cadena de texto que antes se enunciaba.

Pero todavía queda algo sin resolver, y es el caso de que algún carácter no esté en contacto con tan solo una línea. De esta manera los autores de [5], dividen dicha cuestión en dos problemas independientes: la separación de los caracteres que se superponen con otras líneas, y la asignación de la línea de texto. La resolución de ambos queda expuesta de manera precisa y detallada en [5].

En cuanto al uso en la práctica del modelo, decir que se emplea la base de datos ICDAR2013 Handwritten Segmentation Contest [14], que contiene un total de 200 imágenes de entrenamiento, en tres idiomas diferentes. En la siguiente Tabla 2-1 se pueden apreciar los resultados del modelo al compararse con otros de funcionamiento similar. Más detalles acerca de su obtención, así como de las medidas empleadas para evaluarlo y ser comparado, se pueden obtener en [5].

Tabla 2-1. Resultados de la evaluación del modelo [5]

Table 1 Evaluation results

Algorithm	*M*	o2o	DR, %	RA, %	FM, %
CUBS	2677	2595	97.96	96.64	97.45
GOLESTAN-a	2646	2602	98.23	98.34	98.28
INMC	2650	2614	98.68	98.64	98.66
LRDE	2632	2568	96.94	97.57	97.25
MSHK	2696	2428	91.66	90.06	90.85
NUS	2645	2605	98.34	98.49	98.41
QATAR-a	2626	2404	90.75	91.55	91.15
QATAR-b	2609	2430	91.73	93.14	92.43
NCSR(SoA)	2646	2477	92.37	92.48	92.43
ILSP(SOa)	2685	2546	96.11	94.82	95.46
TEI(SoA)	2675	2590	97.77	96.82	97.30
LAG-horizontal (proposed)	2643	2583	97.51	97.73	97.62
LAG-vertical (proposed)	2643	2608	98.45	98.68	98.56

Las medidas empleadas en la Tabla 2-1, expresadas en tanto por ciento, vienen representadas en las siguientes ecuaciones [5]:

$$DR = \frac{o2o}{N}, \qquad RA = \frac{o2o}{M} \tag{2-1}$$

$$FM = \frac{2 \cdot DR \cdot RA}{DR + RA} \tag{2-2}$$

Donde DR se define como *tasa de detección*; RA como *precisión de reconocimiento;* y FM como *métrica de rendimiento*. Por otra parte, tenemos varios parámetros que aparecen en la expresión (2-1), y que deben ser definidos: *N* es el número de elementos del ground-truth; *M* es el número de elementos de los resultados obtenidos y por último *o2o* (one-to-one), que se trata del número de coincidencias entre el ground-truth y los resultados.

Como es posible ver en la Tabla 2-1, los resultados que presenta el modelo propuesto por los autores del artículo (LAG-horizontal y LAG-vertical), son bastante buenos al ser comparados con los demás. Sin embargo, el modelo está lejos de ser perfecto.

Pese a todo, el sistema aquí recogido, presenta una característica muy relevante: consigue adaptarse a varias clases de imágenes de entrada, siempre que su entrenamiento se lleve a cabo con un set de datos apropiado

para la tarea. No obstante, como se señalaba en el párrafo anterior, el modelo no es excelente, ya que no es capaz de resolver con precisión la segmentación de las líneas de texto cuando hay caracteres de diferentes líneas que se tocan.

2.2.2 Artículo 2: DhSegment: A generic deep-learning approach for document segmentation

La solución planteada en la siguiente subsección y referenciada en *Oliveira, S. A., Seguin, B., & Kaplan, F. (2018). "DhSegment: A generic deep-learning approach for document segmentation", 7–12.* [15], tiene como objetivo principal el diseño de las tareas de procesamiento de documentos históricos desde un enfoque más genérico, de manera que se pueda manejar mejor la variabilidad de las series históricas. En concreto, en este artículo los autores intentan resolver varios problemas, como la extracción de páginas, la extracción de las líneas base o baseline, el análisis de la estructura del texto así como las diversas tipologías de las ilustraciones, y la extracción de gráficos o fotografías.

En dicho artículo se expone un programa de código abierto con estructura CNN, que se basa en la predicción de cada pixel del documento para después aplicar un post-procesado en diferentes bloques. Los autores defienden que es suficiente con emplear una sola CNN para resolver los problemas anteriormente planteados. De esta forma, conseguimos obtener un método que presenta una arquitectura idónea para tareas de segmentación semántica aplicadas a documentos históricos.

Por otra parte, en lo que se refiere al sistema completo, que mostramos en la Figura 2-8, se compone de dos bloques principales. El primer bloque, se trata de una FCN (Fully Convolutional Network) a la que se le proporciona como entrada una imagen de un documento histórico, cuyo tamaño no está limitado mientras que la memoria lo permita [15], y cuya salida se trata de un mapa de probabilidades (generado por la función ReLU) de etiquetas para cada pixel de la imagen. En el segundo bloque, ese mapa generado a la salida del primero será convertido en una máscara, de forma que dicho bloque tiene como objetivo transformar el mapa de predicción en la salida deseada del programa.

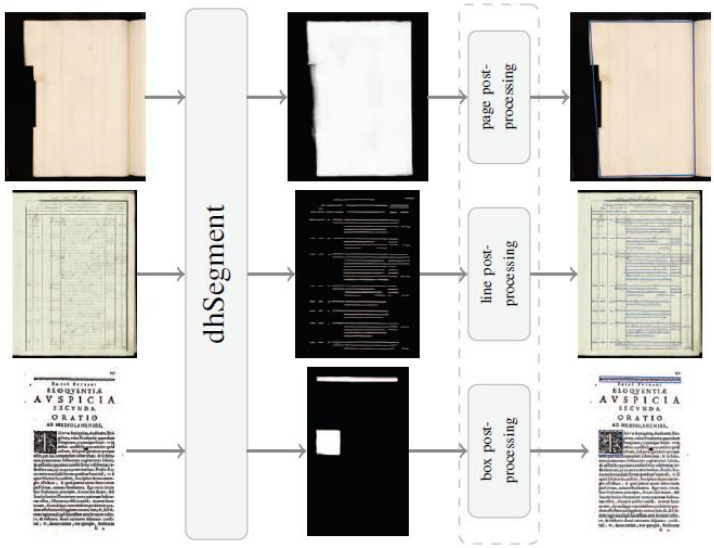


Figura 2-8. Esquema completo del sistema propuesto [15]

Una vez descrito el funcionamiento del sistema completo, es importante continuar hablando de la arquitectura de la red que presenta el método propuesto. Dicha arquitectura es la de una U-Net, de forma que posee dos ramas o caminos: uno de contracción, que consiste básicamente en un conjunto de capas convolucionales que se repiten, de forma que se consigue reducir la información espacial y aumentar la información de las características [13], aunque en el caso de dhSegment este camino se basa en la red residual profunda ResNet-50; y otro de expansión, que combina tanto la información espacial como la de característica a través de una serie de convoluciones y concatenaciones con características de alta resolución del camino de contracción [15]. En la Figura 2-9, podemos observar de forma gráfica la arquitectura que se acaba de describir.

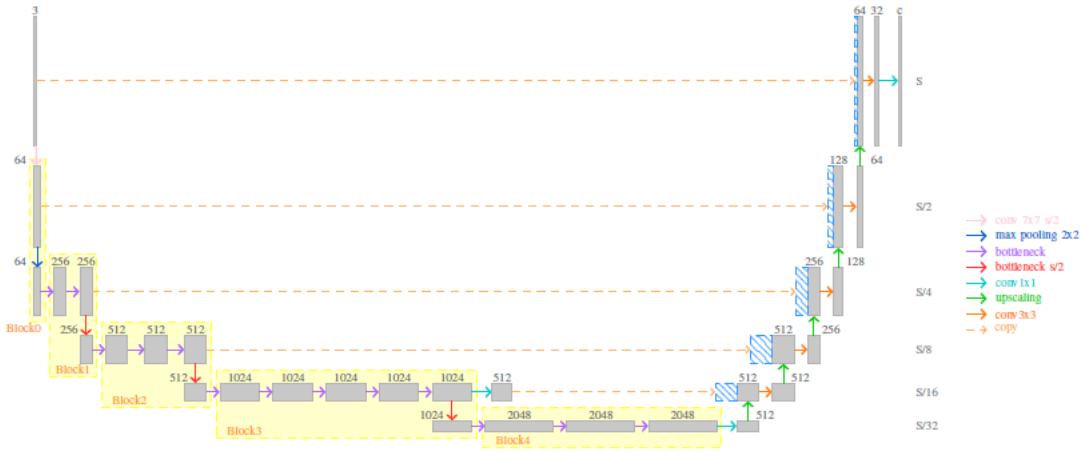


Figura 2-9.Arquitectura de red de dhSegment [15]

Además, la arquitectura de la red tiene un total de 32.8M de parámetros, que gracias a los pesos previamente entrenados en [16] y que pertenecen a la rama de contracción, se reducen considerablemente a 9.36M parámetros que deben de ser entrenados.

En cuanto al post-procesado que se aplica a la salida que resulta de la red, podemos destacar diversas operaciones que se llevan a cabo para conseguirlo:

- **Thresholding:** Esta operación es usada para obtener el mapa binario que se genera a partir de las predicciones que hace la red [15].
- **Operaciones morfológicas:** Se trata de un conjunto de operaciones no lineales que tienen su origen en la teoría de la morfología matemática y son usadas para imágenes y estructuras geométricas. En el caso de DhSegment, las únicas usadas son la erosión y la dilatación.
- **Análisis de componentes conectados (CC):** En el caso de dhSegment, el análisis de CC se utiliza para filtrar los componentes conectados pequeños que pueden haber quedado tras las operaciones de thersholding o las morfológicas [15].
- **Vectorización de formas:** Este paso es completamente necesario para que se pueda transformar la región detectada en un conjunto de coordenadas. Para ello, las formas o manchas del mapa binario se extraen como polígonos.

Siguiendo con el entrenamiento del sistema, destacar que los autores emplean la regularización L2 de forma que se pueda evitar el sobremuestreo. También emplean la inicialización Xavier y el optimizador Adam, previniendo así la falta de diversidad durante el entrenamiento. No hay que olvidar mencionar que se han empleado técnicas de data augmentation (rotación, escalado y mirroring), ofreciendo un menor volumen de datos en el entrenamiento. Todo este proceso de entrenamiento se llevó a cabo en una GPU Nvidia Titan X Pascal. Además, con la ayuda de los pesos previamente entrenados, el tiempo de entrenamiento se ha visto significativamente reducido.

Como se comentaba antes en la introducción del artículo, el sistema construido quiere probar que definitivamente resuelve varios problemas (la extracción de páginas, la extracción de las líneas base o baseline, el análisis de la estructura del texto así como las diversas tipologías de las ilustraciones, y la extracción de gráficos o fotografías), demostrando así su capacidad de generalizar. En este texto, al igual que [13], nos centraremos exclusivamente en las tres primeras tareas, ya que las restantes no aportan información relevante para el desarrollo de este trabajo:

- **Extracción de páginas**: Esta tarea consiste en la localización de los bordes originales del documento cuando la imagen está siendo digitalizada [13] y la existencia del fondo residual está asegurada. Podemos ver en la Figura 2-10, como el sistema realiza esta tarea para el dataset descrito en el Apartado 3.2.

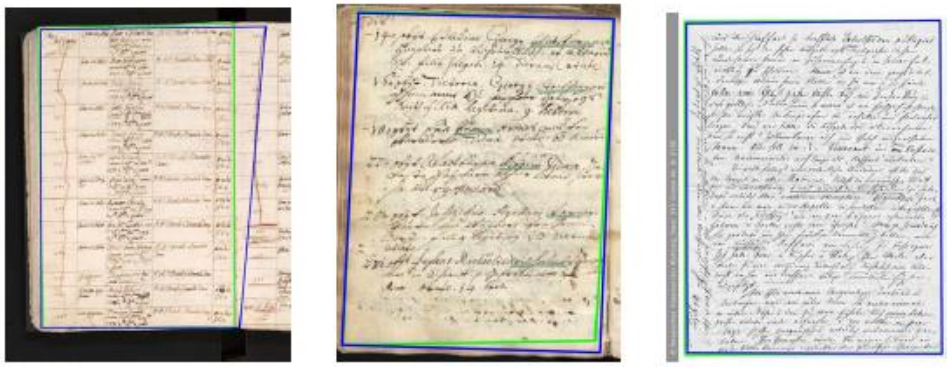


Figura 2-10. Ejemplo de detección de páginas en el conjunto de test cBAD [15]

En la imagen superior, los rectángulos verdes indican la parte de ground_truth, mientras que los que son azules corresponden a las detecciones conseguidas con dhSegment. Como se puede observar, la primera imagen muestra una extracción no muy precisa, ya que la herramienta detecta también la página de al lado. Por otra parte, en la imagen contigua podemos notar que la página es detectada sin problemas, aunque esta no termina de ser completamente precisa ya que ambos rectángulos no terminan de coincidir. Finalmente, la última imagen muestra realmente lo que sería una detección correcta.

En la siguiente Tabla 2-2, se muestran y comparan los resultados de dhSegment con otras herramientas, en el terreno de la extracción de páginas:

Tabla 2-2. Resultados de la tarea de segmentación de páginas [15]

Method	cBAD-Train	cBAD-Val	cBAD-Test
Human Agreement	-	0.978	0.983
Full Image	0.823	0.831	0.839
Mean Quad	0.833	0.891	0.894
GrabCut [21]	0.900	0.906	0.916
PageNet [20]	0.971	0.968	0.974
dhSegment (quads)	$\mathbf{0.98}\pm 6e^{-4}$	$\mathbf{0.98}\pm 8e^{-4}$	$\mathbf{0.98}\pm 8e^{-4}$

- **Detección de la baseline:** Como referíamos en la sección de introducción, una línea de base o baseline, se puede definir como aquella línea "virtual" en la que descansan la mayoría de los caracteres. Las líneas de texto se anotan con una sola línea de base, de manera que los objetos no textuales (dibujos, gráficos etc…) no son anotados. En la siguiente Figura 2-11, observamos como dhSegment realiza este tipo de detección.

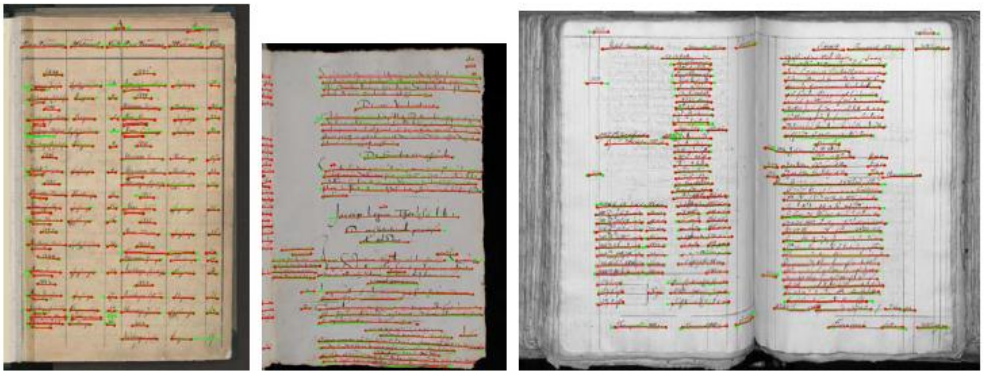


Figura 2-11. Ejemplos de extracción de líneas base en el conjunto de datos cBAD [15]

En la imagen superior, las líneas verdes representan el ground-truth y las rojas representan las baselines predichas. Como podemos distinguir, en la primera página y en la segunda las líneas de texto más cercanas terminan fusionándose, mientras que en la tercera página se detecta también el texto de la página vecina. De esta forma, vemos claramente las limitaciones de la herramienta, aunque en general la detección que hace puede ser considerada buena.

- **Análisis de la estructura del documento:** Esta tarea consiste en segmentar un documento en diferentes regiones, de manera que se cada pixel que conforme la imagen quede etiquetado. Las etiquetas que se pueden encontrar son: regiones de texto, adornos, comentarios y fondo. En la Figura 2-12, observamos un ejemplo de este análisis con el dataset presentado en el Apartado 3.3.

Figura 2-12. Ejemplo del análisis de la estructura de los documentos del dataset DIVA-HisDB [15]

En la imagen superior observamos que la página de la izquierda representa al manuscrito original, a continuación encontramos los píxeles etiquetados por la herramienta dhSegment, y finalmente obtenemos la comparación con el ground-truth. La base de datos empleada para ello es DIVA-HisDB [17]. En la Tabla 2-3, que encontramos a continuación, se muestra una comparación entre este método y otros similares en la realización del análisis de estructura de textos.

Tabla 2-3. Resultados del concurso ICDAR2017 sobre el análisis de la estructura de manuscritos medievales [15]

Method	CB55	CSG18	CSG863	Overall
System-1 (KFUPM)	.7150	.6469	.5988	.6535
System-6 (IAIS)	.7178	.7496	.7546	.7407
System-4.2 (MindGarage-2)	.9366	.8837	.8670	.8958
System-2 (BYU)	.9639	.8772	.8642	.9018
System-3 (Demokritos)	.9675	.9069	.8936	.9227
dhSegment	.974±.001	.928±.002	.905±.007	.936±.004
dhSegment + Page	.978±.001	.929±.002	.909±.006	.938±.004
System-4.1 (MindGarage-1)	.9864	.9357	.8963	.9395
System-5 (NLPR)	.9835	.9365	.9271	.9490

Concluyendo con la revisión de esta herramienta, decir que, como se ha podido comprobar, las salidas que obtiene la misma son bastante competitivas, aunque a veces no sean las mejores. Además, destaca por su entrenamiento eficiente y rápido, debido en gran medida al empleo de pesos pre-entrenados. No debemos olvidar que su gran ventaja es que es de código abierto, de forma que lo podemos usar y modificar cómo queramos.

2.2.3 Artículo 3: Neural text line segmentation of multilingual print and handwriting with recognition-based evaluation

Para comenzar la redacción de la siguiente subsección, decir que el artículo *Schone, P., Hargraves, C., Morrey, J., Day, R., & Jacox, M. (2018). "Neural text line segmentation of multilingual print and handwriting with recognition-based evaluation". In Proceedings of International Conference on Frontiers in Handwriting Recognition, ICFHR* [18], tiene como objetivo plantear un método completamente nuevo, encargado de

detectar las líneas de texto de documentos históricos manuscritos, y también de imágenes de documentos impresos. Básicamente, la técnica empleada en este artículo se centra en la mejora de las redes neuronales profundas, de manera que sea más fácil llevar a cabo la predicción de múltiples clases de píxeles. Además, el método propuesto presenta una nueva técnica que se encarga de unir una línea completa cuando existan espacios en la misma, así como de predecir qué regiones de texto contienen líneas que podrían fusionarse. Según se justifica en [18], el sistema neural propuesto es el primero en ser capaz de predecir el perímetro completo de las líneas de texto, que conforman los documentos analizados, en toda su extensión. De esta manera a la red, en la etapa de entrenamiento, se le administran primero pequeñas regiones de las imágenes a analizar, para después, a medida que el entrenamiento continúa, aumentar el tamaño hasta proporcionar al sistema las imágenes completas. Con todo ello, se revela que lo que se quiere conseguir finalmente con el desarrollo de un sistema de estas características, es que la salida sea apta para ser proporcionada a un sistema de transcripción.

Realmente, la novedad que presenta este algoritmo es que tiene como meta la búsqueda de las fronteras reales del texto, usando para ello decisiones neurales. Así pues, el sistema emplea combinaciones de redes neurales con las que aprende estas fronteras, a partir de un conjunto de datos donde los perímetros de las porciones de texto vienen proporcionados. De esta manera, la red neural necesita como entrada del sistema unas imágenes de entrenamiento con perímetro delimitado, como se describía anteriormente. El programa que empleamos ha sido entrenado y probado con unas 450 imágenes que comprenden tanto textos como gráficos, como se puede observar en la Figura 2-13.

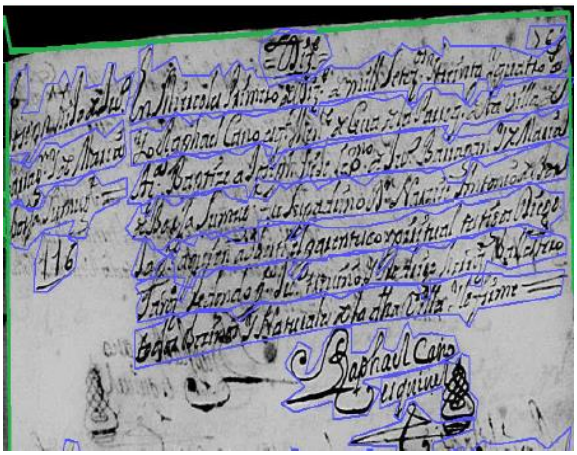


Figura 2-13. Anotación de bloques de texto (en azul) y gráficos/marcos/líneas (en verde) como datos de entrenamiento extraídos de una parte de un registro de la Iglesia española [18]

En cuanto al modelo descrito en [18], decir que está implementado en Tensorflow, que llevaría todo el peso de la parte neural del sistema, para después pasarlo a Java, de manera que se lleve a cabo la parte de postprocesado. Además, el núcleo del sistema de segmentación es una red FCN, de forma que se asegure que el modelo pueda extraer la línea de texto completa y así poder pasarla, posteriormente, a un sistema de transcripción.

Por otra parte, en lo referido a la arquitectura que presenta la red, decir que se trata de una red con forma de reloj de arena, a la que por el asunto de que el algoritmo debe segmentar líneas completas de texto, hay que añadirle varios elementos, como el *etiquetado simultáneo de clases;* el *pegamento textual,* con el que se mantienen juntas las líneas de texto; la *prevención de la fusión de líneas de texto*; o el *mayor contexto sin necesidad de especificaciones de tamaño.* Una descripción más amplia de estos conceptos se puede encontrar tanto en [18], como en [13].

La arquitectura que queda finalmente, se muestra en la Figura 2-14, presentada a continuación:

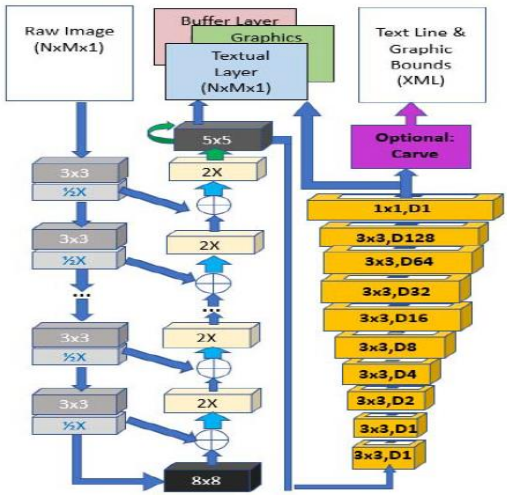


Figura 2-14. Representación gráfica de la arquitectura del sistema propuesto [18]

Además, tenemos que tener en cuenta la figura del "trainer". Este elemento es el encargado de pedir al generador de batches que le proporcione un cierto porcentaje de fragmentos de entrenamiento para tener etiquetas específicas (texto, gráfico, buffer o genérico), de manera que cada uno de sus lotes de imágenes cuenten con ellas. De la misma forma, el generador pregunta por el tamaño deseado para los fragmentos de entrenamiento. Esto se hace, ya que el sistema puede confundirse, a medida que aumentan las iteraciones del algoritmo cuando recibe trozos de imagen muy reducidos, contratiempo que se resuelve con el aumento del tamaño de la imagen conforme se van ejecutando las iteraciones del algoritmo. No obstante, existen situaciones en las que el modelo llega a un empate. Para solventarlas, el sistema se encarga de revisar si esos píxeles situados en la región de empate se extienden o no hasta el final de la región. Posteriormente, un programa dinámico se usa para dividir la región de empate en dos regiones distintas, una superior y otra inferior. Por último, la salida del sistema quedaría tal y como se muestra en la Figura 2-15.

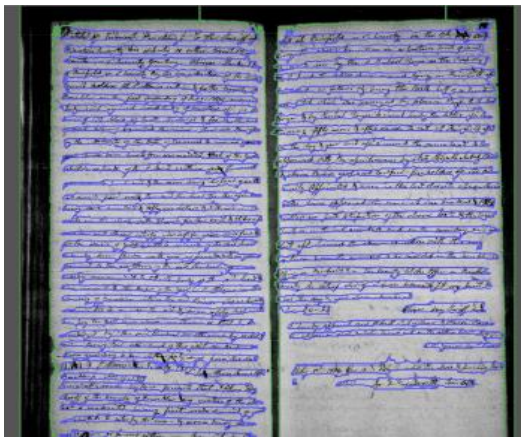


Figura 2-15. Ejemplo del tipo de salida del sistema [18]

Finalmente, para la evaluación del sistema se han empleado varias bases de datos: *IAM Data*, con numerosos documentos de diferentes clases, como libros de cocina, noticias, artículos de carácter científico, que hacen que este conjunto de datos sea bastante complejo; *US Wills and Deeds*, formada por casi 50K de palabras en inglés manuscrito; y *Spanish Church Records*. La siguiente Tabla 2-4, presenta los resultados de la evaluación del modelo con las distintas bases de datos, así como la comparación de la misma con las de otros algoritmos parecidos.

Tabla 2-4. Resultados de la evaluación del modelo con distintas bases de datos, y la comparación de la misma con las de otros algoritmos parecidos [18]

Word Accuracies (100%-WER) Per Dataset by Configuration (and line segmentation cost in minutes/number of output text lines)				
System Configuration	*F(IAM) HWR*	*Historical Newsprint*	*US Wills and Deeds HWR*	*Spanish Church HWR*
CVL [3] (2013)	**60.5%** 9/5697	**18.1%** 25/4375	**68.0%** 20/8734	**36.8%** 43/14.2K
A&S [5] (2014)	**21.6%** 660/10.8K	**57.7%** 236/6634	**63.8%** 628/5210	**57.5%** 753/6727
NCSR [4] (2015)	**71.5%** 5/3425	**77.9%** 3/7206	**72.6%** 40/4830	**58.3%** 13/5300
Neural Fix [8] (2017)	**76.2%** 70/2769	**92.7%** 62/7988	**87.1%** 72/5169	**67.5%** 60/7121
Ours: Neural + Tie Breaking	**86.5%** 102/3640	**93.8%** 47/8001	**88.7%** 71/6110	**71.6%** 76/8142

Para concluir la revisión de este artículo, exponer que, efectivamente, destaca el sistema de entrenamiento presentado por las nuevas técnicas que incluye, tales como: la del pegamento textual, los buffers de separación neural o el crecimiento automático de las imágenes durante los batches. Así pues, se constituye un sistema de segmentación que, según los autores del paper, es capaz de arrojar mejores resultados que otros en la misma línea de trabajo

2.2.4 Artículo 4: Labeling, Cutting, Grouping: an Efficient Text Line Segmentation Method for Medieval Manuscripts

En este apartado, presentamos un cuarto artículo: *Alberti, M., Vogtlin, L., Pondenkandath, V., Seuret, M., Ingold, R., & Liwicki, M. (2019). "Labeling, cutting, grouping: An efficient text line segmentation method for medieval manuscripts". Proceedings of the International Conference on Document Analysis and Recognition, ICDAR, 1200–1206*, que podemos encontrar referenciado en [2]. Este paper expone una forma novedosa de extracción de líneas de texto, enfocada principalmente a documentos manuscritos medievales. Estos se caracterizan por presentar un reto para los sistemas de extracción de líneas, ya que son bastante complejos en cuanto a su estructura. Así pues, el método propuesto en [2] se vale de la segmentación semántica a nivel de píxel como paso de preprocesado, previo a la extracción de las líneas que componen el texto a tratar.

La base de datos empleada es la DIVA-HisDB, que como se describirá posteriormente en el capítulo 3 de este trabajo, se compone por 3 tipos de manuscritos medievales seleccionados por su estructura altamente compleja. En la Figura 2-16, podemos apreciar algunas de las características que hacen de este dataset uno complejo.

(a) CB55, p.25r (b) CB55, p.32v (c) CSG18, p.107 (d) CSG18, p.48 (e) CSG863, p.4 (f) CSG863, p.131

Figura 2-16. Muestras de páginas de los tres manuscritos medievales de DIVA-HisDB, en las que se pueden observar algunas de las múltiples características que hacen de este dataset un reto [2]

Por otra parte, una vez hecha una introducción al método planteado y habiendo revisado la base de datos con la que se va a trabajar, pasamos a describirlo detalladamente. En primer lugar decir que, para explicar correctamente el algoritmo de segmentación de líneas de texto, éste debe separarse en dos pasos principales:

- ***Paso 1:*** Este paso se encarga de realizar la segmentación semántica que ya se comentaba en la introducción. Dicha segmentación consiste básicamente en que para cada píxel, se asigne más de una etiqueta. Así pues, para poder procesar la entrada al sistema expuesto, que está en el dominio RGB como se puede observar en la Figura 2-16, y pasarla al dominio de "*pixel-labelled*", debemos llevar a cabo este paso de segmentación semántica a nivel de pixel, haciendo uso de la red vanilla ResNet-18, que presenta un comportamiento eficiente [19]. Todo esto se lleva a cabo en el entorno de Deep-learning, DeepDIVA, que podemos consultar en [20]. Como la calidad de la segmentación influye especialmente en la extracción de las líneas de texto, antes de pasar al *paso 2,* se pule un poco la salida obtenida aplicando técnicas como la de eliminación de pequeños componentes. Por tanto, una vez lista la salida del sistema, conseguimos una imagen en el dominio de las etiquetas o "*labels*", de modo que es posible seleccionar los pixeles principales del texto, desechando las decoraciones o pequeñas anotaciones. Así pues, tendríamos una imagen final del texto principal, filtrado y limpio, como se puede ver en la Figura 2-17.

Figura 2-17. Imagen en el dominio *pixel-labelled* [2]

Obviamente, antes de pasar al segundo paso, que ya sería la extracción, la imagen debe pasar por otros 3 procesos más: la construcción del mapa de energía del texto, el proceso de fusión de las "costuras" o seams del texto y finalmente la agrupación de los centroides de cada *cluster* (representa a una línea). Para ver estos pasos intermedios con mayor detalle, consultar [2].

- ***Paso 2:*** Una vez determinados los centroides de cada línea, se extraen los polígonos que las encierran. Por tanto, dando un conjunto de centroides, del paso intermedio que comentábamos, por encima, en el *paso 1*, dibujamos sus correspondientes CC (Connected Components) y el MST (Minimum Spanning Tree), que se encarga de conectar dichos centroides en un canvas vacío. A continuación, se hace la convolución de este canvas con un kernel de promediación 5x5, de modo que el resultado final sea una versión borrosa de la línea de texto. Finalmente, se extrae el único CC existente en el canvas, así como sus puntos de contorno, que forman indudablemente el polígono que encierra a la línea. En la Figura 2-18, podemos observar el resultado de este paso:

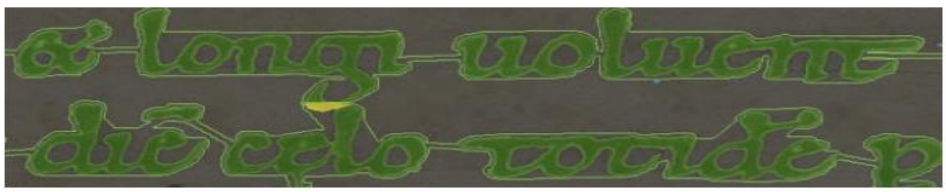


Figura 2-18. Polígonos de salida superpuestos con la imagen RGB [2]

Habiendo ya detallado el funcionamiento del método propuesto, se procede con la discusión de los resultados obtenidos para el mismo. En la Tabla 2-5, se pueden apreciar los valores obtenidos para los parámetros *line iu* y *pixel iu,* (donde *iu* indica IoU, o lo que es lo mismo, la métrica RA que habíamos visto en el apartado 2.2.1), usando el dataset DIVA-HisDB:

Tabla 2-5. Resultados de la extracción de líneas de texto para el dataset DIVA-HisDB [2]

METHOD	LINE IU %	PIXEL IU %
WAVELENGTH [20]	68.58	79.13
BRIGHAM YOUNG UNIVERSITY [3]	81.50	83.07
CITLAB ARGUS LINEDETECT [15]	96.99	93.01
WAVELENGTH* (TIGHT POLYGONS) [3]	97.86	97.05
PROPOSED METHOD*	**99.42**	96.11

Como se observa en la Tabla 2-5, el método presentado en el artículo [2], consigue resultados casi perfectos y supera a algoritmos presentados en su estado del arte (97.86%). De esta manera, podemos afirmar que reduce el error en un 80.7%. A pesar de estos resultados, el algoritmo tiene algunas limitaciones:

- El texto ha de tener la misma orientación en todo el documento
- La presencia de una estructura en el formato del texto más complicado que la disposición del mismo en una o dos columnas, haría necesario un paso previo de preprocesamiento de la estructura del diseño, de forma que se pudieran aislar los diferentes bloques de texto por separado.

En conclusión decir que, en este artículo, los autores proponen la extracción de las líneas de texto específicamente para imágenes de documentos medievales manuscritos. Para ello, se valen de la segmentación semántica a nivel de píxel como paso anterior a la propia extracción. Además, a la vista de los resultados arrojados por el algoritmo, queda patente que se trata de un método novedoso y robusto, al enfrentarse a estructuras de documentos bastante complejas. De esta manera, y a pesar de que presente ciertas limitaciones, los autores consiguen con su algoritmo acercarse un poco más a la mejora completa del tratamiento automatizado de documentos, en su mayoría, históricos.

3 BASES DE DATOS DISPONIBLES

En esta sección se recogen una serie de bases de datos o datasets de documentos históricos, especificando las características más importantes de los mismos, así como el acceso para descargarlos. Nótese que se han encontrado varias dificultades para obtener diferentes bases de datos. Entre estos inconvenientes destacamos: la privatización de muchas de ellas y la poca preparación de las mismas para funcionar con algoritmos de segmentación de líneas (como por ejemplo la ICDAR2019, que no contiene en su fichero PAGE.xml (dónde presenta el ground-truth) las coordenadas poligonales de las líneas del texto, o la ICDAR2017, que está preparada para trabajar con párrafos de texto por lo que tampoco plantea coordenadas específicas de las líneas que conforman el documento). Por otra parte, señalar que de los tres conjuntos de datos presentados en este capítulo, dos de ellos se han empleado en el proyecto desarrollado. A continuación, se hará una descripción de cada uno de ellos y se resaltará el porqué de su elección.

3.1 ICDAR2013 Handwriting Segmentation Contest Dataset

El ICDAR 2013 Handwriting Segmentation Contest [14], se trata de un concurso que tiene como objetivo el uso de prácticas de evaluación bien establecidas, así como la recogida de avances recientes en la segmentación de texto manuscrito off-line. A diferencia de concursos previos, como los de 2007,2009 o 2010, en el de 2013 se incluye, a parte de los idiomas derivados del latín, el bengalí, un idioma hindú. En la Figura 3-1, podemos ver un ejemplo del tipo de imágenes que componen esta base de datos.

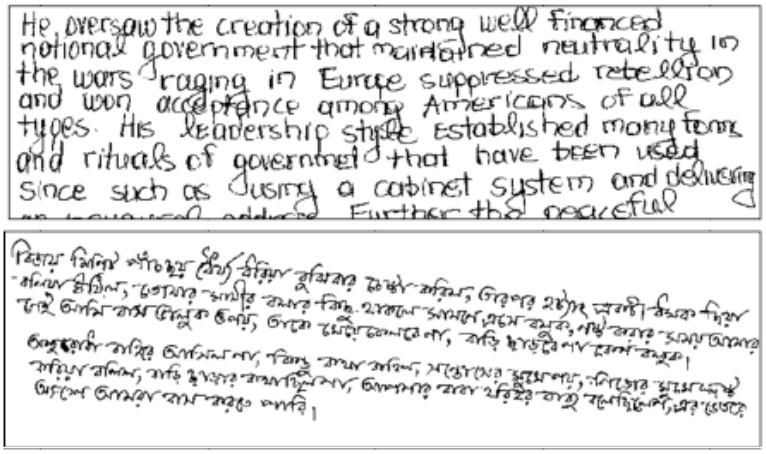


Figura 3-1. Fragmentos de imágenes que componen el dataset ICDAR 2013[14]

Además, este concurso presenta dos nuevos datasets: uno empleado para la segmentación de las líneas de texto y otro empleado para la segmentación de las palabras que conforman esas líneas (sin embargo, no contiene elementos de carácter no textual (líneas o dibujos)). Dichos conjuntos de datos, fueron creados de manera que se pudieran probar y comparar algoritmos relativamente nuevos destinados a la segmentación, en situaciones realistas [14], de imágenes de documentos manuscritos.

De esta manera, dicha base de datos (para la segmentación de líneas) está compuesta por un conjunto de 350 imágenes manuscritas (en blanco y negro) en tres idiomas: inglés (125 imágenes), griego (125 imágenes) y bengalí (100 imágenes). Por otra parte, en cuanto al conjunto de entrenamiento, decir que está compuesto por 200 imágenes (150 imágenes en griego e inglés y 50 imágenes en bengalí), mientras que el conjunto de prueba o test está formado por 150 imágenes en total (50 imágenes en griego, 50 imágenes en inglés y 50 en bengalí). Este dataset se encuentra disponible en la siguiente referencia [21].

La base de datos de ICDAR 2013, ha sido finalmente seleccionada para probar el algoritmo desarrollado en nuestro proyecto. Se ha escogido dicho set de datos ya que cumple con los requisitos de funcionamiento que plantea el algoritmo. De hecho, esta base de datos fue empleada en [3], cuyo algoritmo ha servido como base para la construcción de este proyecto, de modo que es ineludible el uso de dicho dataset para este trabajo. En el siguiente capítulo, se explicará con mayor detalle la estructura de dicha base de datos, además de las modificaciones que ha sufrido.

3.2 cBAD: ICDAR2019 Competition on Baseline Detection

Este dataset contiene un conjunto de imágenes de entrenamiento, evaluación y test que se emplearon en la cBAD: ICDAR2019 Competition on Baseline Detection [22]. Dicho dataset está disponible en la siguiente referencia [23].

Esta competición ha sido organizada usando ScriptNet [22] y, como se ha comentado previamente, los conjuntos de entrenamiento, evaluación y test están publicados en la el portal Zenodo [23], de manera que nos aporta un vista previa al conjunto de datos. Se elige Zenodo para referenciar esta base de datos, debido a que este portal es capaz de conservar documentos a largo plazo; permite el versionado, mejorando así la comparabilidad de resultados; y crea un DOI (Digital Object Identifier) que puede ser empleado para citar el conjunto de datos con independencia de donde esté realmente. Además, cabe destacar que en esta competición se emplea el formato PAGE XML [24], que de hecho se usa habitualmente en tareas de análisis de documentos, y que además es el esquema que mejor se ajusta a las necesidades de esta competición. Podemos ver un ejemplo del formato empleado en la Figura 3-2.

```
<?xml version="1.0" encoding="UTF-8" standalone="yes"?>
<PcGts
  xmlns="http://schema.primaresearch.org/PAGE/gts/pagecontent/2013-07-15"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xsi:schemaLocation="http://schema.primaresearch.org/PAGE/gts/pagecontent/2013-07-15
  http://schema.primaresearch.org/PAGE/gts/pagecontent/2013-07-15/pagecontent.xsd">
    <Metadata>
      <Creator>CVL</Creator>
      <Created>2018-11-29T08:46:03Z</Created>
      <LastChange>2018-11-30T10:18:12Z</LastChange>
    </Metadata>
<Page imageFilename="document.tif" imageWidth="2959" imageHeight="4332">
  <TextRegion id="R0" type="Handwritten">
    <Coords points="2401,228 2647,228 2647,399 2401,399"/>
       <TextLine id="L0">
         <Coords points="2439,306 2574,310 2573,360 2438,356"/>
         <Baseline points="2438,351 2573,355"/>
       </TextLine>
  </TextRegion>
</Page>
</PcGts>
```

Figura 3-2. Ejemplo del formato PAGE XML [24]

Por otra parte, decir que el dataset está compuesto por un total de 3028 imágenes que fueron muestreadas a partir de 175567 imágenes. Esta tarea se realizó mediante el uso de un fichero escrito en Python, encargado de garantizar que los datos hayan sido muestreados de forma uniforme [22]. Una vez seleccionadas las 3028 imágenes de documentos, estas fueron etiquetadas por la herramienta DigiTexx [22], y finalmente, gracias a un proceso de inspección del groundtruth, se obtuvo un set de datos de 3021 imágenes.

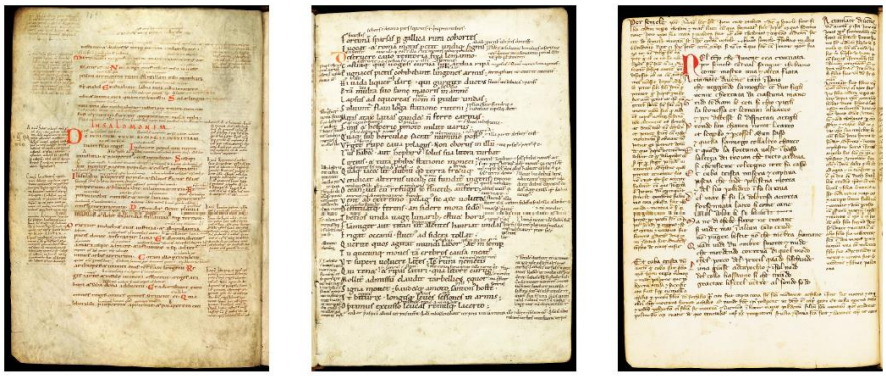
Una vez obtenido el total de imágenes, se procede a la división de la base de datos en tres conjuntos: el conjunto de entrenamiento, que contiene un total de 755 imágenes; el conjunto de evaluación, que al igual que el de entrenamiento contiene un 25% de las imágenes, es decir 755 de ellas; y el conjunto de test que comprende unas 1511 imágenes, suponiendo por tanto un 50% del total. Además, el dataset definido, está formado principalmente por documentos escritos en diferentes idiomas. La colección puede contener tablas, dibujos, documentos manuscritos medievales, grabados, páginas del George Forrest Herbarium y documentos históricos impresos. En la Figura 3-3 vemos un ejemplo de los tipos de imágenes que pueden componer esta base de datos.

Figura 3-3. Imágenes de ICDAR 2019 pertenecientes diferentes colecciones. Hay páginas muy estructuradas (a); páginas poco inscritas (b), (d) y (e); dibujos (b) y grabados (d); y documentos impresos (f) [22]

Por último decir que esta base de datos no fue seleccionada para el proyecto, debido a que no presenta en su ground-truth (que se encuentra en formato PAGE.xml, como se ha destacado anteriormente) las coordenadas poligonales de la línea, siendo así no apta para trabajar con algoritmos que se basan en la segmentación de líneas, como es el que se desarrolla en este trabajo. Cabe mencionar que de haber incluido las coordenadas poligonales de las líneas de texto, habría que haber seleccionado aquellas imágenes que contuvieran solo elementos textuales, por lo que no se hubiera podido probar la base de datos en su conjunto total, ya que al algoritmo presentado solo contempla trabajar con imágenes de texto. Ahora veremos que este problema lo resuelve muy bien el siguiente conjunto de datos.

3.3 DIVA-HisDB Historical Document Image Database (DIVA-HisDB)

DIVA-HisDB [17] es un gran dataset de manuscritos medievales para la evaluación de varias tareas de DIA (Document Image Analysis), tales como: análisis de estructura, segmentación de líneas de texto, binarización e identificación del escritor. En realidad, se trata de una colección formada por tres tipos de manuscritos medievales, seleccionados en función de la complejidad de su estructura: St. Gallen, Stiftsbibliothek, Cod. Sang. 18, codicological unit 4 (CSG18); St. Gallen, Stiftsbibliothek, Cod. Sang. 863 (CSG863); y por último, Cologny-Gen`eve, Fondation Martin Bodmer, Cod. Bodmer 55 (CB55). En la Figura 3-4 podemos observar ejemplos de estas tres clases de manuscritos. Dicha base de datos está disponible en la siguiente referencia [25].



(a) CSG18, p.116 (b) CSG863, p.17 (c) CB55, p.67v

Figura 3-4. Ejemplo de páginas de cada tipo de manuscrito medieval en DIVA-HisDB [17]

Estos manuscritos, han sido elegidos con la mirada puesta en el periodo Carolingio, siendo así representado por las clases CSG18 y CSG63 [17], que contienen documentos del siglo XI y que además están escritos en latín usando el minúsculo guión Carolingio, presentando de esta manera características estructurales muy similares. Cabe destacar que, en estos dos manuscritos, el número de escritores no se ha podido identificar. Así pues, como no es bueno centrarse tan solo en los documentos escritos en un periodo de tiempo concreto, con la clase CB55, los autores consiguen dar mayor dimensión y complejidad a la base de datos. Los manuscritos pertenecientes a este último tipo datan del siglo catorce, mostrando otra clase de guión, otro lenguaje (latín e italiano) y otra estructura diferente. Las páginas seleccionadas de cada manuscrito, para que conformen el conjunto de datos definitivo, son muy diversas. Pueden ser desde las páginas más simples, con pocos brillos interlineales, hasta las muy complejas con muchos brillos, anotaciones e incluso decoraciones.

Esta base de datos está compuesta por un total de 150 páginas anotadas (50 páginas de cada tipo de manuscrito) en color, coincidiendo en su estructura particularmente desafiante. Todas las imágenes que comprenden el dataset, han sido digitalizadas con una resolución de 600 dpi y tienen un tamaño de 20 x 25 cm aproximadamente [17]. Al igual que la base de datos ICDAR 2019, de la que hacíamos referencia en un apartado superior, el ground-truth de este set de datos también sigue el formato PAGE [24], aunque además contamos con su formato a nivel de pixel.

Por otra parte, el conjunto de entrenamiento está formado por unas 60 imágenes (20 páginas por tipo de manuscrito), mientras que tanto para el conjunto de validación y el de test se emplean 30 imágenes respectivamente (10 páginas por manuscrito). En cuanto a las 30 páginas (10 por manuscrito) no utilizadas, decir que se han apartado intencionalmente con objeto de posibilitar la organización de competiciones en el futuro.

Por último, decir que este dataset ha sido seleccionado para probar el algoritmo desarrollado en nuestro proyecto. Se ha escogido dicho set de datos, ya que cumple con los requisitos de funcionamiento que plantea el algoritmo. De hecho, esta base de datos se puede emplear para algoritmos que trabajen elementos textuales y no textuales, así como para aquellos que solo trabajen con imágenes de texto. Así pues, constituye una gran ventaja, pues permite, a diferencia del set de ICDAR 2019, que el algoritmo expuesto pueda aprovechar el set de datos en su totalidad.

.

4 ESTRUCTURA DE LAS BASES DE DATOS SELECCIONADAS

En este capítulo se van a explicar con mayor detalle las características de las dos bases de datos seleccionadas: ICDAR 2013 y DIVA-HisDB. Además se revisará la generación del ground-truth, así como la de los tensores, necesarios para generar los datos de entrenamiento y test del modelo planteado. Todo ello se ve reflejado en el *bloque 1* del diagrama mostrado en la Figura 1-12 del Apartado 1.3, que se desglosa a continuación, y que servirá como esquema fundamental para entender los pasos que se darán en este capítulo (Figura 4-1):

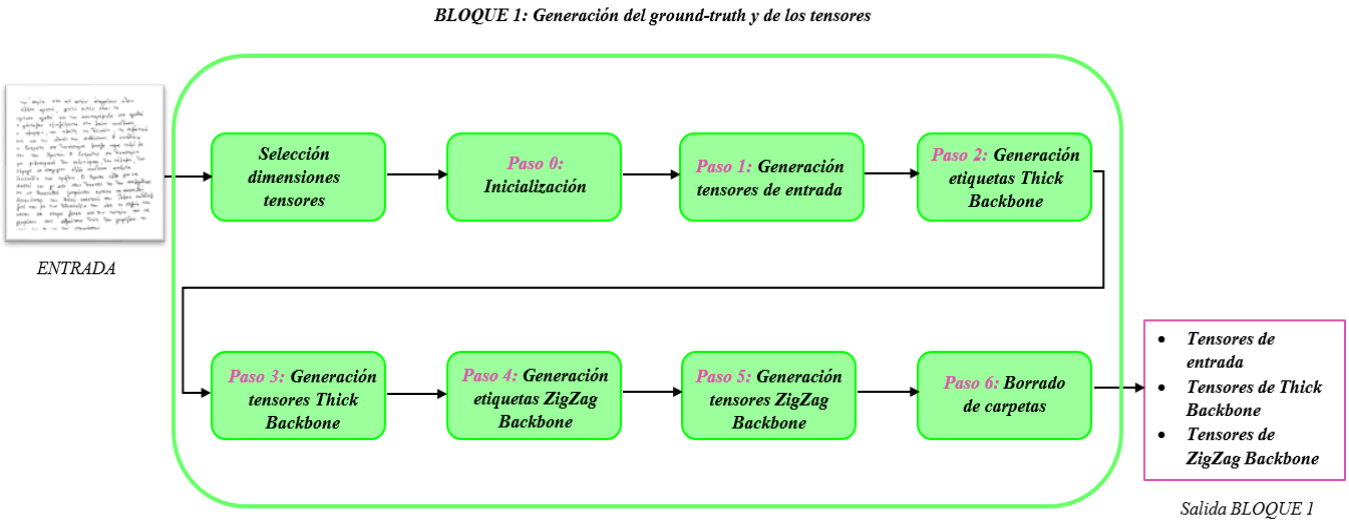


Figura 4-1. Contenido del *Bloque 1* de la estructura "*pipeline*" del proyecto, cuya entrada se trata del conjunto de imágenes (en blanco y negro) que componen el dataset seleccionado, y cuya salida son los tensores de las imágenes de entrada, los tensores de Thick Backbone y los tensores de ZigZag Backbone

4.1 División de las base de datos en conjuntos de entrenamiento, validación y prueba

4.1.1 ICDAR2013 Handwriting Segmentation Contest Dataset

Como ya se comentó en el capítulo anterior, esta base de datos está formada por un total de 350 imágenes manuscritas, en tres idiomas diferentes. Para poder preparar el conjunto de datos de forma que fuera usado en el proyecto, decidimos dividirlo de la misma forma que se hace en [3]. Es cierto que se podrían haber seguido las pautas que se recomiendan en ese artículo, y así conseguir una división sensata de las imágenes que componen el set. De esta manera, deberíamos haber destinado el 20% de las imágenes al conjunto de test, y 80% restante de destinaría a formar el conjunto de entrenamiento y el de validación. No obstante, como bien se explica en [3], el concurso del que salió esta base de datos impone una serie de restricciones a los proyectos que se deben presentar, y una de ellas era precisamente la de destinar las 200 primeras imágenes de la base de datos al conjunto de entrenamiento, y las 150 restantes al conjunto de test. Así, para seguir la línea de [3] y del concurso de 2013, decidimos utilizar esta división también. Por tanto, esta sería la división resultante: 160 imágenes de entrenamiento, 40 de validación y 150 de test o prueba.

4.1.2 DIVA-HisDB Historical Document Image Database (DIVA-HisDB)

En la sección anterior veíamos que la base de datos estaba compuesta por un total de 150 imágenes de tres tipos de documentos distintos. La división del conjunto de datos que hemos planteado, es exactamente la misma que establece esta base de datos, pues al descargarla ya viene dividida en los conjuntos de entrenamiento, validación y test, lista para ser usada. De esta forma, el conjunto de entrenamiento quedaría con 60 imágenes, mientras que el de validación y prueba quedarían con 30 imágenes cada uno. Notamos que efectivamente, este dataset es mucho más pequeño que el anterior, aunque hay que destacar que la estructura de las imágenes que lo componen, es mucho más compleja que las de ICDAR 2013.

4.2 Generación del ground-truth y de los tensores

Este apartado, como ya se enunciaba en la introducción del capítulo, se va a enfocar a la forma en la que se han generado tanto el groundt-truth como los tensores, necesarios para poner en marcha el algoritmo propuesto. Precisamente, para dicha tarea se creó un archivo llamado *computeAllSamples.py,* con el objetivo de generar de forma ordenada estos componentes.

Antes de comenzar con la descripción de la sección, es importante definir el concepto de tensor. Un tensor es básicamente una matriz que posee más de dos dimensiones. Normalmente, en procesos como el que abordamos en este proyecto, es decir, de Machine Learning, las imágenes deben de estar en formato de tensor, de forma que posean 3 dimensiones: alto, ancho y número de canales (indicando que existirán tantas matrices de dos dimensiones (alto x ancho), como canales haya. Siendo tres en caso de que el formato sea RGB, y una dimensión, si el formato es blanco y negro). Por otra parte, señalar que se ha establecido un tamaño máximo de imagen permitido de 4096 x 3072 para la base de datos de ICDAR 2013, mientras que para el dataset DIVA-HisDB se ha fijado en 5120 x 3584. De esta manera, los tensores generados para cada base de datos estarán ajustados a dichas dimensiones específicas. El proceso de ajuste de los tamaños de los tensores tiene que ser de este modo, y ello se debe a que las redes neuronales del tipo U-net (como la que se usa en nuestro proyecto) no admiten dimensiones diferentes de las imágenes de entrada. El proceso de elección de las mismas, para cada base de datos, se explica en el subapartado siguiente.

4.2.1 Selección de las dimensiones de los tensores

En este subapartado se hará una descripción del proceso de selección de las dimensiones de los tensores a crear en el proyecto. Se comenzará detallando el caso del dataset ICDAR 2013, siguiendo con la descripción para el caso del dataset DIVA-HisDB.

4.2.1.1 Selección para la base de datos ICDAR 2013

Al analizar el dataset elegido, nos encontramos con imágenes de tamaños muy distintos. Así, para poder obtener tensores de las mismas dimensiones, se realizó un estudio en el que se reflejaban cuáles eran los valores de altura y anchura, máximos, mínimos y medios (también se calcularon estos valores en relación al número de líneas, ofreciendo así una visión más detallada del conjunto de datos). Dicho análisis ya se llevaba a cabo en [3], de forma que decidimos reutilizar su script de Python (en el que calculan los valores antes descritos) y obtener así los resultados del estudio de dimensiones. Este script se llama *nLinesAndShapeAnalyzer.py* y funciona de la siguiente forma: simplemente le pasamos los archivos *.txt* de la carpeta shape (archivos de texto que contienen las dimensiones originales de cada imagen del dataset) y los de la carpeta nLines (archivos de texto que contienen el número de líneas de las imágenes del dataset), que encontraremos en la ruta /DataBases/ICDAR13/, y nos devuelve un archivo de texto con los resultados del análisis. Además realizamos dicho estudio sin la presencia de las imágenes 224.tif y 327.tif, ya que como se explica en [3], son las que poseen las dimensiones mayores y con ellas el análisis ofrecía un valor máximo de 4496 x 3227 de manera que daba problemas al ser implementado en la red U-Net. En la Tabla 4-1, vemos los resultados obtenidos sin las imágenes mencionadas anteriormente:

Tabla 4–1. Resultados del estudio de dimensiones para ICDAR 2013

	Alto (px)	Ancho (px)	Número de líneas
Máximo	3652	2534	36
Medio	2082.52	2283.48	18.22
Mínimo	651	1427	6

De este modo, con las imágenes 224 y 327 eliminadas, y siguiendo el proceso que se lleva a cabo en [3] (el cual nos sirve como base en todo lo referido a este dataset), se seleccionan como dimensiones fijas los valores de 4096 x 3072. Se escogieron estas cifras ya que se acercan mucho a los máximos del estudio de dimensiones, y también por sus factorizaciones (4096 es 2^{12}, mientras que 3072 es 2^{10} x 3). Es importante destacar que las factorizaciones de los tamaños a fijar son de suma relevancia para elegirlos, ya que si no tiene una factorización que sea potencia de 2 o múltiplo, la red U-Net tiene problemas con la concatenación de las capas que la forman. Además, hemos decidido escoger ese valor máximo y no el medio, para ajustar el tamaño de los tensores, ya que solo vamos a realizar una operación de *zero-padding*, igual que se hace en [3]. Mediante esta operación se añaden ceros fuera de la imagen original hasta que esta alcance las dimensiones fijadas, de manera que estas últimas sean tales que la convolución realizada consiga una imagen final de igual resolución que la original. Así pues, los tensores serán creados con un *shape* de *(4096, 3072,1).*En el subapartado inferior veremos otra aproximación a este problema.

4.2.1.2 Selección para la base de datos DIVA-HisDB

A diferencia del subapartado anterior, en la base de datos DIVA-HisDB nos encontramos con 80 imágenes pertenecientes a los manuscritos CS18 y CS863 que tienen las mimas dimensiones: 4992 x 3328. Por otra parte, las 40 imágenes restantes, pertenecientes al manuscrito CB55, tienen las dimensiones de 6496 x 4872. De esta forma, no es necesario realizar el análisis de dimensiones mediante el script que se mencionaba en el subapartado superior (solo lo debemos emplear en caso de que se quieran obtener también los valores para el número de líneas), simplemente haría falta calcular el valor medio y ya tendríamos la tabla de resultados (Tabla 4-2):

Tabla 4–2. Resultados del estudio de dimensiones para DIVA-HisDB

	Alto (px)	Ancho (px)	Número de líneas
Máximo	6496	4872	33
Medio	5493.33	3842.66	29.18
Mínimo	4992	3328	7

En este caso, para realizar la selección del tamaño de los tensores nos fijamos en el valor medio de la Tabla 4-2. Así pues, elegimos unos valores que estén entorno al valor medio, y que por supuesto tengan unas factorizaciones adecuadas, de modo que no den ningún error al pasárselos a la red U-Net, como ya se explicaba en el punto anterior. Por tanto, la dimensión seleccionada es de 5120 x 3584, con una factorización de 2^{10} x 5 y 2^9 x 7, respectivamente. Este paso de acercarnos al valor medio y no al máximo, como se hacía para la base de datos ICDAR 2013, es debido a que no solo se va a usar la operación de *zero-padding,* que hará que las imágenes de menor tamaño se ajusten a la dimensión seleccionada, sino que también se va a hacer uso de la operación de *downsampling*, que se encargará de que las imágenes de mayor tamaño se ajusten a la dimensión seleccionada, que por supuesto es menor que el tamaño de dichas imágenes. Es importante señalar también que se decidió ajustar a un valor medio, debido a que, en caso contrario, las imágenes con dimensiones mayores verían muy reducida su calidad. Así, los tensores creados tendrán un *shape* de *(5120, 3584,1).* Estas operaciones se verán de forma más ilustrativa en subapartados posteriores.

A continuación pasaremos a explicar el proceso de generación de tensores y ground-truth, que se divide en un

total de seis pasos, contando adicionalmente con un paso 0 de inicialización. Todo ello, como se introdujo anteriormente, queda recogido en el script *computeAllSamples.py* que podremos encontrar en los archivos del proyecto.

4.2.2 Paso 0: Inicialización

Antes de generar tanto el ground-truth como los tensores, se deben de inicializar una serie de parámetros. El primero de ellos es una bandera llamada *DataBaseType*, con la que se indica el tipo de base de datos que se va a emplear y por tanto a la que le debemos extraer los componentes anteriores. En este proyecto contemplamos solo dos conjuntos de datos, el de ICDAR 2013, que indicaríamos poniendo a "1" la bandera, y el de DIVA-HisDB, para el que la bandera se pondría a "0". No obstante, podríamos añadir más bases de datos, asignando otros números a las banderas si tuvieran características distintas a las que ya presentamos en este trabajo. Esto da pie a aclarar que la distinción a través de *DataBaseType* de ambos datasets es porque, efectivamente, trabajan con datos de diferentes extensiones, como se irá viendo más adelante.

Una vez que se indica la base de datos con la que se va a trabajar, se definen una serie de *paths* o rutas, que en esencia, son las rutas a la base de datos y a las imágenes de la misma, de forma que si se usa *computeAllSamples.py* en otro ordenador, se puedan cambiar las rutas de una manera más efectiva y menos tediosa. Se ha de mencionar que en el caso de la base de datos ICDAR 2013 se incluye la ruta a unos archivos guardados en una carpeta llamada *gt_lines*, proporcionados por el propio concurso. Estos ficheros, en formato *tif.dat*, son realmente archivos de ground-truth que contienen la matriz de la imagen de la que llevan el nombre o número. Esta matriz representa el fondo de la imagen (blanco) con valores "0" en esas posiciones, mientras que las líneas de la imagen tienen un valor "n" asociado, de manera que la primera línea de la imagen tendrá el valor "1" en todas su posiciones de la matriz, la segunda línea vendrá representada por el "2" en todas su posiciones, y así sucesivamente hasta completar el número de líneas de la imagen. Para saber más del formato de este tipo de archivos, consultar [3].

Por otra parte, a diferencia de ICDAR 2013, la base de datos DIVA-HisDB no proporciona estos archivos (los que contiene la carpeta *gt_lines*) sino que es necesario generarlos, ya que son imprescindibles para el cálculo del ground-truth total y por tanto para la creación de los tensores. Para esta tarea, se creó una función llamada *funlines* recogida en el archivo *groundTruthFunctions.py*, al que iremos recurriendo a lo largo del capítulo. Esta función recibe como entrada una lista de las rutas necesarias para calcular las matrices de las imágenes (obviamente estas matrices tienen las mismas dimensiones que las imágenes a las que representan). Una vez que recibe dicha lista comienza a extraer las coordenadas de las líneas. Estas coordenadas (que deben ser coordenadas que describan un polígono irregular, indicando así que recoge a la línea con todos sus contornos) las va extrayendo de un archivo con formato PAGE.xml, correspondiente a cada imagen. Cuando se tienen todas las coordenadas de la línea, se rellena el polígono que forman las mismas con el número de la línea correspondiente. Este proceso se repite para cada línea y para cada región de la imagen, y al completarse se tiene la matriz que representa a la imagen, lista para ser almacenada. En el almacenamiento de dicha matriz encontramos otra diferencia con ICDAR 2013, y es que en este caso hemos almacenado cada matriz en formato *.npz*, en vez de en formato *.tif.dat* al estar este último un poco más obsoleto. Además, se generaron también archivos *.txt* que contienen el número de líneas de la imagen, y que se guardan en la carpeta *gt_nLines*.

Seguidamente, después de obtener las *gt_lines*, si es que fuera necesario (se indica con el parámetro *gt_generate* –se pone a "0" si hace falta calcularlas y a "1" en caso contrario-), se terminan de inicializar varios parámetros:

- *generateImages*, que indica con "True" que se deben generar imágenes, normalmente para ilustrar los ground-truths generados, y con "False", que no se deben generar este tipo de imágenes (ya que no se necesitan cuando estamos creando los tensores). En este paso lo dejamos inicializado en "False". Señalar que las imágenes generadas, en el caso de indicar "True", no se emplean en el aprendizaje del algoritmo.
- *cleanAfter,* que indica con "True" que todo excepto los tensores debe ser borrado, puesto que los mismos son los únicos archivos empleados en el aprendizaje del algoritmo. En este paso lo dejamos inicializado a "True", para ahorrar espacio.

Finalmente, se define una ruta principal para guardar los tensores que se vayan creando, y se pasa al siguiente paso.

4.2.3 Paso 1: Generación de los tensores para la entrada de la red usando las imágenes del dataset elegido

Una vez terminado el *Paso 0,* comenzamos con la generación de los tensores para la entrada de la red neuronal. Para generar estos tensores de entrada, se define una ruta donde guardarlos y se llama a la función *createTensor_x (*todas las funciones que aquí se mencionan, se definen en el script *groundTruthFunctions.py*).

Esta función se encarga, en primer lugar, de leer las imágenes del dataset elegido, DIVA-HisDB o ICDAR 2013 (a *createTensor_x* también le indicamos qué dataset queremos utilizar mediante la bandera *DataBaseType* anteriormente definida.) y de almacenarlas en tensores:

- Si se elige la base de datos ICDAR 2013, empezaremos definiendo las dimensiones que va a tener el tensor, que como bien se detallaba en el Subapartado 4.2.1.1, se han fijado a 4096 x 3072. Una vez establecidos los tamaños de los tensores, se continuaría con un paso de renombramiento o reenumeración de las imágenes del dataset. Esto es debido a que los tensores de las imágenes 224 y 327 no serán generados, por lo que habría que volver numerar el set de datos (esto es, desde 1 a 348, en vez de, desde 1 a 350). Seguidamente, se ajustan los tensores a las dimensiones seleccionadas previamente mediante una operación de *zero-padding.* Para terminar, los tensores creados se guardan de forma independiente como ficheros comprimidos en formato *.npz.*
- En el caso de que se elija DIVA-HisDB, empezaremos definiendo las dimensiones de los tensores, de acuerdo a lo discutido en el Subapartado 4.2.1.2. De esta forma quedarán fijadas a 5120 x 3584. A continuación, como durante todo el proceso de generación de dichos componentes, así como de la puesta en marcha del algoritmo, se trabaja con imágenes y datos en blanco y negro, en *createTensor_x* se deben convertir las imágenes a color (RGB, tres dimensiones) de DIVA-HisDB, a imágenes en blanco y negro. Así pues, estaremos trabajando con una dimensión y no con tres, adaptando la base de datos al funcionamiento del algoritmo. Seguidamente, se comprueba si el tamaño de la imagen original es menor a las dimensiones del tensor, si ese es el caso, se procede al ajuste del tensor de la imagen a las dimensiones seleccionadas mediante una operación de *zero-padding*. Si por el contrario, resulta que el tamaño de la imagen original es mayor, se llevarían a cabo dos operaciones:
    - En primer lugar, contaríamos con un paso previo de *zero-padding*, que simplemente sirve para conseguir que en la operación posterior de *downsampling*, podamos obtener el tamaño deseado del tensor en el que estamos guardando la imagen (para ello definimos dos números auxiliares cuyo único fin es hacer que los tamaños cuadren para el paso siguiente).
    - Después, realizaríamos una segunda operación de *downsampling* o submuestreo de la imagen, con la que ajustamos el tensor de la misma (que recordamos que es de dimensión mayor que el tamaño seleccionado para los tensores) a las dimensiones especificadas previamente. Cabe destacar que se ha seleccionado un factor de submuestreo de 2, tanto para el ancho como para el alto de la imagen (bien es cierto que se podría haber escogido cualquier otro).

    Finalmente, los tensores creados se guardan de forma independiente como ficheros comprimidos en formato *.npz.*

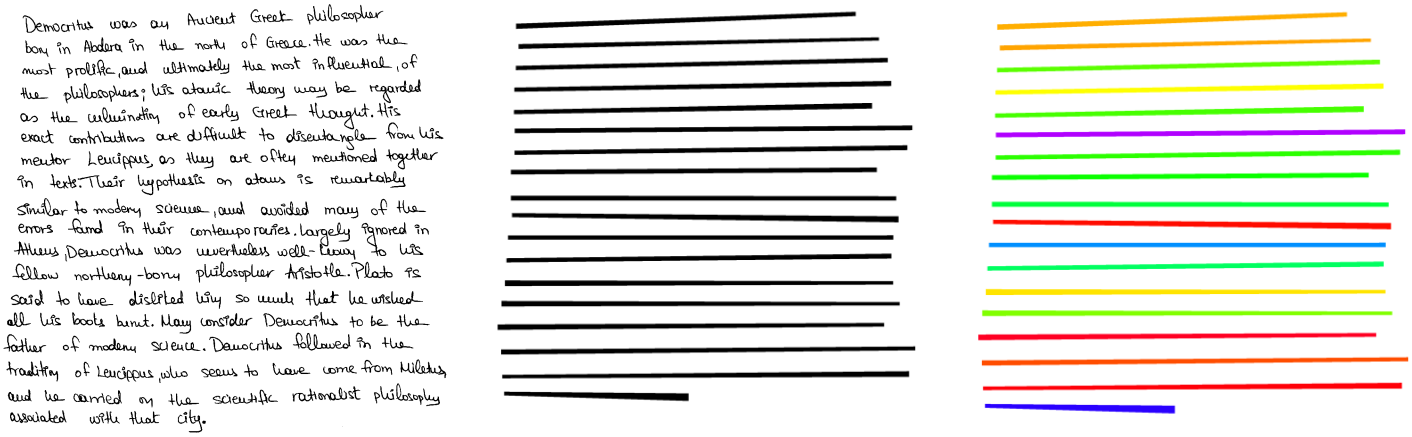
Este último paso de compresión es sumamente importante, pues reduce muchísimo el espacio que consumen dichos tensores. Después de guardarlos, ya en el script principal - *computeAllSamples.py*.-, se llama a la función *checkName* que comprueba si los nombres de los tensores generados son números o no. Si no son números, esta función cambia el nombre original del tensor por un número, que se corresponde con su orden dentro del fichero donde está almacenado. En caso contrario, no se producen cambios. Esto se hace así ya que durante todo el proyecto se trabaja con datos numerados, haciendo más sencillo el manejo de los mismos.

4.2.4 Paso 2: Generación de las etiquetas del problema de Thick Backbone (espina dorsal gruesa)

Cuando tenemos los tensores del *Paso 1*, pasamos a generar las etiquetas del problema de Thick Backbone, o lo que es lo mismo, el ground truth de Thick Backbone (estas etiquetas son en realidad imágenes o matrices donde el fondo de las mismas toma el valor "0", mientras que las líneas que las componen toman cada una un valor distinto). Para poder crearlas, definimos un listado de rutas y llamamos a la función *thickBackboneGroundTruthGenerator.* Esta función se encarga, en primera instancia, de leer las imágenes del dataset elegido, extraer sus dimensiones y almacenarlas en un archivo *.txt.* Además, si el DataBaseType tiene asignado el valor "1", es decir, que se está trabajando con la base de datos ICDAR 2013, esta función extrae el número de líneas y, al igual que con las dimensiones, las almacena en un archivo *.txt* (para el dataset DIVA-HisDB, el número de líneas se extrae en la función *funLines*, de *groundTruthFunctions.py*). Posteriormente, independientemente del dataset seleccionado, se crea un array con el fichero *.npz*, perteneciente a la carpeta *gt_lines* de la que se había hablado anteriormente (con las mismas dimensiones de la imagen original). Este array se pasa a la función *createThickBackbone,* y con ella se crea el esqueleto o espina dorsal de las líneas, siendo este el ground-truth con el cual, en el *Paso 3*, se generaran los tensores de Thick Backbone.

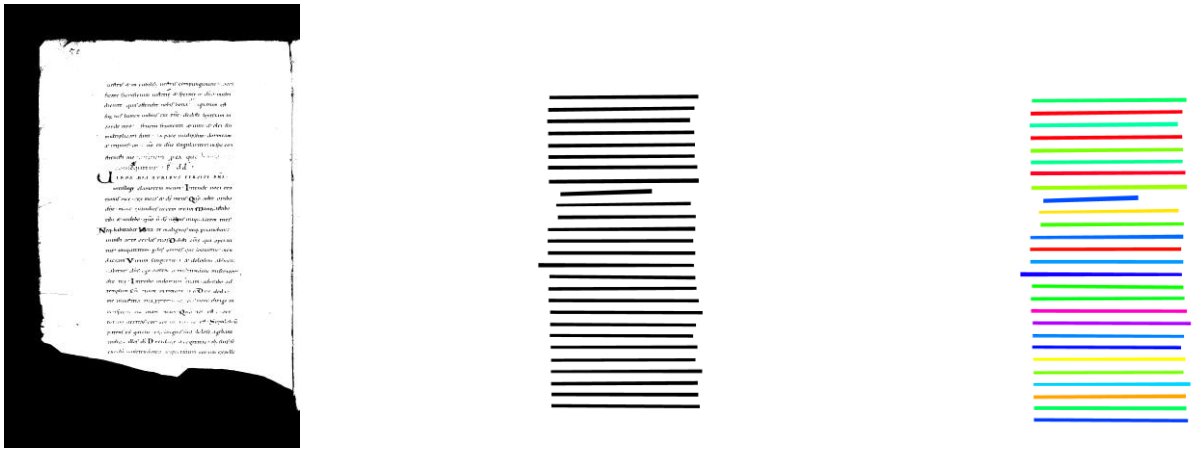
Volviendo a la tarea que desempeña *createThickBackbone,* decir que como bien se explica en [3], esta función comienza con una inspección de la imagen por columnas, cogiendo para cada una el píxel más bajo y el más alto de cada línea. Cuando tenemos esto, llamamos a la función *createLinearRegressionBackbone* (se llama dentro de *createThickBackbone*), que devolverá dos bandas situadas a la altura del mínimo y el máximo de cada línea, de forma aproximada. Para terminar, se rellenan todos los píxeles que hay entre estas dos rectas, de modo que se cree un trapecio, muy parecido a lo que se quiere lograr.

Una vez que *createThickBackbone* ha terminado su tarea, se continúa con la ejecución de *thickBackboneGroundTruthGenerator,* donde lo próximo a realizar es guardar el resultado de la función anterior, que se trata de un fichero binario, en dos formatos diferentes: como tipo entero de 32 bits (int32) a color (donde cada línea tiene asignado un valor distinto), y como tipo Boolean (bool) en blanco y negro (ambos comprimidos en *.npz*), donde se distingue el fondo de las líneas. También generamos las imágenes de los resultados conseguidos, guardándolas en formato *.png.* Y así, tendríamos el ground-truth del problema de Thick Backbone. En la Figura 4-2 y 4-3, se muestran los dos tipos de imágenes de los resultados obtenidos, para ambas bases de datos respectivamente. Todas las funciones mencionadas en este apartado se encuentran en el fichero *groundTruthFunctions.py.*



(a) (b) (c)

Figura 4-2. (a) Imagen 024 perteneciente al dataset ICDAR2013, (b) formato bool del Thick Backbone generado a partir de (a), (c) formato int32 del Thick Backbone generado a partir de (a)



(a) (b) (c)

Figura 4-3. (a) Imagen 031 perteneciente al dataset DIVA-HisDB, (b) formato bool del Thick Backbone generado a partir de (a), (c) formato int32 del Thick Backbone generado a partir de (a)

4.2.5 Paso 3: Generación de los tensores para el problema de Thick Backbone (espina dorsal gruesa)

Al terminar de obtener el ground truth para Thick Backbone, se generan los tensores para este mismo problema. Para ello, se llama a la función *createTensorThickbackbone_y*, que como las otras funciones en los pasos anteriores, pertenece al archivo *groundTruthFunctions.py*. Esta función se encarga de tomar los archivos tipo Boolean (calculados en el paso anterior), y los almacena en tensores que se guardan y se ajustan a las dimensiones seleccionadas, de la misma forma que en el *Paso 1*, como ficheros comprimidos *.npz* y llevando a cabo las operaciones correspondientes a la base de datos seleccionada, respectivamente (también se vuelve a llamar a la función *checkName*).

Seguidamente, antes de completar este paso, si hemos asignamos el valor "True" a la bandera *cleanAfter* en el *Paso 0,* se borraría la carpeta que contiene el ground-truth del Thick Backbone, pues ya no sería necesaria.

4.2.6 Paso 4: Generación de las etiquetas del problema para ZigZag Backbone (contorno de cada línea en forma de zigzag)

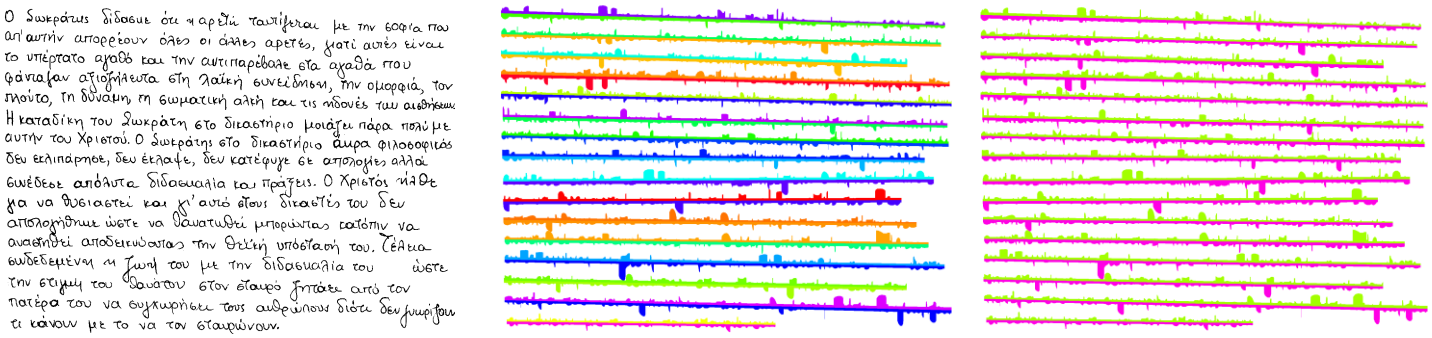
En este cuarto paso, la tarea a realizar es la de generar las etiquetas del problema de ZigZag Backbone (ver Sección 1.2), o lo que es lo mismo, el ground truth de ZigZag Backbone. Para ello, es necesario definir un listado de rutas y llamar a la función *zigzagGroundTruthGenerator*. Esta función comienza leyendo las imágenes del dataset elegido para extraer sus dimensiones y almacenarlas en un archivo *.txt.* Además, de la misma forma que ocurre en el *Paso 2*, si estamos trabajando con el dataset de ICDAR 2013, dicha función también extraerá el número de líneas, almacenándolas en un archivo *.txt.* Posteriormente, independientemente del dataset seleccionado, se crea un array con el fichero *.npz*, perteneciente a la carpeta *gt_lines* de la que se había hablado anteriormente (con las mismas dimensiones de la imagen original). Este array se pasa a la función *createZigZagBackbone*, que crea por una parte, el contorno de las líneas de la imagen que se está procesando (la parte de zigzag), y por otra la espina dorsal en forma de línea. Estas tareas se realizarían con las funciones *createOutlineBackbone* y *createLinearRegressionBackbone*, que para obtener el Zigzag Backbone son imprescindibles, ya que el mismo se consigue mediante la comparación de los resultados obtenidos por las dos funciones anteriores.

De esta forma, lo que se hace a continuación dentro de *createZigZagBackbone*, como se explica en [3], es que dado un número de contorno de línea, se sustituyen los valores de ese mismo contorno, que se encuentran por debajo de la espina dorsal, en forma de línea. Para que dicha tarea resultara más sencilla de realizar, se decidió que para un número "n" de línea, la mitad inferior de su contorno tuviera asignado un valor "-n", mientras que su mitad superior tuviera un valor asignado de "n". Sin embargo, esto es un poco "rebuscado" para que una red

neural lo realice, por lo que, de vuelta en *zigzagGroundTruthGenerator*, se emplea la función *simplifyZigZag*, que lo que hace es simplificar el ground truth obtenido (hasta este punto de la ejecución de la función principal). Para ello, colapsa todas las mitades superiores de los contornos, o lo que es lo mismo, todos los valores positivos en el valor "1", y todas las mitades inferiores, o valores negativos, de los contornos en el valor "-1". Para más información del Outline Backbone, se aconseja consultar [3].

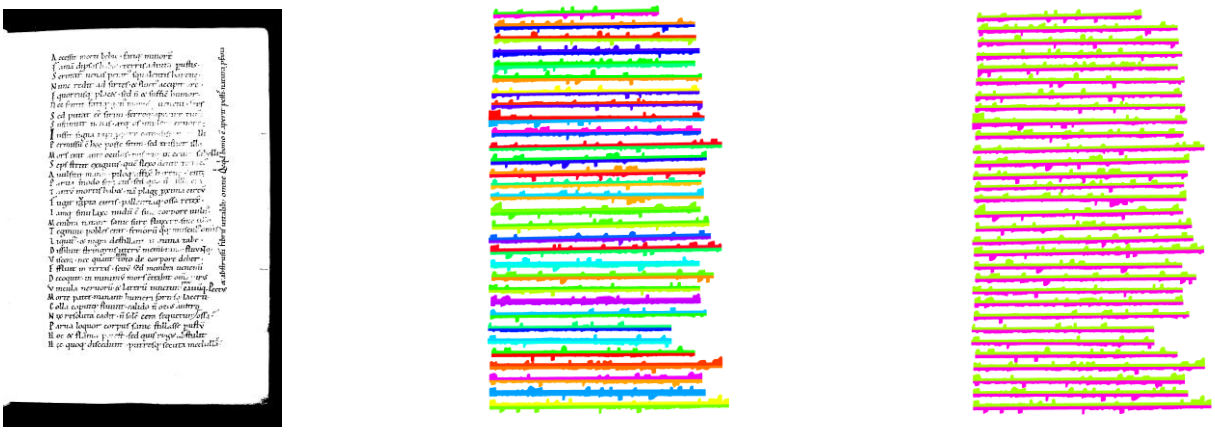
Una vez que *createZigZagBackbone* ha terminado su tarea, se continúa con la ejecución de *zigzagGroundTruthGenerator,* donde lo próximo a realizar es guardar el resultado de la función anterior (fichero binario) en dos formatos de imagen distintos: uno con las líneas numeradas y el otro con el etiquetado de la mitad inferior y superior solamente (ambos como tipo entero de 32 bits (int32)) (ambos en formato comprimido *.npz*). También generamos las imágenes de los resultados conseguidos, guardándolas en formato *.png*. Y así, tendríamos el ground-truth del problema de ZigZag Backbone.

En la Figura 4-4 y 4-5, se muestran los dos tipos de imágenes de los resultados obtenidos, para ambas bases de datos respectivamente. Todas las funciones mencionadas en este apartado se encuentran en el script *groundTruthFunctions.py*, disponible en los archivos del proyecto.



(a) (b) (c)

Figura 4-4. (a) Imagen 025 perteneciente al dataset ICDAR 2013, (b) ZigZag Backbone original de (a), y (c) ZigZag Backbone simplificado de (a)



(a) (b) (c)

Figura 4-5. (a) Imagen 025 perteneciente al dataset DIVA-HisDB, (b) ZigZag Backbone original de (a), y (c) ZigZag Backbone simplificado de (a)

4.2.7 Paso 5: Generación de los tensores para el problema de ZigZag Backbone (contorno de cada línea en forma de zigzag)

Tras completar el paso anterior, se generan los tensores para el mismo problema que se trataba en ese paso. Para ello, se llama a la función *createTensorZigzag_y* (recogida en *groundTruthFunctions.py*). Esta función se encarga de tomar los archivos de ground truth simplificados, obtenidos en el *Paso 4*, y de almacenarlos en tensores, que se guardan y se ajustan de la misma manera que en los *Pasos 1* y *3*. Esto es, en formato comprimido *.npz* y realizando las operaciones correspondientes a la base de datos elegida. Además, al final de esta tarea, se vuelve a llamar a la función *checkName.*

Seguidamente, antes de completar el *Paso 5,* si asignamos el valor "True" a la bandera *cleanAfter* en el paso de inicialización*,* se borrarían los ficheros del ground-truth de ZigZagBackbone, ya que no serían necesarios en pasos posteriores, y si no se eliminan lo único que hacen es consumir un espacio bastante elevado.

4.2.8 Paso 6: Borrado de las carpetas innecesarias

Al llegar a este paso, ya tendríamos todos los tensores y archivos de ground-truth generados (en el caso que no se hubieran borrado en los pasos anteriores). Pero como hemos podido ir viendo a medida que generábamos los tensores, estos al estar comprimidos suponen un 1% des espacio en contraposición con el tamaño original (se ahorra un 99% de espacio), pero para ser generados necesitan emplear los archivos de ground truth, que ocupan varios GB. Para resolver este problema, y como una vez calculados los tensores estos últimos archivos no son necesarios, asignamos a la bandera *cleanAfter* el valor "True", eliminando así todas las carpetas y archivos innecesarios, para poner en marcha el algoritmo propuesto.

Así pues, al final de este proceso de generación de componentes, nos quedaríamos, al menos, con:

- Los tensores con las imágenes de entrada en blanco y negro. Podemos encontrarlos en la ruta de archivos /lineseg/tensors/tensorsDIVA/input/, para la base de datos DIVA-HisDB, y en la ruta /lineseg/tensors/tensorsICDAR2013/input/, para el dataset ICDAR 2013.
- Los tensores binarios de Thick Backbone, accesibles mediante las rutas de archivos:
  - /lineseg/tensors/tensorsDIVA/thickBackboneTensors/, para el dataset DIVA-HisDB
  - /lineseg/tensors/tensorsICDAR2013/thickBackboneTensors/, para el dataset ICDAR 2013
- Los tensores del ZigZag Backbone simplificado, accesibles mediante las rutas de archivos:
  - /lineseg/tensors/tensorsDIVA/zigzagTensors/, para el dataset DIVA-HisDB
  - /lineseg/tensors/tensorsICDAR2013/zigzagTensors/, para el dataset ICDAR 2013

Todos ellos adjuntados, por supuesto, en los archivos del proyecto (que se pueden encontrar en el Anexo B de códigos).

5 PRE-SEGMENTACIÓN

En este capítulo se va a tratar, en primera instancia, una nueva propuesta de implementación de red neural, abordándose de forma muy distinta a lo que se hacía en [3], pero manteniendo el mismo funcionamiento. También nos centraremos en el entrenamiento de la red neural que estamos revisando, así como en los distintos elementos que forman parte de ese proceso. Finalmente, obtendremos los resultados del entrenamiento de la red, de cada base de datos seleccionada (ICDAR 2013 y DIVA-HisDB), para los dos modelos o problemas ya mencionados en el capítulo anterior: Thick Backbone y ZigZag Backbone. Por último, mencionar que a todo este proceso lo llamaremos "pre-segmentación", y se puede apreciar de forma esquemática, siguiendo la estructura "*pipeline"* del proyecto representada en la Figura 1-12 del Apartado 1.3, en la siguiente Figura 5-1:

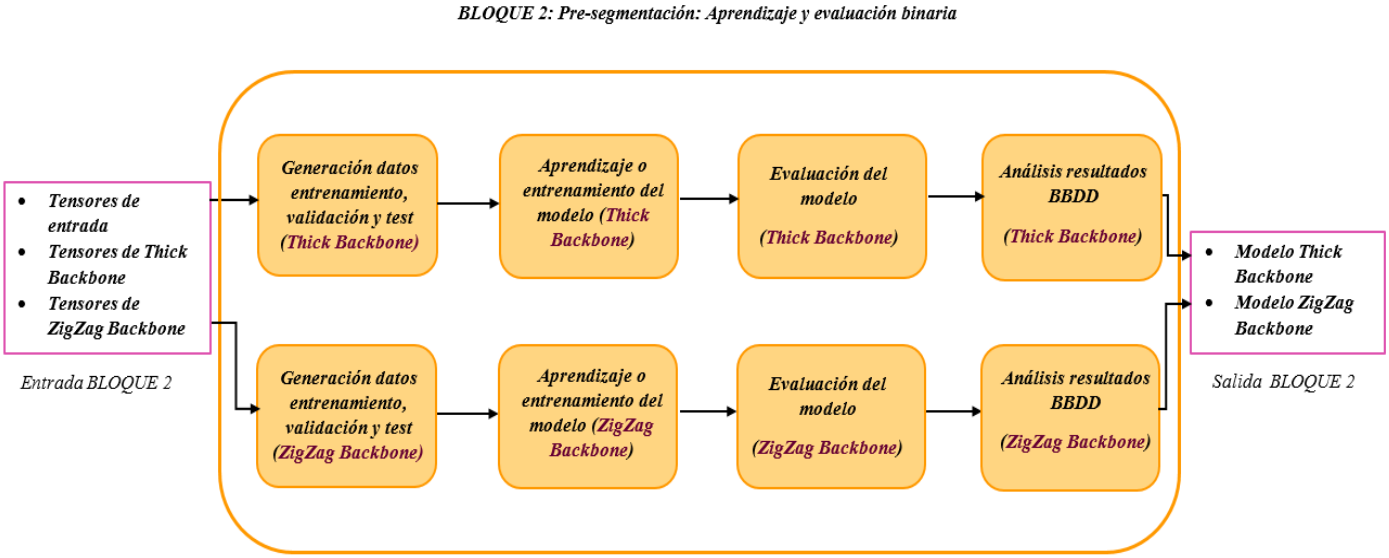


Figura 5-1. Contenido del *Bloque 2* de la estructura "*pipeline*" del proyecto, cuya entrada se trata del conjunto de tensores conseguidos en el *Bloque 1*, y cuya salida son los modelos de Thick Backbone y ZigZag Backbone, listos para el paso de segmentación (*Bloque 3)*

5.1 Nueva propuesta

En [3], que hemos usado como base para el desarrollo del algoritmo propuesto, se describía una implementación de una red neuronal que pudiese aprender a extraer los contornos de las imágenes, con el objetivo de crear una máscara de segmentación para su uso posterior. Además [3] proponía que, para el caso del problema de ZigZag Backbone se entrenaran dos redes U-net por separado, de modo que una fuera capaz de aprender a distinguir la mitad superior de una línea, mientras la otra hiciera lo mismo pero con la mitad inferior. De esta manera, la red aprendería de forma separada a segmentar el contorno superior e inferior de las líneas de las imágenes, que posteriormente quedaría fusionado en una máscara de segmentación.

Sin embargo, una vez analizada esta manera de abordar el problema de ZigZag Backbone, se concluyó que no tenía sentido entrenar la misma red dos veces (generando así dos modelos de entrenamiento distintos). ¿Por qué entrenar la misma red dos veces consumiendo recursos, pudiendo entrenarla una sola vez? Para resolver esta pregunta, se optó por enfocar el problema de un modo distinto. Se propuso extraer solamente la parte superior de las líneas, de forma que la red aprendiera como extraer dicha parte, mientras que la parte inferior se extrajera, reutilizando la misma red, de la imagen rotada unos 180 grados. De esta manera, la red extraería el contorno superior de las líneas de texto para la imagen rotada, pero al estar ésta girada, realmente se estaría extrayendo el contorno inferior de las mismas. Por tanto, para obtener la imagen del contorno inferior de las

líneas que componen el texto estudiado, solo haría falta volver a rotar la figura anterior unos 180 grados. Así solo habría que entrenar una sola red, de la que extraeríamos un solo modelo.

Además de aportar este cambio para optimizar el modelo presentado en [3], se propone la detección y corrección de los diferentes casos de error expuestos en [3] y aquellos nuevos detectados en este trabajo (se revisan en el Capítulo 6). De la misma forma, y como se ha podido comprobar en el Capítulo 4, se propone la adaptación de una nueva base de datos, DIVA-HisDB, al algoritmo desarrollado en el proyecto.

Todo el proceso para llegar a esta nueva propuesta, se refleja en el subapartado siguiente, en el que se explica con detalle cada una de las modificaciones realizadas al modelo planteado por [3]. Mencionar también que el problema de Thick Backbone no ha recibido ninguna alteración (más allá de las modificaciones necesarias para que sea capaz de trabajar con ambos datasets), ya que su funcionamiento es eficiente. En cuanto a la corrección y detección de errores, decir que dichas operaciones serán revisadas en el Capítulo 6, como se ha comentado en el párrafo superior.

5.1.1 Entrenamiento de la red neuronal

En este subapartado se analizarán con mayor detalle todos los aspectos del entrenamiento de la red neural. En primer lugar, se hará una introducción al entrenamiento realizado, así como a los componentes que forman parte del mismo, detallando su funcionamiento en la medida de lo posible. Finalmente, se hará una descripción del mismo para los problemas de Thick Backbone y ZigZag Backbone. Además, se mencionará brevemente, la parte de test realizada de forma posterior a dicho entrenamiento.

5.1.1.1 Introducción y descripción del entrenamiento de la red

Dentro de las herramientas disponibles para el diseño y entrenamiento de ANNs, una de las más utilizadas es la biblioteca Keras (se trata de una biblioteca de redes neuronales open-source escrita en Python). Concretamente, la usamos para el desarrollo de modelos de Deep Learning (DL). En este proyecto, manejamos dos estructuras de DL para el reconocimiento de escritura manuscrita: el caso de Thick Backbone y el de ZigZag Backbone. Por tanto, el entrenamiento en ambos casos se llevará a cabo empleando Keras. Además, hay que destacar que tanto el entrenamiento como el test de ambos modelos, se realizaron en el entorno de programación gratuito de Google Colab (o Google Colaboratory), donde todos los procesos ejecutados durante dichas fases de entrenamiento y test, se llevaron a cabo en uno de los servidores disponibles de Google. A continuación, se explicará con detalle el desarrollo de cada modelo de DL (decir que estos modelos están recogidos en los archivos *CNN_ZigZag.ipynb* y *CNN_thickBackbone.ipynb*, de Google Colab):

Para empezar, lo primero que se necesita en un entorno de Google Colab, en el que desarrollaremos nuestro proyecto de DL, es activar la aceleración por GPU. Esto es así ya que estaremos trabajando con gran cantidad de datos de imagen, por lo tanto, sí, necesitamos mucha memoria. De esta manera, al activar la opción anterior en el entorno de Colab, Google nos proporciona una tarjeta gráfica de alta calidad para nuestro proyecto. En la Figura 5-2 se pueden observar las especificaciones de la tarjeta proporcionada, que como vemos es una NVIDIA-SMI.

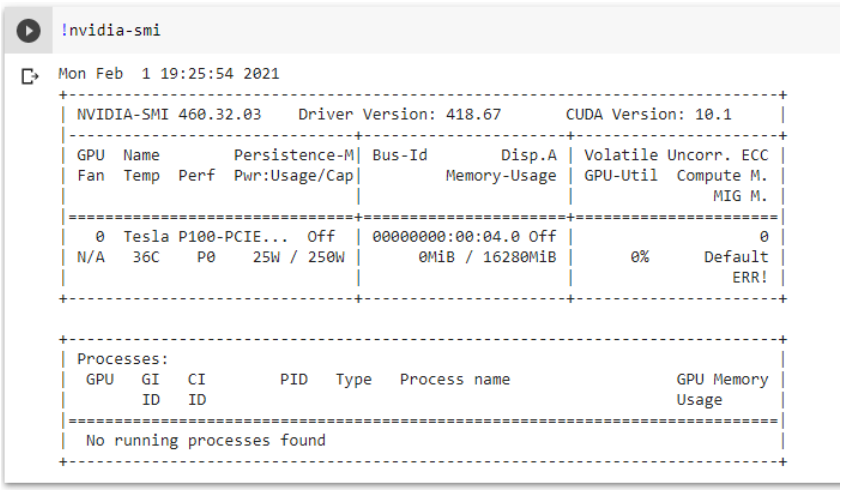
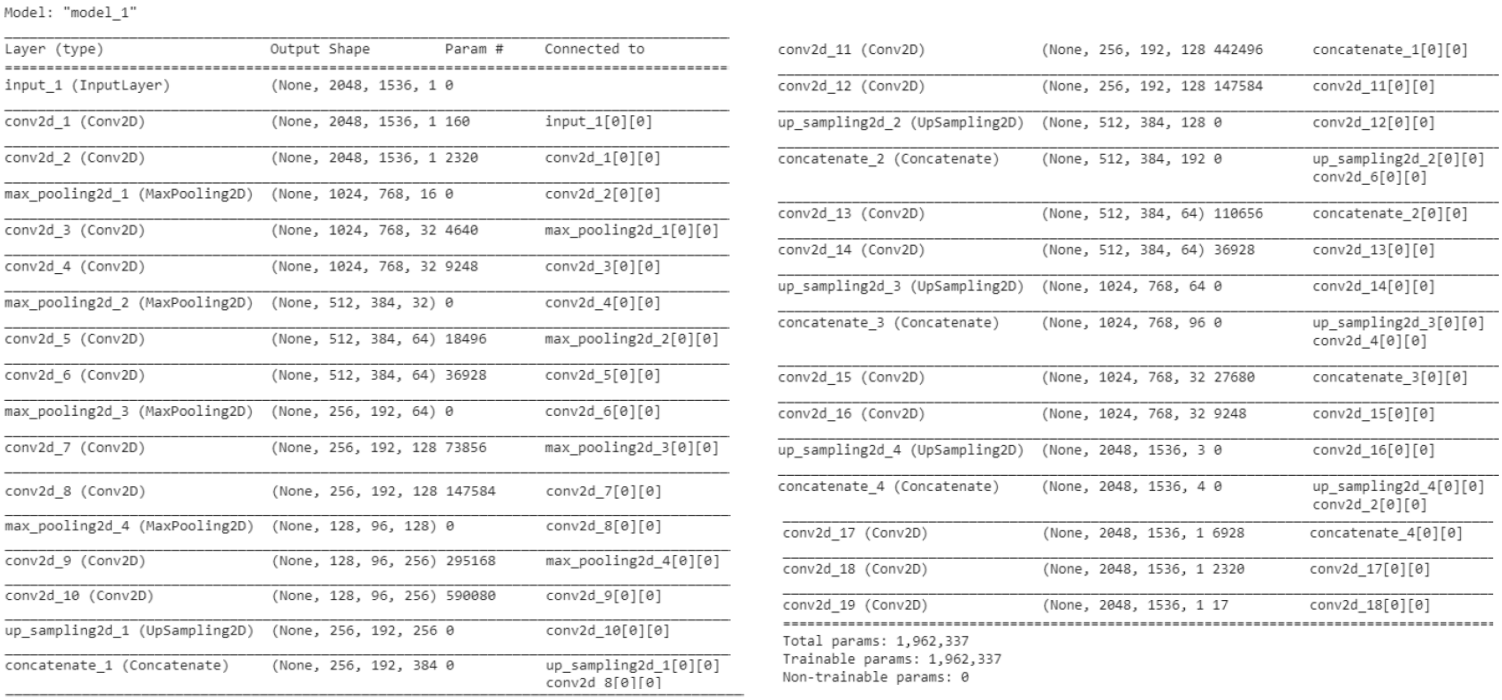


Figura 5-2. Especificaciones de la tarjeta gráfica Nvidia, proporcionada por Google Colaboratory

Como los datos con los que trabajamos en nuestro proyecto se almacenan en Google Drive, el siguiente paso a realizar debe ser el de importar y montar la unidad de Drive. Seguidamente, definimos una serie de rutas necesarias o paths, de manera que sea más sencillo trabajar con el proyecto en el caso de que cambiaríamos de cuenta de Drive o de ordenador. A continuación, se importan todos los ficheros necesarios para el funcionamiento del proyecto, además de realizar una comprobación con la función *makeDirIfNotExist* (definida en el script *groundTruthFunctions.py*), con la que se comprueba si los directorios referenciados en las rutas definidas previamente existen o no. Si no existen, se crean, así no surgen errores derivados de este problema.

Con todo lo anterior dispuesto, se puede comenzar a configurar los parámetros necesarios para realizar el entrenamiento y posterior test. Primero que todo, se selecciona el tipo de dataset a utilizar, mediante el ya conocido parámetro *DataBaseType* (lo ponemos a "0", si queremos usar el set de DIVA-HisDB, o "1", si por el contrario, se quiere usar el set de ICDAR 2013). Dependiendo de ello tendremos unos valores distintos para parámetros como: el índice de entrenamiento y el de test, las dimensiones permitidas de las imágenes de entrada a la red o el tamaño del conjunto de validación.

Una vez configurados los parámetros anteriores, se procede a crear el modelo de la red neuronal elegida. Para nuestro proyecto, como se ha comentado en apartados anteriores, se ha definido una *red tipo U*. Esta red está formada por una secuencia de capas cuya silueta tiene la forma de una U, de ahí su nombre. La red propuesta se recoge en el archivo *Unet_2048.py*, que podemos encontrar entre los ficheros del proyecto (para más información, ver Anexos A y B). Al crear el modelo de la red, este recibe como parámetros las dimensiones de las imágenes a tratar (dimensiones fijas, configuradas anteriormente), divididas entre 2. Esto es así, porque en la red existen capas de Max-pooling, que submuestrean la representación de la entrada de la red tomando el valor máximo sobre la ventana (o recuadro) definida por el parámetro *pool_size* (para cada dimensión de la entrada). Este parámetro es de 2, tanto para el alto como para el ancho de la imagen. Además la ventana anterior se desplaza en pasos o *strides* de 2 en cada dimensión. De esta manera, tendremos que la red será distinta dependiendo del dataset empleado. Si la base de datos utilizada es ICDAR 2013 (*DataBaseType* a "1"), el modelo de U-Net empleado tendrá como dimensiones *2048 x 1536*, de modo que su estructura sería la mostrada en la Figura 5-3:



```
Model: "model_1"
__________________________________________________________________________________________________
Layer (type)                    Output Shape         Param #     Connected to
==================================================================================================
input_1 (InputLayer)            (None, 2048, 1536, 1 0
__________________________________________________________________________________________________
conv2d_1 (Conv2D)               (None, 2048, 1536, 1 160         input_1[0][0]
__________________________________________________________________________________________________
conv2d_2 (Conv2D)               (None, 2048, 1536, 1 2320        conv2d_1[0][0]
__________________________________________________________________________________________________
max_pooling2d_1 (MaxPooling2D)  (None, 1024, 768, 16 0           conv2d_2[0][0]
__________________________________________________________________________________________________
conv2d_3 (Conv2D)               (None, 1024, 768, 32 4640        max_pooling2d_1[0][0]
__________________________________________________________________________________________________
conv2d_4 (Conv2D)               (None, 1024, 768, 32 9248        conv2d_3[0][0]
__________________________________________________________________________________________________
max_pooling2d_2 (MaxPooling2D)  (None, 512, 384, 32) 0           conv2d_4[0][0]
__________________________________________________________________________________________________
conv2d_5 (Conv2D)               (None, 512, 384, 64) 18496       max_pooling2d_2[0][0]
__________________________________________________________________________________________________
conv2d_6 (Conv2D)               (None, 512, 384, 64) 36928       conv2d_5[0][0]
__________________________________________________________________________________________________
max_pooling2d_3 (MaxPooling2D)  (None, 256, 192, 64) 0           conv2d_6[0][0]
__________________________________________________________________________________________________
conv2d_7 (Conv2D)               (None, 256, 192, 128 73856       max_pooling2d_3[0][0]
__________________________________________________________________________________________________
conv2d_8 (Conv2D)               (None, 256, 192, 128 147584      conv2d_7[0][0]
__________________________________________________________________________________________________
max_pooling2d_4 (MaxPooling2D)  (None, 128, 96, 128) 0           conv2d_8[0][0]
__________________________________________________________________________________________________
conv2d_9 (Conv2D)               (None, 128, 96, 256) 295168      max_pooling2d_4[0][0]
__________________________________________________________________________________________________
conv2d_10 (Conv2D)              (None, 128, 96, 256) 590080      conv2d_9[0][0]
__________________________________________________________________________________________________
up_sampling2d_1 (UpSampling2D)  (None, 256, 192, 256 0           conv2d_10[0][0]
__________________________________________________________________________________________________
concatenate_1 (Concatenate)     (None, 256, 192, 384 0           up_sampling2d_1[0][0]
                                                                 conv2d_8[0][0]
__________________________________________________________________________________________________
conv2d_11 (Conv2D)              (None, 256, 192, 128 442496      concatenate_1[0][0]
__________________________________________________________________________________________________
conv2d_12 (Conv2D)              (None, 256, 192, 128 147584      conv2d_11[0][0]
__________________________________________________________________________________________________
up_sampling2d_2 (UpSampling2D)  (None, 512, 384, 128 0           conv2d_12[0][0]
__________________________________________________________________________________________________
concatenate_2 (Concatenate)     (None, 512, 384, 192 0           up_sampling2d_2[0][0]
                                                                 conv2d_6[0][0]
__________________________________________________________________________________________________
conv2d_13 (Conv2D)              (None, 512, 384, 64) 110656      concatenate_2[0][0]
__________________________________________________________________________________________________
conv2d_14 (Conv2D)              (None, 512, 384, 64) 36928       conv2d_13[0][0]
__________________________________________________________________________________________________
up_sampling2d_3 (UpSampling2D)  (None, 1024, 768, 64 0           conv2d_14[0][0]
__________________________________________________________________________________________________
concatenate_3 (Concatenate)     (None, 1024, 768, 96 0           up_sampling2d_3[0][0]
                                                                 conv2d_4[0][0]
__________________________________________________________________________________________________
conv2d_15 (Conv2D)              (None, 1024, 768, 32 27680       concatenate_3[0][0]
__________________________________________________________________________________________________
conv2d_16 (Conv2D)              (None, 1024, 768, 32 9248        conv2d_15[0][0]
__________________________________________________________________________________________________
up_sampling2d_4 (UpSampling2D)  (None, 2048, 1536, 3 0           conv2d_16[0][0]
__________________________________________________________________________________________________
concatenate_4 (Concatenate)     (None, 2048, 1536, 4 0           up_sampling2d_4[0][0]
                                                                 conv2d_2[0][0]
__________________________________________________________________________________________________
conv2d_17 (Conv2D)              (None, 2048, 1536, 1 6928        concatenate_4[0][0]
__________________________________________________________________________________________________
conv2d_18 (Conv2D)              (None, 2048, 1536, 1 2320        conv2d_17[0][0]
__________________________________________________________________________________________________
conv2d_19 (Conv2D)              (None, 2048, 1536, 1 17          conv2d_18[0][0]
==================================================================================================
Total params: 1,962,337
Trainable params: 1,962,337
Non-trainable params: 0
```

Figura 5-3. Estructura de la red para la base de datos ICDAR 2013. Resultado de *model.summary()*

Si por el contrario la base de datos utilizada es DIVA-HisDB (*DataBaseType* a "0"), el modelo de U-Net empleado tendrá como dimensiones *2560 x 1792*, de forma que su estructura sería la que se observa en la Figura 5-4:

Model: "model_1"

Layer (type)	Output Shape	Param #	Connected to
input_2 (InputLayer)	[(None, 2560, 1792,	0	
conv2d_19 (Conv2D)	(None, 2560, 1792, 1	160	input_2[0][0]
conv2d_20 (Conv2D)	(None, 2560, 1792, 1	2320	conv2d_19[0][0]
max_pooling2d_4 (MaxPooling2D)	(None, 1280, 896, 16	0	conv2d_20[0][0]
conv2d_21 (Conv2D)	(None, 1280, 896, 32	4640	max_pooling2d_4[0][0]
conv2d_22 (Conv2D)	(None, 1280, 896, 32	9248	conv2d_21[0][0]
max_pooling2d_5 (MaxPooling2D)	(None, 640, 448, 32)	0	conv2d_22[0][0]
conv2d_23 (Conv2D)	(None, 640, 448, 64)	18496	max_pooling2d_5[0][0]
conv2d_24 (Conv2D)	(None, 640, 448, 64)	36928	conv2d_23[0][0]
max_pooling2d_6 (MaxPooling2D)	(None, 320, 224, 64)	0	conv2d_24[0][0]
conv2d_25 (Conv2D)	(None, 320, 224, 128	73856	max_pooling2d_6[0][0]
conv2d_26 (Conv2D)	(None, 320, 224, 128	147584	conv2d_25[0][0]
max_pooling2d_7 (MaxPooling2D)	(None, 160, 112, 128	0	conv2d_26[0][0]
conv2d_27 (Conv2D)	(None, 160, 112, 256	295168	max_pooling2d_7[0][0]
conv2d_28 (Conv2D)	(None, 160, 112, 256	590080	conv2d_27[0][0]
up_sampling2d_4 (UpSampling2D)	(None, 320, 224, 256	0	conv2d_28[0][0]
concatenate_4 (Concatenate)	(None, 320, 224, 384	0	up_sampling2d_4[0][0] conv2d_26[0][0]
conv2d_29 (Conv2D)	(None, 320, 224, 128	442496	concatenate_4[0][0]
conv2d_30 (Conv2D)	(None, 320, 224, 128	147584	conv2d_29[0][0]
up_sampling2d_5 (UpSampling2D)	(None, 640, 448, 128	0	conv2d_30[0][0]
concatenate_5 (Concatenate)	(None, 640, 448, 192	0	up_sampling2d_5[0][0] conv2d_24[0][0]
conv2d_31 (Conv2D)	(None, 640, 448, 64)	110656	concatenate_5[0][0]
conv2d_32 (Conv2D)	(None, 640, 448, 64)	36928	conv2d_31[0][0]
up_sampling2d_6 (UpSampling2D)	(None, 1280, 896, 64	0	conv2d_32[0][0]
concatenate_6 (Concatenate)	(None, 1280, 896, 96	0	up_sampling2d_6[0][0] conv2d_22[0][0]
conv2d_33 (Conv2D)	(None, 1280, 896, 32	27680	concatenate_6[0][0]
conv2d_34 (Conv2D)	(None, 1280, 896, 32	9248	conv2d_33[0][0]
up_sampling2d_7 (UpSampling2D)	(None, 2560, 1792, 3	0	conv2d_34[0][0]
concatenate_7 (Concatenate)	(None, 2560, 1792, 4	0	up_sampling2d_7[0][0] conv2d_20[0][0]
conv2d_35 (Conv2D)	(None, 2560, 1792, 1	6928	concatenate_7[0][0]
conv2d_36 (Conv2D)	(None, 2560, 1792, 1	2320	conv2d_35[0][0]
conv2d_37 (Conv2D)	(None, 2560, 1792, 1	17	conv2d_36[0][0]

Total params: 1,962,337
Trainable params: 1,962,337
Non-trainable params: 0

Figura 5-4. Estructura de la red para la base de datos DIVA-HisDB. Resultado de *model.summary()*

Después de crear el modelo con la U-Net definida (Figura 5-3 y Figura 5-4), se compila. La compilación exige que se especifiquen diversos parámetros, adaptados particularmente a la formación de la red. En este caso, se contemplan tres parámetros a configurar:

- *optimizer*: representa al algoritmo encargado de la optimización que se emplea para entrenar a la red. Aquí se usa el optimizador que implementa el algoritmo Adam. Este tipo de optimización se basa en un método del gradiente estocástico descendiente, que se fundamenta a su vez en la estimación adaptativa de los momentos de primer y segundo orden.
- *loss*: representa a la función de pérdidas empleada en la evaluación de la red, que es minimizada por el algoritmo de optimización que se describía en el párrafo anterior. Aquí empleamos como función de pérdidas a la entropía cruzada binaria.
- *metrics*: representa a las métricas que se desean recoger al ajustar el modelo (cuando hacemos el paso de *fit*). Normalmente, la métrica más útil es la de precisión o *acc*, que es precisamente la que se recoge en este proyecto (las métricas que deseamos recoger se especifican con su nombre en una matriz).

Una vez compilado el modelo, se especifican otros parámetros como las épocas, el tamaño del batch, las listas de training y validación.

Es importante señalar que, a diferencia de otros entrenamientos que se suelen basar en cargar el dataset completo en la memoria disponible para así, a continuación, entrenar la red, el nuestro se centra en un entrenamiento on-line, o entrenamiento por mini-batches, ya que manejamos datos bastante pesados siendo así la forma convencional un poco agresiva para nuestro proyecto. No obstante, para poder utilizar este tipo de entrenamiento, es necesario generar un fichero que contenga la clase *DataGen.* El que manejamos en el proyecto se llama *DataGen_2048_dataAug_rotated.py,* y a continuación se describirán todos sus detalles:

- Lo primero que encontramos al leer este fichero, es la definición de la clase *DataGen.* Esta clase hereda de la clase *keras.utils.Sequence*, y además implementa 3 métodos de forma obligatoria: *__init__*, que se encarga de inicializar los parámetros definidos; *__len__*, que denota el número de batches por época; y *__getitem__*, que se encarga de generar un batch de datos. A este método se

recurrirá con frecuencia en cada paso del entrenamiento realizado, por lo que es en él donde se han programado todas las instrucciones para la generación, de una forma u otra, de los datos del mismo.

- En lo que se refiere al método *__init__*, destacar que los parámetros que inicializa, a valores por defecto, son los siguientes:
    - *list_image_numbers*: representa a lista de todos los identificadores de "etiquetas" que se utilizarán en el generador (tanto los de la lista de entrenamiento como los de test).
    - *x_path* e *y_path*: ruta a las variables x e y, respectivamente.
    - *batch_size*: tamaño del batch en cada iteración.
    - *num_channels*: número de los canales de la imagen.
    - *shuffle*: sirve para barajar los índices de las etiquetas después de cada época.
    - *to_fit*: sirve para devolver x e y ajustadas (True), o para devolver sólo la predicción de x (False)
    - *zig_zag*: "True" si vamos a devolver el contorno zig_zag, "False" si sólo el contorno regular. Este parámetro solo se activa (True), si queremos emplearlo con el código de [3], el cual contempla el entrenamiento con dos modelos de red. No lo activaremos para nuestro proyecto.
    - *half_outline*: sirve para, si hemos activado el parámetro anterior (*zig_zag*), obtener la mitad superior del contorno (tomando el valor 1) u obtener la mitad inferior (tomando el valor -1). Si *half_outline* toma el valor 0, no produce ningún cambio en el proyecto.
    - *zig_zag_rotated*: básicamente tiene la misma función que *zig_zag*, pero con la diferencia que al activarlo estamos indicando que tan solo se trabajará con una red, y que las imágenes serán rotadas para que se puedan extraer ambas mitades del contorno, sin necesidad de entrenar dos modelos de red. Este parámetro estará activado en nuestro proyecto.
    - *data_augmentation*: cuando toma el valor "True", se devuelven 3 imágenes en lugar de sólo 1, siendo las otras 2 la rotación de 5 grados del original.
    - *debugging*: al tomar el valor "True", muestra los mensajes de depuración.
    - *IMAGES_HEIGHT* e *IMAGES_WIDTH*: representan la altura y anchura de los tensores a tratar, respectivamente.
- Por otra parte, como se ha mencionado antes, es en el método *__getitem__* donde se llevan a cabo las operaciones que definirán el tipo de entrenamiento seleccionado. Entre dichas operaciones, podemos destacar:
    - *Data augmentation:* Una de las operaciones que más llama la atención al recorrer *__getitem__,* es la de data augmentation. Dicha operación aparece por primera vez en la función *__load_batch* (definida en este método), encargada de generar un vector que contenga imágenes y etiquetas de un batch. También aparece, y se desarrolla, en la función *load_single_image*, que genera una imagen si to_fit = "False", y una imagen y su etiqueta en caso contrario. Esta operación está compuesta a su vez por una serie de operaciones que hacen posible multiplicar el número de muestras con las que alimentamos a la red neuronal seleccionada. En nuestro caso las muestras son imágenes, por tanto, para aumentarlas en número, es posible utilizar una gran variedad de técnicas (rotación, darle la vuelta en el eje horizontal o vertical, meterle ruido sal y pimienta, aclarar u oscurecer la imagen, etc…). No obstante, todas las técnicas no son válidas. Es necesario elegir el tipo de data augmentation, de acuerdo con la tarea que vayamos a realizar. Como ya se discutía en [3], la más apropiada es la de rotación: se triplica el dataset de entrenamiento y se termina rotando el ground-truth de cada imagen unos 5 grados en sentido anti horario (aunque también se podría haber elegido rotar la imagen en el sentido horario). Para esta última operación se emplea la función *rotate*, perteneciente a la librería PILLOW de Python. Todo este proceso se lleva a cabo, como ya se adelantaba previamente, en la función *load_single_image*.

- *ZigZag con dos redes*: Esta operación se rescata del código de [3], y aunque no la contemplemos en nuestro proyecto, ya que no trabajamos con ella, es interesante describirla. Dicha operación se activa con el parámetro *zig_zag* al tomar el valor "True", y se encarga de extraer la parte inferior o superior del contorno de la línea (para el problema de ZigZag Backbone), dependiendo del valor que tome *half_outline* (tomará un valor distinto al crear el set de entrenamiento, uno para cada red). El zigzag con dos redes, se puede encontrar en la función *load_single_image*, del método *__getitem__*.
- *ZigZag rotado con una red*: Por otra parte, destacar que esta operación es la que forma parte de la generación de datos para el entrenamiento y test del algoritmo propuesto. Se basa en la operación anterior, con la salvedad de que está programada para que haga la misma función que la anterior pero solo con una red. Para ello, se decide de forma aleatoria si para la imagen tratada se va a rotar su etiqueta o no. Si resulta que la etiqueta no se ha rotado, solo se extrae la parte superior de las líneas de la misma. Pero si ocurre lo contrario, extraemos la parte inferior de la etiqueta, la rotamos y para finalizar, también rotamos la imagen. Esto se hace así porque la red que entrenamos solo acepta las partes superiores de las líneas de la imagen. El zigzag rotado con una red se puede encontrar en la función *load_single_image*, del método *__getitem__*.
- *Downsampling* o *submuestreo*: Finalmente, tenemos la operación de submuestreo. Esta operación fue planteada en [3], para resolver la falta de memoria, derivada del pesado entrenamiento por el que debe pasar nuestro algoritmo, y los límites físicos que existen (demasiadas operaciones para un ordenador convencional). La idea es que, como para sacar el contorno de las líneas de texto no se necesita una calidad de imagen elevada, se decide submuestrear las imágenes del dataset empleado, de manera que se reduzca la calidad de las mismas. Para ello, se emplea el método *block_reduce* perteneciente a *skimage.measure.* Empleando dicho método, se consigue muestrear las imágenes unas tres veces, usando un factor (bloque o recuadro) de 2x2 (alto x ancho) en todas ellas. Además, según se apunta en [3], se empleó una combinación bastante interesante para definir el criterio por el cual se determina qué pixel es el que nos quedamos del bloque 2x2, cada vez que se realiza el submuestreo. Esta combinación se basa en coger el máximo valor del bloque en el primer y tercer submuestreo (con la función *np.max*), para en el segundo elegir el valor mínimo (con la función *np.min*). De este modo, las letras no quedan emborronadas (cosa que pararía si se seleccionara el máximo valor en los tres submuestreos), ni tampoco se hacen demasiado finas (en el caso de que en los tres submuestreos se escogiera el valor mínimo). Esta operación se puede encontrar en la función *load_single_image*, del método *__getitem__*.

Al describir el contenido del script *DataGen_2048_dataAug_rotated.py,* ya podemos generar los datos con los que trabajará el algoritmo propuesto. En concreto, empezaremos generando los datos de training y validación (en el caso de estar ante el problema de ZigZag Backbone, el parámetro *zig_zag_rotated* tomará el valor "True", si no, tomará el valor "False"). Se debe señalar que, para el caso de los datos de training, la opción de *data_augmentation* estará activada, mientras que para los datos de validación no. Después de generar los dos tipos de datos anteriores, nos encontramos con la configuración de la técnica de *"Early stopping"*. Dicha técnica se encarga de comprobar si los valores obtenidos en el entrenamiento de la época anterior, son o no mejores que los obtenidos en la época actual. Si es así, se para el entrenamiento ya que estaremos en el punto más óptimo del mismo. De esta manera, configurando sus parámetros (*monitor*='val_acc', *mode*='max' y *patience*=3), y cuando la precisión en el grupo de validación sea máxima, se para el entrenamiento y se obtiene el modelo tres épocas después del mejor de ellos (ya que se ha configurado el parámetro patience = 3, que indica el número de épocas sin mejora tras las cuales se detendrá el entrenamiento). Esto es un problema, ya que no obtenemos el mejor modelo. Por tanto, para solventar esta situación, creamos un *checkpoint* con el que guardamos los mejores pesos calculados.

Después de realizar todos estos pasos previos y configuraciones, pasamos a ajustar la red, es decir, se adaptan los pesos anteriores al conjunto de datos de entrenamiento y validación. Así pues, el ajuste del modelo requiere que se especifiquen una serie de parámetros de formación: los datos de entrenamiento y validación (*train_gen*

y *valid_gen*); los pasos por época, tanto de entrenamiento como de validación (*train_steps* y *valid_steps*); las épocas (*epochs);* la opción de salida de texto (*verbose)* y por último la lista de callbacks, compuesta por *earlyStoppingCallback* y*, chkpoint* (*callbackList*) . Al terminar el entrenamiento o el ajuste de la red, se obtiene un objeto de tipo historial, que recoge un resumen del rendimiento presentando por el modelo durante el entrenamiento. Guardamos el historial, para no tener que volver a entrenar el modelo (este proceso podría tardar decenas de minutos; depende del tamaño de la red y del volumen de datos que maneje).
El paso siguiente consistirá en evaluar la red sobre dataset separado (los datos de test), es decir, el no empleado durante el entrenamiento. Así, se obtiene una estimación del rendimiento de la red neuronal. Además, al igual que en el caso del ajuste de la red, la evaluación de la misma requiere la especificación de una serie de parámetros: los datos de entrada (*test_gen);* el número de pasos hasta que se completa la evaluación (*test_steps);* y la salida de texto que nos muestra la barra de progreso de la evaluación (verbose). Finalmente, se comprueba visualmente si los resultados de la evaluación del modelo son los que se esperaban. Una vez llegados a este punto, pasamos a extraer y comentar los resultados obtenidos tanto en el entrenamiento del modelo como en el test.

5.2 Resultados de los entrenamientos y test realizados

Como se adelantaba antes, en este apartado comenzaremos exponiendo los resultados obtenidos del proceso de pre-segmentación, tanto durante el entrenamiento del modelo, como en la evaluación del mismo. Por supuesto, esta tarea se llevará a cabo para los dos datasets empleados en el proyecto (ICDAR 2013 y DIVA-HisDB), además de para cada uno de los problemas tratados en el mismo (Thick Backbone y ZigZag Backbone).

Las métricas empleadas, para ambos problemas y datasets, en el entrenamiento del modelo, así como en la evaluación del mismo (solo las dos primeras), son las siguientes:

- *Loss*: representa, por una parte, a la pérdida de entrenamiento, que se calcula haciendo la media de las pérdidas de cada batch de datos de entrenamiento (es el valor de la función de coste para el set de entrenamiento). El modelo cambia con el tiempo, por lo que la *loss* suele ser un poco mayor en las primeras épocas de un dataset que en las últimas.

  Por otra parte, representa también la pérdida en el caso de la evaluación del modelo. En este caso, en el que solo se tiene una época, la *loss* se calcula usando el modelo tal y como es al final de la época, lo que normalmente resulta en una pérdida de test un poco más pequeña.

  La función de pérdidas más utilizada en las redes neuronales profundas, es la entropía cruzada binaria [30], cuya expresión se define a continuación:

$$Cross\ entropy = -\sum_{i=1}^{n}\sum_{j=1}^{m} y_{i,j}\log\left(p_{i,j}\right) \qquad (5-1)$$

  donde $p_{i,j}$ denota la probabilidad, predicha por el modelo, de que la muestra i pertenezca a la clase j [30], mientras que $y_{i,j}$ denota el valor verdadero de la muestra (no estimado por nuestro modelo), i.e., 1 si la muestra i pertenece a la clase j y 0 en caso contrario.

- *Accuracy:* este parámetro representa la precisión media del entrenamiento o test al final de una época. Viene definida por la siguiente expresión [30]:

$$Accuracy = \frac{Number\ of\ correct\ predictions}{Total\ number\ of\ predictions} \qquad (5-2)$$

- *Validation loss:* representa el valor de la función de coste (calculada como se hace con *loss*) para los datos de validación cruzada

- *Validation accuracy*: representa la precisión (calculada como se hace con accuracy) del conjunto de validación al final de una época. Como se ha visto en el apartado anterior, el entrenamiento se debería parar cuando este parámetro deje de aumentar, es decir, cuando llegue a su máximo valor. Sino, el

modelo entrenado podría estar sobremuestreado. Por ello usamos el callback de *early stopping* en el ajuste del modelo.

Destacar, que las métricas anteriores son calculadas de forma automática por los métodos disponibles en la biblioteca Keras [31].

5.2.1 Resultados de los entrenamientos y test con Thick Backbone

Aquí haremos una revisión de los resultados de entrenamiento y test obtenidos para el problema Thick Backbone.

5.2.1.1 Empleando la base de datos ICDAR 2013

En primer lugar, realizamos 10 entrenamientos del modelo y 10 evaluaciones del mismo con el dataset ICDAR 2013. De esta manera, se tiene una visión más completa del entrenamiento y evaluación de la red neuronal propuesta. Así, en la Tabla 5-1, observamos los valores de *loss, accuracy*, *val loss* y *val accuracy* para cada entrenamiento realizado:

Tabla 5–1. Resultados de los 10 entrenamientos realizados para el problema Thick Backbone, con el conjunto de entrenamiento de la base de datos ICDAR 2013 (los valores de *accuracy* y *val accuracy* están en tanto por uno)

Nº entrenamiento	**Loss**	**Accuracy**	**Val loss**	**Val accuracy**
1	0.0463	0.9803	0.0550	0.9766
2	0.0467	0.9800	0.0549	0.9766
3	0.0431	0.9814	0.0520	0.9781
4	0.0421	0.9819	0.0533	0.9782
5	0.0440	0.9812	0.0519	0.9780
6	0.0458	0.9805	0.0541	0.9768
7	0.0403	0.9826	0.0548	0.9779
8	0.0457	0.9807	0.0546	0.9776
9	0.0446	0.9808	0.0534	0.9777
10	0.0471	0.9799	0.0537	0.9774

Si se observa la tabla anterior (Tabla 5-1), se puede apreciar que uno de los mejores entrenamientos realizados ha sido el número 7. Este presenta el mayor valor del parámetro *accuracy* (98.26%), y el menor valor del parámetro *loss* (0.0403) para el set de entrenamiento. Sin embargo, si nos fijamos en los parámetros pertenecientes al set de validación, podemos afirmar que no es el entrenamiento 7 el que más se destaca, aunque sus cifras no son para nada malas. Como se muestra claramente en la Tabla 5-1, es el cuarto entrenamiento el que sobresale por tener el mayor valor del parámetro de *validation accuracy* (97.82 %), mientras que es el quinto el que lo hace por haber conseguido el menor valor del parámetro de *validation loss* (0.0519). Por otra parte, el peor entrenamiento es el décimo, tanto para el parámetro *loss* (0.0471) como para el parámetro *accuracy* (97.99%) del conjunto de entrenamiento. Para los otros dos parámetros, pertenecientes al set de validación, las peores cifras se alcanzan en el primer entrenamiento para *validation loss* (0.0550), y en el segundo para *validation accuracy* (97.66%).

Una vez completado ese número de entrenamientos, se calcula el valor máximo, mínimo y medio de estos. Así pues, se puede hacer un análisis más completo del entrenamiento del modelo. Dichos resultados se muestran en la Tabla 5-2 que representamos a continuación:

Tabla 5–2. Valores máximo, mínimo y medio de los entrenamientos realizados para el problema Thick Backbone, con el conjunto de entrenamiento de la base de datos ICDAR 2013 (los valores de *accuracy* y *val accuracy* están en tanto por uno)

Valores	Loss	Accuracy	Val loss	Val accuracy
Máximo	0.0471	0.9826	0.0550	0.9782
Medio	0.0446	0.9809	0.0538	0.9775
Mínimo	0.0403	0.9799	0.0519	0.9766

Como se puede apreciar en la tabla superior, la Tabla 5-2, se muestran los valores máximos, mínimos y medios de los cuatro parámetros empleados durante el entrenamiento. Al analizar la tabla anterior, Tabla 5-1, se señalaban los entrenamientos con mejores resultados, y bien, si nos fijamos en esta tabla, sus valores combinados darían los mejores resultados del entrenamiento del modelo presentado. Lo mismo ocurre con los peores resultados que describíamos previamente. De esta manera:

- El mejor entrenamiento conseguido tendría los siguientes valores para cada uno los parámetros que lo definen: una *loss* de 0.0403, una *accuracy* del 98.26%, una *validation loss* de 0.0519 y una *validation accuracy* del 97.82%.
- Por otra parte, el peor entrenamiento vendría definido por: una *loss* de 0.0471, una *accuracy* del 97.99%, una *validation loss* de 0.0550 y una *validation accuracy* del 97.66%. Aunque tampoco son unos valores malos, al compararlos con el valor medio de entrenamientos.

No obstante, para evaluar el rendimiento del modelo en su fase de entrenamiento de una forma un poco más equitativa, debemos poner la atención en el valor medio obtenido para cada parámetro. Así pues, se puede decir que se tiene una precisión bastante alta, del 98.09%, con una pérdida del 0.0446, que no está nada mal. Por tanto, vemos que nuestro modelo se comporta bastante bien durante el entrenamiento del mismo, mejor con el conjunto de entrenamiento que con el de validación, aunque la diferencia es cuestión de una décima.

De la misma manera, en la Tabla 5-3 se exponen los resultados de las 10 evaluaciones del modelo, pero en este caso solo para los parámetros *loss* y *accuracy*:

Tabla 5–3. Resultados de las 10 evaluaciones realizadas para el problema Thick Backbone, con el conjunto de test de la base de datos ICDAR 2013 (el valor de *accuracy* está en tanto por uno)

Nº evaluación	Loss	Accuracy
1	0.0610	0.9741
2	0.0601	0.9744
3	0.0590	0.9750
4	0.0612	0.9747
5	0.0592	0.9749
6	0.0618	0.9742
7	0.0613	0.9749
8	0.0621	0.9741
9	0.0602	0.9744
10	0.0608	0.9743

Al observar la Tabla 5-3, se puede percibir que una de las mejores evaluaciones para el conjunto de test ha sido la número 3. Y como se ve, alcanza los valores más óptimos tanto para el parámetro *loss* (0.0590), como para el de *accuracy* (97.50%). Por otra parte, la peor evaluación se alcanza en la octava, de nuevo, para ambos parámetros (*loss* de 0.0621 y *accuracy* de 97.41%).

Cuando se terminan dichas evaluaciones, al igual que para el caso del entrenamiento, se calcula el valor

máximo, mínimo y medio de las mismas, para posteriormente analizar el proceso realizado. En la Tabla 5-4 se exponen los valores conseguidos:

Tabla 5–4. Valores máximo, mínimo y medio de las evaluaciones realizadas para el problema Thick Backbone, con el conjunto de test de la base de datos ICDAR 2013 (el valor de *accuracy* están en tanto por uno)

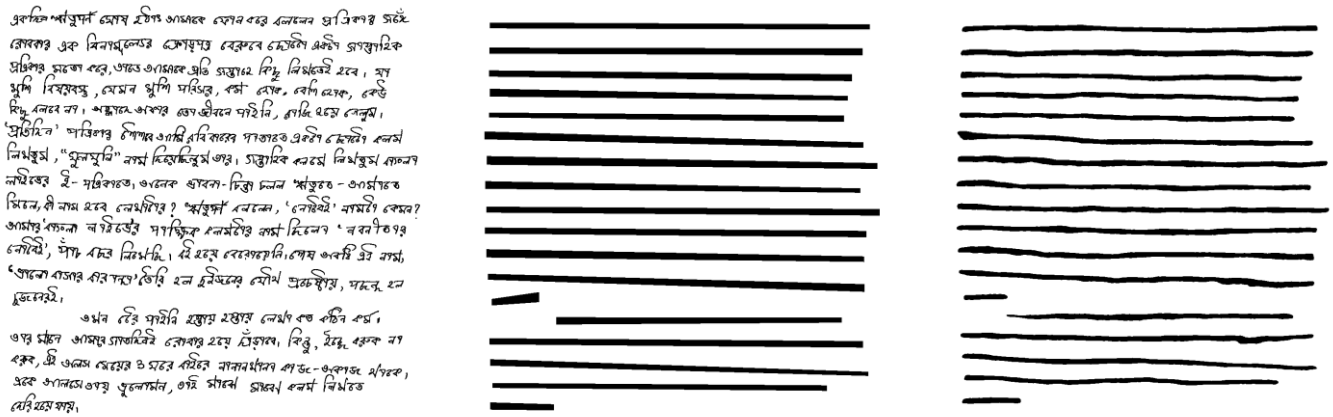
Valores	Loss	Accuracy
Máximo	0.0621	0.9750
Medio	0.0607	0.9745
Mínimo	0.0590	0.9741

Si nos fijamos en la tabla superior, la Tabla 5-4, vemos que en esta se representan los valores máximos, mínimos y medios de los cuatro parámetros empleados durante la evaluación del modelo. Como se podía apreciar en el análisis de la Tabla 5-3, en ella se hacía una mención a los mejores y peores resultados, que como podemos observar en la Tabla 5-4, son los recogidos en la misma. De esta manera:

- La mejor evaluación obtenida tendría los siguientes valores para cada uno los parámetros que la definen: una *loss* de 0.0590 y una *accuracy* del 97.50%. Mucho más bajos que las cifras conseguidas para el entrenamiento.
- Por otra parte, la peor evaluación del modelo vendría definida por: una *loss* de 0.0621 y una *accuracy* del 97.41%. Aunque los valores obtenidos en el peor caso no son tan malos comparados con los medios, siguen siendo un poco peores que la media de entrenamientos e incluso que los peores valores conseguidos en dichos entrenamientos.

No obstante, para medir el rendimiento del modelo a través de la evaluación del mismo, debemos centrarnos en el valor medio obtenido para cada parámetro. Así pues, se puede decir que se tiene una precisión bastante alta, del 97.45% (aunque un poco inferior a la conseguida para el entrenamiento), con una pérdida del 0.0607, que no está nada mal, pero sí que es casi dos décimas superior a la conseguida durante el entrenamiento del modelo. Por tanto podemos decir que nuestro modelo se comporta bastante bien con datos del conjunto de test, aunque no tan bien como con los del set de entrenamiento.

Finalmente, en la Figura 5-5, se muestra un ejemplo de la entrada y la salida del modelo al ser evaluado, así como una predicción del mismo:



(a) (b) (c)

Figura 5-5. (a) Imagen 313 de ICDAR2013, que representa la entrada de la red; (b) Thick Backbone obtenido de (a) conseguido a la salida de la red; (c) Predicción del Thick Backbone de (a).

5.2.1.2 Empleando la base de datos DIVA-HisDB

Al igual que para la base de datos ICDAR 2013, se realizan 10 entrenamientos y 10 evaluaciones del modelo propuesto con el dataset DIVA-HisDB. Así pues, se consigue apreciar mejor el funcionamiento del modelo aquí presentado. Además, hay que mencionar que tanto el entrenamiento como la evaluación del modelo, se han llevado a cabo sin el parámetro de *data_augmentation* activo. Esto se ha debido a problemas de falta de memoria en la GPU de Google Colab, ya que los tensores de esta base de datos pesan algo más que los de ICDAR 2013 al tener dimensiones mayores. Por tanto, en cada batch de imagen, en vez de generar 3 de ellas, solo se genera 1, consiguiendo así que no nos quedemos sin memoria.

De esta forma, en la Tabla 5-5, se pueden observar los valores de *loss, accuracy*, *val loss* y *val accuracy* para cada entrenamiento realizado:

Tabla 5–5. Resultados de los 10 entrenamientos realizados para el problema Thick Backbone, con el conjunto de entrenamiento de la base de datos DIVA-HisDB (los valores de *accuracy* y *val accuracy* están en tanto por uno)

Nº entrenamiento	Loss	Accuracy	Val loss	Val accuracy
1	0.0454	0.9819	0.0603	0.9750
2	0.0460	0.9817	0.0550	0.9783
3	0.0486	0.9806	0.0610	0.9757
4	0.0362	0.9850	0.0461	0.9816
5	0.0368	0.9848	0.0423	0.9830
6	0.0494	0.9802	0.0504	0.9800
7	0.0369	0.9848	0.0426	0.9827
8	0.0386	0.9843	0.0451	0.9822
9	0.0387	0.9845	0.0453	0.9821
10	0.0422	0.9827	0.0457	0.9819

Si se analiza la Tabla 5-5 expuesta, se puede apreciar que uno de los mejores entrenamientos realizados, para el set de entrenamiento, ha sido el número 4. Vemos que, sin duda, dicho entrenamiento presenta la cifra más elevada del parámetro *accuracy* (98.50%) y la más baja para el parámetro *loss* (0.0362). Por otra parte, al fijarnos en los parámetros pertenecientes al set de validación, podemos afirmar que es el quinto entrenamiento el que presenta el mayor valor para el parámetro de *validation accuracy* (98.30 %), así como el menor valor del parámetro de *validation loss* (0.0423). Del mismo modo, el peor entrenamiento, para el set de entrenamiento, que podemos encontrar al analizar la Tabla 5-5, es sin duda el sexto. Este ofrece el mayor valor del parámetro *loss* (0.0494) y el menor valor del parámetro *accuracy* (98.02%). Para los otros dos parámetros, pertenecientes al set de validación, las peores cifras se alcanzan en el tercer entrenamiento, para *validation loss* (0.0610), y en el primero para *validation accuracy* (97.50%).

Cuando se completa ese número de entrenamientos, pasamos a calcular el valor máximo, mínimo y medio de estos. De esta manera, se puede hacer un análisis más completo del entrenamiento del modelo. Dichos resultados se muestran en la Tabla 5-6 que mostramos a continuación:

Tabla 5–6. Valores máximo, mínimo y medio de los entrenamientos realizados para el problema Thick Backbone, con el conjunto de entrenamiento de la base de datos DIVA-HisDB (los valores de *accuracy* y *val accuracy* están en tanto por uno)

Valores	Loss	Accuracy	Val loss	Val accuracy
Máximo	0.0494	0.9850	0.0610	0.9830
Medio	0.0419	0.9831	0.0494	0.9803
Mínimo	0.0362	0.9802	0.0423	0.9750

Al observar la tabla superior, la Tabla 5-6, vemos que esta recoge los valores máximos, mínimos y medios de los cuatro parámetros empleados durante el entrenamiento. Cuando se analizaba la Tabla 5-5, se señalaban los entrenamientos con mejores y peores cifras. Por tanto, si en esta tabla combinamos esos valores que resaltábamos en la anterior, obtendríamos los mejores y peores resultados del entrenamiento del modelo presentado. Así pues:

- El mejor entrenamiento conseguido tendría los siguientes valores para cada uno los parámetros que lo definen: una *loss* de 0.0362, una *accuracy* del 98.50%, una *validation loss* de 0.0423 y una *validation accuracy* del 98.30%.
- Por otra parte, el peor entrenamiento vendría definido por: una *loss* de 0.0494, una *accuracy* del 98.02%, una *validation loss* de 0.0610 y una *validation accuracy* del 97.50%.

No obstante, para evaluar el rendimiento del modelo en su fase de entrenamiento, de una forma un poco más general, debemos fijarnos en el valor medio obtenido para cada parámetro. Por ello, se puede decir que se consigue una precisión bastante alta, del 98.31%, con una pérdida del 0.0419. De ese modo, vemos que nuestro modelo se comporta muy bien durante su entrenamiento. Con lo que se podría decir que funciona un poco mejor con el conjunto de entrenamiento que con el de validación, aunque la diferencia es cuestión de milésimas (por lo que se podría concluir que no hay *overfitting*).

Por otra parte, en la tabla siguiente, Tabla 5-7 se exponen los resultados de las 10 evaluaciones del modelo, pero en este caso solo para los parámetros *loss* y *accuracy*:

Tabla 5–7. Resultados de las 10 evaluaciones realizadas para el problema Thick Backbone, con el conjunto de test de la base de datos DIVA-HisDB (el valor de *accuracy* está en tanto por uno)

Nº evaluación	Loss	Accuracy
1	0.0478	0.9804
2	0.0441	0.9819
3	0.0477	0.9811
4	0.0423	0.9830
5	0.0388	0.9843
6	0.0448	0.9820
7	0.0389	0.9840
8	0.0379	0.9845
9	0.0372	0.9846
10	0.0395	0.9837

Una vez recogidas todas las evaluaciones realizadas en la Tabla 5-7, se puede apreciar que la mejor de ellas, para el conjunto de test, por supuesto, es la novena. En dicha evaluación se observa como conseguimos los valores más óptimos, tanto para el parámetro *loss* (0.0372), como para el de *accuracy* (98.46%). Por otra parte, las peores cifras se alcanzan en la primera evaluación, de nuevo, para ambos parámetros: *loss* de 0.0478 y *accuracy* de 98.04%.

Seguidamente, al igual que para el caso del entrenamiento, se calcula el valor máximo, mínimo y medio de las evaluaciones expuestas en la Tabla 5-7, para posteriormente analizar el proceso realizado. En la Tabla 5-8 se muestran los valores conseguidos:

Tabla 5–8. Valores máximo, mínimo y medio de las evaluaciones realizadas para el problema Thick Backbone, con el conjunto de test de la base de datos DIVA-HisDB (el valor de *accuracy* están en tanto por uno)

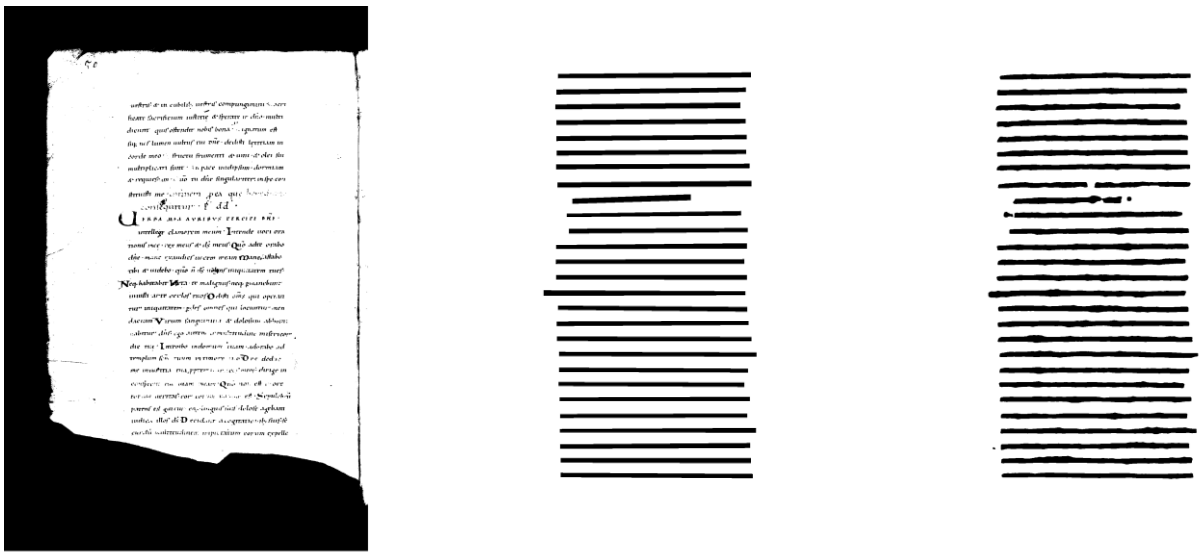
Valores	Loss	Accuracy
Máximo	0.0478	0.9846
Medio	0.0419	0.9830
Mínimo	0.0372	0.9804

Al fijarnos en la Tabla 5-8, observamos que en esta se representan los valores máximos, mínimos y medios de los cuatro parámetros empleados durante la evaluación del modelo. Como se podía apreciar en el análisis de la Tabla 5-7*,* se hacía una mención a los mejores y peores resultados, que como podemos observar en esta tabla, son los recogidos en la misma. De esta manera:

- La mejor evaluación obtenida tendría los siguientes valores para cada uno los parámetros que la definen: una *loss* de 0.0372 y una *accuracy* del 98.46%. Solo del orden de una centésima más baja que las cifras conseguidas para los mismos parámetros durante el entrenamiento del modelo.
- Por otra parte, la peor evaluación del modelo vendría definida por: una *loss* de 0.0478 y una *accuracy* del 98.04%. Aunque los valores obtenidos en el peor caso no son tan malos comparados con los medios, siguen siendo un poco peores que la media de entrenamientos, aunque superan a los valores conseguidos en el peor caso de entrenamiento.

Sin embargo, para evaluar el rendimiento del modelo a través de la evaluación del mismo, debemos centrarnos en el valor medio obtenido para cada parámetro. Así pues, se puede decir que se tiene una precisión muy alta, del 98.30% (solo una milésima inferior a la conseguida en el entrenamiento), con una pérdida del 0.0419, que no está nada mal, ya que incluso coincide con la obtenida para el entrenamiento. Por tanto podemos afirmar que el modelo se comporta de manera bastante notable con datos del conjunto de test, de manera incluso comparable a su comportamiento durante el entrenamiento.

Finalmente, en la Figura 5-6, se muestra un ejemplo de la entrada y la salida del modelo al ser evaluado, así como una predicción del mismo:



(a) (b) (c)

Figura 5-6. (a) Imagen 031 de DIVA-HisDB que representa la entrada de la red; (b) Thick Backbone obtenido de (a) conseguido a la salida de la red; (c) Predicción del Thick Backbone de (a)

Como se puede apreciar en la tercera imagen, la (c) (que es la predicción que realiza el modelo sobre la imagen 031 del set de datos empleado), podemos ver que obtiene con bastante precisión el Thick Backbone de (a), con la salvedad de que la línea número 8 de (c) está un poco separada al compararla con la de (b). No obstante, si se coteja con (a), se aprecia que las palabras en ese punto están un poco separadas, por lo que no podemos tomar esta diferencia como algo defectuoso totalmente. Lo mismo ocurre con la línea siguiente.

5.2.2 Resultados de los entrenamientos y test con ZigZag Backbone

En este subapartado, se hará un análisis de los resultados de entrenamiento y test obtenidos para el problema ZigZag Backbone, empezando con los conseguidos para la base de datos ICDAR 2013 y terminando con los de DIVA-HisDB.

5.2.2.1 Empleando la base de datos ICDAR 2013

Al igual que para el problema de Thick Backbone, en ZigZag se realizan también 10 entrenamientos del modelo propuesto y 10 evaluaciones del mismo. Todo ello con el set de datos de ICDAR 2013. Así, pues se pueden apreciar las variaciones existentes en las diferentes iteraciones. En la Tabla 5-9, se exponen los valores de las cuatro métricas descritas al comienzo del apartado 5.2, para cada uno de los diez entrenamientos completados:

Tabla 5–9. Resultados de los 10 entrenamientos realizados para el problema ZigZag Backbone, con el conjunto de entrenamiento de la base de datos ICDAR 2013 (los valores de *accuracy* y *val accuracy* están en tanto por uno)

Nº entrenamiento	Loss	Accuracy	Val loss	Val accuracy
1	0.0431	0.9814	0.0476	0.9790
2	0.0361	0.9842	0.0408	0.9823
3	0.0391	0.9829	0.0429	0.9815
4	0.0404	0.9824	0.0438	0.9813
5	0.0361	0.9843	0.0449	0.9800
6	0.0402	0.9825	0.0422	0.9816
7	0.0382	0.9834	0.0454	0.9800
8	0.0384	0.9834	0.0410	0.9824
9	0.0400	0.9827	0.0436	0.9809
10	0.0383	0.9831	0.0427	0.9819

Si se observa la Tabla 5-9, se puede apreciar que uno de los mejores entrenamientos realizados ha sido el quinto. Este presenta el mayor valor del parámetro *accuracy* (98.43%) y el menor valor del parámetro *loss* (0.0361), para el set de entrenamiento. Sin embargo, si nos fijamos en los parámetros pertenecientes al set de validación, se puede observar que el entrenamiento que destaca por tener un menor valor del parámetro *validation loss* es el segundo (0.0408), mientras que el octavo lo hace por haber conseguido el valor más alto del parámetro *validation accuracy* (98.24%). Por otra parte, el peor entrenamiento es el primero. Este presenta las peores cifras para los cuatros parámetros que componen el entrenamiento, como podemos observar en la primera fila de la tabla, donde están recogidos.

Cuando se completa este número de entrenamientos, se pasa a calcular el valor máximo, mínimo y medio de los mismos. Así, es mucho más sencillo analizar el rendimiento del modelo. Los valores calculados se muestran en la Tabla 5-10:

Tabla 5–10. Valores máximo, mínimo y medio de los entrenamientos realizados para el problema ZigZag Backbone, con el conjunto de entrenamiento de la base de datos ICDAR 2013 (los valores de *accuracy* y *val accuracy* están en tanto por uno)

Valores	Loss	Accuracy	Val loss	Val accuracy
Máximo	0.0431	0.9843	0.0476	0.9824
Medio	0.0390	0.9830	0.0435	0.9811
Mínimo	0.0361	0.9814	0.0408	0.9790

En esta tabla, como se puede comprobar, se muestran los valores máximos, mínimos y medios de los cuatro parámetros empleados durante el entrenamiento. Al analizar la tabla anterior, Tabla 5-9, se señalaban los entrenamientos con mejores y peores resultados. Si ahora nos fijamos bien en la Tabla 5-10, los valores combinados de esos entrenamientos darían, por consiguiente, los mejores y peores resultados del entrenamiento del modelo presentado respectivamente. De esta manera:

- El mejor entrenamiento conseguido, tendría los siguientes valores para cada uno los parámetros que lo definen: una *loss* de 0.0361, una *accuracy* del 98.43%, una *validation loss* de 0.0408 y una *validation accuracy* del 98.24%.
- Por otra parte, el peor entrenamiento vendría definido por: una *loss* de 0.0431, una *accuracy* del 98.14%, una *validation loss* de 0.0476 y una *validation accuracy* del 97.90%. Aunque tampoco son unos valores demasiado malos, al compararlos con el valor medio de entrenamientos.

Sin embargo, para evaluar el rendimiento del modelo en su fase de entrenamiento, de una forma más general, debemos centrarnos en el valor medio obtenido para cada parámetro. Así pues, se puede decir que se tiene una precisión bastante alta, del 98.24%, con una pérdida bastante baja, del 0.0361, que no está nada mal. Por tanto, vemos que nuestro modelo se comporta muy bien durante la fase de entrenamiento, mejor con el conjunto de entrenamiento que con el de validación, aunque la diferencia entre ambos es ínfima.

Seguidamente, al igual que se hizo anteriormente para el entrenamiento, en la Tabla 5-11 se exponen los resultados de las 10 evaluaciones del modelo, pero en este caso solo para los parámetros *loss* y *accuracy*:

Tabla 5–11. Resultados de las 10 evaluaciones realizadas para el problema ZigZag Backbone, con el conjunto de test de la base de datos ICDAR 2013 (el valor de *accuracy* está en tanto por uno)

Nº evaluación	Loss	Accuracy
1	0.0533	0.9770
2	0.0465	0.9797
3	0.0510	0.9786
4	0.0493	0.9788
5	0.0474	0.9796
6	0.0476	0.9792
7	0.0536	0.9773
8	0.0487	0.9795
9	0.0525	0.9777
10	0.0480	0.9791

Al observar la Tabla 5-11, se puede percibir que, la mejor evaluación del modelo para el conjunto de test, ha sido la segunda, ya que alcanza los valores más óptimos de *loss* (0.0465) y de *accuracy* (97.97%). Por otra parte, la peor evaluación se alcanza en la número 1, para el parámetro *accuracy* (97.70%), mientras que para el parámetro *loss* se consigue en la séptima evaluación (0.0536).

Cuando se tienen todos los valores para las 10 evaluaciones realizadas, de la misma forma que para el caso del

entrenamiento, se calcula: el valor máximo, mínimo y medio de las mismas. En la Tabla 5-12 se exponen estos valores:

Tabla 5–12. Valores máximo, mínimo y medio de las evaluaciones realizadas para el problema ZigZag Backbone, con el conjunto de test de la base de datos ICDAR 2013 (el valor de *accuracy* está en tanto por uno)

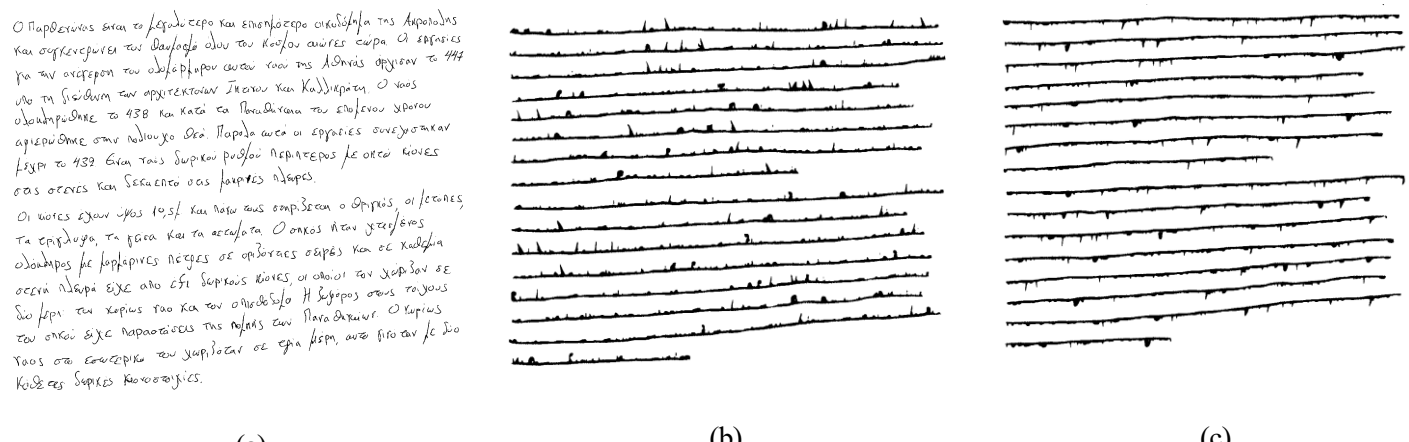
Valores	Loss	Accuracy
Máximo	0.0536	0.9797
Medio	0.0498	0.9787
Mínimo	0.0465	0.9770

Si nos fijamos la tabla superior, la Tabla 5-12, vemos que en esta se representan los valores máximos, mínimos y medios de los cuatro parámetros empleados durante la evaluación del modelo. Como se podía apreciar en el análisis de la Tabla 5-11, se hacía referencia a los mejores y peores resultados, que como podemos observar en la Tabla 5-12, son los recogidos en la misma. De esta manera:

- La mejor evaluación obtenida tendría los siguientes valores para cada uno los parámetros que la definen: una *loss* de 0.0465 y una *accuracy* del 97.97%. Cifras un poco inferiores a las conseguidas en el entrenamiento del modelo.
- Por otra parte, la peor evaluación del modelo vendría definida por: una *loss* de 0.0536 y una *accuracy* del 97.70%. Aunque los valores obtenidos en el peor caso no son tan malos comparados con los medios, siguen siendo un poco peores que la media de entrenamientos e incluso que los peores valores conseguidos en dichos entrenamientos.

En cambio, para evaluar el rendimiento del modelo a través de la evaluación del mismo, es necesario centrarse en el valor medio obtenido para cada parámetro. Así pues, se puede afirmar que se tiene una precisión bastante alta, del 97.87%, pero un poco inferior a la conseguida en la fase de entrenamiento, con una pérdida del 0.0498, que no está nada mal. Por tanto, podemos decir que el modelo funciona bien con el dataset de test, aunque, como se ha visto en tablas anteriores, lo hace mejor en la fase de entrenamiento.

Finalmente, en la Figura 5-7 se muestra el tipo de entrada que recibe la red, y la salida que arroja de cada parte del contorno extraído (las hemos colocado ambas en la misma posición, de otra forma, la imagen (c) estaría justo al revés). Hay que destacar que, en este caso, no se muestra una imagen de predicción, ya que para poder realizar dicho paso, deberíamos unir ambas imágenes, es decir fusionarlas. Así pues, la imagen final tendría tanto el contorno superior como el inferior extraído. No obstante, este proceso se explicará con mayor detalle en el próximo capítulo.



(a) (b) (c)

Figura 5-7. (a) Imagen 201 de ICDAR2013 que representa la entrada de la red; (b) Salida de la red con la parte superior extraída; (c) Salida de la red con la parte inferior extraída

5.2.2.2 Empleando la base de datos DIVA-HisDB

De la misma manera que se actuó con el dataset ICDAR 2013, en este subapartado se realizan 10 entrenamientos y 10 evaluaciones del modelo propuesto con el dataset DIVA-HisDB. Con ello se consigue una visión más completa del funcionamiento del modelo expuesto. Además, como ya comentábamos en el subapartado 5.2.1.2, hay que mencionar que tanto el entrenamiento como la evaluación del modelo, se han llevado a cabo sin el parámetro de *data_augmentation* activo. Dicho cambio se debe a problemas de falta de memoria en la GPU de Google Colab, ya que los tensores de esta base de datos pesan algo más que los de ICDAR 2013, al tener dimensiones mayores. Por tanto, en cada batch de imagen, en vez de generar 3 de ellas, solo se genera 1, consiguiendo así que la memoria no se agote.

Seguidamente, en la Tabla 5-13 que encontramos a continuación, se exponen los valores de las cuatro métricas descritas al comienzo del apartado 5.2, para cada uno de los diez entrenamientos completados:

Tabla 5–13. Resultados de los 10 entrenamientos realizados para el problema ZigZag Backbone, con el conjunto de entrenamiento de la base de datos DIVA-HisDB (los valores de *accuracy* y *val accuracy* están en tanto por uno)

Nº entrenamiento	Loss	Accuracy	Val loss	Val accuracy
1	0.0472	0.9820	0.0465	0.9811
2	0.0453	0.9825	0.0568	0.9781
3	0.0480	0.9817	0.0479	0.9810
4	0.0423	0.9838	0.0537	0.9797
5	0.0391	0.9846	0.0467	0.9812
6	0.0401	0.9842	0.0432	0.9830
7	0.0388	0.9851	0.0424	0.9831
8	0.0453	0.9832	0.0509	0.9797
9	0.0376	0.9854	0.0381	0.9855
10	0.0374	0.9854	0.0376	0.9851

Al observar la tabla superior, se puede apreciar que el mejor entrenamiento conseguido, trabajando con el set de entrenamiento, es el décimo. Esto es así dado que presenta el mayor valor del parámetro *accuracy* (98.54%) y el menor valor del parámetro *loss* (0.0374). De la misma manera, si nos fijamos en los parámetros pertenecientes al set de validación, se puede observar que vuelve a ser el décimo entrenamiento el que presenta el menor valor del parámetro *validation loss* (0.0376), mientras que es el noveno el que alcanza el mayor valor de *validation accuracy* (98.55%). Por otra parte, el peor entrenamiento que podemos encontrar en la tabla anterior es sin duda el segundo, pues consigue las peores cifras para los cuatros parámetros que componen el entrenamiento.

Cuando se completa dicho número de entrenamientos, se pasa a calcular el valor máximo, mínimo y medio de los mismos. Así, es mucho más sencillo analizar el rendimiento del modelo. Los valores calculados se muestran en la Tabla 5-14:

Tabla 5–14. Valores máximo, mínimo y medio de los entrenamientos realizados para el problema ZigZag Backbone, con el conjunto de entrenamiento de la base de datos DIVA-HisDB (los valores de *accuracy* y *val accuracy* están en tanto por uno)

Valores	Loss	Accuracy	Val loss	Val accuracy
Máximo	0.0480	0.9854	0.0568	0.9855
Medio	0.0421	0.9838	0.0464	0.9818
Mínimo	0.0374	0.9817	0.0376	0.9781

Como se comentaba en el párrafo anterior, en la Tabla 5-14 se muestran los valores máximos, mínimos y medios de los cuatro parámetros empleados durante el entrenamiento. Como se podía apreciar en el análisis de la Tabla 5-13, se hacía una mención a los mejores y peores resultados, que como podemos observar en la Tabla 5-14, son los recogidos en la misma. Así pues:

- El mejor entrenamiento conseguido, tendría los siguientes valores para cada uno los parámetros que lo definen: una *loss* de 0.0374, una *accuracy* del 98.54%, una *validation loss* de 0.0376 y una *validation accuracy* del 98.55%.
- Por otra parte, el peor entrenamiento vendría determinado por: una *loss* de 0.0480, una *accuracy* del 98.17%, una *validation loss* de 0.0568 y una *validation accuracy* del 97.81%. Aunque al compararlos con la segunda fila de la Tabla 5-14, no resultan cifras demasiado malas, de hecho están bastante bien.

Sin embargo, para evaluar el rendimiento del modelo en su fase de entrenamiento, de una forma más general, debemos atender al valor medio obtenido para cada parámetro. Por tanto, se puede afirmar que se tiene una precisión muy alta, del 98.38%, con una pérdidas bastante buenas, del 0.0421. Por tanto, vemos que nuestro modelo se comporta muy bien durante la fase de entrenamiento. Hay que decir que funciona solo un poco mejor con el conjunto de entrenamiento que con el de validación, aunque la diferencia es muy pequeña.

A continuación, de la misma forma que para el entrenamiento, en la Tabla 5-15 se exponen los resultados de las 10 evaluaciones del modelo, pero en este caso solo para los parámetros *loss* y *accuracy*:

Tabla 5–15. Resultados de las 10 evaluaciones realizadas para el problema ZigZag Backbone, con el conjunto de test de la base de datos DIVA-HisDB (el valor de *accuracy* está en tanto por uno)

Nº evaluación	Loss	Accuracy
1	0.0447	0.9828
2	0.0446	0.9823
3	0.0443	0.9827
4	0.0481	0.9814
5	0.0377	0.9853
6	0.0382	0.9850
7	0.0403	0.9840
8	0.0494	0.9816
9	0.0345	0.9864
10	0.0358	0.9856

Analizando la Tabla 5-15, se puede apreciar que la mejor evaluación del modelo para el conjunto de test, ha sido la novena, ya que alcanza los valores más óptimos tanto para el parámetro *loss* (0.0345), y de *accuracy* (98.64%). Por otra parte, la peor evaluación se alcanza en la número 8, para el parámetro *loss* (0.0494), mientras que para el parámetro *accuracy* (98.14%), se consigue en la cuarta evaluación

Cuando se tienen todos los valores para las 10 evaluaciones realizadas, se calculan: el valor máximo, mínimo y medio de las mismas. En la Tabla 5-16 se exponen estos valores:

Tabla 5–16. Valores máximo, mínimo y medio de las evaluaciones realizadas para el problema ZigZag Backbone, con el conjunto de test de la base de datos DIVA-HisDB (el valor de *accuracy* está en tanto por uno)

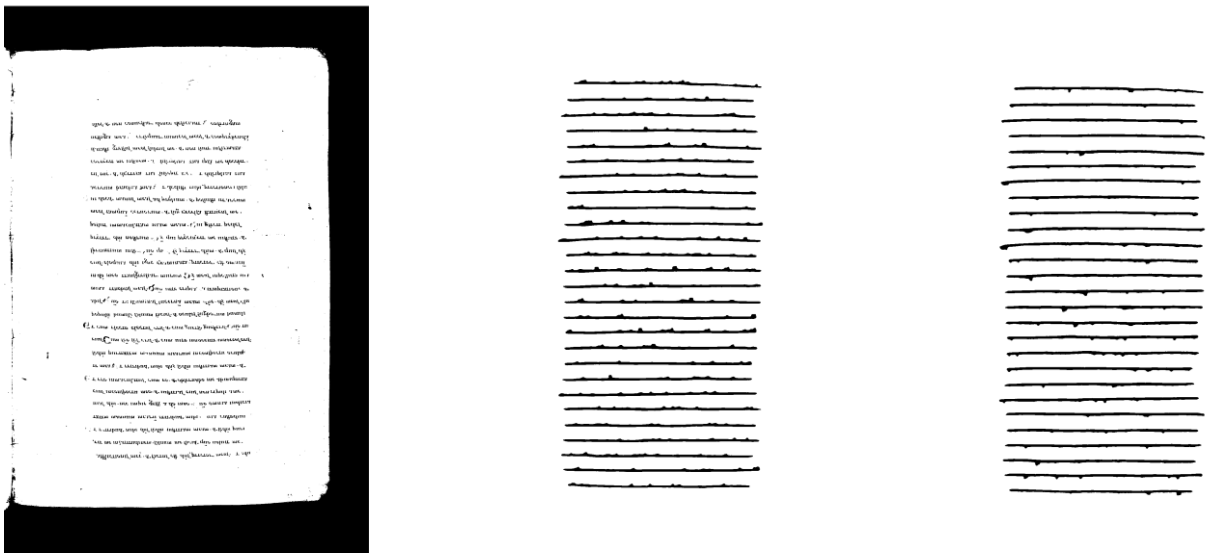
Valores	Loss	Accuracy
Máximo	0.0494	0.9864
Medio	0.0418	0.9837
Mínimo	0.0345	0.9814

Estudiando la Tabla 5-16, se puede observar que en esta se representan los valores máximos, mínimos y medios de los cuatro parámetros empleados durante la evaluación del modelo. Cuando se analizaba la tabla anterior, Tabla 5-15, se señalaban los entrenamientos con mejores y peores cifras. Por tanto, si en esta tabla combinamos esos valores que resaltábamos en la anterior, obtendríamos los mejores y peores resultados del entrenamiento del modelo aquí expuesto. Así pues:

- La mejor evaluación obtenida tendría los siguientes valores para cada uno los parámetros que la definen: una *loss* de 0.0345 y una *accuracy* del 98.64%. Cifras que se asemejan mucho a las obtenidas en el entrenamiento del modelo, incluso siendo la precisión en este caso de test un poco mayor a la conseguida con el conjunto de entrenamiento.
- Por otra parte, la peor evaluación del modelo estaría dada por los siguientes valores: una *loss* de 0.0494 y una *accuracy* del 98.14%. Si comparamos estos parámetros con los valores medios de los mismos, vemos que no hay tanta diferencia. Incluso dichas cifras no distan mucho de las conseguidas en el peor caso de entrenamiento.

En cambio, para estudiar el rendimiento del modelo a través de la evaluación del mismo, debemos atender al valor medio obtenido para cada parámetro. Por tanto, se puede decir que el modelo planteado consigue una precisión muy alta, del 98.37%, casi igual a la obtenida en la fase de entrenamiento, con una pérdida del 0.0418, que no está nada mal. Por lo que se afirma que el modelo funciona bastante bien con el dataset de test, y lo hace alcanzando valores muy parecidos a los conseguidos con el set de entrenamiento.

Para finalizar este apartado decir que, en la Figura 5-8 se muestra el tipo de entrada que recibe la red, y la salida que arroja de cada parte del contorno extraído (las hemos colocado ambas en la misma posición, de otra forma, la imagen (c) estaría justo al revés). Hay que destacar que, en este caso, al igual que se dijo en el subapartado anterior, no se muestra una imagen de predicción, ya que para poder realizar este paso, deberíamos unir ambas imágenes, es decir fusionarlas. Así pues la imagen final tendría tanto el contorno superior como el inferior extraído. En el próximo capítulo se extenderá la explicación de este último proceso.



(a) (b) (c)

Figura 5-8. (a) Imagen 091 de la base de datos DIVA-HisDB que representa la entrada de la red; (b) Salida de la red con la parte superior extraída; (c) Salida de la red con la parte inferior extraída

6 SEGMENTACIÓN

En este capítulo ponemos el foco en la *tarea de segmentación* del texto, que es el siguiente paso a realizar con el fin de alcanzar el objetivo principal del proyecto. Para ello, se requiere una especie de *máscara*, la cual presenta un número (valor) distinto para cada línea del texto, de manera que queda representada con colores con la ayuda de un *colormap* (quedaría una *máscara* donde cada línea del texto estaría representada con un color diferente). Dicha *máscara,* se compara píxel a píxel con el texto original, dando lugar así a la segmentación del texto deseada. No obstante, para construir esta *máscara*, al contar con 2 modelos capaces de extraer los contornos superiores e inferiores (lo hace un solo modelo) y la espina dorsal gruesa del texto, es necesario, además, otro proceso capaz de fusionar las tres piezas anteriores. A este proceso se le llama *proceso de fusión*.

Por tanto, en este Capítulo se detallarán tanto el proceso de fusión como el de segmentación, ambos bien definidos en el fichero de Python *segmentation.py*. Además, se hará un repaso de los errores existentes en el proceso de fusión, así como de sus posibles soluciones (todas ellas implementadas en la función *fixFunction.py*, que es llamada desde *segmentation.py*). Todo el proceso realizado en el Capítulo se refleja de forma esquemática, siguiendo la estructura "*pipeline"* del proyecto representada en la Figura 1-12 del Apartado 1.3, en la siguiente Figura 6-1:

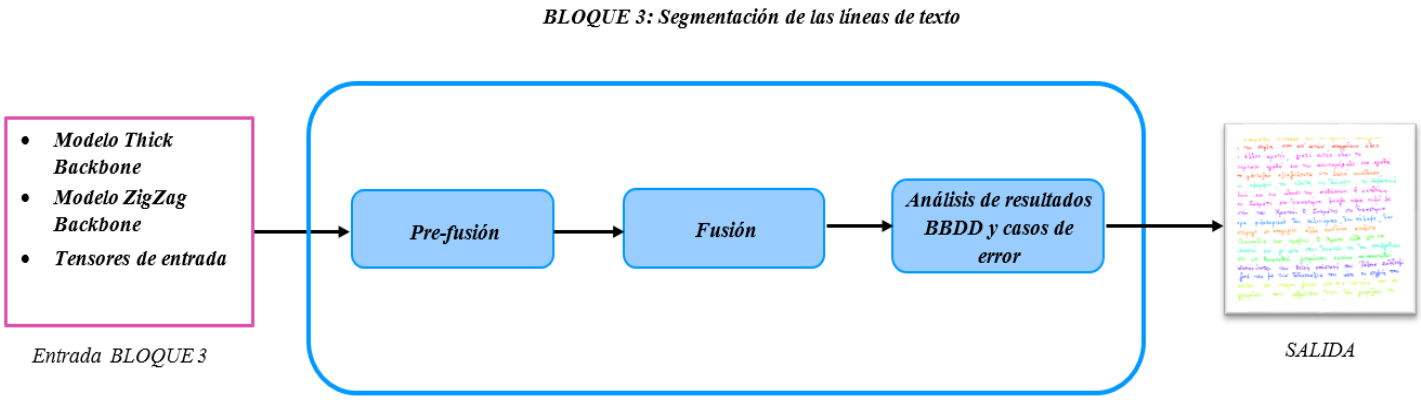


Figura 6-1. Contenido del *Bloque 3* de la estructura "*pipeline*" del proyecto, cuya entrada se trata del conjunto de tensores de entrada conseguidos en el *Bloque 1*, y los modelos de Thick Backbone y ZigZag Backbone. Por otra parte, su salida, es la salida del sistema y se trata de las segmentaciones de las imágenes de la base de datos seleccionada

6.1 Descripción del proceso de fusión

En esta sección se va a describir el proceso de fusión llevado a cabo, con el fin de construir la *máscara de segmentación*. En primer lugar describiremos el paso previo a la fusión, en el que se obtienen los elementos que posteriormente completarán la *máscara.* Sin embargo, antes de comenzar con la explicación de este *paso previo,* es necesario definir un par de conceptos:

- *Connected Components:* Un CC (Connected Component), concepto visto brevemente en el Capítulo 2 de esta memoria y adquirido de la teoría de grafos, representa básicamente una región o polígono cerrado dentro de la imagen a tratar (en nuestro caso serían las líneas de texto). Además, se caracteriza por el hecho de que los píxeles que forman parte de dicha región están conectados y tienen el mismo valor numérico [3]. Así pues, en un CC se pueden distinguir dos tipos de regiones: una región interior, en la que los píxeles que la conforman están rodeados completamente por otros con el mismo valor (en cualquier posición de las ocho posibles); y una región exterior en la que sus píxeles están rodeados, como mínimo, por un píxel de igual valor a los de dicha región y por al menos uno de un valor distinto. Con el fin de etiquetar e identificar los *Connected Components* en nuestras imágenes, independientemente de la base de datos empleada, se usa la función *scipy.ndimage.label*, que

podemos encontrar en [32]. En la Figura 6-2, se puede observar el funcionamiento de la función anterior.

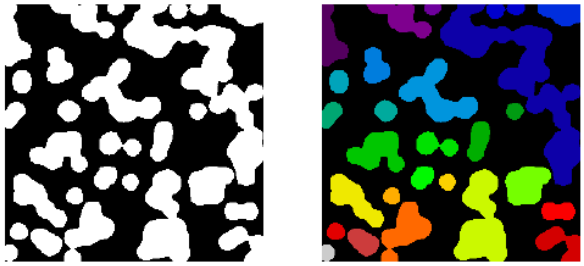


Figura 6-2. Funcionamiento *scipy.ndimage.label* [3]

- *Mapa de la máscara:* Otro concepto o más bien, apunte, que se debe hacer antes de ver las máscaras que podemos construir, es que el mapa de color empleado para representar la *máscara de segmentación* es completamente aleatorio, pues se consigue extrayendo los $n + 1$ colores aleatorios del colormap *rainbow* [3], correspondientes a las n líneas del texto (se fuerza que el primero de los colores sea blanco y esté asociado con el "0", que es el fondo de las imágenes). De esta manera, debido a la limitación en el mapa de colores y a la aleatoriedad de la asignación de dichos colores, puede parecer que dos líneas ostenten el mismo tono de color, pero en realidad serán distintos.

Una vez realizadas las definiciones anteriores, pasamos a explicar el **paso de *pre-fusión***, en el que se consiguen los elementos necesarios para poder crear nuestra *máscara de segmentación.* Con objeto de ilustrar este proceso y así poder comprenderlo un poco mejor, se muestra un esquema recogido en la Figura 6-3 (donde el símbolo  sirve para denotar los resultados intermedios del proceso), basado en el desarrollado en [3]:

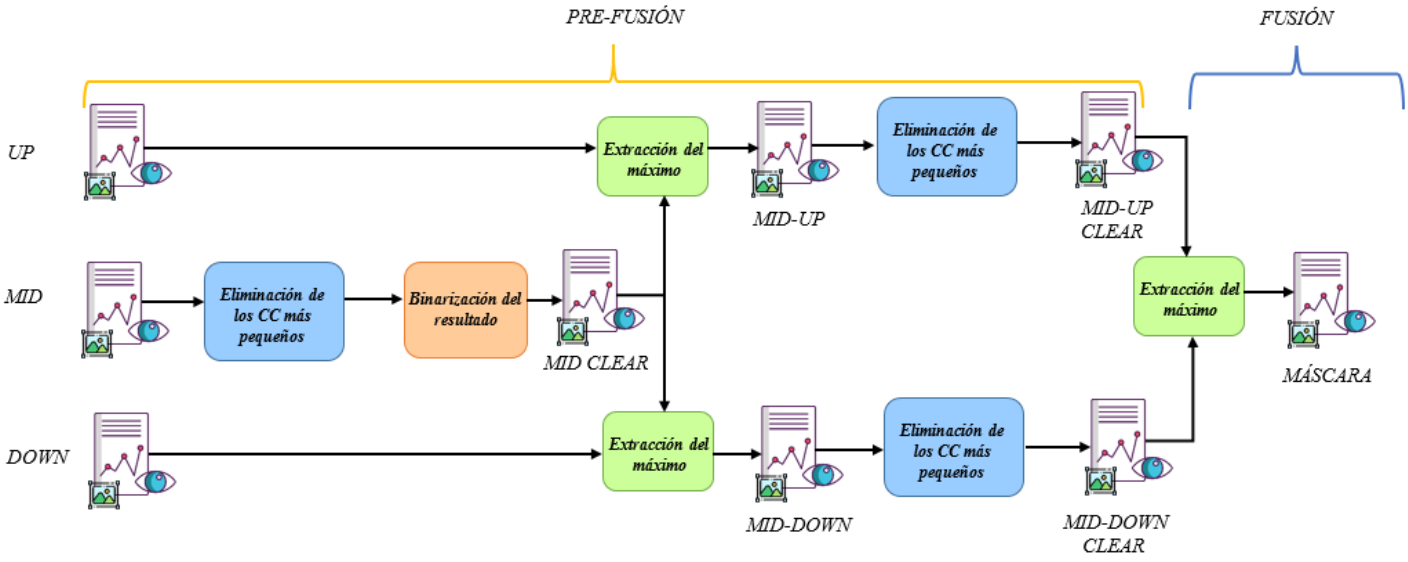


Figura 6-3. Esquema que ilustra los procesos de *pre-fusión* y *fusión*

Como se observa en la Figura 6-3, el paso de *pre-fusión* comienza con la existencia de tres predicciones de los modelos propuestos. En el caso del modelo dispuesto para el problema de Thick Backbone, hablamos de la predicción del esqueleto o espina dorsal gruesa del texto tratado, lo que en el esquema anterior se denomina como *MID*. Por otra parte, en el caso del modelo construido para el problema de ZigZag Backbone, manejamos dos tipos de predicciones extraídas con la misma red neuronal: la predicción de las partes o contornos superiores de las líneas que componen el texto tratado, lo que en el diagrama superior encontramos como *UP*; y la predicción de los contornos inferiores de las líneas del texto, lo que en el esquema anterior nombramos como *DOWN*. En la Figura 6-4, que se muestra a continuación, ponemos un ejemplo de las tres matrices de predicción.

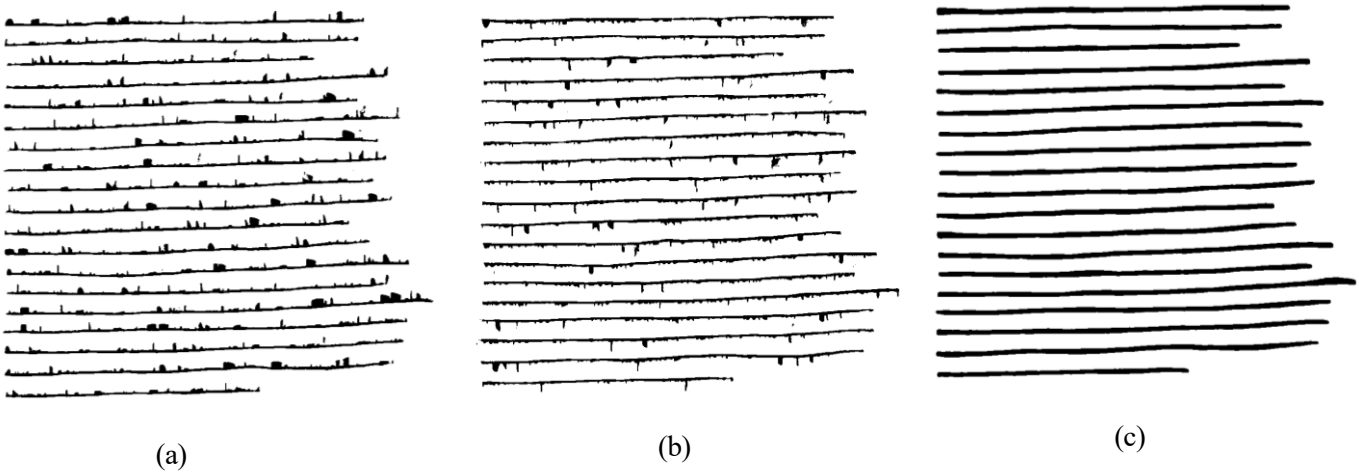


Figura 6-4. Representación de las tres matrices de predicción de la imagen 005 del dataset ICDAR 2013: (a) Predicción del contorno superior de las líneas de la imagen 005 (*UP*) ; (b) Predicción del contorno inferior de las líneas de la imagen 005 (*DOWN*); y (c) Predicción del esqueleto o espina dorsal gruesa de las líneas de la imagen 005 (*MID*).

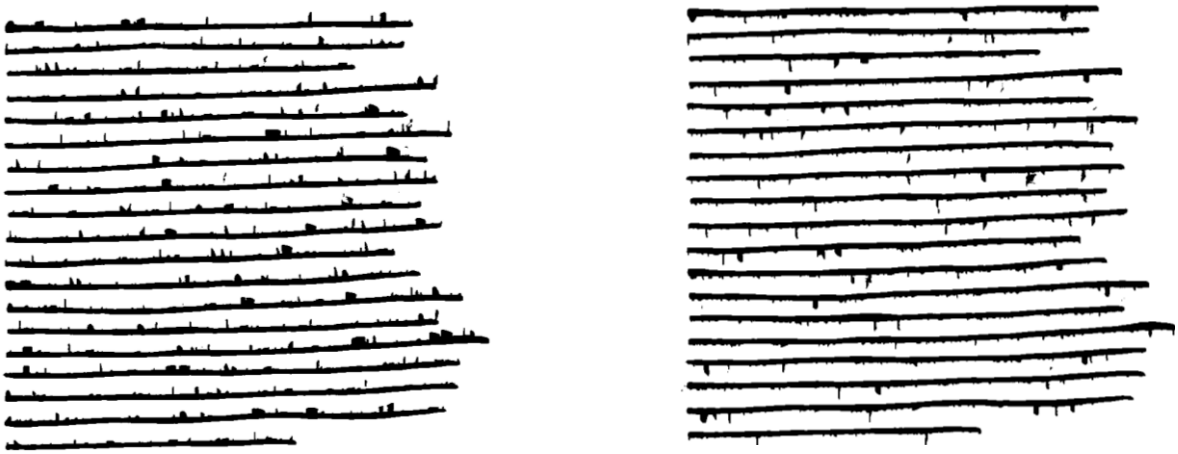
Una vez que tenemos las predicciones anteriores, lo primero que se debe hacer es aplicar la función *removeSmallCC*, definida en el archivo *segmentation.py,* a la predicción de Thick Backbone, es decir a *MID*. Esta función se encarga de etiquetar los CCs de los que se hablaba antes (asigna un valor diferente a cada línea de texto: los píxeles de la primera línea llevarán el valor numérico 1, los de la segunda el 2 y así sucesivamente hasta completar el número total de líneas) y eliminar los que tengan un tamaño menor que una tolerancia impuesta. En nuestro trabajo se mantuvo la tolerancia elegida por [3], que es de 2800 (*CC_AREA_TOLERANCE_MID* = 2800). En la Figura 6-5, se muestra el resultado de aplicar la función anterior a *MID*:

Figura 6-5. Imagen resultante de aplicar la función *removeSmallCC* a la *MID* de la imagen 005 del dataset ICDAR 2013

Después de etiquetar y eliminar los CCs más pequeños, se pasa a binarizar el resultado conseguido en la figura anterior. Así pues nos quedaría una imagen en blanco y negro, donde los píxeles negros tendrían un valor de 1, mientras que los blancos, que constituyen el fondo de la imagen, estarían a 0.

Al terminar de binarizar la *MID* pasada por *removeSmallCC*, hacemos dos operaciones con el resultado de esa binarización, que denominamos *MID CLEAR*:

- Se hace el máximo, elemento a elemento, entre *UP* y *MID CLEAR*, obteniendo así la imagen o matriz *MID-UP*, es decir se fusionan ambas partes. En la Figura 6-6 (a), vemos un ejemplo de *MID-UP*.
- Se hace el máximo, elemento a elemento, entre *DOWN* y *MID CLEAR*, obteniendo así la imagen o matriz *MID-DOWN,* es decir se fusionan ambas partes. En la Figura 6-6 (b), vemos un ejemplo de *MID-DOWN*.

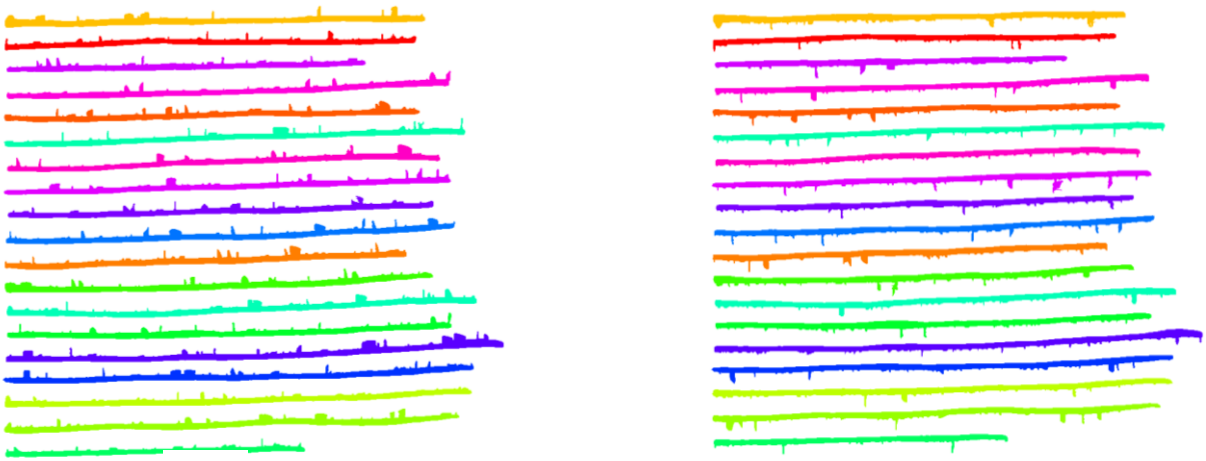


(a) (b)

Figura 6-6. (a) *MID-UP* y (b) *MID-DOWN* de la imagen 005 del dataset ICDAR 2013

De esta forma, se consigue que tanto el contorno superior de las líneas como el inferior estén mejor delimitados, como se puede comprobar en la imagen anterior.

El siguiente paso a realizar, será el de pasar, tanto *MID-UP* como *MID-DOWN* a la función *removeSmallCC,* para, al igual que se hizo con *MID,* etiquetar sus CCs y eliminar los que sean más pequeños que una cierta tolerancia. En este caso, la tolerancia será de 2500 (*CC_AREA_TOLERANCE_UPDOWN* = 2500). En la Figura 6-7, se muestran los resultados de dicha operación: *MID-UP CLEAR* y *MID-DOWN CLEAR*.



(a) (b)

Figura 6-7. (a) *MID-UP CLEAR* y (b) *MID-DOWN CLEAR* de la imagen 005 del dataset ICDAR 2013

Llegados a este punto, se tienen los elementos necesarios para comenzar con el **proceso de *fusión***. Este proceso es verdaderamente sencillo, ya que lo único que necesitamos es coger el máximo, elemento a elemento, entre *MID-UP CLEAR* y *MID-DOWN CLEAR*. Y finalmente, después de esta operación, conseguiríamos la máscara deseada, lista para el proceso de segmentación (Apartado 6.2). En la Figura 6-8, mostramos un ejemplo de *MÁSCARA*.

Figura 6-8. *MÁSCARA* de la imagen 005 del dataset ICDAR 2013

6.1.1 Recopilación de los errores encontrados durante el proceso de fusión

Como es natural, para que la máscara resultante del proceso de fusión esté bien construida, tanto la mitad superior de la imagen segmentada, como la inferior, es decir, *MID-UP CLEAR* y *MID-DOWN CLEAR* en el diagrama anterior, deben de mantener las mismas posiciones y coincidir en el número de CC, de manera que al unirlas (por supuesto con la espina dorsal también) para formar la *máscara de segmentación*, encajen a la perfección. No obstante, en un gran número de imágenes, tanto del dataset DIVA-HisDB como del ICDAR 2013, se apreciaban diversos problemas de segmentación. Al igual que se hace en [3], como el patrón de fallos de segmentación se repite en diferentes imágenes, realizamos una clasificación de los mismos en varios tipos de errores encontrados durante el proceso de fusión. En este subapartado, se describirá en qué consisten, el proceso seguido para detectarlos, así como la manera de corregirlos en determinados casos. Finalmente, se expondrán los resultados de dicha corrección.

6.1.1.1 Caso A: Líneas unidas

Este caso [3] se da cuando, en el paso de *pre-fusión* previamente descrito, al fusionar *MID CLEAR* con *UP* y *DOWN* (generándose así *MID-UP* y *MID-DOWN*), dos líneas de cualquiera de las dos fusiones, se unen por error. Esta unión se produce debido a que la línea inferior (superior) posee un caracter más largo de la cuenta (rozando la línea superior (inferior)), y éste se ha unido con la línea justamente superior (inferior), dando lugar a un error. Así pues, el algoritmo etiquetaría ambas líneas unidas por dicho caracter (o caracteres) como la misma.

Si además de ello, esa unión de líneas se produce en solo una de las dos mitades, algo bastante frecuente, la enumeración de las líneas de ambas partes no sería la misma, y por lo tanto al intentar construir la *máscara de segmentación*, nos encontraríamos con líneas que han juntado valores de una parte y de la otra. De esta manera, la *máscara de segmentación*, a partir de dichas líneas, sería errónea. En la Figura 6-9 se puede identificar el caso descrito de forma gráfica.

(a) (b) (c)

Figura 6-9. Caso A en la imagen 089 del dataset ICDAR 2013: (a) *MID-UP CLEAR*, donde se detectó el Caso A; (b) *MID-DOWN CLEAR*; y (c) *MÁSCARA* errónea

Como se puede observar en la Figura 6-9, solo la imagen (a) presenta el Caso A en la línea 15, el cual encontramos señalado con un círculo negro. De esta manera, al formar la *máscara de segmentación* (c), podemos apreciar que ésta es claramente incorrecta, ya que a partir de la línea 15, la enumeración de las líneas de las 3 imágenes que forman la máscara, no concuerda. A continuación, se explicará la detección del Caso A.

Para empezar, decir que la detección de este caso se lleva a cabo en la función *fixFunction* (que se llama siempre desde la función *funSegFus,* declarada en *segmentation.py*) del archivo de Python, *fixFunction.py*. Dicha función recibe como parámetros: *MID-UP CLEAR*, *MID-DOWN CLEAR*, la binarización de *MID* después de haber pasado por la función *removeSmallCC* y un diccionario creado previamente, donde se recogen las dimensiones de cada CC de *MID-UP CLEAR* y *MID-DOWN CLEAR*. Así, con este diccionario, el algoritmo (que es el de la función *funSegFus* del archivo *segmentation.py*, ya que es ahí donde llamamos a *fixFunction*), crea un archivo de depuración, con el contenido del mismo.

En *fixFunction* nos valemos de elementos de estadísticos, de modo que podamos estimar las líneas mal segmentadas, y clasificar dichos errores de segmentación en cada caso identificado. Para ello, se definen dos variables aleatorias, que se usarán también en casos posteriores, y que se tratan de: *X_midUp* (que en el código la encontramos como *y_clear_midUp*), v.a. que representa el tamaño, en términos de píxel, de las líneas que conforman *MID-UP CLEAR*; y *X_midDown* (que en el código la encontramos como *y_clear_midDown*), v.a. que representa, el tamaño, en términos de píxel, de las líneas que forman *MID-DOWN CLEAR*.

Con los conceptos anteriores definidos, podemos plantear el proceso seguido para la ***detección*** de este caso. Decimos que nos encontramos ante un caso de tipo A, cuando el CC de alguna de las líneas del texto tiene un valor muy grande dentro del grupo de los tamaños de CC detectados (normalmente ese tamaño suele ser el de la suma de los CC de dos líneas). Ahora, expresando esto en términos matemáticos, decimos que un Caso A es detectado, cuando el tamaño de la línea donde se ha encontrado este error, sea superior a la media de tamaños de líneas del texto sumándole 2.3 veces la desviación típica [3]. Esto se hace tanto para *MID-UP CLEAR*, como para *MID-DOWN CLEAR*.

Una vez que se detecta el caso actual en alguna de las líneas del texto analizado, se llama a la función *fixCaseA*, recogida en *fixFunction.py*, para proceder a la ***corrección*** de dicho caso. Dicha función, *fixCaseA,* recibe como entrada la imagen a corregir, bien sea *MID-UP CLEAR* o *MID-DOWN CLEAR,* y la línea donde se ha encontrado el error. El **método de corrección** seguido por la función anterior, recoge los siguientes pasos:

- Primero, se extrae la línea errónea, la que, según hemos detectado, se habría unido con la siguiente, creando una *máscara de segmentación* equivocada.
- A continuación, para separar ambas líneas fusionadas, usando para ello la función *createLinearRegressionBackbone* perteneciente a *groundTruthFunctions.py*, construimos una recta de regresión que divida la fusión anterior por la mitad.
- Una vez separadas, ya no nos sirve la recta de regresión, luego asociados el valor "0" a cada uno de sus píxeles, haciendo que ésta sea parte del fondo de la imagen. El resultado conseguido se pasa a la función *removeSmallCC,* para que ésta etiquete bien los CCs de las líneas y elimine aquellos que tengan un tamaño menor que la tolerancia previamente impuesta.
- Si la función anterior nos devuelve dos valores de CC distintos, hemos corregido el error, de modo que se vuelven a enumerar bien las líneas erróneas y las consecutivas a estas. Si por el contrario, resulta que *removeSmallCC* no devuelve dos valores de CC, esto indica que no se ha podido corregir el error, y por tanto se devolverá la imagen sin corregir, señalando por supuesto que tiene un error que no se ha podido arreglar (se imprime un código de error).

En la Figura 6-10, se puede apreciar la corrección de una imagen que presenta el Caso A:

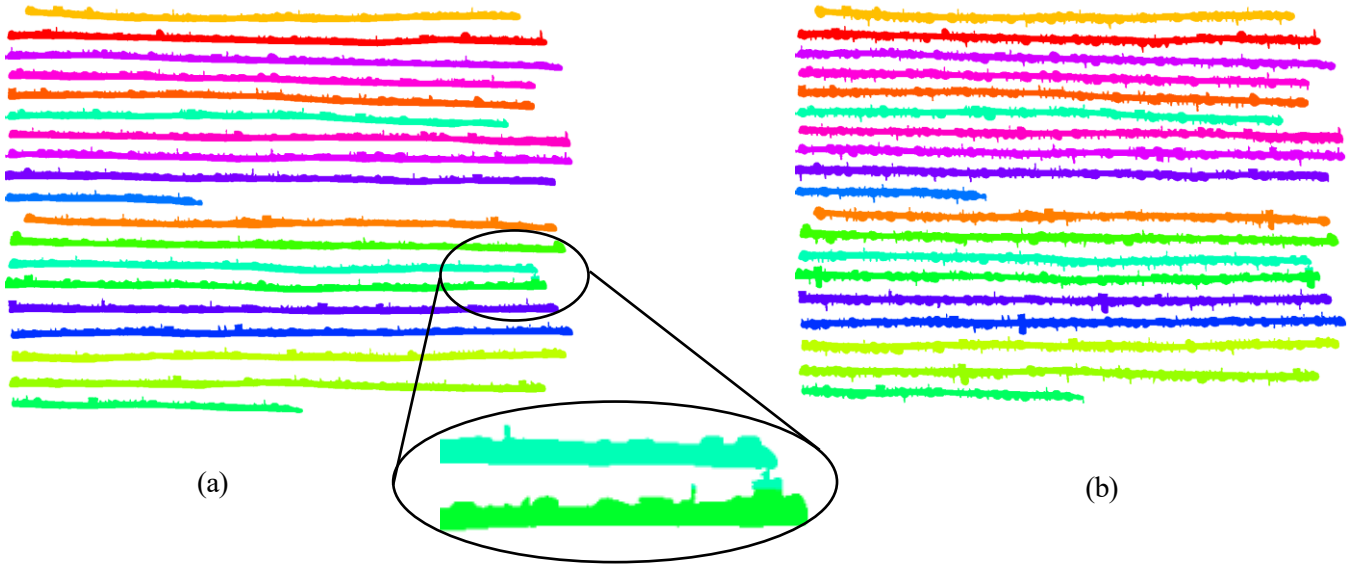


Figura 6-10. Corrección del Caso A en la imagen 246 del dataset ICDAR 2013: (a) *MID-UP CLEAR* donde se ha corregido el Caso A; (b) *MÁSCARA* sin errores

Por tanto, una vez corregido el Caso A qué se muestra en la Figura 6-9, se puede afirmar que la *máscara de segmentación* final ya es completamente correcta.

6.1.1.2 Caso B: Hueco en la línea detectado por solo una parte de la misma

Este caso ocurre sobre todo cuando la imagen original presenta un hueco en alguna de sus líneas, o también cuando algunos caracteres de las mismas están distanciados unos de otros. Además de la presencia del hueco en la imagen, para que clasifiquemos el error encontrado en la segmentación dentro del Caso B, dicho hueco solo puede aparecer en una de las dos mitades de la línea donde se haya encontrado, bien aquella perteneciente a *MID-UP CLEAR* o en aquella de *MID-DOWN CLEAR*, pero nunca en ambas, de forma que en una de las mitades la línea tenga dos partes diferenciadas, con dos tipos de CC, mientras que en la otra mitad la línea esté completa y por tanto mantenga un solo valor de CC. Por tanto, los números de las líneas no coincidirán y por tanto al construir la *máscara de segmentación* obtendremos un resultado erróneo. En la Figura 6-11, podemos ver este caso ilustrado.

(a) (b) (c)

Figura 6-11. Caso B en la imagen 053 del dataset DIVA-HisDB: (a) *MID-UP CLEAR*, donde se detecta el Caso B en la primera línea; (b) *MID-DOWN CLEAR,* donde se detecta el Caso B en la línea 19; y (c) *MÁSCARA* errónea

Como se aprecia en la imagen superior, el Caso B se puede detectar en la línea 1 de (a) y en la 19 de (b), donde se observa la existencia de un hueco en el texto. Si ahora nos fijamos en la línea 1 de (b) y en la 19 de (a), vemos que no hay ningún hueco, luego efectivamente se trata de un Caso B. De esta forma, la *máscara de segmentación* conseguida (c), es incorrecta, pues ha tenido una enumeración errónea a partir de dichas líneas. Además, como hemos señalado, en la Figura 6-11 conviven 2 casos B, cosa que puede suceder con otros tipos de casos en una misma imagen, dando lugar a *situaciones mixtas* en ella.

Por otra parte, una vez descrito el tipo de caso al que nos enfrentamos, pasamos a detallar el ***método de detección*** seguido para detectarlo, valga la redundancia. Al igual que se explicaba para el Caso A, la tarea de detección del Caso B se lleva a cabo en la función *fixFunction* (que se llama siempre desde la función *fugSegFus*, definida en *segmentation.py*). Por tanto, para identificar un Caso B en cualquiera de las imágenes que forman el dataset seleccionado, se deben cumplir las siguientes condiciones:

- La línea, donde posiblemente se encuentre el error, debe tener un tamaño inferior a la media de los tamaños de las líneas de la imagen. No obstante, esta condición es necesaria pero no es suficiente para indicar la presencia de un Caso B en la imagen tratada. Esto es debido a que, normalmente, en los documentos manuscritos la última línea de texto es menor que la media de las demás, por tanto se necesita otra condición para determinar la presencia del Caso B en la imagen.
- Como sabemos, la línea donde se encuentra el hueco, es decir la línea errónea, que debe estar en una de las imágenes que representan las mitades de todas las líneas, debe ser mucho menor que su homóloga en la otra imagen, ya que así es como describíamos que debía ser un Caso B. De esta manera, con dicha característica en mente, planteamos lo siguiente:
    1. Empleamos las variables (suponemos que son variables normales) que describíamos en el Caso A, *X_midUp* (*y_clear_midUp*) y *X_midDown* (*y_clear_midDown)*, para crear otra variable aleatoria, resultado de la resta de las mismas: $Y = X\_midUp - X\_midDown$, a la que por supuesto se le puede extraer tanto su media como desviación típica.
    2. De esta forma, podemos emplear *Y* para comprobar si el resultado de la resta entre ambas líneas es mayor o menor que la media y la desviación típica de la misma.

3. Así pues, consideramos que la línea es una *línea pequeña* si su tamaño es inferior a la media menos 1.2 veces la desviación típica de *y_clear_midUp* o *y_clear_midDown*, dependiendo que imagen estemos comprobando (no obstante, se tiene que hacer para las dos).
4. Una vez que tengamos la detección de una *línea pequeña*, en cualquiera de las dos imágenes anteriores, comprobamos si estamos o no ante un Caso B: la resta de *y_clear_midUp* - *y_clear_midDown* (siempre en este orden), debe estar fuera del intervalo definido por la media de la variable $Y \pm 0.6$ veces su desviación típica (en el código la encontramos con el nombre de *midUp_midDown_std*), y además la línea que presente el hueco debe de tener las dos rectas, en las que ha quedado fraccionada la línea principal, a la misma altura. Para ello hacemos uso de la función *detectSameHeight*, que describiremos con mayor detalle en el Subapartado 6.1.1.3.

Al saber positivamente que estamos ante un caso de tipo B, se inicia el ***proceso para corregirlo***. Para ello se llama a la función *fixCaseB,* recogida también en el archivo *fixFunction.py*. Nuestra función de corrección recibe como entrada los mismos parámetros que *fixCaseA:* la imagen a corregir y la línea errónea. No obstante, los pasos seguidos por *fixCaseB* para corregir este caso son distintos:

- Primero se extraen, tanto la línea donde ha sido detectado el caso B, como la siguiente (que sería la otra parte de la línea dividida por el hueco, donde está el error).
- Se calculan:
  - La media de las posiciones donde comienzan las líneas de la imagen tratada (*leftPos*)
  - La media de las posiciones donde acaban las líneas de la imagen tratada (*rightPos*)
- Después se saca la parametrización de la recta de regresión de las líneas extraídas en el primer punto.
- Seguidamente, para extraer las coordenadas que unirán, si se precisa, dichas líneas, se evalúan ambas rectas de regresión, con 3 píxeles de ancho (de forma que sean gruesas, para que al pasarlas por *removeSmallCC* no haya ningún problema) para los valores de las medias anteriores. Y una vez que tengamos las coordenadas de la *línea de unión*, las rellenamos con el número que le corresponde a la línea corregida.
- Así pues, se llama a *removeSmallCC* y se le pasa el resultado anterior, de modo que se vuelva a realizar el etiquetado de los CCs. Si obtenemos dos tipos de CCs distintos, significa que las dos líneas no son dos líneas, sino partes de una sola, por lo que deberían unirse, para que el programa al realizar la segmentación las considerara como una única línea en la máscara. Entonces, *fixCaseB* se encargaría de unirlas mediante la fusión de las mismas con la *línea de unión*, conseguida previamente, volviendo a enumerar las líneas posteriores a la corregida, con el fin de que la *máscara de segmentación* sea correcta. Si por el contrario no se consiguen dos tipos de CCs distintos, significa que las dos partes de la línea no estaban juntas, de manera que se devolvería la imagen al programa principal en el estado que esté, pero con un código de error.

En la Figura 6-12, que encontramos a continuación, se puede apreciar la corrección de una imagen que presenta el Caso B:

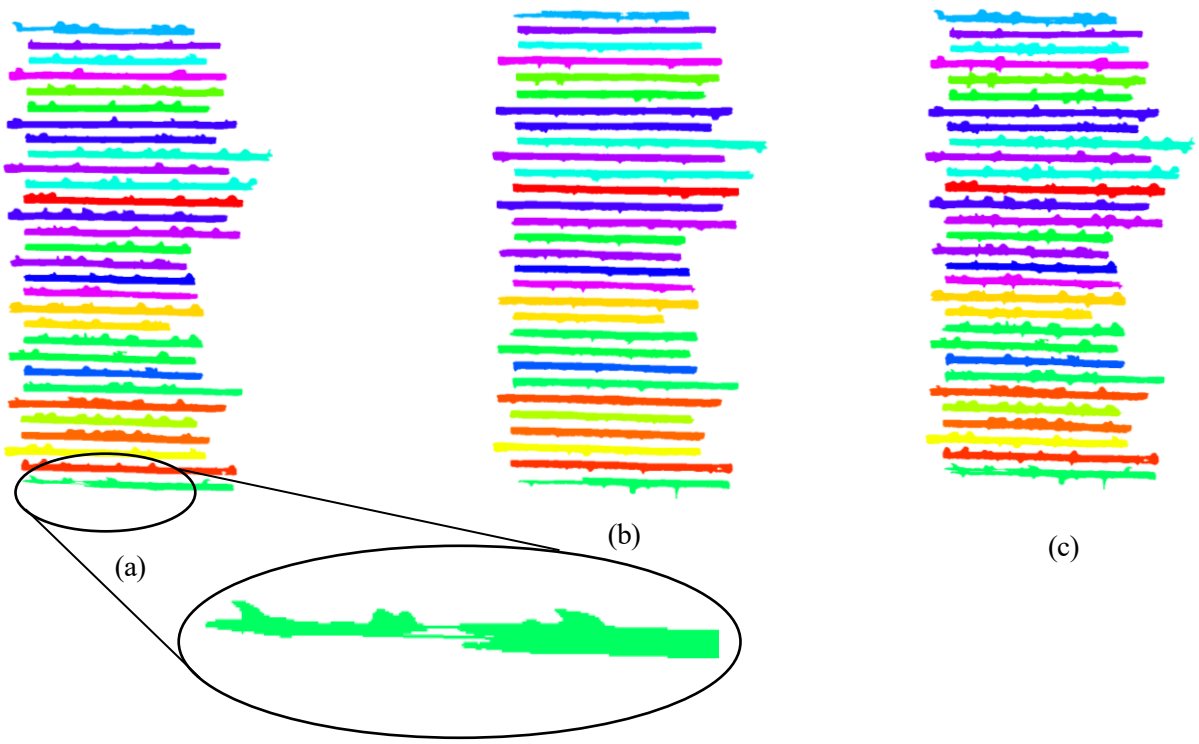


Figura 6-12. Corrección del Caso B en la imagen 030 del dataset DIVA-HisDB: (a) *MID-UP CLEAR*, donde se corrige el Caso B; (b) *MID-DOWN CLEAR*; y (c) *MÁSCARA* sin errores

Una vez corregido el Caso B de la imagen superior, se puede decir que la *máscara de segmentación* final ya es completamente correcta.

6.1.1.3 Caso C: Hueco en la línea detectado por ambas partes de la misma

En lo referido al Caso C, podemos decir que se da, al igual que el anterior, cuando en alguna línea del texto de la imagen encontramos un hueco. No obstante, para que el error se clasifique dentro del Caso C, el hueco debe ser detectado tanto en la imagen que contiene las partes superiores de las líneas, como en la que contiene las inferiores. Como se observa en la Figura 6-13, así quedaría el Caso C representado:

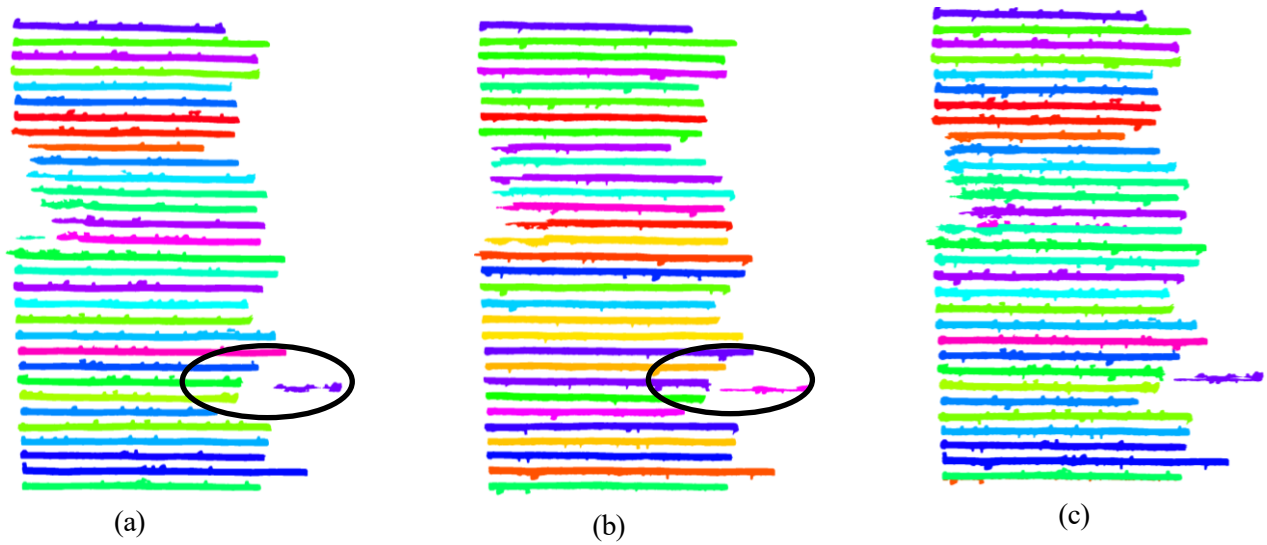


Figura 6-13. Caso C en la imagen 018 del dataset DIVA HisDB: (a) *MID-UP CLEAR*, donde se detecta el Caso C en la línea 26; (b) *MID-DOWN CLEAR* donde se detecta el Caso C en la línea 26; y (c) *MÁSCARA* errónea

Como se puede apreciar en la imagen superior, vemos que la línea 26, tanto en (a) como en (b), está dividida por un hueco, dando lugar al conocido Caso C (vemos que tenemos también un Caso B, en la línea 15 de (a), aunque en este análisis no es relevante su mención). Sin embargo, al tratarse de un caso "local", la *máscara de segmentación* obtenida en (c), es errónea solo en esa línea prácticamente. Por lo tanto, se puede comprobar que este caso no es tan destructivo como los dos anteriores, el A y el B.

En cuanto a la ***detección*** de dicho caso, decir que, de la misma manera que se hacía para los dos casos expuestos previamente, esta tarea se ejecuta en la función *fixFunction.* En primer lugar, se comprueba, mediante el uso de la función *detectSameHeight*, si existe un posible Caso C tanto en *MID-UP CLEAR* como en *MID-DOWN CLEAR*, ya que es solo en esta situación en la que se puede declarar la presencia de un Caso C en la imagen analizada. Para entender mejor la tarea desempeñada por esta última función, definida en *fixFunction.py*, a continuación se hará una explicación detallada de su funcionamiento.

El primer punto que se debe definir acerca de *detectSameHeight*, es sin duda su finalidad. Esta función, como su propio nombre indica, se encarga precisamente de detectar si dos líneas están a la misma altura. Para ello, recibe como parámetros, tanto la imagen a analizar, como el número de línea que está siendo analizada. A partir de aquí, su modo de operar es muy sencillo. Simplemente, extrae la línea con el posible Caso C y la siguiente, de manera que se pueda comprobar si ambas, la primera recorrida de izquierda a derecha y la segunda de derecha a izquierda, comparten un cierto número de píxeles a la misma altura. Es necesario establecer un umbral en lo referente al número de píxeles, ya que se puede dar el caso que haya líneas inferiores o superiores a la analizada, que cuenten con ciertas protuberancias capaces de compartir algunos píxeles con la línea estudiada, dando lugar así a un Caso C mal detectado. Seguidamente, si es cierto que ambas líneas extraídas comparten ese número impuesto de píxeles, o lo que es lo mismo, están a la misma altura, significa que en esa imagen se da la posibilidad de que exista un Caso C, por lo que pondremos una bandera, *flagD* a "1", indicando claramente este suceso.

Finalmente, volviendo a *fixFunction,* y habiendo comprobado que tanto para *MID-UP CLEAR* como para *MID-DOWN CLEAR*, las banderas *flagC_midUp* y *flagC_midDown*, respectivamente, devuelvan "1", se pasa a corregir este Caso. Realmente, la ***corrección*** es bastante sencilla, pues se vuelve a emplear la función *fixCaseB* (descrita previamente durante la explicación del Caso B), ya que ambos casos padecen el mismo problema, aunque como bien se sabe, en circunstancias distintas, lo que los hace diferentes. En la Figura 6-14, que se muestra a continuación, se puede observar la corrección del mismo.

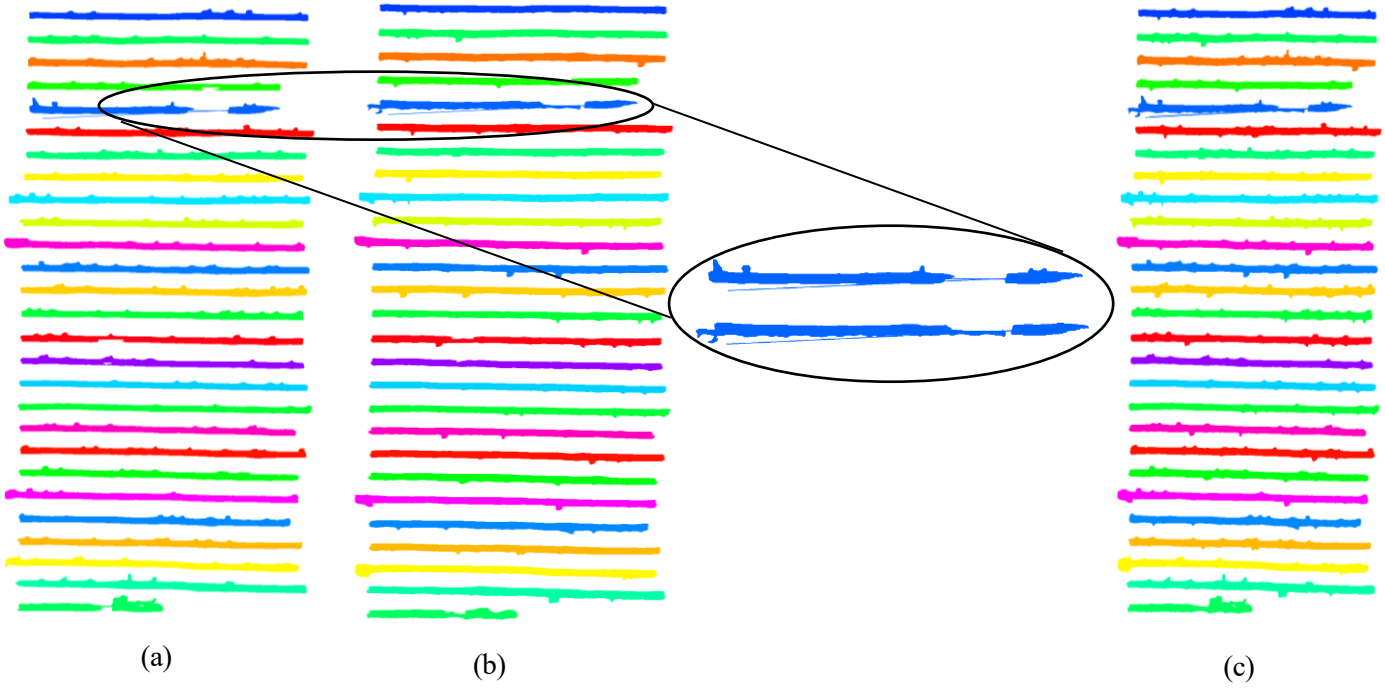
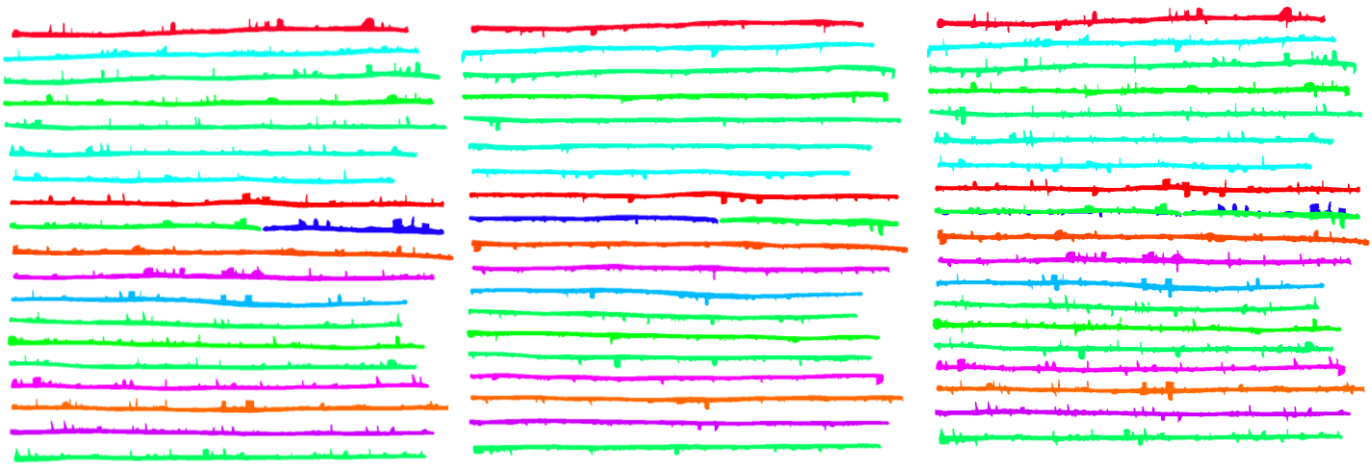


Figura 6-14. Corrección del Caso C en la imagen 092 del dataset DIVA-HisDB: (a) *MID-UP CLEAR*, donde se corrige el Caso C en la línea 5; (b) *MID-DOWN CLEAR* donde se corrige el Caso C en la línea 5; y (c) *MÁSCARA* sin errores

Como se puede observar en la Figura 6-14, al corregir el Caso C, la *máscara de segmentación* conseguida queda sin errores.

6.1.1.4 Caso D: Derivado del Caso C

Seguidamente, se presenta uno de los casos menos frecuentes encontrados en este proyecto: el Caso D, que no es más que una derivación del Caso C. Concretamente, el fallo lo comete la función empleada para etiquetar las CC (*scipy.ndimage.label*), que es la encargada de hacer ese proceso de etiquetado de izquierda a derecha, de manera que si nos encontramos un hueco en alguna línea de la imagen puede ser que en *MID-UP CLEAR* se etiquete primero la parte izquierda y después la derecha; y que en *MID-DOWN CLEAR* se haga al contrario. Esta numeración errónea haría que al construir la *máscara de segmentación*, ésta no fuera correcta, pero solo en esa línea. En la Figura 6-15 podemos ver este error de forma gráfica:



(a) (b) (c)

Figura 6-15. Caso D en la imagen 043 del dataset ICDAR 2013: (a) *MID-UP CLEAR*, donde se observa el Caso D en la línea 9; (b) *MID-DOWN CLEAR* donde se observa el Caso D en la línea 9; y (c) *MÁSCARA* errónea

En cuanto a la detección y corrección del Caso D, hemos decidido no plantearlas al ser un caso que tan solo se da en una imagen del dataset ICDAR 2013, siendo así un error puntual. Pues, como se aprecia en la imagen superior, el error no se extiende más allá de la línea donde fue detectado.

6.1.1.5 Caso F: Líneas con más de un hueco

Finalmente, presentamos el último de los casos de error encontrados durante el análisis del proyecto. Este caso se da cuando una misma línea, tanto de *MID-UP CLEAR* como de *MID-DOWN CLEAR*, presenta más de un hueco, o lo que es lo mismo, está fraccionada en varias líneas pequeñas. De esta manera, la *máscara de segmentación* obtenida será errónea. En la Figura 6-16, se puede ver este tipo de error.

En lo referido a la detección y corrección del Caso F, es preciso decir que, como es un caso que se da en muy pocas imágenes del dataset DIVA-HisDB, arreglarlo no supondría una mejora excesiva de la segmentación que lleva a cabo el algoritmo diseñado. De todas formas, se propone su corrección en la sección de mejoras del proyecto, ya que al hacerlo, mejoraría la robustez del sistema.

(a) (b) (c)

Figura 6-16. Caso F en la imagen 093 del dataset DIVA-HisDB: (a) *MID-UP CLEAR*, donde se observa el Caso F en la línea 14; (b) *MID-DOWN CLEAR* donde se observa el Caso F en la línea 14; y (c) *MÁSCARA* errónea

6.1.2 Resultados de la corrección de los errores detectados, con la base de datos ICDAR 2013

Una vez aplicadas las correcciones propuestas al proceso de segmentación, y habiendo éste concluido al terminar de analizar y segmentar cada imagen perteneciente al dataset ICDAR 2013, destacamos que:

- De las 348 imágenes que componen la base de datos, en 37 de ellas se han detectado uno o varios posibles casos de error.
- De esas 37 imágenes con presuntos errores detectados, 12 de ellas han sido devueltas con dichos errores corregidos. Entre las mismas encontramos, con anterioridad a su corrección, los siguientes errores:
    - 10 casos A, 1 caso B y 2 casos C.

  Por otra parte, de las 25 imágenes restantes, algunas de ellas no se han podido corregir de ninguna manera, en otras se detectaba la presencia de algún error pero no se indicaba de qué clase se trataba. No obstante, sí que es cierto que aquellas imágenes contienen algún tipo de error, probablemente de los anteriormente expuestos, pues las tres imágenes que forman parte de la *máscara de segmentación* final, cuentan con un número de líneas distinto, señal clara de error (en su mayoría se tratan de casos A no detectados). Aunque bueno, al final se tenían muchas otras en las que los posibles errores fueron corregidos de forma parcial .Por tanto, a continuación haremos un resumen de los errores encontrados:
    - 9 casos A, 8 casos B, 9 casos C y 5 casos sin error aparente, que serían aquellos nuevos errores planteados en el apartado anterior, o tratarse de un fallo del programa por falta de precisión, como veremos más adelante en el capítulo 8 de mejoras.

6.1.3 Resultados de la corrección de los errores con la base de datos DIVA-HisDB

Al terminar el algoritmo propuesto de ejecutar los procesos de detección y corrección de las imágenes segmentadas del dataset DIVA-HisDB, pasamos a analizar y estudiar los resultados conseguidos:

- De las 120 imágenes que componen la base de datos, en 62 de ellas se han detectado uno o varios posibles casos de error. Como vemos, la estructura de las imágenes de este dataset es mucho más compleja que la de las imágenes del dataset anterior, de manera que se detectarán muchos más errores que en ICDAR 2013.

- De esas 62 imágenes con presuntos errores detectados, 15 de ellas han sido devueltas con dichos errores corregidos. Entre las mismas encontramos, con anterioridad a su corrección, los siguientes errores:
    - 2 casos A, 6 casos B y 10 casos C.

  Por otra parte, de las 47 imágenes restantes, al igual que pasaba para el dataset de ICDAR 2013, tenemos algunas en las que los errores presentes no se han podido corregir, otras en que sí se han corregido parcialmente (esto lo podemos apreciar en la Figura 6-17), o se ha intentado corregir y por supuesto, existen otras imágenes en las que se detectaba la presencia de algún error pero no se indicaba de cuál se trataba. A continuación, recopilamos los errores encontrados:
    - 13 casos A, 20 casos B, 17 casos C y 19 casos sin error aparente, que podrían ser aquellos nuevos errores planteados en el apartado anterior, o tratarse de un fallo del programa por falta de precisión, como veremos más adelante en el capítulo 8 de mejoras.

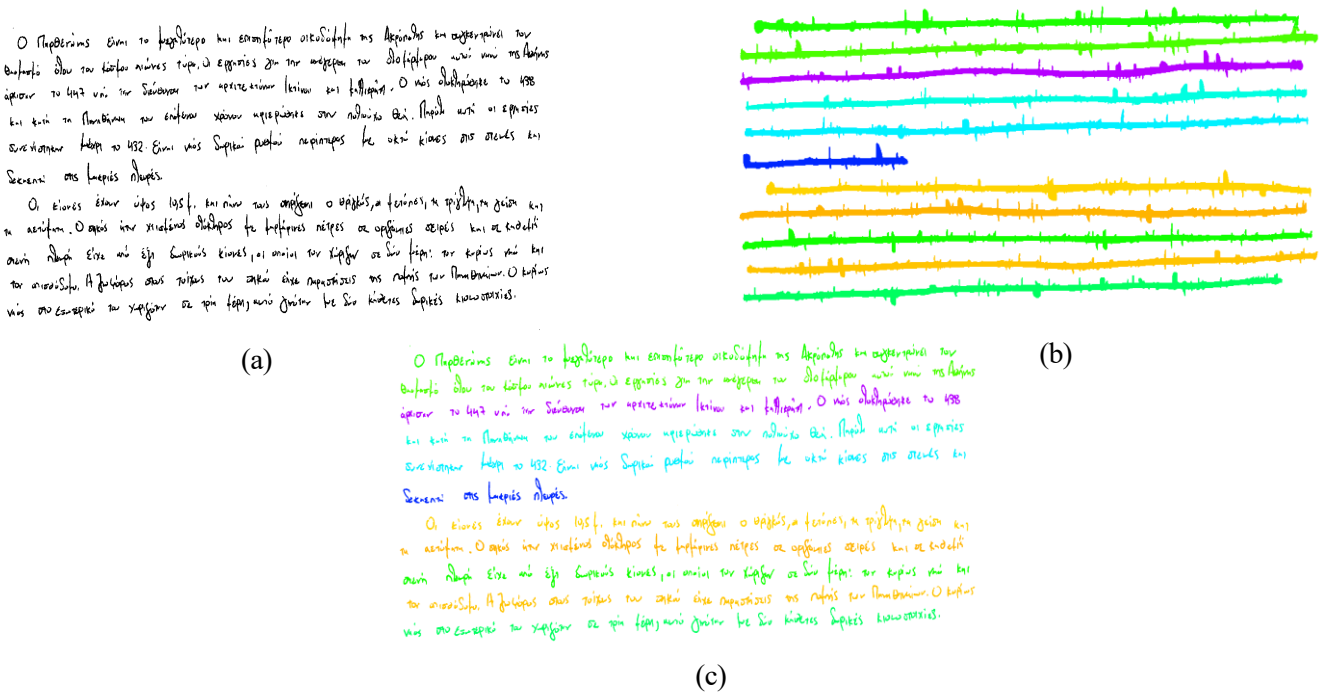
(a) (b)

Figura 6-17. Corrección parcial de los errores de la imagen 086 del dataset DIVA-HisDB (a) *MID-UP CLEAR*; y (b) *MID-DOWN CLEAR*

Si observamos la imagen superior, se puede apreciar que el Caso B de la línea 2 no ha sido corregido mientras que el Caso C de la línea 3 sí. Así pues, decimos que esta imagen ha sido parcialmente corregida.

6.2 Descripción del proceso de segmentación

Una vez obtenida la *máscara de segmentación* del proceso de *fusión* descrito en el Apartado 6.1, pasamos a realizar el propio proceso de *segmentación*. Este proceso viene recogido en la función *segmentation* del archivo *segmentation.py*, y consiste en la comparación, elemento a elemento de la imagen original con la *máscara de segmentación*. De esta manera, conseguimos construir una nueva imagen en la que los píxeles correspondientes al texto, tendrán el mismo valor que aquellos de la máscara situados en la misma posición. Podemos ver un ejemplo de dicho proceso en la Figura 6-18.



(a) (b)

(c)

Figura 6-18. Proceso de segmentación de la imagen 248 del dataset ICDAR 2013: (a) Imagen original; (b) *MÁSCARA*; y (c) Segmentación de (a)

6.2.1 Error encontrado durante el proceso de segmentación

Es muy común que en las imágenes resultantes del proceso de segmentación, ciertos píxeles de las mismas no hayan sido segmentados. Es decir, se trata de píxeles que forman parte de alguna línea de texto, por lo tanto suelen ser negros en la imagen original, o sea, tienen el valor 1. No obstante, en la imagen segmentada, aparecen sin ningún valor asignado.

Para arreglar este error, aquí se propone el diseño de un algoritmo que sea capaz de asignar a los píxeles no segmentados, el valor de la *máscara* de aquellos que estén más próximos. Sin embargo, dicho error en la segmentación no tiene una solución trivial, y eso se debe principalmente a los llamados *caracteres conflictivos*. Estos, como ya se había revisado previamente, están muy cerca de una línea a la que no pertenecen y por tanto si llevamos la asignación de valores por proximidad, descrita antes, sin tener en cuenta estos *caracteres*, es muy posible que nos equivoquemos. En la siguiente Figura 6-19, se muestran los dos casos existentes, para el dataset ICDAR 2013, de los píxeles posibles sin segmentar.

(a) (b)

Figura 6-19. (a) Recreación del final de la línea 24 de la imagen 055 del dataset ICDAR 2013; y (b) Recreación de la parte intermedia entre las líneas 10 y 11 de la imagen 032 del dataset ICDAR 2013

Como se aprecia en la Figura 6-19, se tiene:

- Error en el plano horizontal (a): La mala segmentación puede darse al inicio o al final de una línea, de forma que debería asignarse el valor de CC más próximo a la misma
- Error en el plano vertical (b): El error se encuentra en el plano vertical de la imagen, complicando la resolución del mismo.

Para corregir ambos tipos de error de una forma eficaz, se plantea un algoritmo iterativo de búsqueda de píxeles sin segmentar (lo podemos encontrar en la función *segmentation*). El patrón de búsqueda del algoritmo se basa en los siguientes pasos:

1. Primero se crea una variable llamada *segmentation* que contiene los píxeles con los valores de aquellos de la *máscara de segmentación,* así como los sin segmentar, pertenecientes a la imagen original. Estos últimos, al no estar segmentados, presentarán un valor de "1", que es el negro, como se veía antes en la Figura 6-19.
2. Una vez dentro del bucle de repetición, al ser este un algoritmo iterativo, se construye una matriz con todos los píxeles no segmentados, llamada *nonSegmentatedPixels*. Dicha matriz se va actualizando conforme se pasa a la siguiente iteración, evitando pues, posibles segmentaciones erróneas. El algoritmo se repetirá hasta que no quede ningún píxel sin segmentar.
3. Seguidamente, se comprueba si nos encontramos o no en un bucle infinito, situación que puede suceder al tratarse de un proceso iterativo, como ya se ha señalado previamente. Si es así, se para la ejecución del algoritmo y se indica que se ha podido evitar un bucle infinito con el número de píxeles no segmentados mayores que una cierta tolerancia (*TOLERANCE_NON_SEGMENTATED* = 0).
4. Si no se da el caso de bucle infinito, comenzamos con la asociación de valores a los píxeles no segmentados. En primer lugar, se crea un array con el mismo número de ceros que los píxeles no segmentados, llamado *newSegmentatedPixels.* A continuación, para cada píxel de la matriz *nonSegmentatedPixels*, con el valor "1", se le asigna el valor del CC más próximo.
5. Para ello, se emplea la función *findNearestCC*, recogida en *segmentation.py,* encargada de encontrar el CC más cercano. Esta función buscará, de izquierda a derecha, el valor requerido del CC más próximo en un rectángulo de búsqueda con dimensiones de 200 píxeles de ancho x 60 píxeles de alto. En caso de no encontrar dicho valor en primera instancia, el proceso de búsqueda procede así:
   - Se situará un píxel más arriba y volverá a buscar en la componente horizontal.
   - Si tampoco encuentra el valor deseado de esa forma, debe situarse un píxel más abajo e iniciar de nuevo la búsqueda en la componente horizontal.

   Así pues, la función planteada se encarga de inspeccionar primero la componente horizontal para después comenzar con la vertical.
6. Por tanto, una vez empleada dicha función, se comprueba si el valor devuelto por la misma es el adecuado. Si el valor conseguido es igual a "-1", indica que no se ha podido encontrar el CC más próximo, y de esta manera, la segmentación sería errónea por lo que se debería repetir el proceso. Si por el contrario, se consigue un valor distinto a "-1", podemos afirmar que se ha encontrado el valor del CC más cercano. Así pues, este último se almacenaría en el array *newSegmentatedPixels,* en la misma posición que el píxel sin segmentar. El algoritmo se repite hasta que no quede ningún píxel sin segmentar.
7. Finalmente, al terminar de asociar valores a los píxeles no segmentados, se completa la *máscara de segmentación* con ellos y se vuelve a segmentar, de forma que quede corregido este error.

Dicho proceso es el mismo para las dos bases de datos tratadas en este trabajo. No obstante, antes de emplear la función *segmentation* con las imágenes del dataset DIVA HisDB, se deben limpiar los bordes negros de las mismas, ya que de lo contrario tomaría dichos bordes como píxeles pendientes de segmentar, como se contempla en el paso número 1, que describíamos anteriormente. Para ello, decidimos emplear la función *cleanDIVAborders,* definida en el fichero *segmentation.py.* Esta función simplemente, extrae las coordenadas de la página de texto del ground-truth, las procesa para que coincidan con las coordenadas de la imagen de

entrada (recordemos que habíamos procesados las imágenes del dataset para que todas tuvieran las mismas dimensiones) y las vuelca en una matriz de ceros, es decir, en una imagen en blanco. De esta manera, tenemos una imagen sin bordes negros (o unos), donde lo único que existe de color negro son las letras el texto a segmentar.

Una vez que se obtiene la imagen segmentada y se guarda, pasamos a redimensionarla para que pueda servir como entrada del software de evaluación que veremos en el Apartado 6.3. Se guarda esta imagen tanto en formato *.png*, como en *.tif.dat* (solo para el dataset ICDAR 2013, ya que no se puede conseguir para el de DIVA-HisDB por motivos que se explicarán posteriormente). Este último formato es el que deben de tener las imágenes que hemos segmentado, con el fin de poder emplear el programa de evaluación con ellas, como se verá en el apartado que describimos a continuación.

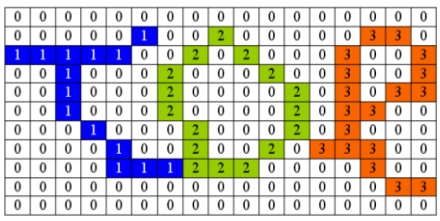
6.3 Resultados finales del proyecto para cada base de datos

En este apartado se exponen los resultados finales del proyecto desarrollado. Como ya se hacía en [3], para evaluar el rendimiento de nuestro algoritmo, emplearemos las medidas establecidas en las bases del concurso de ICDAR 2013, tanto para el dataset sacado del mismo, como para el de DIVA-HisDB. Así pues, podremos comparar los resultados de ambos con aquellos algoritmos que participaron en el concurso de 2013, de la misma manera que con el expuesto en [3].

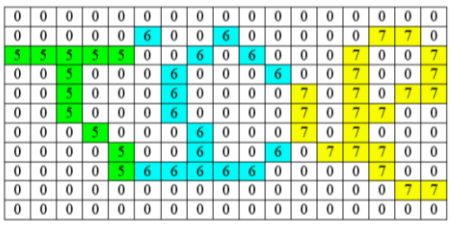
Para empezar, según se refiere en [26] y en [14], el método de evaluación del rendimiento del algoritmo desarrollado, se basa principalmente en el recuento del número de coincidencias presentes entre las entidades detectadas por dicho algoritmo, y aquellas encontradas en el ground-truth del dataset. De esta manera, se usa una tabla *MatchScore* que tiene como objetivo calcular el grado de similitud entre los píxeles de una región determinada del ground-truth con los de los resultados obtenidos. Así pues, dicha tabla, que mostramos en la Figura 6-20, se puede obtener siguiendo la fórmula expresada a continuación:

$$MatchScore\ (i,j) = \frac{T\ (\ Gj\ \cap\ Ri\ \cap\ I)}{T\ (\ (Gj\ \cap\ Ri)\ \cap\ I)} \qquad (6-1)$$

Donde I representa todos los píxeles de la imagen; G_j representa todos los píxeles contenidos en la región j del ground-truth; R_i representa todos los píxeles contenidos en la región i de los resultados obtenidos de la segmentación realizada por el algoritmo; $T(s)$ función que se encarga de contar los píxeles presentes en un conjunto s (total) [26]. En la Figura 6-20, que mostramos seguidamente, se puede apreciar un **ejemplo del cálculo** de la tabla $MatchScore\ (i,j)$, mencionada previamente, que encontramos en la página web del concurso de ICDAR 2013 [21]:



(a)



(b)

GT \ RES	5	6	7
1	.79	.11	0
2	0	.68	.08
3	0	0	.86

(c)

Figura 6-20. (a) Imagen que representa la segmentación del ground-truth, (b) Imagen que representa la segmentación del resultado obtenido y (c) que es la tabla *MatchScore (i, j)*

Como se puede apreciar, la Tabla MatchScore (Figura 6-20 (c)) se encarga de calcular la similitud entre una determinada región del ground-truth con otra de los resultados obtenidos por nuestro algoritmo. Así pues, una vez se han obtenido los valores mostrados en la tabla anterior se comparan con un cierto umbral, denominado umbral de aceptación del evaluador, T_a, de forma que si alguno de los valores recogidos en la Tabla MatchScore (Figura 6-20 (c)) es igual o superior al de T_a , se considera que la segmentación realizada de la zona estudiada es correcta, lo que en inglés se define como *one-to-one match* (o2o) [26]. Como su nombre indica, T_a es ajustable, y en este proyecto, al igual que en [3], el umbral escogido es el del 95%, pues es el que se seleccionó en el concurso de ICDAR 2013. Como no se tienen referencias para el dataset de DIVA-HisDB, se seleccionará, en principio, el mismo umbral para este dataset.

Finalmente, para evaluar el rendimiento de nuestro algoritmo, se deben determinar dos medidas clave, la tasa de detección (DR, detection rate) y la precisión de reconocimiento (RA, recognition accuracy) [26], que nos darán la *FM* (*Final Measurement*), métrica que realmente calculará el rendimiento conseguido por el algoritmo diseñado (se calcula para cada imagen que compone el dataset, y después se hace la media entre todas ellas para obtener la *FM* en valor medio). Estas métricas ya se definirieron previamente en el Apartado 2.2.1, mediante las ecuaciones (2-1) y (2-2).

Con el fin de poder obtener los valores medios de dichas medidas de forma automática, para el dataset analizado, se emplea un programa llamado ICDAR 2013 Handwriting Segmentation Contest Viewer and Evaluator v.2 [21]. En la Figura 6-21, mostramos los elementos de este programa de evaluación:

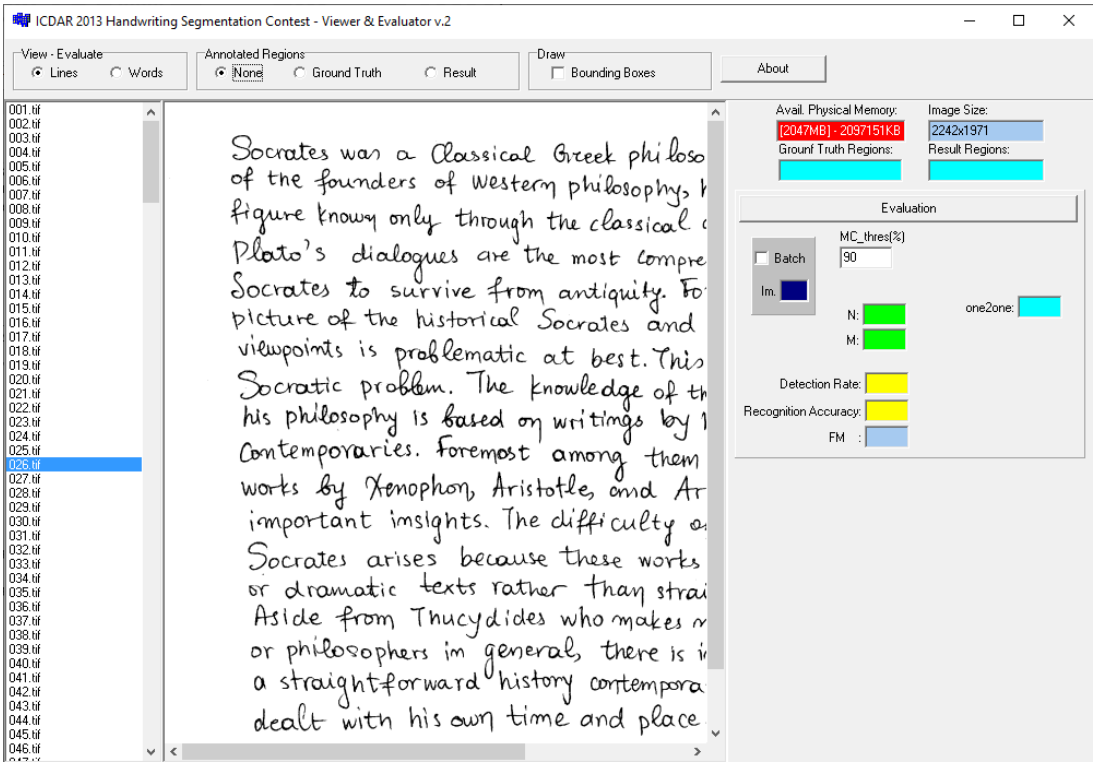


Figura 6-21. Programa de evaluación del rendimiento del algoritmo en términos de segmentación.

Como podemos ver en la imagen superior, se debe seleccionar el componente de segmentación a analizar, líneas o palabras. En nuestro caso seleccionamos las líneas. Seguidamente debemos introducir en las carpetas de *images*, *gt* y *results,* situadas en el directorio donde se ha instalado el programa, los archivos correspondientes en los formatos adecuados. En *images* introducimos las imágenes del dataset a analizar, mientras que en *gt* debemos colocar las líneas en formato *.tif.dat* (o lo que es lo mismo las *gt_lines* de las que se hablaba en secciones anteriores). Finalmente, en *results* debemos meter los resultados de la segmentación, también en formato *.tif.dat.* Sin embargo, antes de darle al botón de *Evaluation*, debemos fijar el *MC_thres*, que no es otro que el umbral T_a, y picar la casilla de *Batch* (en el caso de ICDAR 2013 el *Batch* tendría un valor de 348, mientras que para DIVA-HisDB, tendría un valor de 120) indicando así que vamos a analizar un conjunto de imágenes.

6.3.1 Resultados finales con la base de datos ICDAR 2013

En este subapartado se exponen los resultados de DR, RA y FM, obtenidos para el dataset ICDAR 2013. Como se puede apreciar en la Figura 6-22, que aparece a continuación, la DR alcanza una puntuación del 93.77%, mientras que la RA obtiene una del 93.27%.

Por otra parte, combinando las dos medidas anteriores, la FM consigue un valor del 93.51%. Posteriormente, se compararán estos resultados con otros obtenidos por los diferentes algoritmos que participaron en la competición de ICDAR 2013.

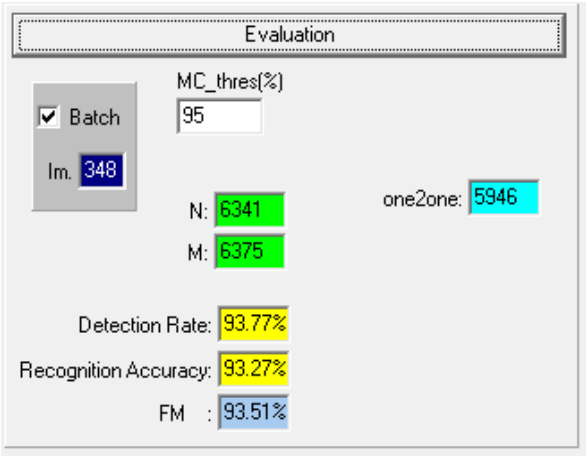


Figura 6-22. Resultados de la evaluación, obtenida gracias al programa proporcionado en la competición de ICDAR 2013 (*Batch = 348*), de la segmentación realizada por el algoritmo propuesto.

6.3.2 Resultados finales con la base de datos DIVA-HisDB

En relación a los resultados de segmentación del dataset DIVA-HisDB, decir que no se han podido evaluar con el evaluador del concurso de ICDAR 2013. Esto es debido a que, como se ha mencionado anteriormente, los archivos de resultados, así como los de ground-truth de las líneas, han de estar en un formato específico: *.tif.dat.* En un principio, se intentó cambiar el formato de estos archivos para que tuvieran la extensión necesaria para ser evaluados, pero no resultó efectivo. En su lugar, solo se conseguían archivos pesados que no contenían la información que dichos archivos tenían realmente.

Este problema de formatos no es la primera vez que surge en el proyecto. Sin ir más lejos, en el capítulo 4, en el que se detallaba la estructura de la base de datos DIVA-HisDB, destacábamos que en la función *funLines,* al extraer los archivos de ground-truth de las líneas, que en ICDAR 2013 se proporcionan a los participantes del concurso, estos no se pudieron guardar en *.tif.dat*, si no que fueron almacenados con extensión *.npz,* por los mismos problemas que ahora tenemos. Por tanto, esta situación, nos da pie a pensar que los archivos de ground-truth proporcionados por el concurso de ICDAR 2013 tienen una información añadida, que falta en los nuestros para DIVA-HisDB.

Así pues, como no disponemos de los formatos correctos en nuestros archivos, el evaluador no puede extraer el valor medio de las medidas recogidas en las ecuaciones (6-2) y (6-3), para los resultados de segmentación conseguidos con esta base de datos. De este modo, dejamos dicho asunto como mejora del proyecto, pues, por falta de tiempo no hemos podido solventarlo.

La idea, como bien explicaremos en el Capítulo 8, sería la de programar un evaluador específico para esta base de datos, que quizá debería construirse de forma parecida o con alguna influencia del trabajo expuesto en [2], del que hablábamos en el Capítulo 2. De manera que no tuviéramos que emplear esos archivos *gt_lines*, si no otra clase de ficheros más adecuada a la estructura de DIVA-HisDB, como se propone en ese mismo artículo.

6.4 Comparación de los resultados del algoritmo con los de otros algoritmos

Una vez evaluados los resultados de segmentación del dataset ICDAR 2013 (ya que no ha sido posible para los de DIVA-HisDB), se realizará una comparación entre estos y los obtenidos por otros algoritmos. Casi todos los algoritmos que compararemos participaron en el concurso de ICDAR 2013, por lo que podremos incluso señalar el puesto que ocuparía nuestro algoritmo si hubiera participado en esta competición. En la imagen inferior, Figura 6-23, se puede observar un gráfico con los resultados de FM para cada algoritmo de segmentación de líneas de texto:

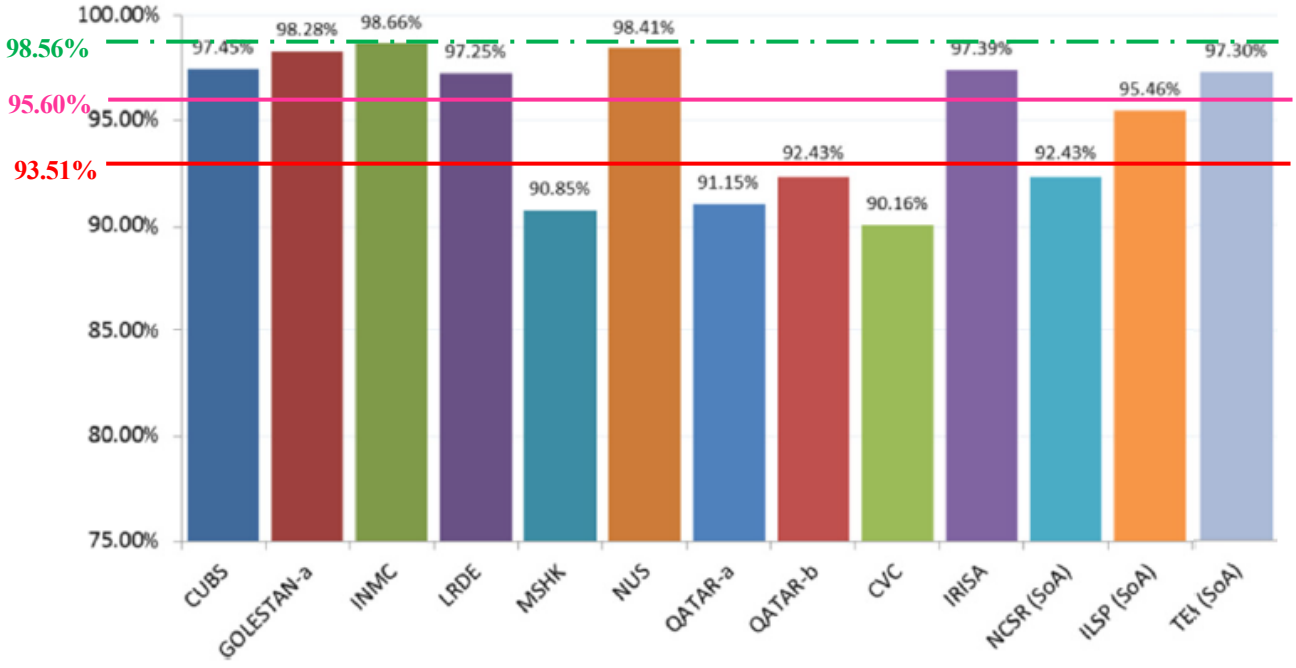


Figura 6-23. Evaluación del rendimiento de los algoritmos competidores en el concurso de ICDAR 2013 en lo referente a la segmentación de líneas de texto

En esta imagen, como se puede apreciar, se incluyen también: nuestro algoritmo, que es representado por una línea roja; el algoritmo recogido en [3] (ya que nos ha servido de referencia durante todo el proyecto), representado con una línea rosa; y por último el algoritmo recogido en [5], representado por una línea verde de puntos y rayas (para que sean visibles los resultados de los otros algoritmos a comparar).

Analizando los valores de FM alcanzados por los competidores de ICDAR 2013 y comparándolos con el conseguido por nuestro algoritmo, se puede decir que aunque el concurso se llevó a cabo hace 8 años, y hemos empleado una tecnología más nueva y pulida que la de 2013, el sistema propuesto se hubiera clasificado como el noveno mejor, teniendo todavía cinco algoritmos por debajo de él.

No obstante, si metemos en esta competición los algoritmos encontrados en [3] y [5], nuestro algoritmo se desplazaría hasta alcanzar la undécima posición en el ranking. En el caso de [5], se puede entender que obtenga una puntuación de FM más alta que la nuestra, ya que emplea la denominada técnica LAG, y esta es mucho más precisa y robusta que la desarrollada en este proyecto. Por otra parte, hay que destacar que la bajada en la FM obtenida por nuestro sistema, en comparación a la conseguida en [3], que si recordamos es el trabajo que nos ha servido como punto de partida para nuestro proyecto, se puede deber a que al implementar el aprendizaje de los contornos inferiores y superiores de forma conjunta en vez de separada (se genera un modelo en vez de dos), se pierda un poco de precisión a cambio de un menor gasto computacional y de recursos.

En conclusión, vistos los resultados obtenidos y el puesto conseguido en las dos clasificaciones planteadas, se puede afirmar que el rendimiento del algoritmo es bastante bueno.

7 CONCLUSIONES

En la introducción del proyecto se hacía referencia a la importancia de los textos manuscritos para la historia de la humanidad, y lo relevante que era su digitalización, de manera que muchas personas del campo de la investigación pudieran acceder a ellos , manipularlos de una forma más sencilla y por su puesto aplicar tareas de automatización en los mismos. Por esta razón, es muy importante que la segmentación de líneas de texto en aquellos documentos manuscritos digitalizados se lleve a cabo, pues permite que dichas tareas de automatización se puedan realizar sin problemas. Por supuesto, esto no es fácil, como hemos podido comprobar a lo largo de nuestro trabajo, y desde luego existen numerosas maneras de desarrollar algoritmos que se encarguen de la segmentación de las líneas de texto de manuscritos históricos. Sin embargo, en el trabajo planteado, se propone y revisa un solo algoritmo, de cuyo funcionamiento extraeremos una serie de conclusiones que se irán detallando conforme avancemos en el capítulo.

Como bien sabemos, nuestro algoritmo surge de la optimización del modelo propuesto en [3]. El método planteado, consiste en emplear una sola red neural para entrenar al algoritmo, de manera que ésta aprenda solamente a extraer el contorno superior de las líneas del texto, y para conseguir el contorno inferior de las mismas, simplemente se le pase la imagen rotada 180 grados. Gracias a ello, se reduce el consumo de recursos, pues se emplean dos modelos de red en vez de tres (como ocurría en [3]), aunque como hemos visto en el capítulo 6, se baja un poco la precisión de la segmentación final del texto.

Es en esta fase de entrenamiento y posterior test de la red, en la que se obtienen los primeros resultados e imágenes de "pre-segmentación". De este proceso se extrajeron diversos resultados cuyas conclusiones son las siguientes:

- **Modelo de Thick Backbone:** Al comparar los valores conseguidos para este modelo en las bases de datos analizadas en el proyecto, se puede decir que:
    - En la fase de entrenamiento, el rendimiento del algoritmo es ligeramente peor si empleamos el dataset ICDAR 2013 que si se utiliza el de DIVA-HisDB. No obstante, el funcionamiento con cualquier base de datos es bastante bueno.
    - En la fase de evaluación, nos encontramos que al emplear la base de datos DIVA-HisDB se observa una mejora importante en el funcionamiento del algoritmo, con respecto al presentado por el mismo con el dataset ICDAR 2013. Esta mejoría se aprecia especialmente en el valor del parámetro de pérdidas, que es mucho menor para el dataset de DIVA-HisDB.

  Por tanto, se puede afirmar que el modelo propuesto para el problema de Thick Backbone, presenta mejores resultados con el dataset de DIVA-HisDB, funcionando así, mejor con él.

- **Modelo de ZigZag Backbone:** Al comparar los valores conseguidos para este modelo en las bases de datos analizadas en el proyecto, se puede decir que:
    - En la fase de entrenamiento, es el dataset ICDAR 2013 el que hace que nuestro modelo se comporte mejor, pues presenta unos valores de pérdidas especialmente bajos, y aunque es cierto que el dataset DIVA-HisDB dota de mayor precisión al modelo, la diferencia en estos términos entre ambos datasets es ínfima. De todas formas, con cualquier base de datos, el funcionamiento del algoritmo es bastante bueno.
    - En la fase de evaluación, se puede afirmar que es el dataset DIVA-HisDB el que presenta los mejores valores para todos los parámetros estudiados en esta etapa. Por tanto, claramente es éste el que ofrece al algoritmo un mejor funcionamiento. Pero al igual que en la fase anterior, se podría usar cualquier dataset, pues la diferencia entre sus valores es muy pequeña.

  Luego, se puede decir que el funcionamiento del modelo propuesto, para el problema de ZigZag Backbone, es muy bueno con ambas bases de datos. Aun así, se destacan ligeramente los resultados conseguidos con el dataset DIVA-HisDB.

Una vez completada la fase de segmentación, nos ocupamos de la verdadera *tarea de segmentación* del texto, que es el siguiente paso a realizar con el fin de alcanzar el objetivo principal del proyecto. Para ello, es

fundamental la creación de una *máscara de segmentación*, que al ser comparada con la imagen original, nos devuelve la segmentación del texto deseada. Para poder ver el rendimiento del proceso de segmentación, evaluamos los resultados de la segmentación de las imágenes de la base de datos de ICDAR 2013, ya que como sabemos, el dataset de DIVA-HisDB requiere un evaluador específico que se ajuste a sus características. Así pues, al comparar dichos resultados de segmentación con los de otros algoritmos, se extrajeron las siguientes conclusiones:

- Nuestro algoritmo destaca por encima de otros cinco presentados al concurso, no obstante hay que señalar que emplea una tecnología mucho más nueva y pulida que la que usan los competidores de dicho concurso, aunque es cierto que es bastante más simple.
- Al comparar el algoritmo propuesto en nuestro proyecto con los presentados en [3] y [5], nos dimos cuenta que aunque habíamos reducido el gasto computacional de la red al generar dos modelos en vez de tres, estábamos perdiendo precisión, a pesar de haber corregido un caso de error que en [3] no estaba ni siquiera detectado. Además, volvemos a notar que debido a la simplicidad del sistema construido, la robustez del mismo se ve afectada.
- También se aprecia, como el algoritmo desarrollado es poco robusto en la corrección y detección de errores, aunque sí bastante preciso en esta tarea.
- Sin embargo, los resultados conseguidos para las medidas descritas anteriormente, ponen de manifiesto que el rendimiento del algoritmo es bastante notable.

Por lo tanto, a la vista de todo lo conseguido durante la realización del proyecto, se puede afirmar que el algoritmo desarrollado presenta un funcionamiento muy bueno. Aun así, debemos plantear una serie de mejoras para hacerlo todavía más competitivo. Dichos puntos de mejora se detallan en el capítulo siguiente.

8 MEJORAS DEL PROYECTO

Como se ha señalado en el Capítulo 7, ha quedado claro que el algoritmo desarrollado, a pesar de ser bastante eficiente, presenta fallos en varios puntos de su funcionamiento. Por tanto, en este Capítulo nos centraremos en enumerar varias ideas que, desarrolladas, podrían mejorar el rendimiento del algoritmo propuesto. Es preciso señalar, que las mejoras que aquí se exponen no se han podido implementar en este trabajo debido a la falta de tiempo o preparación para llevarlas a cabo.

A continuación se detallan las 4 mejoras propuestas.

8.1 Primera mejora: Aumentar la precisión de la máscara de segmentación

La primera mejora propuesta para el algoritmo desarrollado en este proyecto, se centra en el aumento de la precisión de las máscaras de segmentación conseguidas, manifestándose de la siguiente forma:

Al analizar las segmentaciones de las imágenes pertenecientes a ambas bases de datos, hemos podido comprobar que en los casos donde algunas letras son muy largas, la precisión de dicha segmentación no es demasiado buena. Podemos observar esta situación en la Figura 8-1:

Figura 8-1. Fragmento de las líneas 9, 10 y 11 de la segmentación de la imagen 031 del dataset ICDAR 2013

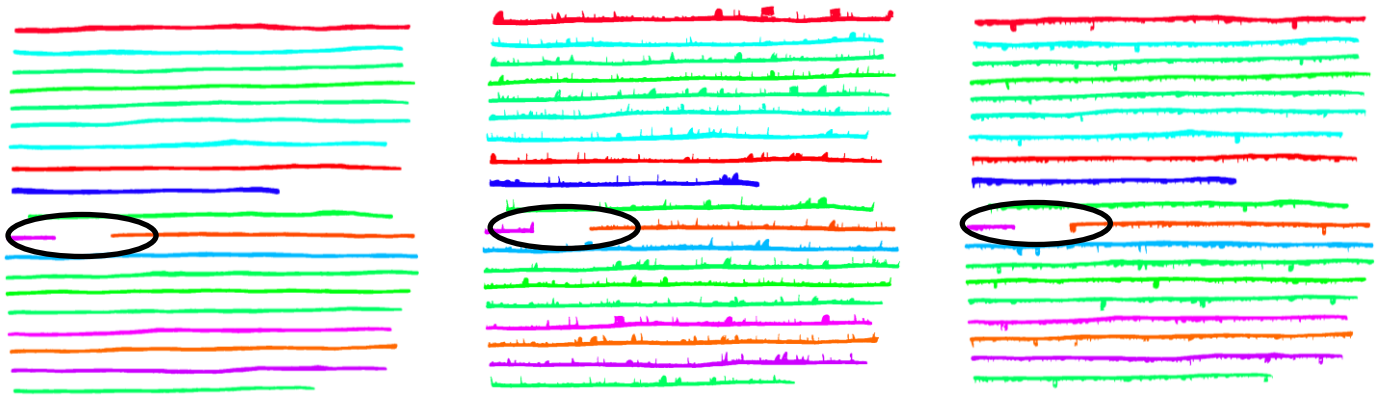
Como se puede apreciar en la imagen superior, vemos como el algoritmo segmenta parte las letras más largas como pertenecientes a la línea justamente inferior, poniendo de manifiesto la falta de precisión de la *máscara de segmentación* obtenida. Dicha falta de precisión, podría deberse a la implementación de un algoritmo como el nuestro, que se centra en el concepto de segmentar, con una misma red, tanto el contorno superior como el inferior de las líneas que forman el texto de la imagen. Si en lugar de este código tan sencillo, se hubieran utilizado las ideas planteadas en [5], donde se hace referencia a la tecnología LAG, nuestra segmentación hubiera sido muchísimo más exacta.

8.2 Segunda mejora: Mejorar los procesos de detección y corrección de errores

En cuanto a la segunda mejora se debe destacar que, si se llevara a cabo, sería la que más beneficios traería tras su aplicación. No obstante, es conveniente señalar que su desarrollo podría ser algo laborioso, como cualquier tarea de optimización.

Antes de empezar a definir esta mejora, es necesario resaltar que, como se ha comprobado en capítulos anteriores, la función de detección de errores desempeñada por nuestro algoritmo, si bien es precisa, no es demasiado robusta. Esto se debe a que, como veíamos en la función *funSegFus*¸ la condición para que una imagen sea detectada como posible portadora de algún tipo de error, y sea procesada para clasificarlos, es que el número de CC no sea el mismo entre las imágenes *MID CLEAR--MID-UP CLEAR--MID-DOWN CLEAR*. O lo que es lo mismo, que la resta entre el número de CC de estas imágenes sea distinta de "0". Dicho proceso para seleccionar qué imágenes podrían tener algún tipo de error, no funciona siempre, no es robusto. De hecho existen varias situaciones en las que la condición de detección no sirve para su cometido:

1. La primera situación encontrada que se salta esta condición es la siguiente: Si en una imagen del dataset tenemos un Caso A, que como sabemos, disminuye en una unidad el número de CC de la imagen, y también un Caso B, que aumenta en una unidad el número de CC de la misma, de manera que sea par el número de errores presente en ella, la resta de los CC de *MID CLEAR-MID-UP CLEAR-MID-DOWN CLEAR*, sería "0". Por tanto, la imagen analizada nunca se procesaría por *fixFunction.py*, y en consecuencia, sus errores jamás serían detectados ni mucho menos corregidos. Así pues, se pone de manifiesto que haría falta una condición de detección mucho más robusta.
2. La segunda situación, es muy parecida a la anterior, solo que en esta ocasión nos encontramos con uno o varios casos del tipo C en las tres imágenes. Como tienen el mismo número de CC, la resta es "0" y la imagen nunca se procesaría. Por lo tanto, se reafirma la necesidad de mejorar dicha condición de detección, aunque como veremos en la tercera situación, habría que tener en cuenta otros puntos adicionales. En la Figura 8-2, se muestra un ejemplo de la segunda situación:



(a) (b) (c)

Figura 8-2.(a) *MID CLEAR*; (b) *MID-UP CLEAR*; y (c) *MID-DOWN CLEAR* de la imagen 204 del dataset ICDAR 2013

3. La tercera situación encontrada se centra en la detección de un error falso, o "falsa alarma". Este problema puede aparecer de dos formas:

    - Puede ocurrir que en alguna de las imágenes que componen el trío anterior (*MID CLEAR, MID-UP CLEAR* o *MID-DOWN CLEAR*), se haya eliminado un CC con un área menor a la tolerancia impuesta en el código. Como ese CC desaparecería de una o de dos de las imágenes, la resta entre sus correspondientes CC sería distinta de "0", dando lugar a la detección de un error, a todas luces falso.
      Esto se produce debido a que la eliminación de CC pequeños, llevada a cabo por la función *removeSmallCC*, se basa fundamentalmente en una medida de las áreas de los mismos. El inconveniente es que existen líneas que son muy pequeñas y se quedan por debajo de la tolerancia, como bien explicamos antes. Esta situación podría solventarse al introducir un mecanismo que permitiera a la red neural adjuntar a su predicción de cualquier línea, un porcentaje que representara la certeza que tiene la red de haber detectado una posible línea. Podemos observar este problema en la Figura 8-3:

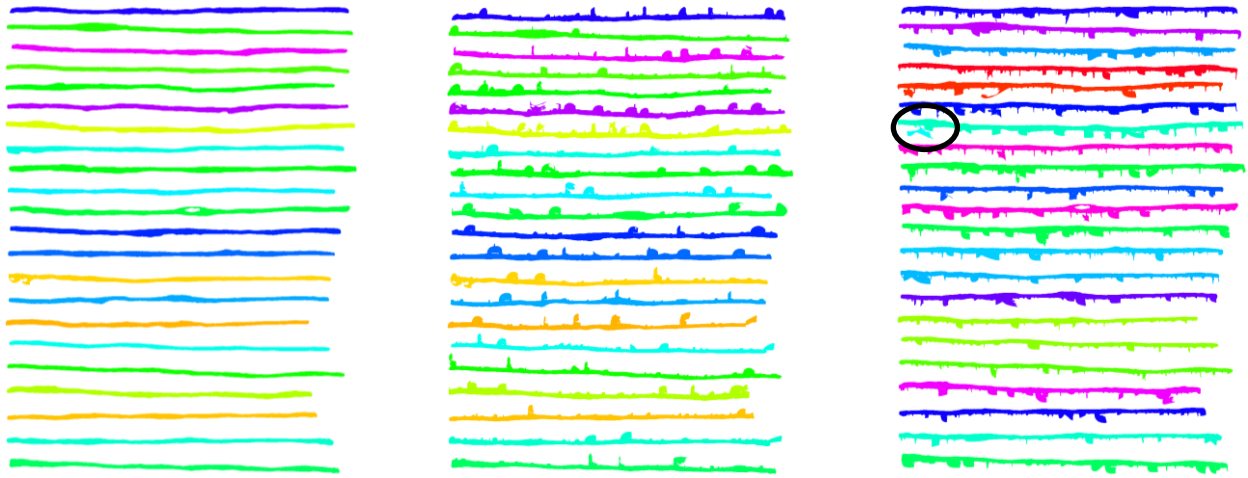


Figura 8-3.(a) *MID CLEAR*; (b) *MID-UP CLEAR*; y (c) *MID-DOWN CLEAR* de la imagen 347 del dataset ICDAR 2013

- Otra forma de "falsa alarma", se da cuando en el Thick Backbone del texto (*MID CLEAR*) hay un error del tipo C, mientras que en *MID-UP CLEAR* y *MID-DOWN CLEAR* no hay rastro de él. Además, cuando se forma la *máscara de segmentación*, fruto de la unión de las tres imágenes anteriores, ésta tampoco presenta ningún tipo de error. Ello se debe a que no se analiza ni corrige la imagen de *MID CLEAR*, simplemente nos remitimos a comprobar los errores que hay en las otras dos imágenes. Dicha situación podría solventarse de forma parecida al Caso C planteado, aunque habría que replantear el código para que trabajara con *MID CLEAR* también. Podemos apreciar la situación descrita, en la Figura 8-4 que mostramos a continuación:

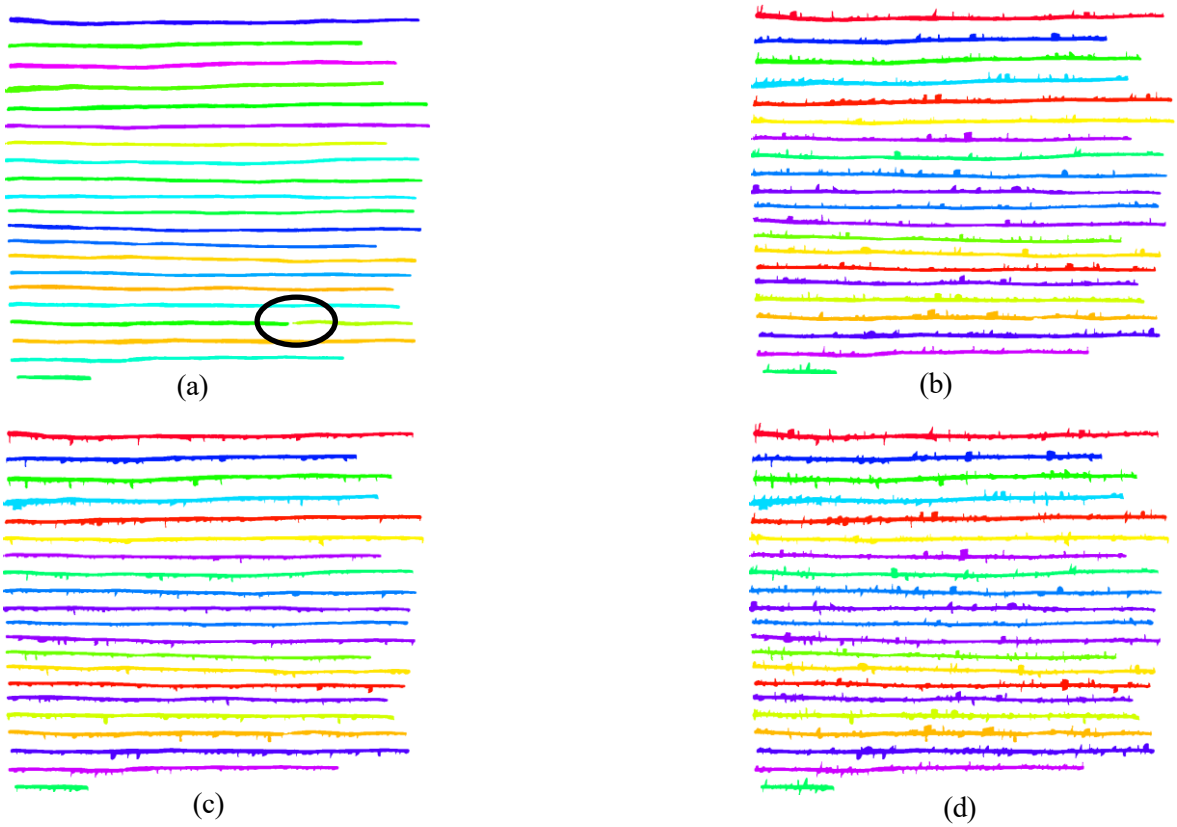


Figura 8-4.(a) *MID CLEAR*; (b) *MID-UP CLEAR*; (c) *MID-DOWN CLEAR;* y (d) *MÁSCARA* de la imagen 111 del dataset ICDAR 2013

Por otra parte, de la misma manera que el mecanismo de detección de errores del algoritmo no es todo lo robusto que hubiéramos querido, el mecanismo de corrección de dichos errores también es bastante simple. Como ya se ha descrito en capítulos anteriores, la corrección de los casos A, B y C se ha basado exclusivamente en el uso de rectas de regresión. Si bien es cierto que en el C y en el B hemos introducido la condición de que se comprobara la posición de las líneas a estudiar, los umbrales y tolerancias impuestas para distinguir la presencia de ambos casos y así corregirlos siguen siendo poco robustos. Por tanto, se debería incidir en el desarrollo de un método de corrección más complejo que se basara, por ejemplo, en el uso de los contornos superiores e inferiores de las imágenes para averiguar donde empiezan y acaban las líneas que forman parte de su texto. De ese modo, la recta de regresión construida sería más precisa y en consecuencia, la corrección de la imagen analizada.

8.3 Tercera mejora: Arreglar el Caso F

En el Capítulo 6 se describía el Caso F, señalándose que el algoritmo propuesto en este proyecto no arreglaba ni detectaba dicho error. En esta parte se propone, como tercera mejora, su detección y corrección, de forma que al plantearla, aunque no haya muchos casos del estilo, haría mucho más robusta la segmentación de las imágenes. Ello se debe a que, de una manera u otra, este Caso tienen un peso importante en la puntuación de las métricas del evaluador, ya que implica que la imagen en la que se detecta presente varios huecos en las líneas de la misma, haciendo que la red neuronal detecte dichas líneas como 3 o 4 diferentes, en vez de una sola.

8.4 Cuarta mejora: Programación de un evaluador para la base de datos DIVA HisDB

La cuarta y última mejora propuesta, es la de desarrollar y programar un evaluador que saque los valores medios de las medidas reflejadas en las ecuaciones del Capítulo 2, (2-1) y (2-2), para las segmentaciones de las imágenes que conforman la base de datos DIVA-HisDB (de manera que se pueda evaluar el rendimiento del algoritmo en la tarea de segmentación), que por falta de tiempo, no se ha podido realizar en este proyecto. Una solución para desarrollar dicho evaluador, podría ser plantearlo de forma parecida al de [2], ya que en este artículo, como hemos visto en el Capítulo 2, también se trabaja con la base de datos DIVA-HisDB.

REFERENCIAS

[1] Li, X. H., Yin, F., Xue, T., Liu, L., Ogier, J. M., & Liu, C. L. (2019). Instance aware document image segmentation using label pyramid networks and deep watershed transformation. Proceedings of the International Conference on Document Analysis and Recognition, ICDAR, 514–519. https://doi.org/10.1109/ICDAR.2019.00088

[2] Alberti, M., Vogtlin, L., Pondenkandath, V., Seuret, M., Ingold, R., & Liwicki, M. (2019). Labeling, cutting, grouping: An efficient text line segmentation method for medieval manuscripts. Proceedings of the International Conference on Document Analysis and Recognition, ICDAR, 1200–1206. https://doi.org/10.1109/ICDAR.2019.00194

[3] Mendoza, J.L., Murillo, J.J., (2020). Trabajo de Fin de Grado en Ingeniería de las Tecnologías de Telecomunicación. Herramienta para segmentación de líneas de texto en imágenes basada en Deep Learning

[4] Du, X., Pan, W., & Bui, T. D. (2009). Text line segmentation in handwritten documents using Mumford-Shah model. Pattern Recognition, 42(12), 3136–3145. https://doi.org/10.1016/j.patcog.2008.12.021

[5] Vo, Q. N., Kim, S. H., Yang, H. J., & Lee, G. S. (2018). Text line segmentation using a fully convolutional network in handwritten document images. IET Image Processing, 12(3), 438–446. https://doi.org/10.1049/iet-ipr.2017.0083

[6] Amarnath, R., Nagabhushan, P., & Javed, M. (n.d.). Word and character segmentation directly in run-length compressed handwritten document images, 1-17.

[7] Gruning, T., Labahn, R., Diem, M., Kleber, F., & Fiel, S. (2018). READ-BAD: A new dataset and evaluation scheme for baseline detection in archival documents. Proceedings - 13th IAPR International Workshop on Document Analysis Systems, DAS 2018, 351–356. https://doi.org/10.1109/DAS.2018.38

[8] O'Shea .K.T & Nash .R (2015). An Introduction to Convolutional Neural Networks. ArXiv e-prints.

[9] Ronneberger, O., Fischer, P., & Brox, T. (2015). U-Net: Convolutional Networks for Biomedical Image Segmentation, 1–8.

[10] Shorten, C., (2019).Introduction to ResNets. Accessed 25 April 2021. Available at: https://towardsdatascience.com/introduction-to-resnets-c0a830a288a4

[11] Siu, C. (2019). Residual networks behave like boosting algorithms. Proceedings - 2019 IEEE International Conference on Data Science and Advanced Analytics, DSAA 2019, 31–40. https://doi.org/10.1109/DSAA.2019.00017

[12] Kabonire, R., (2018). Residual Network Architecture. Accessed 25 April 2021. Available at: https://medium.com/datadriveninvestor/residual-network-architecture-8e478adabfec

[13] Ugarte, J., Murillo, J.J., (2019). Final Master Project in Master in Telecommunications Engineering. Deep Learning: segmentation of documents from the Archivo General de Indias with DhSegment and

NeuralLineSegmenter

[14] Stamatopoulos, N., Gatos, B., Louloudis, G., Pal, U., & Alaei, A. (2013). ICDAR 2013 handwriting segmentation contest. In Proceedings of the International Conference on Document Analysis and Recognition, ICDAR. https://doi.org/10.1109/ICDAR.2013.283

[15] Oliveira, S. A., Seguin, B., & Kaplan, F. (2018). DhSegment: A generic deep-learning approach for document segmentation, 7–12. https://doi.org/10.1109/ICFHR-2018.2018.00011

[16] Jia Deng, Wei Dong, Socher, R., Li-Jia Li, Kai Li, & Li Fei-Fei. (2009). ImageNet: A large-scale hierarchical image database. https://doi.org/10.1109/cvprw.2009.5206848

[17] Simistira, F., Seuret, M., Eichenberger, N., Garz, A., Liwicki, M., & Ingold, R. (2016). DIVA-HisDB: A precisely annotated large dataset of challenging medieval manuscripts. In Proceedings of International Conference on Frontiers in Handwriting Recognition, ICFHR. https://doi.org/10.1109/ICFHR.2016.0093

[18] Schone, P., Hargraves, C., Morrey, J., Day, R., & Jacox, M. (2018). Neural text line segmentation of multilingual print and handwriting with recognition-based evaluation. In Proceedings of International Conference on Frontiers in Handwriting Recognition, ICFHR. https://doi.org/10.1109/ICFHR-2018.2018.00054

[19] He, K., Zhang, X., Ren, S., & Sun, J. (2016). Deep residual learning for image recognition. In Proceedings of the IEEE Computer Society Conference on Computer Vision and Pattern Recognition. https://doi.org/10.1109/CVPR.2016.90

[20] Alberti, M., Pondenkandath, V., Wursch, M., Ingold, R., & Liwicki, M. (2018). DeepDIVA: A highly-functional python framework for reproducible experiments. In Proceedings of International Conference on Frontiers in Handwriting Recognition, ICFHR. https://doi.org/10.1109/ICFHR-2018.2018.00080

[21] ICDAR 2013 Handwriting Segmentation Contest -Resources. Users.iit.demokritos.gr. 2013 [on-line]. Available at: https://users.iit.demokritos.gr/~nstam/ICDAR2013HandSegmCont/resources.html

[22] Diem, M., Kleber, F., Sablatnig, R., & Gatos, B. (2019). CBAD: ICDAR2019 competition on baseline detection. In Proceedings of the International Conference on Document Analysis and Recognition, ICDAR. https://doi.org/10.1109/ICDAR.2019.00240

[23] CBAD: ICDAR2019 competition on baseline detection -Resources. Zenodo. 2019 [on-line]. Available at: https://doi.org/10.5281/zenodo.2567397

[24] Pletschacher, S., & Antonacopoulos, A. (2010). The PAGE (Page Analysis and Ground-truth Elements) format framework. In Proceedings - International Conference on Pattern Recognition. https://doi.org/10.1109/ICPR.2010.72

[25] Simistira, F., Seuret, M., Eichenberger, N., Garz, A., Liwicki, M., & Ingold, R. (2016). DIVA-HisDB: A precisely annotated large dataset of challenging medieval manuscripts. In Proceedings of International Conference on Frontiers in Handwriting Recognition, pp. 471-476. Resources. [on-line]. Available at: https://diuf.unifr.ch/main/hisdoc/diva-hisdb

[26] Gatos, B., Stamatopoulos, N., & Louloudis, G. (2010). ICFHR 2010 handwriting segmentation contest. In Proceedings - 12th International Conference on Frontiers in Handwriting Recognition, ICFHR 2010. https://doi.org/10.1109/ICFHR.2010.120

[27] Pérez, E. (2018). Notebloc, escanea documentos desde el móvil con este sencillo y preciso digitalizador. [online]Xatakandroid.com. Disponible en: https://www.xatakandroid.com/aplicaciones-android/notebloc-escanea-documentos-desde-el-movil-con-este-sencillo-y-preciso-digitalizador

[28] Tizaylapiz.com .Cómo se escribe una carta.[online] Disponible en: https://www.tizaylapiz.com/2016/05/como-se-escribe-una-carta.html

[29] "Review: DeepLabv1 & DeepLabv2 — Atrous Convolution (Semantic Segmentation)." [Online]. Available at: https://towardsdatascience.com/review-deeplabv1-deeplabv2-atrousconvolution-semantic-segmentation-b51c5fbde92d

[30] Kumar, H., 2021. Loss vs Accuracy. [online] Kharshit.github.io. Available at: https://kharshit.github.io/blog/2018/12/07/loss-vs-accuracy [Accessed 27 May 2021].

[31] Team, K., 2021. Keras documentation: Metrics. [online] Keras.io. Available at: https://keras.io/api/metrics/ [Accessed 27 May 2021].

[32] 3.3.9.8. Labelling connected components of an image Scipy lecture notes. Scipy lectures.org [online]. Available at: https://docs.scipy.org/doc/scipy/reference/generated/scipy.ndimage.label.html.

ANEXO A: ESTRUCTURA DE DIRECTORIOS DEL CÓDIGO

En este Anexo se va a detallar el sistema de ficheros en el que se estructura el código adjuntado a la memoria del proyecto. Para empezar, decir que encontramos el código dispuesto en tres directorios raíces o principales, que contienen los siguientes archivos:

1. **Directorio "algorithm"**

   Contiene los Jupyter Notebooks, *CNN_ZigZag.ipynb* y *CNN_thickBackbone.ipynb*, listos para cargarlos en Google Colab, y empezar las fases de entrenamiento y test de los modelos creados en ellos.

2. **Directorio "DataBases"**

   Contiene los dos datasets seleccionados, ICDAR 2013 y DIVA-HisDB, cuya estructura ha sido modificada. Estos aparecen comprimidos para ocupar el menor espacio posible.

3. **Directorio "lineseg"**

   Contiene los subdirectorios necesarios para llevar a cabo los objetivos del proyecto, relatados en el Capítulo 1 de la memoria:

   a) **Directorio "models"**

      Contiene los modelos resultantes del entrenamiento de Thick Backbone y ZigZag Backbone en formato *.json*, y sus parámetros en formato *.h5*.

      En total habría 4 pares *json-h5* en este directorio, pues se ha trabajado con 2 datasets: ICDAR 2013 y DIVA-HisDB.

   b) **Directorio "PythonFiles"**

      Contiene todos los archivos *.py* empleados en el desarrollo del proyecto, los cuales han sido descritos en capítulos anteriores:

      - *groudTruthFunctions.py:* contiene todas las funciones que extraen los ficheros de ground-truth de la imagen original, tanto para el problema de Thick Backbone, como para el de ZigZag Backbone. Además de estas funciones, contiene otras encargadas de la creación de los tres tipos de tensores mencionados en el capítulo 4 (tensores de entrada, y los tensores de Thick Backbone y de ZigZag Backbone).
      - *computeAllSamples.*py: es el fichero encargado de crear los archivos de ground-truth, así como de generar los tensores deseados.
      - *nLinesAndShapeAnalyzer.py*: es el fichero que genera las estadísticas que vimos en la sección 4.2.1 de la memoria. Un ejemplo del archivo *.txt* generado como resultado de la ejecución de este fichero, lo podemos encontrar en este directorio (se trata de las estadísticas para el dataset DIVA-HisDB).
      - *Unet_2048.py*: contiene la definición de la arquitectura de red empleada.
      - *DataGen_2048_dataAug_rotated.py:* contiene el código del *Data Generator* que

hemos empleado para generar los datos de entrenamiento, validación y test, con los que se ha alimentado a los modelos creados en los notebooks anteriores.

- *segmentation.py*: fichero que contiene las funciones necesarias para realizar la segmentación de las imágenes de cualquiera de los datasets elegidos.
- *fixFunction.py:* fichero que contiene las funciones necesarias para la detección y corrección de los errores derivados del proceso de fusión visto en el Capítulo 6.

c) Directorio "Results"

Contiene las imágenes resultantes del:

- Proceso de segmentación realizado por nuestro algoritmo. Encontramos dos carpetas con resultados para las dos bases de datos seleccionadas.
- Proceso de segmentación en blanco y negro. Se muestran algunos ejemplos del tipo de salida y entrada que recibe el algoritmo, así como, en el caso del problema de Thick Backbone, de la predicción realizada por el mismo. Todo ello clasificado en las carpetas: results_TBB y results_ZBB.

d) Directorio "tensors"

Contiene los tensores generados y comprimidos, para cada base de datos seleccionada: ICDAR 2013 y DIVA-HisDB.

ANEXO B: CÓDIGOS DEL PROYECTO

En este Anexo se van a adjuntar lo códigos empleados para la realización del proyecto. Estos son aquellos que se describían anteriormente en el Anexo A.

B.1 Códigos del directorio */lineseg/PythonFiles/*

B.1.1 Código de *groundTruthFunctions.py*

```
'''
    Created by Jose Luis Mendoza for his Final Disertation,
    "A tool for text lines segmentation in images bases on deep learning"
    2019-2020
    University of Seville, Spain
    Modified by Julia Moreno Casanova

'''
from PIL import Image
import random, matplotlib.pyplot as plt, matplotlib as mpl
import numpy as np
import os, errno
from skimage.transform import downscale_local_mean
from lxml import objectify
import xml.etree.ElementTree as ET
import cv2

#import matplotlib.pyplot as plt

from matplotlib.path import Path
from IPython import get_ipython
#get_ipython().run_line_magic('matplotlib', 'qt')




def createCmap():
    """This function returns the color map that will be used to plot and save
    figures.

    Returns:
    cmap:the color map

    """
    # We take the gist_rainbow color map which already exists
    cmap = plt.cm.gist_rainbow
    # We extract the 256 colors there are in form of a list
    cmaplist = [cmap(i) for i in range(cmap.N)]
    # We shuffle the list with this particular random seed
    random.seed(4)
    random.shuffle(cmaplist)
    # We force the first color to be white, which belongs to the background
    cmaplist[0] = ('w')
```

```
    # We create the new color map
    cmap = mpl.colors.LinearSegmentedColormap.from_list(
            'Custom cmap', cmaplist, cmap.N)

    return cmap

#############################################################################
##

def createCmapZigZag(image):
    """This function return the color map that will be used to plot and save
    figures made of zig zag images

    Returns:
    cmap:the color map

    """
    colorsImage = len(np.unique(image))
    # We take the gist_rainbow color map which already exists
    cmap = plt.cm.gist_rainbow
    # We extract the 256 colors there are in form of a list
    cmaplist = [cmap(i) for i in range(cmap.N)]

    # We shuffle the list with this particular random seed
    random.seed(4)
    random.shuffle(cmaplist)

    # Take only the colorsImage first colors
    cmaplist = cmaplist[:colorsImage]

    # The central of all thos colors will be white (0)
    whiteIndex = int(colorsImage/2)

    # We force the white color to be white, which belongs to the background
    cmaplist[whiteIndex] = ('w')
    # We create the new color map
    cmap = mpl.colors.LinearSegmentedColormap.from_list(
            'Custom cmap', cmaplist, colorsImage)

    return cmap

#############################################################################
##

def createPixelWiseBackbone(image):
    """This function creates the "Pixel-Wise Backbone" from a ground truth
    image. This new image consists in one pixel per each x coordenate and
    text-line number. This pixel represents the middle of that column for
each
    text-line number.

    Parameters:
    image (array): an array which represents the ground truth image from the
    ICDAR2013 Handwritten Segmentation Contest dataset

    Returns:
    array: the Pixel-Wise Backbone

    """
    # The number of lines in the ground truth image (remember: a number for
    # each color + the background is the number 0)
    linesN = len(np.unique(image)) - 1
```

```
    # We will need this for the loop
    height, width = image.shape

    # The array we are going to return
    pixelWiseBackbone = np.array([])

    # Main loop
    for j in range(width):
        # We initialize an "all background" (all 0's) column. Then, we
        # will fill this column with the backbone pixels. The result will
        # be appended to the array which we will return
        columnToInsert = np.zeros((height, 1)).astype(int)

        # We take a column to analyze
        columnFromImage = image[:, j]

        for n in range(1, linesN+1):
            # We extract the index where the column matches the number
            index = np.array(np.where(columnFromImage == n))

            # If there's no match, the array "index" will be empty
            if index.size > 0:
                # Take the mean and round it. Then parse to int array
                iBackboneIndex = np.around(np.mean(index)).astype(int)
                columnToInsert[iBackboneIndex] = n

        # When the loop is over, we will have a column to insert in the BB
        try:
            pixelWiseBackbone = np.column_stack((pixelWiseBackbone,
columnToInsert))
        except:
            # The first time it will throw a ValueError Exception which we
have
            # to handle
            pixelWiseBackbone = columnToInsert

    return pixelWiseBackbone
##############################################################################
##

def createLinearRegressionBackbone(image):
    """This function creates the "Linear Regression Backbone" from a ground
    truth image's "Pixel-Wise Backbone". This new image consists in the
    regression line that goes throgh the middle of every component of the
    text-line.

    Parameters:
    pixelWiseBackbone (array): an array which represents "Pixel-Wise
Backbone"
    of a ground truth image returned by the function
"createPixelWiseBackbone"

    Returns:
    array: the Linear Regression Backbone

    """
    # The number of lines in the ground truth image (remember: a number for
    # each color + the background is the number 0)
#    linesN = len(np.unique(pixelWiseBackbone)) - 1
    linesValues = np.unique(image) #JJMF202011
```

```
    # The array we are going to return
    linearRegressionBackbone = np.zeros(image.shape)


    #for n in range(1, linesN+1):        #JJMF 202011 for n in range(1,
linesN+1):
    lineCounter = 0
    for n in linesValues[1:,]:
        lineCounter += 1
        # We extract the X and Y position of every pixel corresponding to the
        # line "n". Be careful because X and Y are given the other
        (Y_n, X_n) = np.where(image == n) #n JJMF202011

        # We check if the coordinates are not empty
        if X_n.size > 0:
            # We are going to take the linear regression. The line will be
            #    y_linearRegression = m*X_n + b
            (m, b) = np.polyfit(X_n, Y_n, 1)

            # Now, we have to extract the minimum and maximum values of "X"
for
            # this text line.
            X_line = np.array(range(np.amin(X_n), np.amax(X_n)))

            # The linear regression polynomial will be evaluated from the
            # beggining to the very last pixel of the text line
            Y_linearRegression = np.around(np.polyval([m, b],
X_line)).astype(int)

            # We fill the coordinates we got with the number of the line
            linearRegressionBackbone[Y_linearRegression, X_line] =
lineCounter #n JJMF202011

    return linearRegressionBackbone

##########################################################################
##

def createOutlineBackbone(image):
    """This function creates the outline of every text line from a ground
    truth image

    Parameters:
    image (array): an array which represents the ground truth image from the
    ICDAR2013 Handwritten Segmentation Contest dataset

    Returns:
    array: the outline image

    """
    # The number of lines in the ground truth image (remember: a number for
    # each color + the background is the number 0)
    linesN = len(np.unique(image)) - 1
    linesValues = np.unique(image) #JJMF202011


    # We will need this for the loop
    height, width = image.shape

    # The arrays we are going to use as a help
    auxArray1 = np.array([]) # to store the maximums of each line
    auxArray2 = np.array([]) # to store the minimums of each line
```

```
    auxArray3 = np.array([]) # to store the linear regression of the max
    auxArray4 = np.array([]) # to store the linear regression of the min

    # The array we are going to return
    outlineBackbone = np.array([])

    # Main loop
    for j in range(width):
        # We initialize an "all background" (all 0's) column. Then, we
        # will fill this column with the max and min pixels. The result will
        # be appended to the auxiliars arrays 1 and 2
        columnToInsert1 = np.zeros((height, 1)).astype(int)
        columnToInsert2 = np.zeros((height, 1)).astype(int)

        # We take a column to analyze
        columnFromImage = image[:, j]


        #JJMF 202011 for n in range(1, linesN+1):
        lineCounter = 0
        for n in linesValues[1:,]:
            lineCounter +=1
 #       for n in range(1, linesN+1):
            # We extract the index where the column matches the number
            index = np.array(np.where(columnFromImage == n))

            # If there's no match, the array "index" will be empty
            if index.size > 0:
                # Take the max and min of the indexes
                iMaxIndex = np.amax(index)
                iMinIndex = np.amin(index)
                columnToInsert1[iMaxIndex] = lineCounter #JJMF202011 n
                columnToInsert2[iMinIndex] = lineCounter #JJMF202011 n

        # When the loop is over, we will have two columns to insert in the
        # proper auxiliar arrays
        try:
            auxArray1 = np.column_stack((auxArray1, columnToInsert1))
            auxArray2 = np.column_stack((auxArray2, columnToInsert2))
        except:
            # The first time it will throw a ValueError Exception which we
have
            # to handle
            auxArray1 = columnToInsert1
            auxArray2 = columnToInsert2

    # Now, here it is the second part of the function. We will take the
linear
    # regression from this arrays. We will use the upper function for this.
    auxArray3 = createLinearRegressionBackbone(auxArray1)
    auxArray4 = createLinearRegressionBackbone(auxArray2)

    # The formula used to create the "outlineBackbone" array is the next: we
are
    # going to fill an array with the line number "n" between this two values
    #       1st value: max{auxArray1, auxArray3}
    #       2nd value: min{auxArray2, auxArray4}
    for j in range(width):
        # We initialize an "all background" (all 0's) column.
        columnToInsert = np.zeros((height, 1)).astype(int)

        # We take  column to analyze
```

```
        columnFromAux1 = auxArray1[:, j]
        columnFromAux2 = auxArray2[:, j]
        columnFromAux3 = auxArray3[:, j]
        columnFromAux4 = auxArray4[:, j]

        for n in range(1, linesN+1):
            # We extract the index where the columns match the number
            index1 = np.array(np.where(columnFromAux1 == n))
            index2 = np.array(np.where(columnFromAux2 == n))
            index3 = np.array(np.where(columnFromAux3 == n))
            index4 = np.array(np.where(columnFromAux4 == n))

            # We know that the indexes 3 and 4 have the same domain, and we
            # also know that, unless there's no blank space in the text line,
            # their domain is greater than the indexes 1 and 2.
            # That's why the first condition is to evaluate if index3 (or 4,
we
            # don't really mind) is empty or not.
            if index3 > 0:
                if index1 > 0:
                    maxIndex = max(index1[0][0], index3[0][0])
                else:
                    maxIndex = index3[0][0]
                if index2 > 0:
                    minIndex = min(index2[0][0], index4[0][0])
                else:
                    minIndex = index4[0][0]

                # Now we will loop within the values we have taken to fill
the
                # column to insert afterwards
                for i in range(minIndex, maxIndex + 1):
                    columnToInsert[i] = n

        # When the loop is over, we will have to insert the new column
        try:
            outlineBackbone = np.column_stack((outlineBackbone,
columnToInsert))
        except:
            # The first time it will throw a ValueError Exception which we
have
            # to handle
            outlineBackbone = columnToInsert

    return outlineBackbone

##########################################################################
##

def createZigZagBackbone(image, verbose = 0):
    """

    Parameters:
    image (array): an array which represents the ground truth image from the
    ICDAR2013 Handwritten Segmentation Contest dataset

    Returns:
    array: the outline image

    """
    # The number of lines in the ground truth image (remember: a number for
    # each color + the background is the number 0)
```

```
    linesN = len(np.unique(image)) - 1

    # We will need this later
    height, width = image.shape

    # We get both of the backbones because zigzag is created comparing them
    outlineBackboneImage = createOutlineBackbone(image)
    if verbose:
        plt.figure(3)
        plt.imshow(np.array(outlineBackboneImage))
        plt.show()
    linearRegressionBackboneImage = createLinearRegressionBackbone(image)
    if verbose:
        plt.figure(4)
        plt.imshow(np.array(linearRegressionBackboneImage))
        plt.show()

    # The returning array
    zigZagBackbone = np.zeros(image.shape)

    # Main loop
    for n in range(1, linesN+1):
        # We copy the line n in a separeted array
        auxArray = np.zeros(image.shape)
        auxArray[outlineBackboneImage == n] = n
        auxArray[linearRegressionBackboneImage == n] = -n

        # When selecting a column from the image, we have one of these two
situations:
        #   1.- We don't find any number apart from 0 in it
        #   2.- We find a sequence of numbers "n" until one unique pixel of
"-n"
        #       so that we should change the numbers n below for -n
        for j in range(width):
            columnFromImage = auxArray[:, j]

            index = np.array(np.where(columnFromImage == -n))

            # Only 2 possibilities, etiher length of index == 1 or == 0
            if index.size == 1:
                # We take indices of the elements of the line
                index2 = np.array(np.where(columnFromImage == n))

                # Separate those which are beyond our "-n" index
                index2 = index2[0][index2[0] > index[0][0]]

                # The case of having only 1 or 2  pixel of thickness is not
considered

                # Substitute the n with -n
                columnFromImage[index2] = -n

                # Update the column
                auxArray[:, j] = columnFromImage

        zigZagBackbone = zigZagBackbone + auxArray

    return zigZagBackbone
```

```
##############################################################################
##

def createThickBackbone(image):
    # The number of lines in the ground truth image (remember: a number for
    # each color + the background is the number 0)
    linesN = len(np.unique(image)) - 1
    linesValues = np.unique(image) #JJMF202011
    # We will need this for the loop
    height, width = image.shape

    # The arrays we are going to use as a help
    auxArray1 = np.array([]) # to store the linear regression of the max
    auxArray2 = np.array([]) # to store the linear regression of the min

    # The array we are going to return
    thickBackbone = np.array([])


    # Main loop
    for j in range(width):
        # We initialize an "all background" (all 0's) column. Then, we
        # will fill this column with the max and min pixels. The result will
        # be appended to the auxiliars arrays 1 and 2
        columnToInsert1 = np.zeros((height, 1)).astype(int)
        columnToInsert2 = np.zeros((height, 1)).astype(int)

        # We take a column to analyze
        columnFromImage = image[:, j]


        #JJMF 202011 for n in range(1, linesN+1):
        lineCounter = 0
        for n in linesValues[1:,]:
            lineCounter += 1
            # We extract the index where the column matches the number
            index = np.array(np.where(columnFromImage == n))

            # If there's no match, the array "index" will be empty
            if index.size > 0:
                # Take the max and min of the indexes
                iMaxIndex = np.amax(index)
                iMinIndex = np.amin(index)
                columnToInsert1[iMaxIndex] = int(lineCounter) #n  JJJMF202011
                columnToInsert2[iMinIndex] = int(lineCounter) #n  JJJMF202011

        # When the loop is over, we will have two columns to insert in the
        # proper auxiliar arrays
        try:
            auxArray1 = np.column_stack((auxArray1, columnToInsert1))
            auxArray2 = np.column_stack((auxArray2, columnToInsert2))
        except:
            # The first time it will throw a ValueError Exception which we
have
            # to handle
            auxArray1 = columnToInsert1
            auxArray2 = columnToInsert2

    #plt.imshow(auxArray1)
    # Now, here it is the second part of the function. We will take the
linear
    # regression from this arrays. We will use the upper function for this.
```

```
    auxArray1 = createLinearRegressionBackbone(auxArray1)
    auxArray2 = createLinearRegressionBackbone(auxArray2)

    # Now we fill between those two lines
    for j in range(width):
        # We initialize an "all background" (all 0's) column.
        columnToInsert = np.zeros((height, 1)).astype(int)

        # We take  column to analyze
        columnFromAux1 = auxArray1[:, j]
        columnFromAux2 = auxArray2[:, j]

        for n in range(1, linesN+1):
            # We extract the index where the columns match the number
            index1 = np.array(np.where(columnFromAux1 == n))
            index2 = np.array(np.where(columnFromAux2 == n))

            if index1 > 0:
                maxIndex = index1[0][0]
                minIndex = index2[0][0]

                # Now we will loop within the values we have taken to fill
the
                # column to insert afterwards
                for i in range(minIndex, maxIndex + 1):
                    columnToInsert[i] = n

        # When the loop is over, we will have to insert the new column
        try:
            thickBackbone = np.column_stack((thickBackbone, columnToInsert))
        except:
            # The first time it will throw a ValueError Exception which we
have
            # to handle
            thickBackbone = columnToInsert

    return thickBackbone


##########################################################################
##

def simplifyZigZag(image):
    """This function collapses all the negatives values in a zig-zag backbone
    image into -1, 0 and 1

    Parameters:
    image (array): the zig-zag backbone representation of an image

    Returns:
    array: the simplified zig zag backbone

    """
    newImage = np.copy(image)
    newImage[newImage<0]=-1
    newImage[newImage>0]=1
    return newImage.astype(int)

##########################################################################
##
def lineThickener(x):
```

```
    """This function takes an image and puts a pixel above and under each
pixel
    with the same number of the pixel

    Parameters:
        image(array): the image

    Returns:
        array: the image thckened
    """
    image = np.copy(x)
    # We will need this later
    height, width = image.shape

    for i in range(width):
        indices = np.where(image[:, i] != 0)
        indices = np.array(indices[0])

        indicesUp = indices + 1
        image[indicesUp, i] =  image[indices, i]

        indicesDown = indices - 1
        image[indicesDown, i] =  image[indices, i]

    return image


"""
################################################################################

The following added JJMF 202011 and adaptated for DIVA database by JMC 20211

We include next the functions to be called when creating the ground truth


################################################################################

"""


def thickBackboneGroundTruthGenerator(listPaths, DataBaseType, generateImages
= False, verbose = 0):
    """
    thickBackbonesGroundTruthGenerator(listPaths,verbose=0)

    listPaths is a dictionary expected to have the following entries:
        - destination_route_int32
        - destination_route_bool
        - destination_route_nLines
        - destination_route_images
        - route_lines
        - route_images


    DataBaseTypeis a variable that indicates which database will be used:
        - DataBaseType = 0 : DIVA with images in jpg format and route_lines
in npz format
        - DataBaseType = 1: ICDAR2013 with images in tiff format and
route_lines .tiff.dat format

   Set verbose to 1 if you want online information on the computations. The
   number of the sample processed is always printed.
```

```
    """


##############################################################################
##

    # First we create a variable with the list of the files that we are going
to use
    dir_files = sorted(os.listdir(listPaths["route_images"]))



    for x in range(len(dir_files)):


        file = listPaths["route_images"] + dir_files[x]
        base=os.path.basename(file)
        file_name=os.path.splitext(base)[0]
        print(file_name)

        if (DataBaseType == 0):

            if verbose: print('DIVA DataBase: Reading paths\n')
            route_image_x = listPaths["route_images"]  + file_name + ".jpg"
            route_x =  listPaths["route_lines"] + file_name + ".npz"
            destination_x_int32 = listPaths["destination_route_int32"] +
file_name + ".npz"
            destination_x_bool = listPaths["destination_route_bool"]  +
file_name + ".npz"



        elif DataBaseType :

            if verbose: print('ICDAR13 DataBase: Reading paths\n')
            route_image_x = listPaths["route_images"]  + file_name + ".tif"
            route_x = listPaths["route_lines"] + file_name + ".tif.dat"
            destination_x_int32 = listPaths["destination_route_int32"] +
file_name + ".tif.dat"
            destination_x_bool = listPaths["destination_route_bool"] +
file_name + ".tif.dat"
            destination_x_nLines = listPaths["destination_route_nLines"] +
file_name + ".txt"


        destination_x_shape = listPaths["destination_route_shape"]  + "/" +
file_name + ".txt"
        name_int32_png = listPaths["destination_route_images"] + "/" +
file_name + "int32.png"
        name_bool_png = listPaths["destination_route_images"] + "/" +
file_name + "bool.png"

        if verbose: print('Reading images\n')
        # Extract the shape of the real image
    #    import time
        image_x = Image.open(route_image_x)

        if verbose:
            plt.figure(1)
            plt.imshow(np.array(image_x))
            plt.show()
```

```
        shape_x = np.array(image_x).shape

        # We save the shape in a text file
        if verbose: print('Saving shapes\n')
        f = open(destination_x_shape,"w")
        shape_tofile_x = str(shape_x[0]) + " " + str(shape_x[1])
        f.write(shape_tofile_x) # write the tuple into a file
        f.close() # close the file

        # Read the binary files: We check again the DataBaseType we are using
        if DataBaseType :

            array_x = np.fromfile(route_x, dtype='int32')
            array_x = np.reshape(array_x, shape_x)

            # Save the number of lines in a different text file
            if verbose: print('Saving # of lines \n')
            nLines_x= len(np.unique(array_x)) - 1
            f = open(destination_x_nLines,"w")
            f.write(str(nLines_x))
            f.close()

        elif (DataBaseType == 0):

            b = np.load(route_x)
            array_x=b['array1']

            # We only want an array with one dimension not with 3
            # We could also convert the image to black and white with only
one dimension
            dim=(shape_x[0],shape_x[1])
            array_x =np.reshape(array_x,dim)

            if verbose: print('The nlines of the file are already saved \n')

        # Create the targeted array
        if verbose: print('Creating the backbone\n')
        thickBackbone = createThickBackbone(array_x)

        if verbose:
            plt.figure(2)
            plt.imshow(thickBackbone)
            plt.show()

        # Save the new binary file in 2 different forms
        thickBackbone.astype('int32').tofile(destination_x_int32)
        thickBackbone.astype('bool').tofile(destination_x_bool)

        if generateImages:
            mpl.image.imsave(name_int32_png, thickBackbone.astype('int32'),
cmap=createCmap() )
            mpl.image.imsave(name_bool_png,  thickBackbone.astype('bool') ,
cmap=mpl.cm.binary)



def zigzagGroundTruthGenerator(listPaths, DataBaseType, generateImages =
False, verbose = 0):
    """
    zigzagGroundTruthGenerator(listPaths,verbose=0)
```

```
    listPaths is a dictionary expected to have the following entries:
        - destination_route_original
        - destination_route_simplified
        - destination_route_nLines
        - destination_route_images
        - route_Lines
        - route_images


    DataBaseTypeis a variable that indicates which database will be used:
        - DataBaseType = 0 : DIVA with images in jpg format and route_lines
in npz format
        - DataBaseType = 1: ICDAR2013 with images in tiff format and
route_lines .tiff.dat format

   Set verbose to 1 if you want online information on the computations. The
   number of the sample processed is always printed.

    """


##############################################################################
##

    # First we create a variable with the list of the files that we are going
to use
    dir_files = sorted(os.listdir(listPaths["route_images"]))

    for x in range(len(dir_files)):

        file = listPaths["route_images"] + dir_files[x]
        base=os.path.basename(file)
        file_name=os.path.splitext(base)[0]
        print(file_name)

        if (DataBaseType == 0):

            if verbose: print('DIVA DataBase: Reading paths\n')
            route_image_x = listPaths["route_images"]  + file_name + ".jpg"
            route_x =  listPaths["route_lines"] + file_name + ".npz"
            destination_x_original = listPaths["destination_route_original"]
+ file_name + ".npz"
            destination_x_simplified =
listPaths["destination_route_simplified"] + file_name + ".npz"



        elif DataBaseType :

            if verbose: print('ICDAR13 DataBase: Reading paths\n')
            route_image_x = listPaths["route_images"]  + file_name + ".tif"
            route_x = listPaths["route_lines"] +  file_name + ".tif.dat"
            destination_x_original = listPaths["destination_route_original"]
+ file_name + ".tif.dat"
            destination_x_simplified =
listPaths["destination_route_simplified"] + file_name + ".tif.dat"
            destination_x_nLines = listPaths["destination_route_nLines"] +
file_name + ".txt"
```

```
        destination_x_shape = listPaths["destination_route_shape"] +
file_name + ".txt"
        name_original_png = listPaths["destination_route_original"]  +
file_name + ".png"
        name_simplified_png = listPaths["destination_route_simplified"] +
file_name + ".png"


        # Read the image and extract the shape of the real image
        if verbose: print('Reading images\n')
        image_x = Image.open(route_image_x)
        shape_x = np.array(image_x).shape

        if verbose:
            plt.figure(1)
            plt.imshow(np.array(image_x))
            plt.show()


        # We save the shape in a text file
        f = open(destination_x_shape,"w")
        if verbose: print('Saving Shape\n')
        shape_tofile_x = str(shape_x[0]) + " " + str(shape_x[1])
        f.write(shape_tofile_x) # write the tuple into a file
        f.close() # close the file


        # Read the binary files: We check again the DataBaseType we are using
        if DataBaseType :

            array_x = np.fromfile(route_x, dtype='int32')
            array_x = np.reshape(array_x, shape_x)

            # Save the number of lines in a different text file
            if verbose: print('Saving # of lines \n')
            nLines_x= len(np.unique(array_x)) - 1
            f = open(destination_x_nLines,"w")
            f.write(str(nLines_x))
            f.close()

        elif (DataBaseType == 0):

            b = np.load(route_x)
            array_x=b['array1']

            # We only want an array with one dimension not with 3
            # We could also convert the image to black and white with only
one dimension
            dim=(shape_x[0],shape_x[1])
            array_x =np.reshape(array_x,dim)

            if verbose: print('The nlines of the file are already saved \n')

        # Create the targeted array
        if verbose: print('Creating the zig-zag backbone\n')
        zigZagBackbone = createZigZagBackbone(array_x,verbose)
        simplifiedZigZagBackbone = simplifyZigZag(zigZagBackbone)

        if verbose:
            plt.figure(2)
            plt.imshow(zigZagBackbone)
            plt.show()
```

```
        # Save the new binary file in 2 different forms
        if verbose: print('Saving the result into two images formats, with
numbered lines or just with up/down labeling\n')
        zigZagBackbone.astype('int32').tofile(destination_x_original)

simplifiedZigZagBackbone.astype('int32').tofile(destination_x_simplified)

        if generateImages:
            if verbose: print('Saving images into data\n')
            mpl.image.imsave(name_original_png, zigZagBackbone,
cmap=createCmapZigZag(zigZagBackbone))
            mpl.image.imsave(name_simplified_png, simplifiedZigZagBackbone,
cmap=createCmapZigZag(simplifiedZigZagBackbone))

"""
##############################################################################

The following functions added  by JMC 20211 for the DIVA database adaptation

We include a function capable of extracting the number of lines of the
PAGE.xml file. Also, its main function is to create a matrix from the
original image, in order to generate the groundTruth of the project


##############################################################################

"""



def funlines(listPaths):
    """
    Scope:
    Function that creates an array representing the treated image, where the
background of the image
    is represented by zeros, while each line of the image is filled (since it
is an irregular polygon)
    of 1's if it is the first line, of 2's if it is the second and so on
until it reaches nLines
    In addition, it extracts the number of lines of the image


    listPaths is a dictionary expected to have the following entries:

    - route_list_files: path to the .xml files corresponding to the images
    - destination_matrix: path to save the npz file containing the image
array
    - destination_nLines: path to save files containing the number of lines
in the image


    """

##############################################################################
##


    # First we create a variable with the list of the files that we are going
to use
    dir_files = sorted(os.listdir(listPaths["route_list_files"]))
```

```
    # Process to extract the xml coordinates


    for n in range(len(dir_files)):
        file= listPaths["route_list_files"] + "/"+ dir_files[n]
        file_root = objectify.parse(file).getroot()
        # We count the text regions in order to access the lines
        regions=file_root.Page.countchildren()
        # We establish the number that will carry the line in the image
matrix
        lineOrder=0
        # Number of lines we have initially
        nlines=0
        # To obtain the imageWidth and the imageHeight
        tree = ET.parse(file)
        root = tree.getroot()
        W = int(root[1].get('imageWidth'))
        H = int(root[1].get('imageHeight'))

        # We create a zero-matrix
        image_matrix = np.zeros((H, W),dtype='int32')
        print("This is the file:"+ str(n+1) + "/"+ str(len(dir_files)))
        for i in range(regions):

            # Number of lines per region
            # We remove two children because there are two elements in the
region that are not lines :
            #    - the coordinates of the region
            #    - a text_eval element

            num_lines_reg=file_root.Page.TextRegion[i].countchildren()-2
            # Number of lines per image
            nlines= nlines + num_lines_reg

            # Next, we extract the coordinates of each line
            for j in range(num_lines_reg):


                # This indicates that the first line will carry the number 1
in the matrix
                lineOrder = lineOrder + 1
                # The coordinates are like this: pairs formed of (H,W)==
(y,x)
                # When the line coordinates are extracted, we convert them to
str,
                # in orderto extract the pairs of points that form the line
                coord =
file_root.Page.TextRegion[i].TextLine[j].Coords.get('points')
                string_coord = str(coord)

                # Next, we check how many pairs of points there are
                num_commas = string_coord.count(",")
                # We create an array of int zeros in order to fill the
polygon containing the text line
                z_array = np.random.randint(1, size=(num_commas, 2))
                # Parameter that acts as an index at the position of the
vector
                len_z_array = 0

                while num_commas > 0:
                    extension = len(string_coord)
```

```
                    pos_comma = string_coord.find(",") # We see where the X
is
                    pos_space = string_coord.find(" ") # We see where the Y
ends

                    # We get x and y, and we convert them to int
                    x = string_coord[0:pos_comma]
                    x = int(x)
                    # We check if it is the last pair of x and y coordinates,
and we convert them to int
                    if num_commas == 1:
                        y = string_coord[(pos_comma+1):extension]
                        y = int(y)

                    else:
                        y = string_coord[(pos_comma+1):pos_space]
                        y = int(y)
                        # Now, we remove x and y from the entire chain
                        string_coord = string_coord[pos_space+1:extension]

                    # We replace in the matrix the corresponding line number

                    image_matrix[y,x] = lineOrder
                    #image_matrix[H,W]

                    # We save the pair of points
                    z_array[len_z_array,0] = x
                    z_array[len_z_array,1] = y

                    # We decrease or increase the indexes
                    len_z_array = len_z_array+1
                    num_commas=num_commas-1

                # We fill the entire line

                nx, ny = W, H #dimensions
                poly_verts = z_array # polygon vertices


                # We create the coordinates of the vertices for each cell of
the mesh

                x_p, y_p = np.meshgrid(np.arange(nx), np.arange(ny))
                x_p, y_p = x_p.flatten(), y_p.flatten()

                # Here are the points of the polygon filling
                points = np.vstack((x_p,y_p)).T

                # We fill in the polygon delimited by the vertices with the
points calculated above
                path = Path(poly_verts)
                grid = path.contains_points(points)
                #The grid returns an array of True and False,
                # where True are the positions of the fill of the polygon
                grid = grid.reshape((ny,nx))

                # We look for the coordinates of the filling
                filling_coord=np.where(grid==True)
                filling_x=filling_coord[0]
                filling_y=filling_coord[1]
                # Length of the coordinates extracted for the filling. Both
must have the same number
```

```
                len_filling=len(filling_x)
                for zz in range(len_filling):
                    pos_filling_y=filling_y[zz]

                    pos_filling_x=filling_x[zz]

                    image_matrix[pos_filling_x,pos_filling_y]=lineOrder

                # We have finished filling in the line, we continue with the
next one until we fill in all of them



        # Next, we save the matrix in the type of file we have chosen
        base=os.path.basename(file)
        file_name=os.path.splitext(base)[0]

        """
        # We show the image matrix with the filled lines
        vals = np.linspace(0,1,256)
        np.random.shuffle(vals)
        cmap2 = plt.cm.colors.ListedColormap(plt.cm.jet(vals))

        plt.imshow(image_matrix,cmap=cmap2)
        """

        #Binary data
        matrix_file= listPaths["destination_matrix"] + file_name + ".npz"

        np.savez_compressed(matrix_file, array1=image_matrix)

        # Finally, we save the number of lines in a txt file
        file_nLines = listPaths["destination_nLines"]+ file_name + '.txt'
        f = open(file_nLines,"w")
        f.write(str(nlines))
        f.close()

##############################################################################
#####
def makeDirIfNotExist(dirname, verbose = False):
    #if not os.path.exists(os.path.dirname(filename)):
    if not os.path.exists(dirname):
        if verbose: print('Directory does not exist')
        try:
            os.makedirs(dirname)
            if verbose: print('Directory made')
        except OSError as exc: # Guard against race condition
            if exc.errno != errno.EEXIST:
                if verbose: print('Error creating directory')
                raise

##############################################################################
#####


def checkName (xPath):

    # With this function we check if the files name is a string or not
    # If is a string, we change the name for the proper number

    ### Author Julia Moreno Casanova
```

```
    dir_files = sorted(os.listdir(xPath))

    for x in range(len(dir_files)):

      file1 = xPath + dir_files[x]


      base = os.path.basename(file1)
      file_name=os.path.splitext(base)[0]

      if (file_name.isdigit() == False):
        # The file name is not a number, so we have to numerate the file
        file_name = "{:03d}".format(x+1)

        # We save the file with the changed name
        new_file_x = xPath + file_name + ".npz"

        os.rename(file1, new_file_x)
##########################################################################
#####

"""
   FUNCTION TO GREATE TENSORS
"""


def createTensor_x(route_images,route_x_save,DataBaseType):
    """
    With this version we save in one tensor all images, compressed.
JJMF202010
    Modified by JMC 20211: For this function to work with the DIVA dataset as
well as the ICDAR13 dataset

    """

##########################################################################
##

    # First we create a variable with the list of the files that we are going
to use
    dir_files = sorted(os.listdir(route_images))

    for x in range(len(dir_files)):
        file = route_images + dir_files[x]
        base=os.path.basename(file)
        file_name=os.path.splitext(base)[0]
        print(file_name)

        if DataBaseType :
            # Maximum size allowed
            TENSOR_HEIGHT = 4096
            TENSOR_WIDTH = 3072
            # According to this tensors will not be generated for larger
images in the
            # ICDAR13 dabase (this applies to images 224 or x == 327)
            if(x == 224 or x == 327):
                print("X, skipping the image " + str(x) + ".tif, the
enumeration will be altered")
            else:
                #  All tensors will be set to this size, with zero padding.
```

```
                x2save = np.full((TENSOR_HEIGHT, TENSOR_WIDTH,1),
0).astype("int8")


                route_x = route_images + file_name + ".tif"

                x_array = np.logical_not(np.array(Image.open(route_x)))

                shape_x = x_array.shape


                x_array = np.reshape(x_array, (shape_x[0], shape_x[1],1)) #
To add third dimension

                # We want the variables saved to be "201 - 348", not "201 -
350"
                # with 2 missing
                #print(x)
                if(x>224):
                    x = x - 1
                if(x>=327):
                    x= x - 1
                print(x)

                x2save[:shape_x[0], :shape_x[1], :] = x_array.astype("int8")

                route2savecmp = route_x_save + file_name +  ".npz"
                np.savez_compressed(route2savecmp, x=x2save)



        elif (DataBaseType == 0) :

            # Maximum size allowed: all tensors will be set to this size
            TENSOR_HEIGHT = 5120
            TENSOR_WIDTH = 3584

            # Images of DIVA DataBase are in RGB format, so we need to
convert them
            # to B&W format
            route_x = route_images + file_name + ".jpg"

            originalImage = cv2.imread(route_x)
            grayImage = cv2.cvtColor(originalImage, cv2.COLOR_BGR2GRAY)
            (thresh, blackAndWhiteImage) = cv2.threshold(grayImage, 127, 255,
cv2.THRESH_BINARY)
            cv2.imwrite(route_x, blackAndWhiteImage)

            x_array = np.logical_not(np.array(Image.open(route_x)))


            shape_x = x_array.shape

            # Tensors that are below the maximun size allowed,(those with
this size: 4992 x 3328)
            # will be set to it with zero padding.
            # However, tensors that are above the maximun size allowed,
(those with this size: 6496 x 4872)
            # will be set to it with downsampling.

            if ((shape_x[0] < TENSOR_HEIGHT) & (shape_x[1] < TENSOR_WIDTH)):
```

```
                # Zero padding.
                print("DOING ZERO-PADDING")
                x2save = np.full((TENSOR_HEIGHT, TENSOR_WIDTH,1),
0).astype("int8")
                x_array = np.reshape(x_array, (shape_x[0], shape_x[1],1)) #
To add third dimension
                x2save[:shape_x[0], :shape_x[1], :] = x_array.astype("int8")
                route2savecmp = route_x_save + file_name +  ".npz"
                np.savez_compressed(route2savecmp, x2save)

            else:
                # Downsampling

                # First. we set the downsampling factor in order to achieve
the maximun size allowed
                # for the tensors
                factor_Height = 2
                factor_Width = 2

                # In order to obtain the desired size, first we must do a
previous zero padding step
                # We calculate 2 auxiliar numbers in order to obtain the
wanted dimensions
                numaux_height = (TENSOR_HEIGHT*factor_Height) / shape_x[0]
                numaux_width = (TENSOR_WIDTH*factor_Width) / shape_x[1]
                aux_x = np.full((int(shape_x[0]*numaux_height),
int(shape_x[1]*numaux_width)), 0).astype("int8")
                aux_x[:shape_x[0], :shape_x[1]] = x_array


                 # Now we begin the downsampling step
                print("DOING DOWNSAMPLING")

                image_downscaled = downscale_local_mean(aux_x,
(factor_Height, factor_Width))
                # We save and compress the tensor.
                x2save = np.reshape(image_downscaled,
(TENSOR_HEIGHT,TENSOR_WIDTH, 1))

                route2savecmp = route_x_save + file_name +  ".npz"
                np.savez_compressed(route2savecmp, x2save)




def
createTensorThickbackbone_y(route_gt,route_shape,route_y_save,DataBaseType):

   # First we create a variable with the list of the files that we are going
to use
   dir_files = sorted(os.listdir(route_gt))

   for y in range(len(dir_files)):
        file = route_gt + dir_files[y]
        base=os.path.basename(file)
        file_name=os.path.splitext(base)[0]
        print(file_name)

        if DataBaseType :
            # Maximum size allowed
```

```
            TENSOR_HEIGHT = 4096
            TENSOR_WIDTH = 3072
            # We will skip 224 and 327 for the huge dimensions they have
            if(y == 224 or y == 327):
                print("Skipping the labelling " + str(y) + ".tif.data, the
enumeration will be altered")
            else:
                #  All tensors will be set to this size, with zero padding.

                y2save = np.full((TENSOR_HEIGHT, TENSOR_WIDTH,1),
0).astype("int8")
                print("Y: Loading " + str(y) + " out of 350")

                route_y = route_gt + file_name + ".tif.dat"
                route_shape_y = route_shape  + file_name + ".txt"
                f = open(route_shape_y,"r")
                shape_y = f.read().split()
                f.close()
                shape_y[0] = int(shape_y[0])
                shape_y[1] = int(shape_y[1])
                y_array = np.reshape(np.fromfile(route_y, dtype="bool"),
(shape_y[0],shape_y[1], 1))
                # We want the variables saved to be "1 - 348", not "1 - 350
with 2 missing"
                if(y>224):
                    y = y - 1
                if(y>=327):
                    y= y - 1
                y2save[:shape_y[0], :shape_y[1], :] = y_array

                route2savecmp = route_y_save + file_name +  ".npz"
                np.savez_compressed(route2savecmp, y=y2save)


        elif (DataBaseType == 0) :

            # Maximum size allowed: all tensors will be set to this size
            TENSOR_HEIGHT = 5120
            TENSOR_WIDTH = 3584

            route_y = route_gt + file_name + ".npz"
            route_shape_y = route_shape  + file_name + ".txt"
            f = open(route_shape_y,"r")
            shape_y = f.read().split()
            f.close()
            shape_y[0] = int(shape_y[0])
            shape_y[1] = int(shape_y[1])



            # Tensors that are below the maximun size allowed,(those with
this size: 4992 x 3328)
            # will be set to it with zero padding.
            # However, tensors that are above the maximun size allowed,
(those with this size: 6496 x 4872)
            # will be set to it with downsampling.

            if ((shape_y[0] < TENSOR_HEIGHT) & (shape_y[1] < TENSOR_WIDTH)):
                # Zero padding.
                print("DOING ZERO-PADDING")
                y2save = np.full((TENSOR_HEIGHT, TENSOR_WIDTH,1),
0).astype("int8")
```

```
                y_array = np.reshape(np.fromfile(route_y, dtype="bool"),
(shape_y[0],shape_y[1], 1))
                y2save[:shape_y[0], :shape_y[1], :] = y_array
                 # We save and compress the tensor.
                route2savecmp = route_y_save + file_name +  ".npz"
                np.savez_compressed(route2savecmp, y2save)

            else:
                # Downsampling

                # First. we set the downsampling factor in order to achieve
the maximun size allowed
                # for the tensors
                factor_Height = 2
                factor_Width = 2
                # In order to obtain the desired size, first we must do a
previous zero padding step
                # We calculate 2 auxiliar numbers in order to obtain the
wanted dimensions
                numaux_height = (TENSOR_HEIGHT*factor_Height) / shape_y[0]
                numaux_width = (TENSOR_WIDTH*factor_Width) / shape_y[1]
                aux_y = np.full((int(shape_y[0]*numaux_height),
int(shape_y[1]*numaux_width)), 0).astype("int8")
                y_array = np.fromfile(route_y, dtype="bool")
                y_array_reshaped = y_array.reshape(shape_y[0],shape_y[1])
                aux_y[:shape_y[0], :shape_y[1]] = y_array_reshaped



                # Now we begin the downsampling step
                print("DOING DOWNSAMPLING")
                label_downscaled = downscale_local_mean(aux_y,(factor_Height,
factor_Width))

                # We save and compress the tensor.

                y2save = np.reshape(label_downscaled,
(TENSOR_HEIGHT,TENSOR_WIDTH, 1))

                route2savecmp = route_y_save + file_name +  ".npz"
                np.savez_compressed(route2savecmp, y2save)







def createTensorZigzag_y(route_gt,route_shape,route_y_save,DataBaseType):
    # First we create a variable with the list of the files that we are going
to use
    dir_files = sorted(os.listdir(route_gt))
    for y in range(len(dir_files)):
        file = route_gt + dir_files[y]
        base=os.path.basename(file)
        file_name=os.path.splitext(base)[0]
        print(file_name)

        # Maximum size allowed

        if DataBaseType :
```

```
            TENSOR_HEIGHT = 4096
            TENSOR_WIDTH = 3072
            # We will skip 224 and 327 for the huge dimensions they have
            if(y == 224 or y == 327):
                print("Skipping the labelling " + str(y) + ".tif.data, the
enumeration will be altered")
            else:
                 #  All tensors will be set to this size, with zero padding.
                y2save = np.full((TENSOR_HEIGHT, TENSOR_WIDTH,1),
0).astype("int8")
                print("Y: Loading " + str(y) + " out of 350")

                route_y = route_gt  + file_name + ".tif.dat"

                route_shape_y = route_shape + file_name + ".txt"
                f = open(route_shape_y,"r")
                shape_y = f.read().split()
                f.close()
                shape_y[0] = int(shape_y[0])
                shape_y[1] = int(shape_y[1])
                y_array = np.reshape(np.fromfile(route_y,
dtype="int32").astype("int8"), (shape_y[0],shape_y[1], 1))
                # We want the variables saved to be "1 - 348", not "1 - 350
with 2 missing"
                if(y>224):
                    y = y - 1
                if(y>=327):
                    y= y - 1
                y2save[:shape_y[0], :shape_y[1], :] = y_array

                route2savecmp = route_y_save + file_name +  ".npz"
                np.savez_compressed(route2savecmp, y=y2save)



        elif (DataBaseType == 0) :
            # Maximum size allowed: all tensors will be set to this size
            TENSOR_HEIGHT = 5120
            TENSOR_WIDTH = 3584

            route_y = route_gt + file_name + ".npz"
            route_shape_y = route_shape  + file_name + ".txt"
            f = open(route_shape_y,"r")
            shape_y = f.read().split()
            f.close()
            shape_y[0] = int(shape_y[0])
            shape_y[1] = int(shape_y[1])

            # Tensors that are below the maximun size allowed,(those with this
size: 4992 x 3328)
            # will be set to it with zero padding.
            # However, tensors that are above the maximun size allowed, (those
with this size: 6496 x 4872)
            # will be set to it with downsampling.

            if ((shape_y[0] < TENSOR_HEIGHT) & (shape_y[1] < TENSOR_WIDTH)):
                # Zero padding.
                print("DOING ZERO-PADDING")
                y2save = np.full((TENSOR_HEIGHT, TENSOR_WIDTH,1),
0).astype("int8")
```

```
                y_array = np.reshape(np.fromfile(route_y, dtype="int32"),
(shape_y[0],shape_y[1], 1))
                y2save[:shape_y[0], :shape_y[1], :] = y_array
                 # We save and compress the tensor.
                route2savecmp = route_y_save + file_name +  ".npz"
                np.savez_compressed(route2savecmp, y2save)

            else:
                # Downsampling

                # First. we set the downsampling factor in order to achieve
the maximun size allowed
                # for the tensors
                factor_Height = 2
                factor_Width = 2
                # In order to obtain the desired size, first we must do a
previous zero padding step
                # We calculate 2 auxiliar numbers in order to obtain the
wanted dimensions
                numaux_height = (TENSOR_HEIGHT*factor_Height) / shape_y[0]
                numaux_width = (TENSOR_WIDTH*factor_Width) / shape_y[1]
                aux_y = np.full((int(shape_y[0]*numaux_height),
int(shape_y[1]*numaux_width)), 0).astype("int8")
                y_array = np.fromfile(route_y, dtype="int32")
                y_array_reshaped = y_array.reshape(shape_y[0],shape_y[1])
                aux_y[:shape_y[0], :shape_y[1]] = y_array_reshaped



                # Now we begin the downsampling step
                print("DOING DOWNSAMPLING")
                label_downscaled = downscale_local_mean(aux_y,(factor_Height,
factor_Width))

                # We save and compress the tensor.

                y2save = np.reshape(label_downscaled,
(TENSOR_HEIGHT,TENSOR_WIDTH, 1))

                route2savecmp = route_y_save + file_name +  ".npz"
                np.savez_compressed(route2savecmp, y2save)
```

B.1.2 Código de *computeAllSamples.py*

```
#!/usr/bin/env python3
# -*- coding: utf-8 -*-
"""
Created on Thu Nov 12 12:13:02 2020

@author: Juan José Murillo-Fuentes

Based on functions by J.L. Mendoza
Modified by J. Moreno (adaptation to DIVA database)

SCOPE:
```

```
This script  can read two types of database, but not at the same time.
We indicate with a flag which type we would like it to read.
So it can read the 350 images in the ICDAR13 dataset to compute the
groundtruth
of the proposed algorithm. An also, it can do the same with the 120 images in
the DIVA dataset.


Maximum size allowed is 4096 x 3072. All tensors will be set to this size,
with
zero padding.

It:
    Initializes
    Generate tensors for the input using the images of the dataset
    Generate labels for the thick backbone problem
    Generate the tensors for the labels of the thick backbone problem
    Generate labels for the zig zag problem
    Generate the tensors for the labels of the zig zag problem
    Erases not needed folders (if requested)

    Images of the labels can be generated, but are not needed for the
training
    and test stage.

    Tensors are compressed saving 99% of the original size.

    In the generation of these tensors some GB (About 5 GB) of HD are needed,
    that may be erased in the end.

"""
from groundTruthFunctions import thickBackboneGroundTruthGenerator, \
zigzagGroundTruthGenerator, createTensor_x, createTensorThickbackbone_y, \
createTensorZigzag_y, makeDirIfNotExist,funlines, checkName



import shutil
import os

from IPython import get_ipython
get_ipython().run_line_magic('matplotlib','qt') # 'inline'))



"""
0. INITIALIZATION
"""

# To see further information on the inner steps and show images. Nice for one
# one image
verbose = 1

if verbose: print("INITIALIZATION\n")

DataBaseType = 0 #1 is for ICDAR13 and 0 is for DIVA

if DataBaseType :
    # ICDAR13 data files
    if verbose: print("ICDAR13 DATASET\n")
    #Download and decompress the images and gt_lines
```

```
    mainPathIcdar13 = '/Users/julia_000/Desktop/DataBases/ICDAR13'
    route_lines = mainPathIcdar13 + "/gt_lines/"
    route_images = mainPathIcdar13 + "/images/"

    """
    #Tensors will be generated for images in this range
    range(1,351) # 224 and 327 will not be proccessed
    #may be changed to compute a subset
    """
elif (DataBaseType == 0) :
    # DIVA data files
    if verbose: print("DIVA DATASET\n")
    #Download and decompress the images and gt_lines
    mainPathDIVA = '/Users/julia_000/Desktop/DataBases/DIVA-HisDB'

    route_list_files = mainPathDIVA + "/test_train/gt_PAGE_xml_TASK2/"
    destination_matrix = mainPathDIVA + "/gt_lines/"
    destination_nLines = mainPathDIVA + "/gt_nLines/"
    route_images = mainPathDIVA + "/test_train/data/"

    listPathslineDIVAGt={"route_list_files": route_list_files,\
                         "destination_matrix":destination_matrix,\
                         "destination_nLines":destination_nLines}

     # If gt_lines and gt_nLines are not given, they are generated
     # in this section. Note that for ICDAR13, they are already calculated.

    gt_generate = 1 # 0 if they are not generated, 1 if they are.

    if gt_generate:

       if verbose: print("The gt_lines and gt_nLines are already generated
for DIVA dataset\n")

    else:
       if verbose: print("Generating the gt_lines and gt_nLines for DIVA
dataset ...\n")
       # We call the function to generate the groundtruth lines

       funlines(listPathslineDIVAGt)

       if verbose: print("Generation COMPLETED\n")


    route_lines = destination_matrix
#Images may be generated to illustrate the generated ground truths, but are
not
#needed to generate the tensors
generateImages = False

#Set it to true if at the end everything but the tensors is deleted
cleanAfter = True



#Get current path, tensors will be created in a folder in this path

mainPath = os.path.abspath(os.getcwd())  #We will generate all within this
directory
if verbose: print("Current path:", mainPath)
```

```
"""
1. GENERATING TENSORS FOR INPUTS

We read images, and store them into tensors that are independently saved as
compressed files. Compression is crucial to save space, reducing the files to
1%


According to this tensors will not be generated for larger images, in the
ICDAR13 dabase this applies to images number 224 and 327 that will be not
included
updating the numeration to images from 1 to 348.

This applies to generated tensors for the labels, later computed

"""

if verbose: print("1. GENERATING TENSORS FOR THE INPUT\n")
# Routes where to save the variables
# Folders must exist, otherwise we get an error
if DataBaseType :

    route_x_save = mainPath + "/tensorsICDAR13/input/x/"

elif (DataBaseType == 0):

    route_x_save = mainPath + "/tensorsDIVA/input/x/"

""" Check if it does not exist the directory and creates it does not """

makeDirIfNotExist(route_x_save)

createTensor_x(route_images,route_x_save,DataBaseType)

# Once we have created the tensors for the inputs, we check if their names
are numbers
# or not. If not we must change it to numbers, because its easier to work
with numerated files.
# For that reason, we call the checkName function
checkName(route_x_save)
#exit
#sys.exit()


"""
 2. GENERATING LABELS FOR THICK BACKBONE
"""

if verbose: print("2. GENERATING LABELS FOR THICK BACKBONE\n")

if DataBaseType :

    thickBackBoneGtMainPath = mainPath
+"/datasets/DS_ICDAR13/thickBackBoneGt"

elif (DataBaseType == 0):

    thickBackBoneGtMainPath = mainPath +"/datasets/DS_DIVA/thickBackBoneGt"

destination_route_int32 = thickBackBoneGtMainPath + "/int32/"
destination_route_bool = thickBackBoneGtMainPath + "/bool/"
```

```
destination_route_shapes = thickBackBoneGtMainPath + "/shape/"
destination_route_nLines = thickBackBoneGtMainPath + "/nLines/"
destination_route_images = thickBackBoneGtMainPath + "/images/"

listPathsthickBackBoneGt={"destination_route_int32":
destination_route_int32,\
                          "destination_route_bool":destination_route_bool,\

"destination_route_nLines":destination_route_nLines,\

"destination_route_shape":destination_route_shapes,\

"destination_route_images":destination_route_images,\
                          "route_lines":route_lines,\
                          "route_images":route_images}

""" Check if it does not exist the directory and creates if it does not """

for keydir in listPathsthickBackBoneGt:
    dirname = listPathsthickBackBoneGt[keydir]
    if verbose: print(dirname)
    makeDirIfNotExist(dirname, verbose)


thickBackboneGroundTruthGenerator(listPathsthickBackBoneGt,DataBaseType,gener
ateImages ,verbose)



"""
3. GENERATING TENSORS FOR THICKBACKBONE
"""

if verbose: print("3. GENERATING TENSORS FOR THICKBACKBONE\n")

if DataBaseType :
    route_gt = mainPath + "/datasets/DS_ICDAR13/thickBackboneGt/bool/"
    route_shape = mainPath + "/datasets/DS_ICDAR13/thickBackboneGt/shape/"
    route_y_save = mainPath + "/tensorsICDAR13/thickBackboneTensors/y/"
elif (DataBaseType == 0):
    route_gt = mainPath + "/datasets/DS_DIVA/thickBackboneGt/bool/"
    route_shape = mainPath + "/datasets/DS_DIVA/thickBackboneGt/shape/"
    route_y_save = mainPath + "/tensorsDIVA/thickBackboneTensors/y/"


makeDirIfNotExist(route_y_save)

createTensorThickbackbone_y(route_gt,route_shape,route_y_save,DataBaseType)

# Once we have the ThickBackbone tensors, we check if their names are numbers
or not.
# If not, we must change them with the checkName function.
checkName(route_y_save)

if cleanAfter:
    shutil.rmtree(thickBackBoneGtMainPath)


"""
4. GENERATING LABELS FOR ZIGZAG

"""
```

```
if verbose: print("4. GENERATING LABELS FOR ZIG ZAG\n")

if DataBaseType :
    zigzagMainFolder = mainPath +"/datasets/DS_ICDAR13/zigzagGt/"
elif (DataBaseType == 0):
    zigzagMainFolder = mainPath +"/datasets/DS_DIVA/zigzagGt/"


destination_route_original = zigzagMainFolder+"original/"
destination_route_simplified = zigzagMainFolder+"simplified/"
destination_route_shapes = zigzagMainFolder+"shape/"
destination_route_nLines = zigzagMainFolder+"nLines/"
destination_route_images = zigzagMainFolder+"images/"


listPathszigzagGt={"destination_route_original": destination_route_original,\

"destination_route_simplified":destination_route_simplified,\

"destination_route_nLines":destination_route_nLines,\

"destination_route_shape":destination_route_shapes,\

"destination_route_images":destination_route_images,\
                           "route_lines":route_lines,\
                           "route_images":route_images}


""" Check if it does not exist the directory and creates if it does not """

for keydir in listPathszigzagGt:
    dirname = listPathszigzagGt[keydir]
    if verbose: print(dirname)
    makeDirIfNotExist(dirname)


zigzagGroundTruthGenerator(listPathszigzagGt,DataBaseType,generateImages
,verbose)



"""
5.  GENERATING TENSORS FOR ZIGZAG
"""
if verbose: print("5. GENERATING TENSORS FOR ZIG ZAG\n")

route_gt = zigzagMainFolder+ "simplified/"
route_shape = zigzagMainFolder+ "shape/"
if DataBaseType :
    route_y_save = mainPath + "/tensorICDAR13/zigzagTensors/y/"
elif (DataBaseType == 0):
    route_y_save = mainPath + "/tensorDIVA/zigzagTensors/y/"

""" Check if it does not exist the directory and creates if it does not """
makeDirIfNotExist(route_y_save)

createTensorZigzag_y(route_gt,route_shape,route_y_save,DataBaseType)
# Once we have the ZIgZag Backbone tensors, we check if their names are
numbers or not.
# If not, we must change them with the checkName function.
checkName(route_y_save)
```

```
"""
6. CLEANING: ERASING NOT NEEDED FOLDERS
"""

if verbose: print("ERASING NOT NEEDED FOLDERS\n")
if cleanAfter:
    shutil.rmtree(mainPath +"/datasets")
```

B.1.3 Código de *nLinesAndShapeAnalyzer.py*

```
'''
    Created by Jose Luis Mendoza for his Final Disertation,
    "A tool for text lines segmentation in images bases on deep learning"
    2019-2020
    University of Seville, Spain

    The following code makes a dataset analyzing

    The dataset can be found in the following URL:

        http://users.iit.demokritos.gr/~nstam/ICDAR2013HandSegmCont/
    Modified by Julia Moreno Casanova
'''
# If you feel more comfortable, then create it locally and then move it
wherever
route_shape = '/Users/julia_000/Desktop/DataBases/DIVA/shape/'
route_nLines = '/Users/julia_000/Desktop/DataBases/DIVA/gt_nLines/'

list_heights = []
list_widths = []
list_nLines = []

import statistics
import os
# First we create a variable with the list of the files that we are going to
use
dir_files = sorted(os.listdir(route_shape))
for x in range(len(dir_files)):
        file = route_shape + dir_files[x]
        base=os.path.basename(file)
        file_name=os.path.splitext(base)[0]
        print(file_name)

        route_shape_x = route_shape + file_name + ".txt"
        route_nLines_x = route_nLines + file_name + ".txt"

        f = open(route_shape_x,"r")
        shape_x = f.read().split()
        list_heights.append(int(shape_x[0]))
        list_widths.append(int(shape_x[1]))
        f.close()

        f = open(route_nLines_x,"r")
        nLines_x = f.read()
        list_nLines.append(int(nLines_x))
        f.close()
```

```
print("The minimum - average - maximum values of HEIGHT are:\n\t\t" + \
      str(min(list_heights)) + " - " +  str(statistics.mean(list_heights)) +
" - " + str(max(list_heights)))

print("\nThe minimum - average - maximum values of WIDTH are:\n\t\t" + \
      str(min(list_widths)) + " - " + str(statistics.mean(list_widths)) + " -
" + str(max(list_widths)))

print("\nThe minimum - average - maximum values of NLINES are:\n\t\t" + \
      str(min(list_nLines)) + " - " + str(statistics.mean(list_nLines)) + " -
" + str(max(list_nLines)))
```

B.1.4 Código de *Unet_2048.py*

```
import keras

def down_block(x, filters, kernel_size=(3, 3), padding="same", strides=1):
    c = keras.layers.Conv2D(filters, kernel_size, padding=padding,
strides=strides, activation="relu")(x)
    c = keras.layers.Conv2D(filters, kernel_size, padding=padding,
strides=strides, activation="relu")(c)
    p = keras.layers.MaxPool2D((2, 2), (2, 2))(c)
    return c, p

def up_block(x, skip, filters, kernel_size=(3, 3), padding="same",
strides=1):
    us = keras.layers.UpSampling2D((2, 2))(x)
    concat = keras.layers.Concatenate()([us, skip])
    c = keras.layers.Conv2D(filters, kernel_size, padding=padding,
strides=strides, activation="relu")(concat)
    c = keras.layers.Conv2D(filters, kernel_size, padding=padding,
strides=strides, activation="relu")(c)
    return c

def bottleneck(x, filters, kernel_size=(3, 3), padding="same", strides=1):
    c = keras.layers.Conv2D(filters, kernel_size, padding=padding,
strides=strides, activation="relu")(x)
    c = keras.layers.Conv2D(filters, kernel_size, padding=padding,
strides=strides, activation="relu")(c)
    return c

# Custom layer made for this project
def MinPoolingBool2D( x ):
    minPool = -keras.layers.MaxPool2D((2, 2), (2, 2))(-x);
    return minPool

def UNet(height, width):
    f = [16, 32, 64, 128, 256, 512, 1024, 2048, 4092]
    inputs = keras.layers.Input((height, width, 1))



    p0 = inputs
```

```
    c1, p1 = down_block(p0, f[0]) #128 -> 64
    c2, p2 = down_block(p1, f[1]) #64 -> 32
    c3, p3 = down_block(p2, f[2]) #32 -> 16
    c4, p4 = down_block(p3, f[3]) #16->8

    bn = bottleneck(p4, f[4])

    u1 = up_block(bn, c4, f[3]) #8 -> 16
    u2 = up_block(u1, c3, f[2]) #16 -> 32
    u3 = up_block(u2, c2, f[1]) #32 -> 64
    u4 = up_block(u3, c1, f[0]) #64 -> 128

    outputs = keras.layers.Conv2D(1, (1, 1), padding="same",
activation="sigmoid")(u4)




    model = keras.models.Model(inputs, outputs)
    return model
```

B.1.5 Código de *DataGen_2048_dataAug_rotated.py*

```
'''
    Created by Jose Luis Mendoza for his Disertation,
    "A tool for text lines segmentation in images bases on deep learning"
    2019-2020
    University of Seville, Spain
    Modified by Julia Moreno Casanova

    The following code is used by the CNN for taking a batch of images and
labels
    to be processed.


'''

import keras
import os
import numpy as np
from skimage.measure import block_reduce
from PIL import Image
import random

class DataGen(keras.utils.Sequence):
    """Generates data for Keras
    Sequence based data generator. Suitable for building data generator for
training and prediction.
    The images will be downsampled to half the resolution saved.
    """
    # Routes where the variables are. Obviously, this would vary between
implementations
    route_x_train = "C:\\variables_TFG\\x_train\\"
    route_y_train = "C:\\variables_TFG\\y_train\\"
    route_x_test = "C:\\variables_TFG\\x_test\\"
    route_y_test = "C:\\variables_TFG\\y_test\\"
```

```
    def __init__(self, list_image_numbers = range(1,201),
                 x_path="C:\\variables_TFG\\dataset\\x",
                 y_path="C:\\variables_TFG\\dataset\\y",
                 batch_size=10, num_channels = 1, shuffle = True,
                 to_fit = True, zig_zag = False, half_outline =
0,zig_zag_rotated= False,
                 data_augmentation=True, debugging=False, IMAGES_HEIGHT =
4096, IMAGES_WIDTH = 3072,DataBaseType=1 ):

        """Initialization

        :param list image numbers: list of all 'label' ids to use in the
generator (DEFAULT: the ones for fitting -- 1-200.tif (ICDAR 2013))
        :param x_path: path to x variables location (DEFAULT:
"C:\\variables_TFG\\dataset\\x")
        :param y_path: path to y variables location (DEFAULT:
"C:\\variables_TFG\\dataset\\y")
        :param batch_size: batch size at each iteration (DEFAULT: 10)
        :param num_channels: number of image channels (DEFAULT: 1 - Black and
White)
        :param shuffle: True to shuffle label indexes after every epoch
(DEFAULT: True)
        :param to_fit: True to return x and y [fitting], False to return x
only [prediction] (DEFAULT: True)
        :param zig_zag: True if we are going to return zig_zag outline, False
if just regular outline (DEFAULT: False)
        :param half_outline: taking 0 it does not affect to the project.
However, taking the value +1 makes this return the
                             upper half of the outline and taking the value -
1, the the other half (DEFAULT: 0)
        :param zig_zag_rotated: True if we are going to return zig_zag
outline working with just an U-Net, False if just regular outline (DEFAULT:
False)
        :param data_augmentation: when True, 3 images are returned instead of
only 1, being the 2 others the 5 degrees rotation of the original (DEFAULT:
True)
        :param debugging: True to show debugging messages (DEFAULT: False)
        :param IMAGES_HEIGHT: image height (DEFAULT: 4096)
        :param IMAGES_WIDTH: image width (DEFAULT: 3072)
        """
        self.list_image_numbers = list_image_numbers
        self.x_path = x_path
        self.y_path = y_path
        self.batch_size = batch_size

        self.num_channels = num_channels
        self.shuffle = shuffle
        self.to_fit = to_fit
        self.zig_zag = zig_zag
        if(not zig_zag):
            self.half_outline =  0 # if zig_zag == False there's no point in
taking any half
        else:
            self.half_outline = half_outline

        self.zig_zag_rotated = zig_zag_rotated
        self.data_augmentation = data_augmentation
        self.debugging = debugging
        self.on_epoch_end() # to generate the indexes variables and shuffling
        self.IMAGES_HEIGHT = IMAGES_HEIGHT
        self.IMAGES_WIDTH = IMAGES_WIDTH
```

```
        image_shape=(int(IMAGES_HEIGHT/2), int(IMAGES_WIDTH/2))
        self.image_shape = image_shape
        self.DataBaseType = DataBaseType


    def __len__(self):
        """Denotes the number of batches per epoch

        :return: number of batches per epoch
        """
        return int(np.floor(len(self.list_image_numbers) / self.batch_size))

    def __getitem__(self, batch_index):
        """Generate one batch of data

        :param batch_index: index of the batch BEGINNING WITH 0
        :return: x and y when fitting. x only when predicting
        """
        if(self.debugging):
            print("Function: getItem(" + str(batch_index) + ")")

        # If this is the very last batch (meaning that next one starts out of
        # range) we are going to get only the last indexes available.
        # EXAMPLE: with a batch size of 7 and 200 elements, we have 28 full
        # batches and last batch with batch_index=28 will have only 4
elements.
        # if( (28+1)*7 > 200 ):
        #   batch_size = 200 - 28*7 = 4
        if(batch_index+1)*self.batch_size > len(self.list_image_numbers):
            self.batch_size = len(self.list_image_numbers) -
batch_index*self.batch_size


        # Extract the ids of this batch from the list.
        # EXAMPLE: with batch_index = 1 and the same situation described
before,
        # a list made by list_image_numbers = range(1,201)
{1,2,3...198,199,200}
        # we should store in elements_batch the sub-list {8,9,10,11,12,13,14}
        # so we will have to take list_image_numbers[7:13]
        # HOWEVER, we should take that "sub-list" from the object "indexes"
        # which will be properly shuffled
        elements_batch = self.indexes[batch_index*self.batch_size :
(batch_index+1)*self.batch_size]

        if(self.debugging):
            print("Debugging1: batch_size = " + str(self.batch_size))
            print("Debugging2: indexes = " + str(self.indexes))
            print("Debugging3: elements_batch = " + str(elements_batch))



        if(self.to_fit):
            x, y = self.__load_batch(elements_batch)
            #print("\t\tfin de getItem(" + str(batch_index) + ")")
            return x, y
        else:
            x = self.__load_batch(elements_batch)
            return x


    def __load_batch(self, elements_batch):
```

```
        """Generates an array containing images and labels of a batch

        :param elements_batch: list of label ids to load
        :return: batch of images and labels
        """
        # If data augmentation is activated, we will need 3 more images
        if(self.data_augmentation):
            images_to_load = 3*self.batch_size
        else:
            images_to_load = self.batch_size

        if(self.to_fit):

           x = np.empty((images_to_load, self.image_shape[0],
self.image_shape[1], self.num_channels), dtype="float32")
           y = np.empty((images_to_load, self.image_shape[0],
self.image_shape[1], self.num_channels), dtype="float32")


           if(self.debugging):
                print("Debugging4: x, yshapes are: " + str(x.shape) + " " +
str(y.shape))

           if(self.data_augmentation):
                for i in range(0,elements_batch.size):
                    # load single image will return 3 images from each one
                    aux_x, aux_y =
self.load_single_image(self.list_image_numbers[elements_batch[i]])

                    for j in range(0, 3):
                        x[3*i+j,], y[3*i+j,] =aux_x[j,], aux_y[j,]


           else:
                for i in range(0,elements_batch.size):
                    x[i,], y[i,] =
self.load_single_image(self.list_image_numbers[elements_batch[i]])


           return x, y
        else:


            x = np.empty((images_to_load, self.image_shape[0],
self.image_shape[1], self.num_channels), dtype="float32")


            if(self.debugging):
                print("Debugging4: x shape is: " + str(x.shape))

            if(self.data_augmentation):
                for i in range(0,elements_batch.size):
                    # load single image will return 3 images from each one
                    aux_x =
self.load_single_image(self.list_image_numbers[elements_batch[i]])

                    for j in range(0, 3):
                        x[3*i+j,] =aux_x[j,]


            else:
                for i in range(0,elements_batch.size):
```

```
                    x[i,] =
self.load_single_image(self.list_image_numbers[elements_batch[i]])


            return x


    def load_single_image(self, image_number):
        """Generates one image if to_fit==False and one image and its label
if else

        :param elements_batch: list of label ids to load
        :return: batch of images
        """

        if(self.debugging):
            print("Debugging5: load single image number: " +
str(image_number))

        if(self.to_fit):
            # Path
            image_number = "{:03d}".format(image_number)
            image_path = os.path.join(self.x_path, image_number) + ".npz"
            label_path = os.path.join(self.y_path, image_number) + ".npz"

            if(self.debugging):
                print("Debugging6: looking for the images in:")
                print("\timage_path="+image_path)
                print("\tlabel_path="+label_path)

            # Load variables and unzip them, they are in npz format to make
them lighter.

            xz = np.load(image_path)
            yz =np.load(label_path)
            if(self.DataBaseType == 0):
              x=xz['arr_0'].astype('float32')
              y=yz['arr_0'].astype('float32')
            elif(self.DataBaseType == 1):
              x=xz['x'].astype('float32')
              y=yz['y'].astype('float32')

            if(self.zig_zag):
                if(self.half_outline != 0):
                    # If half_outline = 1 --> upper half || half_outline = -1
--> other half
                    y =  np.floor((self.half_outline * y + 1) / 2)

            # Now we check if we are in the modified case (in which only one
network is trained)
            elif(self.zig_zag_rotated):

              # We randomly choose whether the image is rotated or not.
              rpick = random.randint(0, 1)

              # We start with random rotation
              if rpick:
                  # It is not rotated, only the upper part of the line is
extracted
                  y=  np.floor((y + 1) / 2)
              else:
```

```
                # We extract the bottom part (label) and rotate it. This way
we will have the image rotated with the extraction also rotated.
                 y= np.floor((-y + 1) / 2) # We take the bottom part (label
/label)
                 y= np.flipud(y) # Now we rotate the label
                 x= np.flipud(x) # We also rotate the image


            else:
                # In case it is not ZigZag and it is Thick Backbone
                y = np.absolute(y)

            # Downscaling: we do it to solve the lack of memory

            x = np.reshape(x, (2*self.image_shape[0], 2*self.image_shape[1]))
            y = np.reshape(y, (2*self.image_shape[0], 2*self.image_shape[1]))

            x = block_reduce(x, block_size=(2, 2), func=np.max)

            y = block_reduce(y, block_size=(2, 2), func=np.max)


            x = np.reshape(x, (self.image_shape[0],self.image_shape[1],
self.num_channels))
            y = np.reshape(y, (self.image_shape[0], self.image_shape[1],
self.num_channels))


            # Data augmentation
            if(self.data_augmentation):
                x_image = Image.fromarray(np.reshape(x, (self.image_shape[0],
self.image_shape[1])))
                y_image = Image.fromarray(np.reshape(y, (self.image_shape[0],
self.image_shape[1])))

                # Rotate +5 degrees
                x_pos5 = np.array(Image.Image.rotate(x_image, 5))
                y_pos5 = np.array(Image.Image.rotate(y_image, 5))

                # Rotate -5 degrees
                x_neg5 = np.array(Image.Image.rotate(x_image, -5))
                y_neg5 = np.array(Image.Image.rotate(y_image, -5))

                # Array to return
                x_augmentated = np.empty((3, self.image_shape[0],
self.image_shape[1], self.num_channels), dtype = 'float32')
                y_augmentated = np.empty((3, self.image_shape[0],
self.image_shape[1], self.num_channels), dtype = 'float32')

                # Fill the arrays
                x_augmentated[0,] = np.reshape(x_pos5, (self.image_shape[0],
self.image_shape[1], self.num_channels))
                x_augmentated[1,] = x
                x_augmentated[2,] = np.reshape(x_neg5, (self.image_shape[0],
self.image_shape[1], self.num_channels))
                y_augmentated[0,] = np.reshape(y_pos5, (self.image_shape[0],
self.image_shape[1], self.num_channels))
                y_augmentated[1,] = y
                y_augmentated[2,] = np.reshape(y_neg5, (self.image_shape[0],
self.image_shape[1], self.num_channels))

                if(self.debugging):
```

```
                    print("Debugging7: x_augmentated, y_augmentated shapes
are: " + str(x_augmentated.shape) + " " + str(y_augmentated.shape))

                return x_augmentated, y_augmentated

            else:
                if(self.debugging):
                    print("Debugging7: x, y shapes are: " + str(x.shape) + "
" + str(y.shape))
                return x, y


        # If "to_fit == False" then we do the same but only with x
        else:
            # Path
            image_number = "{:03d}".format(image_number)
            image_path = os.path.join(self.x_path, image_number) + ".npz"

            if(self.debugging):
                print("Debugging6: looking for the images in:")
                print("\timage_path="+image_path)


            # Load variable
            xz = np.load(image_path)
            if(self.DataBaseType == 0):
              x=xz['arr_0'].astype('float32')
            elif(self.DataBaseType == 1):
              x=xz['x'].astype('float32')


            if(self.zig_zag_rotated & (int(image_number) % 2 == 0)):
                x= np.flipud(x)


            # Downscaling
            x = np.reshape(x, (2*self.image_shape[0], 2*self.image_shape[1]))

            x = block_reduce(x, block_size=(2, 2), func=np.max)

            x = np.reshape(x, (self.image_shape[0], self.image_shape[1],
self.num_channels))

            # Data augmentation
            if(self.data_augmentation):
                x_image = Image.fromarray(np.reshape(x, (self.image_shape[0],
self.image_shape[1])))

                # Rotate +5 degrees
                x_pos5 = np.array(Image.Image.rotate(x_image, 5))

                # Rotate -5 degrees
                x_neg5 = np.array(Image.Image.rotate(x_image, -5))

                # Array to return
                x_augmentated = np.empty((3, self.image_shape[0],
self.image_shape[1], self.num_channels), dtype = 'float32')

                # Fill the arrays
                x_augmentated[0,] = np.reshape(x_pos5, (self.image_shape[0],
self.image_shape[1], self.num_channels))
                x_augmentated[1,] = x
```

```
                x_augmentated[2,] = np.reshape(x_neg5, (self.image_shape[0],
self.image_shape[1], self.num_channels))

                if(self.debugging):
                    print("Debugging7: x_augmentated shape is: " +
str(x_augmentated.shape))

                return x_augmentated

            else:
                if(self.debugging):
                    print("Debugging7: x shape is: " + str(x.shape))
                return x
    def on_epoch_end(self):
        """Updates indexes after each epoch

        """
        # Create an attribute called "indexes", which is a numpy copy of
list_image_number
        self.indexes = np.arange(len(self.list_image_numbers))
        # and shuffle it if shuffle==True
        if self.shuffle == True:
            np.random.shuffle(self.indexes)
```

B.1.6 Código de *segmentation.py*

```
"""
Created on Wed Jan 13 21:21:43 2021

@author: Julia Moreno Casanova
Based on functions by J.L. Mendoza
"""
from keras.models import model_from_json
import numpy as np
import cv2
import os
import matplotlib as mpl
from skimage.measure import block_reduce
from groundTruthFunctions import createCmap
from fixFunction import fixFunction
import scipy
import itertools
from PIL import Image
from lxml import objectify



 # If a pixel is not segmentated, it will find the nearest connected
component
 # in a rectangle with this dimensions
TOLERANCE_X = 60 # increases vertical shape of the rectangle
TOLERANCE_Y = 200 # increses horizontal shape of the rectangle
TOLERANCE_NON_SEGMENTATED = 0 # if an infinite loop happens, will get out of
it and print if non segmentated pixels are greater than this number

def funSegFus(list_test, listPathsSegFus, DataBaseType):
    """
```

```
    list_test: the list of test images from which we want to get the final
prediction

    listPathsSegFus is a dictionary expected to have the following entries:
        - route_xPath: path to the npz files corresponding to the images
        - route_modelPathCP:path to the h5 files corresponding to the models
        - route_modelPath:path to the json files corresponding to the models
        - route_resultsPath:path to the folder where segmentations results
are stored

    DataBaseTypeis a variable that indicates which database will be used:
        - DataBaseType = 0 : DIVA with images in jpg format and route_lines
in npz format
        - DataBaseType = 1: ICDAR2013 with images in tiff format and
route_lines .tiff.dat format

    """

    list_fixes = []
    list_fails = []



    CC_AREA_TOLERANCE_MID = 2800
    CC_AREA_TOLERANCE_UPDOWN = 2500
    AREA_DEBUG = True


    # Calculate the min and max index
    # max_index_test: index representing the number of the last sample of the
set
    # min_index_test: index representing the number of image by which the set
starts

    min_index_test= list_test[0]
    max_index_test=list_test[len(list_test)-1]



    for i in range(min_index_test,max_index_test +1):

        num2 = "{:03d}".format(i)
        print(num2)
        if DataBaseType:
            IMAGE_HEIGHT = 4096
            IMAGE_WIDTH  = 3072

            # remember that 224 and 327 were removed... the routes will be
searched with "file_name"
            # but the images and results, with "file_name_2"
            if(i>=224):
                i = i + 1
            if(i>=327):
                i= i + 1
            num = "{:03d}".format(i)
        elif (DataBaseType == 0):
            IMAGE_HEIGHT = 5120
            IMAGE_WIDTH  = 3584
            num = num2
```

```
        ########################
        #    ZIG_ZAG BACKBONE   #
        ########################

        # Load json and create model

        json_file_ZZ = open(listPathsSegFus["route_modelPath_ZZ"] , 'r')
        model_UD_json = json_file_ZZ.read()
        json_file_ZZ.close()
        model_UD = model_from_json(model_UD_json)
        # load weights into new model
        model_UD.load_weights(listPathsSegFus["route_modelPathCP_ZZ"] )
        print("Loaded model_UD from disk")

        xz = np.load(listPathsSegFus["route_xPath"]  + num2 + ".npz")

        if (DataBaseType == 0):

          x=xz['arr_0'].astype('float32')
        elif (DataBaseType == 1):

          x=xz['x'].astype('float32')

        # We rotate the input image so that the model returns the down part
        x_r =  np.flipud(x)

        x_downscaled = block_reduce(x.reshape(IMAGE_HEIGHT, IMAGE_WIDTH),
block_size=(2, 2), func=np.max)

        x_downscaled_r= block_reduce(x_r.reshape(IMAGE_HEIGHT, IMAGE_WIDTH),
block_size=(2, 2), func=np.max)


        y_up =   np.round(np.reshape(model_UD.predict(x_downscaled.reshape(1,
int(IMAGE_HEIGHT/2), int(IMAGE_WIDTH/2), 1).astype('float32')),
(int(IMAGE_HEIGHT/2), int(IMAGE_WIDTH/2))))
        y_down =
np.round(np.reshape(model_UD.predict(x_downscaled_r.reshape(1,
int(IMAGE_HEIGHT/2), int(IMAGE_WIDTH/2), 1).astype('float32')),
(int(IMAGE_HEIGHT/2),int(IMAGE_WIDTH/2))))

        y_up    = cv2.resize(y_up  , (IMAGE_WIDTH, IMAGE_HEIGHT),
interpolation = cv2.INTER_AREA).astype('bool')
        y_down  = cv2.resize(y_down, (IMAGE_WIDTH, IMAGE_HEIGHT),
interpolation = cv2.INTER_AREA).astype('bool')


        # We flip y_down so that when joining midUp and midDown they are in
the same position

        y_down= np.flipud(y_down)
        mpl.image.imsave(listPathsSegFus["route_resultsPath"] + num2 + "-
01y_up.png", y_up, cmap=mpl.cm.binary)
        mpl.image.imsave(listPathsSegFus["route_resultsPath"] + num2 + "-
02y_down.png", y_down, cmap=mpl.cm.binary)


        ########################
        #     THICK BACKBONE    #
        ########################
```

```
        # Load json and create model
        json_file_TBB = open(listPathsSegFus["route_modelPath_TBB"], 'r')
        model_thickBackbone_json = json_file_TBB.read()
        json_file_TBB.close()
        model_thickBackbone = model_from_json(model_thickBackbone_json)
        # Load weights into new model

model_thickBackbone.load_weights(listPathsSegFus["route_modelPathCP_TBB"])
        print("Loaded model_thickBackbone from disk")

        y_thickbackbone =
np.round(np.reshape(model_thickBackbone.predict(x_downscaled.reshape(1,
int(IMAGE_HEIGHT/2), int(IMAGE_WIDTH/2), 1).astype('float32')),
(int(IMAGE_HEIGHT/2), int(IMAGE_WIDTH/2)))).astype('bool')

        y_thickbackbone = cv2.resize(y_thickbackbone.astype('float32'),
(IMAGE_WIDTH, IMAGE_HEIGHT), interpolation = cv2.INTER_AREA).astype('bool')

        mpl.image.imsave(listPathsSegFus["route_resultsPath"]  + num2 + "-
03y_thickbackbone.png", y_thickbackbone, cmap=mpl.cm.binary)

        ##########################
        #      PRE-FUSION        #
        ##########################


        # we remove the small cc from the thickbackbone because it is
difficult to be wrong
        y_clear_thickbackbone, nLines_thickbackbone =
removeSmallCC(y_thickbackbone, CC_AREA_TOLERANCE_MID)
        mpl.image.imsave(listPathsSegFus["route_resultsPath"]  + num2 + "-
04y_clear_thickbackbone.png", y_clear_thickbackbone, cmap=createCmap())

        # but with the y_up and the y_down the process is different because
sometimes the line is
        # so thin that it is splitted in two parts. For that reason, we are
going to
        # join the y_up with the clear thickBackbone and rename it midUp.
Same with midDown
        y_midUp   = np.maximum(y_clear_thickbackbone.astype('bool'), y_up  )

        y_midDown = np.maximum(y_clear_thickbackbone.astype('bool'), y_down)


        mpl.image.imsave( listPathsSegFus["route_resultsPath"]  + num2 + "-
05y_midUp.png"  , y_midUp  , cmap = mpl.cm.binary)
        mpl.image.imsave(listPathsSegFus["route_resultsPath"]  +  num2 + "-
06y_midDown.png", y_midDown, cmap = mpl.cm.binary)

        # Now we filter the smaller CC
        y_clear_midUp  , nLines_midUp   = removeSmallCC(y_midUp  ,
CC_AREA_TOLERANCE_UPDOWN)
        y_clear_midDown, nLines_midDown = removeSmallCC(y_midDown,
CC_AREA_TOLERANCE_UPDOWN)



        if(AREA_DEBUG):
            if(nLines_midUp - nLines_thickbackbone or nLines_thickbackbone -
nLines_midDown or nLines_midUp - nLines_midDown):
                print("midUp-mid-midDown = " + str(nLines_midUp) + " - " +
str(nLines_thickbackbone)  + " - " + str(nLines_midDown))
```

```
                # There must be an error, we are going to analyse the size of
each line as a Random Variable
                line_pixels_mid     = []
                line_pixels_midUp   = []
                line_pixels_midDown = []


                # Area of CC's
                for i in np.unique(y_clear_thickbackbone):
                    if i != 0:
                        line_pixels_mid.append(np.sum(y_clear_thickbackbone
== i))

                for i in np.unique(y_clear_midUp):
                    if i != 0:
                        line_pixels_midUp.append(np.sum(y_clear_midUp == i))

                for i in np.unique(y_clear_midDown):
                    if i != 0:
                        line_pixels_midDown.append(np.sum(y_clear_midDown ==
i))


                # Get mean and standard deviation and put them in position 0
and 1
                array_line_pixels_mid     = np.array(line_pixels_mid    )
                array_line_pixels_midUp   = np.array(line_pixels_midUp  )
                array_line_pixels_midDown = np.array(line_pixels_midDown)
                line_pixels_mid.insert(0, array_line_pixels_mid.mean())
                line_pixels_mid.insert(1, array_line_pixels_mid.std())
                line_pixels_midUp.insert(0, array_line_pixels_midUp.mean())
                line_pixels_midUp.insert(1, array_line_pixels_midUp.std())
                line_pixels_midDown.insert(0,
array_line_pixels_midDown.mean())
                line_pixels_midDown.insert(1,
array_line_pixels_midDown.std())

                # Save debug list in a file
                f = open(num2 + "-DEBUG.txt","w")
                f.write("Image name: " + num2 + "\n\n")
                f.write("Y_CLEAR_MID\n")
                f.write("mean   = " + str(line_pixels_mid[0])   + "\n")
                f.write("std    = " + str(line_pixels_mid[1])   + "\n")
                f.write("nLines = " + str(nLines_thickbackbone) + "\n")
                f.write("lines_pixels = \n")
                for i in range(2, len(line_pixels_mid)):
                    f.write("\t" + "{:02d}".format(i-1) + ": " +
str(line_pixels_mid[i]) + "\n")
                f.write("==============================\n")
                f.write("Y_CLEAR_MIDUP\n")
                f.write("mean   = " + str(line_pixels_midUp[0]) + "\n")
                f.write("std    = " + str(line_pixels_midUp[1]) + "\n")
                f.write("nLines = " + str(nLines_midUp)         + "\n")
                f.write("lines_pixels = \n")
                for i in range(2, len(line_pixels_midUp)):
                    f.write("\t" + "{:02d}".format(i-1) + ": " +
str(line_pixels_midUp[i]) + "\n")
                f.write("==============================\n")
                f.write("Y_CLEAR_MIDDOWN\n")
                f.write("mean   = " + str(line_pixels_midDown[0]) + "\n")
                f.write("std    = " + str(line_pixels_midDown[1]) + "\n")
```

```
                f.write("nLines = " + str(nLines_midDown)          + "\n")
                f.write("lines_pixels = \n")
                for i in range(2, len(line_pixels_midDown)):
                    f.write("\t" + "{:02d}".format(i-1) + ": " +
str(line_pixels_midDown[i]) + "\n")
                f.close() # close the file

                # Create dictionary for auxiliary function
                debugDict = {
                    "y_clear_mid" : {
                        "mean"   : line_pixels_mid[0],
                        "std"    : line_pixels_mid[1],
                        "nLines" : nLines_thickbackbone,
                        "lines_pixels" : line_pixels_mid[2:]
                        },
                    "y_clear_midUp" : {
                        "mean"   : line_pixels_midUp[0],
                        "std"    : line_pixels_midUp[1],
                        "nLines" : nLines_midUp,
                        "lines_pixels" : line_pixels_midUp[2:]
                        },
                    "y_clear_midDown" : {
                        "mean"   : line_pixels_midDown[0],
                        "std"    : line_pixels_midDown[1],
                        "nLines" : nLines_midDown,
                        "lines_pixels" : line_pixels_midDown[2:]
                        }
                    }
                try:
                    y_clear_midUp, y_clear_midDown, error_value =
fixFunction(y_clear_thickbackbone, y_clear_midUp, y_clear_midDown, debugDict)
                    if(error_value):
                        print('Fix function could not help, BAD
SEGMENTATION')
                        list_fails.append(num2)
                    else:
                        print('Fixed!')
                        list_fixes.append(num2)
                except Exception as e:
                    print(e)
                    print('Fix function could not help, BAD SEGMENTATION')
                    list_fails.append(num2)



        mpl.image.imsave(listPathsSegFus["route_resultsPath"] + num2 + "-
07y_clear_midUp.png", y_clear_midUp, cmap=createCmap())
        mpl.image.imsave(listPathsSegFus["route_resultsPath"] + num2 + "-
08y_clear_midDown.png", y_clear_midDown, cmap=createCmap())

        ##########################
        #         FUSION         #
        ##########################

        fusion = np.maximum(y_clear_midUp, y_clear_midDown)
        mpl.image.imsave(listPathsSegFus["route_resultsPath"] + num2 + "-
09fussion.png", fusion, cmap=createCmap())

        ##########################
        #      SEGMENTATION      #
        ##########################
```

```
        if (DataBaseType == 0):
            # We have to extract the black borders of the DIVA image in order
to
            # start the segmentation process
            [new_image,Y_array_s,X_array_s] =
cleanDIVAborders(x,listPathsSegFus,num2)
            x_segmentated =
segmentation(new_image.astype('int32').reshape(IMAGE_HEIGHT, IMAGE_WIDTH),
fusion)
            # To respet the format of the evaluation software

        else:
            x_segmentated =
segmentation(x.astype('int32').reshape(IMAGE_HEIGHT, IMAGE_WIDTH), fusion)

        mpl.image.imsave(listPathsSegFus["route_resultsPath"]+ num2 + "-
10x_segmentated.png", x_segmentated, cmap=createCmap())

        ##############################
        #  EVALUATION SOFTWARE INPUT #
        ##############################


        # The segmentated image takes the shape of the original image, in
order to
        # have the right dimensions for the evaluation software input
        f = open(listPathsSegFus["route_shape"] + num + ".txt","r")
        shape_original = f.read().split()
        f.close()
        shape_original[0] = int(shape_original[0])
        shape_original[1] = int(shape_original[1])
        # To save the segmetated image with the original shape for
downsampled images
        if (DataBaseType == 0):

            mpl.image.imsave(listPathsSegFus["route_resultsPath"] + num2+
"-11x_segmentated_originalshape.png", x_segmentated, cmap=createCmap())


        else:
            x_segmentated = x_segmentated[:shape_original[0],
:shape_original[1]]
            mpl.image.imsave(listPathsSegFus["route_resultsPath"] + num2+ "-
11x_segmentated_originalshape.png", x_segmentated, cmap=createCmap())


x_segmentated.astype('int32').tofile(listPathsSegFus["route_resultsPath"]+
"evaluation_software_input/" + num + ".tif.dat")


    print("len(list_fixes) = " + str(len(list_fixes)))
    print("list_fixes = " + str(list_fixes))

    print("len(list_fails) = " + str(len(list_fails)))
    print("list_fails = " + str(list_fails))


#####################################################
#                   OTHER FUNCTIONS                 #
#####################################################

def removeSmallCC(image, tolerance):
```

```
            # Extract all the conected components from image
            label_img, cc_num = scipy.ndimage.label(image)
            # CC = scipy.ndimage.find_objects(label_img)

            # Calculate the length of every CC
            cc_areas = scipy.ndimage.sum(image, label_img, range(cc_num+1))

            # Mark any area below the tolerance limit and remove it (0 =
background)
            area_mask = (cc_areas < tolerance)
            label_img[area_mask[label_img]] = 0

            # Now we may have a messy list with some terms missing (e.g: 0,
1, 2, 4, 6, 8)
            textLineNumber = 0
            frequencyAnalysis = np.unique(label_img)
            for i in sorted(frequencyAnalysis):
                label_img[label_img == i] = textLineNumber
                textLineNumber = textLineNumber + 1

            return label_img, textLineNumber

def findNearestCC(mask, posX, posY, tolerance_x=TOLERANCE_X,
tolerance_y=TOLERANCE_Y):
            # Used by the function "segmentation(original, mask)"
            # If value == -1, it didn't find the connected component
            value = -1

            # So, given a coordinate X, e.g 7 and tolerance_x 3, we will have
to find in the next sequence:
            # find horizontally in posX = 7, then in 8 - 6 - 9 - 5 - 10 - 4
            list2findX = [posX]
            newElement = posX
            for i in range(1, 2*tolerance_x+1):
                newElement = newElement + pow(-1, i)*i
                list2findX.append(newElement)

            # given a coordinate Y, e.g "10" and tolerance = 5 we want to
find in the following positions
            #        9 - 11- 8 - 12 - 7 - 13 - 6 - 14 - 5 - 15 (zigzagging)
            # this list will be saved in "list2find"
            list2findY = [posY]
            newElement = posY

            for i in range(1,2*tolerance_y+1):
                newElement = newElement + pow(-1,i)*i
                list2findY.append(newElement)

            # Now we have the lists, let's do the cartesian product of them
both
            coordinates2find = itertools.product(list2findX, list2findY)

            for x,y in coordinates2find:
                if mask[x][y] != 0:
                    value = mask[x][y]
                    break

            return value
```

```
def segmentation(original, mask,tolerance_y=TOLERANCE_Y,
tolerance_x=TOLERANCE_X,
tolerance_non_segmentated=TOLERANCE_NON_SEGMENTATED):
            segmentation = np.zeros_like(original)
            segmentation[original == 1] = mask[original == 1]


            # There are pixels which are not segmentated, lets assign them
the value of
            # nearest connected component
            debug_repeated_times = 0
            repeat = True

            last_non_segmentated_pixels = -1 # initialized to an invalid
value

            while repeat:
                repeat = False
                # We will repeat until we segmentate them all
                # first, we find those non-segmentated pixels
                nonSegmentatedPixels =
np.logical_and(np.logical_not(segmentation),
original.astype('bool')).astype('int')



                # If some pixels are not segmentated
                if np.count_nonzero(nonSegmentatedPixels) > 0:

                    # check if we ended up in an infinite loop (last
situation was not improved)
                    if last_non_segmentated_pixels ==
np.count_nonzero(nonSegmentatedPixels):

                        # we are going to get out, and only print if greater
than tolerated
                        if np.count_nonzero(nonSegmentatedPixels) >
tolerance_non_segmentated:
                            print("\tDEBUG: Infinite loop avoided with " +
str(np.count_nonzero(nonSegmentatedPixels)) + " non segmentated pixels")

                    # If not, let's continue
                    else:
                        newSegmentatedPixels =
np.zeros_like(nonSegmentatedPixels)

                        # now, for each pixel being 1 in
"nonSegmentatedPixels" group, we assign the value
                        # of the more or less "closest" connected component,
following the next rules:
                        #   First, we will find the connected component
horizontally, for a maximum of 10 pixels on each side
                        #   If it doesn't find it, it will go one pixel up
and try again horizontally. If not, one pixel below and again...

                        for i in np.argwhere(nonSegmentatedPixels==1):

                            segmentationValue = findNearestCC(mask, i[0],
i[1])
                            if segmentationValue == -1: # wrong segmentation,
we have to repeat
                                repeat = True
```

```
                            else: # good segmentation
                                newSegmentatedPixels[i[0],i[1]] =
segmentationValue

                        # After the loop, we improve the mask and re-
segmentate

                        mask = np.maximum(mask, newSegmentatedPixels)
                        segmentation[original == 1] = mask[original == 1]

                        # Update last non segmentated pixels counter
                        last_non_segmentated_pixels =
np.count_nonzero(nonSegmentatedPixels)

                        # Only for debugging reasons
                        print("\tDEBUG: Repeated for " +
str(debug_repeated_times) + " times and " + str(last_non_segmentated_pixels)
+ " non segmentated pixels")
                        debug_repeated_times = debug_repeated_times + 1

            return segmentation


########################################################################
# Author: Julia Moreno Casanova #
# This function is only to use it with the DIVA dataset
########################################################################
TENSOR_WIDTH = 3584
TENSOR_HEIGHT = 5120
def cleanDIVAborders(image,listPaths, num ):

    # First we create a variable with the list of the files that we are going
to use
    dir_files = sorted(os.listdir(listPaths["route_gt_page"]))

    # Process to extract the Page coordinates

    file= listPaths["route_gt_page"] + "/"+ dir_files[int(num)-1]
    file_root = objectify.parse(file).getroot()

    coord = file_root.Page.TextRegion.Coords.get('points')
    string_coord = str(coord)

    # Next, we check how many pairs of points there are
    num_commas = string_coord.count(",")

    # We create an array of zeros and size 4, because there are only 4 pairs
    # of points
    X_array = np.zeros((4,), dtype=int)
    Y_array = np.zeros((4,), dtype=int)
    #Index
    index_xy_array = 0

    while num_commas > 0:
        extension = len(string_coord)
        pos_comma = string_coord.find(",") # We see where the X is
        pos_space = string_coord.find(" ") # We see where the Y ends

        # We get x and y, and we convert them to int
        x = string_coord[0:pos_comma]
        x = int(x)
```

```
        X_array[index_xy_array] = x

        # We check if it is the last pair of x and y coordinates, and we
convert them to int
        if num_commas == 1:
            y = string_coord[(pos_comma+1):extension]
            y = int(y)

            Y_array[index_xy_array] = y


        else:
            y = string coord[(pos comma+1):pos space]
            y = int(y)

            Y_array[index_xy_array] = y


            # Now, we remove x and y from the entire chain
            string_coord = string_coord[pos_space+1:extension]

        # We save the pair of points

        # We decrease or increase the indexes
        index_xy_array = index_xy_array+1
        num_commas=num_commas-1

    # Once we have obtained the Page limits, we turn the black borders
    # of the original image, into white ones, so we dont have problems in the
    # segmentation step
    # We know there are only 4 pairs of points and that they form a rectangle
so:
    X_array_s=sorted(list(set(X_array)))
    Y_array_s=sorted(list(set(Y_array)))

    # The images are resized to fit the H and W of the net, so we have to
modify the coordinates

    f = open(listPaths["route_shape"] + num + ".txt","r")
    shape_image = f.read().split()
    f.close()
    shape_image[0] = int(shape_image[0])
    shape_image[1] = int(shape_image[1])

    blank_image = np.zeros((TENSOR_HEIGHT, TENSOR_WIDTH),dtype='int32')

    if ((shape_image[0] < TENSOR_HEIGHT) and (shape_image[1] <
TENSOR_WIDTH)):
        # Zero padding.
        X_array_s [0] = round(X_array_s[0]*(TENSOR_WIDTH/shape_image[1]))
        X_array_s [1] = round(X_array_s[1]*(TENSOR_WIDTH/shape_image[1]))
        Y_array_s [0] = round(Y_array_s[0]*(TENSOR_HEIGHT/shape_image[0]))
        Y_array_s [1] = round(Y_array_s[1]*(TENSOR_HEIGHT/shape_image[0]))


    else:
        # ZP and Down-sampling
        factor_Height = 2
        factor_Width = 2
        resized_Height =TENSOR_HEIGHT*factor_Height
        resized_Width = TENSOR_WIDTH*factor_Width
```

```
        X_array_s [0] = round(X_array_s[0]*(resized_Width/shape_image[1]))
        X_array_s [0] = round(X_array_s [0]/4)
        X_array_s [1] = round(X_array_s[1]*(resized_Width/shape_image[1]))
        X_array_s [1] = round(X_array_s [1]/4)

        Y_array_s [0] = round(Y_array_s[0]*(resized_Height/shape_image[0]))
        Y_array_s [0] = round(Y_array_s [0]/4)
        Y_array_s [1] = round(Y_array_s[1]*(resized_Height/shape_image[0]))
        Y_array_s [1] = round(Y_array_s [1]/4)


    for i in range (Y_array_s[0],Y_array_s[1]+1):
            for j in range (X_array_s[0],X_array_s[1]+1):

                blank_image[i,j] = int(image[i,j])



    return blank_image,Y_array_s,X_array_s
```

B.1.7 Código de *fixFunction.py*

```
# -*- coding: utf-8 -*-
"""
Created on Tue Dec 15 12:30:29 2020

@author: Julia Moreno Casanova
Based on functions by J.L. Mendoza
"""

from math import sqrt
import numpy as np


from groundTruthFunctions import createLinearRegressionBackbone, createCmap
import matplotlib as mpl


# Bugs cases:
# A: a sentence has been connected with the following one, meaning it will
have double size
# to spot case A, check if the size of a line is greater than mean+2,3std
#
# B: a gap in the middle of a sentence has been detected as a gap in one or
two of the segmentation images but not the others
# B can be confused with a short sentence, so this is the way to proceed:
# A possible line in "B" case has to be spotted if its size is less than
mean-0.8std. This may seem too much, but it is easy
# to discard so we better spot it accurately
# After checking a "small line", we will substract its contrary pair. Meaning
that if line 3 of midUp is "small", we will substract the line 3 of midDown
# Then, we will check if this value is within the mean+-0.8*std of MID_UP-
MID_DOWN random variable. If it is, that's because it's just a small sentence
# if not, B case has been spotted
# C: a gap in the middle of a sentence has been detected as a gap in one or
two of the segmentation images but not the others
```

```
def fixFunction(mid, midUp, midDown, debugDict):
    # fixed --> if it is completely fixed
    # oneFix --> if one fix has been done in this iteration
    fixed = False
    oneFixDone = False
    # Record of all the fixes done
    fixes = []


    # nLines of y_clear_midUp and y_clear_midDown
    nLinesUpDown = debugDict['y_clear_midUp']['nLines'] +
debugDict['y_clear_midDown']['nLines']

    # counters for the loop. The idea is to zig-zag between up and down
    counterUp = 0;
    counterDown = 0

    upTurn = True; # upTurn = True means it's up turn. upTurn = False means
it's down turn.



    # We are going to treat 3 random variables, namely MID, MIDUP and MIDDOWN
(aproximately gaussian random variable)
    # midUp - midDown have mean = mean_midUp - mean_midDown and std =
sqrt(std_midUp^2 + std_midDown^2)
    midUp_midDown_mean = debugDict['y_clear_midUp']['mean'] -
debugDict['y_clear_midDown']['mean']
    midUp_midDown_std  = sqrt((debugDict['y_clear_midUp']['std'] ** 2) +
(debugDict['y_clear_midDown']['std'] ** 2))
    print(midUp_midDown_mean)
    print(midUp_midDown_std)



    list_midUp = debugDict['y_clear_midUp']['lines_pixels']
    list_midDown = debugDict['y_clear_midDown']['lines_pixels']

    while not fixed:
        try:
            for i in range(0, nLinesUpDown-2):

                if not oneFixDone:
                    if upTurn:
                        midUp_line = list_midUp[counterUp]


                        # See if this line' size is normal or not
                        if(midUp_line > debugDict['y_clear_midUp']['mean'] +
2.3*debugDict['y_clear_midUp']['std']):
                            # We have the A case
                            print("A case spotted in midUp " + str(counterUp
+ 1))
                            midUp = fixCaseA(midUp, counterUp + 1)

                            # Update values
                            debugDict['y_clear_midUp']['nLines'] =
debugDict['y_clear_midUp']['nLines'] + 1
                            list_midUp = []
                            for i in np.unique(midUp):
                                if i != 0:
                                    list_midUp.append(np.sum(midUp == i))
```

```
                            oneFixDone = True;

                        elif(midUp_line < debugDict['y_clear_midUp']['mean']
- 1.2*debugDict['y_clear_midUp']['std']):
                            midDown_line = list_midDown[counterUp]


                            # MID UP IS EVALUATED BEFORE MID DOWN, IF A "CASE
A" HAS NOT BEEN FIXED IN MIDDOWN, THE PROGRAM
                            # WILL DETECT A "CASE B" WITHOUT BEEN TRUE. The
first condition of the "if" prevents that.
                            # We can have either the B case or a short
sentence
                            flagB_midUp = detectSameHeight(midUp, counterUp +
1)
                            flagB_midDown = detectSameHeight(midDown,
counterUp + 1)

                            if ((midDown_line <=
debugDict['y_clear_midDown']['mean'] +
2.3*debugDict['y_clear_midDown']['std']) and
                                ((midUp_line - midDown_line) <
(midUp_midDown_mean - 0.6*midUp_midDown_std))  and
                               ((flagB_midUp == 1 and flagB_midDown == 0) or
(flagB_midUp == 0 and flagB_midDown == 1))):
                                print("B case spotted in midUp " +
str(counterUp + 1))

                                # Can raise an exception andr return to
segmentation

                                midUp = fixCaseB(midUp, counterUp + 1)

                                # Update values
                                debugDict['y_clear_midUp']['nLines'] =
debugDict['y_clear_midUp']['nLines'] - 1
                                list_midUp = []
                                for i in np.unique(midUp):
                                    if i != 0:
                                        list_midUp.append(np.sum(midUp == i))

                                oneFixDone = True;

                            elif((midDown_line <=
debugDict['y_clear_midDown']['mean'] +
2.3*debugDict['y_clear_midDown']['std'])):

                                # To check if there is a C Case, we must
check midUp and midDown
                                flagC_midUp = detectSameHeight(midUp,
counterUp + 1)
                                flagC_midDown = detectSameHeight(midDown,
counterUp + 1)

                                if((flagC_midUp == 1) and (flagC_midDown ==
1) ):

                                    print("C Case spotted in midUp " +
str(counterUp + 1))
```

```
                                    # Can raise an exception and return to
segmentation
                                    midUp = fixCaseB(midUp, counterUp + 1)
                                    midDown = fixCaseB(midDown, counterUp +
1)

                                    # Update values
                                    debugDict['y_clear_midUp']['nLines'] =
debugDict['y_clear_midUp']['nLines'] - 1
                                    list_midUp = []
                                    debugDict['y_clear_midDown']['nLines'] =
debugDict['y_clear_midDown']['nLines'] - 1
                                    list_midDown = []
                                    for i in np.unique(midUp):
                                        if i != 0:
                                            list_midUp.append(np.sum(midUp ==
i))
                                    for i in np.unique(midDown):
                                        if i != 0:

list_midDown.append(np.sum(midDown == i))

                                    oneFixDone = True;
                                else:
                                    print("B and C Case not spotted, only a short
sentence in midUp " + str(counterUp + 1))



                            counterUp = counterUp + 1


                        else:

                            midDown_line = list_midDown[counterDown]


                            # See if this line' size is normal or not
                            if(midDown_line >
debugDict['y_clear_midDown']['mean'] +
2.3*debugDict['y_clear_midDown']['std']):
                                # We have the A case
                                print("A case spotted in midDown " +
str(counterDown + 1))
                                midDown = fixCaseA(midDown, counterDown + 1)

                                # Update values
                                debugDict['y_clear_midDown']['nLines'] =
debugDict['y_clear_midDown']['nLines'] + 1
                                list_midDown = []
                                for i in np.unique(midDown):
                                    if i != 0:
                                        list_midDown.append(np.sum(midDown == i))

                                oneFixDone = True;

                            elif(midDown_line <
debugDict['y_clear_midDown']['mean'] -
1.2*debugDict['y_clear_midDown']['std']):
                                midUp_line = list_midUp[counterDown]
```

```
                            flagB_midUp = detectSameHeight(midUp, counterDown
+ 1)

                            flagB_midDown = detectSameHeight(midDown,
counterDown + 1)

                            # We can have either the B case or a short
sentence
                            if(((midUp_line - midDown_line) >
(midUp_midDown_mean + 0.6*midUp_midDown_std)) and
                                ((flagB_midUp == 1 and flagB_midDown == 0) or
(flagB_midUp == 0 and flagB_midDown == 1))):
                                print("B case spotted in midDown " +
str(counterDown + 1))

                                # Can raise an exception andr return to
segmentation
                                midDown = fixCaseB(midDown, counterDown + 1)

                                # Update values
                                debugDict['y_clear_midDown']['nLines'] =
debugDict['y_clear_midDown']['nLines'] - 1
                                list_midDown = []
                                for i in np.unique(midDown):
                                    if i != 0:
                                        list_midDown.append(np.sum(midDown ==
i))

                                oneFixDone = True;



                            else:
                                # To check if there is a C Case, we must
check midUp and midDown
                                flagC_midUp = detectSameHeight(midUp,
counterDown + 1)

                                flagC_midDown = detectSameHeight(midDown,
counterDown + 1)


                                if((flagC_midUp == 1) and (flagC_midDown ==
1) ):
                                    print("C Case spotted in midDown " +
str(counterDown + 1))

                                    # Can raise an exception and return to
segmentation
                                    midDown = fixCaseB(midDown, counterDown +
1)
                                    midUp = fixCaseB(midUp, counterDown + 1)
                                    # Update values
                                    debugDict['y_clear_midDown']['nLines'] =
debugDict['y_clear_midDown']['nLines'] - 1
                                    list_midDown = []
                                    debugDict['y_clear_midUp']['nLines'] =
debugDict['y_clear_midUp']['nLines'] - 1
                                    list_midUp = []
                                    for i in np.unique(midDown):
                                        if i != 0:
```

```
list_midDown.append(np.sum(midDown == i))
                                    for i in np.unique(midUp):
                                        if i != 0:
                                            list_midUp.append(np.sum(midUp ==
i))

                                    oneFixDone = True;




                                else:
                                    print("B and C case not spotted, only a
short sentence in midDown " + str(counterDown + 1))

                            counterDown = counterDown + 1

                    upTurn = not upTurn
            except: # A case has not been fixed, we return the image as it is an
error code
                print(" A case has not been fixed, we return the image as it is
an error code")
                return midUp, midDown, 1

            # After the "for" loop, if one fix has been done we are going to
            # evaluate if it is definitely fixed or not
            fixes.append(oneFixDone)

            # If in the first time no case was spotted, it doesn't make sense to
continue. Neither does if infinite loop occurs
            if((len(fixes) == 1 and oneFixDone==False) or len(fixes) >
nLinesUpDown/1.5):
                print(" If in the first time no case was spotted, it doesn't make
sense to continue. Neither does if infinite loop occurs")
                return midUp, midDown, 1
            elif( oneFixDone==False ):
                fixed = True # It has been fixed!!!:


            # Restore the value of oneFixDone for next loop and change loop size
            oneFixDone = False
            nLinesUpDown = debugDict['y_clear_midUp']['nLines'] +
debugDict['y_clear_midDown']['nLines']
            counterUp = 0
            counterDown = 0

    return midUp, midDown, 0

##################################################
#                   FIXING CASES                 #
##################################################


def fixCaseA(image, nLine):
    # Draw a line  in order to separate two fusioned lines
    onlyN = np.zeros_like(image)
    onlyN[image == nLine] = nLine
    #mpl.image.imsave("fixCaseADebug-00.png", image, cmap=createCmap())
    #mpl.image.imsave("fixCaseADebug-01.png", onlyN, cmap=createCmap())
```

```
    line =
createLinearRegressionBackbone(onlyN.astype('bool').astype('int32')).astype('
int32')
    #mpl.image.imsave("fixCaseADebug-02.png", line, cmap=mpl.cm.binary)

    onlyN[line == 1] = 0

    # Re-use code my friend
    fix, n_fix= removeSmallCC(onlyN)
    #mpl.image.imsave("fixCaseADebug-04.png", fix, cmap=createCmap())


    if (n_fix != 3): # background + 2 more
        raise ValueError("A fix couldn't help")

    fix[fix==1] = nLine
    fix[fix==2] = nLine + 1

    # shift all the numbers beyond the one given
    for i in reversed(range(nLine + 1, np.unique(image).size)):
        image[image == i] = i+1

    image = np.maximum(image, fix)

    return image



LEFT_TOLERANCE = 2000
RIGHT_TOLERANCE = 1000
def fixCaseB(image, nLine):
    # If one gap has been spotted, both sides must be joined. The following
number
    # to the one spotted MUST be the other half of the line, but we are not
sure
    # if it's the left one or the right one.
    # We could just change numbers of this and the following line, but this
way
    # give us more chance to detect wether it's a case B or anything else.

   # mpl.image.imsave("fixCaseBDebug-00.png", image, cmap=createCmap())

    height, width = image.shape
    # Analyse the horizontal position of the lines
    leftIndices  = []
    rightIndices = []


    for i in range(height):
        rowToAnalyse = image[i][:]
        nonZeroIndices = np.nonzero(rowToAnalyse)
        if(nonZeroIndices[0].size > 0):
            if(nonZeroIndices[0][0] < LEFT_TOLERANCE):
                leftIndices.append(nonZeroIndices[0][0])
            if(nonZeroIndices[0][-1] > RIGHT_TOLERANCE):
                rightIndices.append(nonZeroIndices[0][-1])


    leftPos = np.array(leftIndices).mean()
    rightPos = np.array(rightIndices).mean()
```

```
    print("leftPos - rightPos = " + str(leftPos) + " - " + str(rightPos))

    # Now extract N and N+1 lines
    extract = np.zeros_like(image)
    extract[image == nLine] = nLine
    extract[image == nLine + 1] = nLine + 1

    #mpl.image.imsave("fixCaseBDebug-01.png", extract, cmap=createCmap())

    for i in range(nLine, nLine + 2):
        # Regression line
        (Y, X) = np.where(extract == i)
        (m, b) = np.polyfit(X, Y, 1)

        # The linear regression polynomial will be evaluated from the leftPos
and rightPos calculated before
        x_linearRegression = np.array(range(int(np.floor(leftPos)),
int(np.ceil(rightPos))))

        # The line MUST be thick or else it could have diagonals and
"removeSmallCC" will not count them properly
        for j in [-1, 0, +1]:
            y_linearRegression = np.around(np.polyval([m, b+j],
x_linearRegression)).astype(int)

            # We fill the coordinates we got with the number of the line
            extract[y_linearRegression, x_linearRegression] = i

    #mpl.image.imsave("fixCaseBDebug-02.png", extract, cmap=createCmap())

    # We do the labelling again
    fix, n_fix= removeSmallCC(extract.astype('bool').astype('int32'))

    #mpl.image.imsave("fixCaseBDebug-03.png", fix, cmap=createCmap())

    if n_fix != 2: # background + only 1 more
        raise ValueError("B fix couldn't help")

    else:

        # Change the following color to the one given
        image[image == nLine + 1] = nLine
        fix[fix == 1] = nLine

        # shift all the numbers beyond the one given
        for i in range(nLine + 2, np.unique(image).size):
            image[image == i] = i-1

        # merge fix and image
        image = np.maximum(image, fix)

        #mpl.image.imsave("fixCaseBDebug-04.png", image, cmap=createCmap())


    return image

#######################################################
#                   OTHER FUNCTIONS                   #
#######################################################
import scipy
CC_AREA_TOLERANCE = 8000
```

```
def removeSmallCC(image):
    # Extract all the conected components from image
    label_img, cc_num = scipy.ndimage.label(image)
    # CC = scipy.ndimage.find_objects(label_img)

    # Calculate the length of every CC
    cc_areas = scipy.ndimage.sum(image, label_img, range(cc_num+1))

    # Mark any area below the tolerance limit and remove it (0 = background)
    area_mask = (cc_areas < CC_AREA_TOLERANCE)
    label_img[area_mask[label_img]] = 0

    # Now we may have a messy list with some terms missing (e.g: 0, 1, 2, 4,
6, 8)
    textLineNumber = 0
    frequencyAnalysis = np.unique(label_img)
    for i in sorted(frequencyAnalysis):
        label_img[label_img == i] = textLineNumber
        textLineNumber = textLineNumber + 1

    return label_img, textLineNumber


THOLD_TOLERANCE = 60
def detectSameHeight(image, nLine):
    # Function that detects if two lines are at the same height, by Julia
Moreno Casanova
    # First, we set the flagD to zero, which means that we haven`t spot a
Case yet
    flagD=0
    # Now extract N and N+1 lines
    extract = np.zeros_like(image)
    extract[image == nLine] = nLine
    extract[image == nLine + 1] = nLine + 1




    #mpl.image.imsave("fixCaseBDebug-01.png", extract, cmap=createCmap())

    # We must chek if there are two points we the same height (each one must
below to a different line of course)
    # nLine: check the points of the right side of the line
    # nLine: check the points of the left side of the line

    (Y_nLine, X_nLine) = np.where(extract == nLine)
    (Y_nLine1, X_nLine1) = np.where(extract == (nLine +1))
    length_nLine = len(X_nLine)
    length_nLine1 = len(X_nLine1)

    # So we dont run out of indexes in the following for's
    if (length_nLine < length_nLine1):
        actual_length = length_nLine

    else:
        actual_length = length_nLine1

    # With this for we check:
    # whether the height of the points forming the right half of the line
(Y_nLine[i])
    # is equal to the height of the points forming the left half of the other
line(Y_nLine1[j])
```

```
    cont = actual_length -1 # To not create another for
   # To determine that a detected Case C is truly a Case c and
   # not two letters on different lines that are at the same height.
    thold_case = 0
   #int(actual_length/2)-1
    for i in range(cont,0,-1):

        j = cont -i

        if (Y_nLine[i] == Y_nLine1[j]):

            thold case = thold case + 1

            if (thold_case >= THOLD_TOLERANCE):
               # We have detected the C case
               flagD = 1
            else:
               flagD = 0



    return
```

B.2 Códigos del directorio */algorithm/*

B.2.1 Código del Notebook *CNN_thickBackbone.ipynb* (Google Colab)

```
!nvidia-smi

################################################################################
# 1.In order to take the files that we have in the drive, we have to mount
the unit
################################################################################
from google.colab import drive
drive.mount('/content/gdrive')

################################################################################
# 2.We create a Path in order to make it easier to use the file with
different
# file structures
################################################################################
mainPath = '/content/gdrive/My Drive/TFM/lineseg/'
import sys
sys.path.append(mainPath + 'PythonFiles')

# If we use the DIVA dataset

xPath = mainPath + 'tensors/tensorsDIVA/input/x/'
yPath = mainPath + 'tensors/tensorsDIVA/thickBackboneTensors/y/'
modelPathCP_TBB = mainPath + 'models/contest_model_thickBackbone_DIVA.h5'
modelPath_TBB = mainPath + 'models/contest_model_thickBackbone_DIVA.json'

"""
# If we use the ICDAR2013 dataset
```

```
xPath = mainPath + 'tensors/tensorsICDAR2013/input/x/'
yPath = mainPath + 'tensors/tensorsICDAR2013/thickBackboneTensors/y/'
modelPathCP_TBB = mainPath + 'models/contest_model_thickBackbone_ICDAR13.h5'
modelPath_TBB = mainPath + 'models/contest_model_thickBackbone_ICDAR13.json'
"""
historyPath = mainPath +'models/History_TBB.npy'
# We create a path to save the images and then visualize them
ResultsPath = mainPath + 'Results/results_TBB/'

#############################################################################
# 3.We import what we need and check whether the directories referred to in
the above
# paths already exist or not
#############################################################################

import DataGen_2048_dataAug_rotated as DataGen
import numpy as np
import Unet_2048
import matplotlib as mpl
import matplotlib.pyplot as plt
from PIL import Image
from keras.callbacks import EarlyStopping, ModelCheckpoint
from segmentation import funSegFus
from groundTruthFunctions import makeDirIfNotExist

# Check if the directory exists and creates it does not. We do that for every
Path already defined

makeDirIfNotExist(mainPath)
makeDirIfNotExist(mainPath + 'PythonFiles')
makeDirIfNotExist(mainPath + 'models')
makeDirIfNotExist(xPath)
makeDirIfNotExist(yPath)
makeDirIfNotExist(ResultsPath)

#############################################################################
# 4.We create all the necessary objects for the training
#############################################################################

DataBaseType = 0 # 1 is for ICDAR13 and 0 is for DIVA

if DataBaseType:
  IMAGES_HEIGHT = 4096
  IMAGES_WIDTH = 3072
  indexes = np.arange(1, 349) # from 001 to 348
  train_indexes = indexes[:199]
  test_indexes  = indexes[200:]
  validation_size = 40
  np.random.seed(42)
  np.random.shuffle(train_indexes)
  data_aug = True
elif (DataBaseType == 0):
  IMAGES_HEIGHT = 5120
  IMAGES_WIDTH = 3584
  indexes = np.arange(1, 121) # from 001 to 120
  train_indexes = indexes[:89]
  test_indexes  = indexes[90:]
  validation_size = 30

  data_aug = False
```

```
model_thickBackbone = Unet_2048.UNet(int(IMAGES_HEIGHT/2),
int(IMAGES_WIDTH/2))
model_thickBackbone.compile(optimizer="adam", loss="binary_crossentropy",
metrics=["acc"])
model_thickBackbone.summary()

epochs = 5000
batch_size = 1



list_validation = list(train_indexes[0:validation_size-1])
list training = list(train indexes[validation size:])

# We print the lists to see what they have
print("Validation list = " + str(list_validation))
print("Training list = " + str(list_training))

#############################################################################
# 5.We generate the Data
#############################################################################
train_gen= DataGen.DataGen(list_training, batch_size=batch_size, x_path=
xPath, y_path= yPath, zig_zag=False,
                                half_outline = 0,zig_zag_rotated= False,
data_augmentation=data_aug, IMAGES_HEIGHT = IMAGES_HEIGHT, IMAGES_WIDTH =
IMAGES_WIDTH,
                                DataBaseType = DataBaseType)
valid_gen = DataGen.DataGen(list_validation, batch_size=batch_size, x_path=
xPath, y_path=yPath, zig_zag=False,
                                half_outline = 0,zig_zag_rotated= False,
data_augmentation=False, IMAGES_HEIGHT = IMAGES_HEIGHT, IMAGES_WIDTH =
IMAGES_WIDTH,
                                DataBaseType = DataBaseType)

#  So we don't run out of input data
train_steps = len(list_training)//batch_size
valid_steps = len(list_validation)/batch_size

# But when we introduce the patience, we got a problem, since we set
patience=3, we won't get the best model, but the model three epochs after the
best model.
earlyStoppingCallback = EarlyStopping(monitor='val_acc', mode='max',
patience=3)

# Solution: checkpoint to save the weights of the best
chkpoint   = ModelCheckpoint(filepath=modelPathCP_TBB  , monitor='val_acc',
mode='max', save_weights_only=True, save_best_only=True)

callbackList   = [earlyStoppingCallback, chkpoint]

# We save the model
model_json_thickBackbone = model_thickBackbone.to_json()
with open(modelPath_TBB, "w") as json_file:
    json_file.write(model_json_thickBackbone)

#############################################################################
# 6.We train the network with the training package
#############################################################################

history_TBB = model_thickBackbone.fit(train_gen, validation_data=valid_gen,
steps_per_epoch=train_steps, validation_steps=valid_steps, epochs=epochs,
verbose=1, callbacks=callbackList)
```

```
################################################################################
# 7.We save the history
################################################################################

np.save(historyPath,history_TBB.history)

################################################################################
# 8.We show and save the figures
################################################################################
history_TBBload=np.load(historyPath,allow_pickle='TRUE').item()

# Plot training & validation accuracy values
plt.plot(history_TBB.history['acc'])
plt.plot(history_TBB.history['val_acc'])
plt.title('Model accuracy')
plt.ylabel('Accuracy')
plt.xlabel('Epoch')
plt.legend(['Train', 'Validation'], loc='upper left')
plt.show()

# Plot training & validation loss values
plt.plot(history_TBB.history['loss'])
plt.plot(history_TBB.history['val_loss'])
plt.title('Model loss')
plt.ylabel('Loss')
plt.xlabel('Epoch')
plt.legend(['Train', 'Validation'], loc='upper left')
plt.show()

################################################################################
# 9.We analyze the training and generate the test data
################################################################################
# Load the weights
model_thickBackbone.load_weights(modelPathCP_TBB)

batch_size_test = 3
list_test = list(test_indexes)
test_steps = len(list_test)//batch_size_test
test_gen= DataGen.DataGen(list_test, batch_size=batch_size_test,
x_path=xPath, y_path=yPath,
                                          zig_zag=False, half_outline = 0,
zig_zag_rotated= False, data_augmentation=False,
                                           IMAGES_HEIGHT = IMAGES_HEIGHT,
IMAGES_WIDTH = IMAGES_WIDTH,DataBaseType = DataBaseType)

score = model_thickBackbone.evaluate(test_gen, steps=test_steps,verbose=1)
print(score)
################################################################################
# 10.We check the result visually
################################################################################
# TBB
# Returns x,y, which are of type DataGen
x_test_TBB, y_test_TBB = test_gen.load_single_image(40)

# Test Input
x_test_TBB = x_test_TBB.astype('int8')
mpl.image.imsave(ResultsPath + "TBB_TESTX.png",
x_test_TBB.reshape(int(IMAGES_HEIGHT/2), int(IMAGES_WIDTH/2)),
cmap=mpl.cm.binary)
plt.figure(figsize=(12,20))
```

```
plt.imshow(x_test_TBB.reshape(int(IMAGES_HEIGHT/2), int(IMAGES_WIDTH/2)),
cmap=mpl.cm.binary)
plt.axis("off")
# Test Output
y_test_TBB = y_test_TBB.astype('int8')
mpl.image.imsave(ResultsPath +"TBB_TESTY.png",
y_test_TBB.reshape(int(IMAGES_HEIGHT/2), int(IMAGES_WIDTH/2)),
cmap=mpl.cm.binary)

plt.figure(figsize=(12,20))
plt.imshow(y_test_TBB.reshape(int(IMAGES_HEIGHT/2), int(IMAGES_WIDTH/2)),
cmap=mpl.cm.binary)
plt.axis("off")

# Prediction


y_predicted = model_thickBackbone.predict(x_test_TBB.reshape(1,
int(IMAGES_HEIGHT/2), int(IMAGES_WIDTH/2), 1).astype('float32'))

y_predicted = np.round(y_predicted).astype('bool')

mpl.image.imsave(ResultsPath +"TBB_PREDICTEDY.png",
y_predicted.reshape(int(IMAGES_HEIGHT/2), int(IMAGES_WIDTH/2)),
cmap=mpl.cm.binary)
plt.figure(figsize=(12,20))
plt.imshow(y_predicted.reshape(int(IMAGES_HEIGHT/2), int(IMAGES_WIDTH/2)),
cmap=mpl.cm.binary)
plt.axis("off")
#############################################################################
# 11.Now we get the final prediction: we modify segmentation.py to make it
easier to
# get the predictions from the test images: To see this, we must go to
CNN_ZigZag
#############################################################################
```

B.2.2 Código del Notebook *CNN_ZigZag.ipynb* (Google Colab)

```
!nvidia-smi

#############################################################################
# 1.In order to take the files that we have in the drive, we have to mount
the unit
#############################################################################
from google.colab import drive
drive.mount('/content/gdrive')

#############################################################################
# 2.We create a Path in order to make it easier to use the file with
different
# file structures
#############################################################################
mainPath = '/content/gdrive/My Drive/TFM/lineseg/'
datasetPath = '/content/gdrive/My Drive/TFM/DataBases/'
import sys
sys.path.append(mainPath + 'PythonFiles') # Here we save the required
files.py
```

```
# If we use the DIVA dataset
xPath = mainPath + 'tensors/tensorsDIVA/input/x/'
yPath = mainPath + 'tensors/tensorsDIVA/zigzagTensors/y/'
shapesPath = datasetPath + 'DIVA-HisDB/gt_shape/'
gt_pagesPath = datasetPath + 'DIVA-HisDB/test_train/gt_PAGE_xml_TASK2/'
modelPathCP_TBB = mainPath + 'models/contest_model_thickBackbone_DIVA.h5'
modelPath_TBB = mainPath + 'models/contest_model_thickBackbone_DIVA.json'
modelPathCP_ZZ = mainPath + 'models/contest_model_UD_DIVA.h5'
modelPath_ZZ = mainPath + 'models/contest_model_UD_DIVA.json'
# We create a path to save the images and then visualize them
ResultsPath = mainPath + 'Results/results_ZBB/'
# We create a path to save the segmentation results
segResultsPath = mainPath + 'Results/Segmentation_results/SEG_DIVA/'

"""
# If we use the ICDAR2013 dataset
xPath = mainPath + 'tensors/tensorsICDAR2013/input/x/'
yPath = mainPath + 'tensors/tensorsICDAR2013/zigzagTensors/y/'
shapesPath = datasetPath + 'ICDAR13/shape/'
modelPathCP_TBB = mainPath + 'models/contest_model_thickBackbone_ICDAR13.h5'
modelPath_TBB = mainPath + 'models/contest_model_thickBackbone_ICDAR13.json'
modelPathCP_ZZ = mainPath + 'models/contest_model_UD_ICDAR13.h5'
modelPath_ZZ = mainPath + 'models/contest_model_UD_ICDAR13.json'
# We create a path to save the images and then visualize them
ResultsPath = mainPath + 'Results/results_ZBB/'
# We create a path to save the segmentation results
segResultsPath = mainPath + 'Results/Segmentation_results/SEG_ICDAR2013/'
"""
historyPath = mainPath + 'models/History_UD.npy'
###############################################################################
# 3.We import what we need and check whether the directories referred to in
the above
# paths already exist or not
###############################################################################

import DataGen_2048_dataAug_rotated as DataGen
import numpy as np
import Unet_2048
import matplotlib as mpl
import matplotlib.pyplot as plt
from PIL import Image
from keras.callbacks import EarlyStopping, ModelCheckpoint
from segmentation import funSegFus
from groundTruthFunctions import makeDirIfNotExist

# Check if the directory  exist and creates it does not. We do that for every
Path already defined

makeDirIfNotExist(mainPath)
makeDirIfNotExist(datasetPath)
makeDirIfNotExist(mainPath + 'PythonFiles')
makeDirIfNotExist(mainPath + 'models')
makeDirIfNotExist(xPath)
makeDirIfNotExist(yPath)
makeDirIfNotExist(ResultsPath)
makeDirIfNotExist(segResultsPath)
makeDirIfNotExist(segResultsPath + 'evaluation_software_input/') # only
required for ICDAR 2013  dataset
makeDirIfNotExist(shapesPath)
makeDirIfNotExist(gt_pagesPath) # only required for DIVA dataset
```

```
###############################################################################
# 4.We create all the necessary objects for the training
###############################################################################

DataBaseType = 0 # 1 is for ICDAR13 and 0 is for DIVA

if DataBaseType:
  IMAGES_HEIGHT = 4096
  IMAGES_WIDTH = 3072
  indexes = np.arange(1, 349) # from 001 to 348
  train_indexes = indexes[:199]
  test_indexes  = indexes[200:]
  validation_size = 40
  data_aug = True #(only for the training and test set)
  np.random.seed(42)
  np.random.shuffle(train_indexes)

elif (DataBaseType == 0):
  IMAGES_HEIGHT = 5120
  IMAGES_WIDTH = 3584
  indexes = np.arange(1, 121) # from 001 to 120
  train_indexes = indexes[:89]
  test_indexes  = indexes[90:]
  validation_size = 30
  data_aug = False # Because if not, we run out of memory (only for the
training and test set)


model_UD = Unet_2048.UNet(int(IMAGES_HEIGHT/2),int(IMAGES_WIDTH/2))
model_UD.compile(optimizer="adam", loss="binary_crossentropy",
metrics=["acc"])
model_UD.summary()

epochs = 5000
batch_size = 1



list_validation = list(train_indexes[0:validation_size-1])
list_training = list(train_indexes[validation_size:])

# We print the lists to see what they have
print("Validation list = " + str(list_validation))
print("Training list = " + str(list_training))

print(len(list_training))

###############################################################################
# 5.We generate the Data
###############################################################################
train_gen_UD = DataGen.DataGen(list_training, batch_size=batch_size, x_path=
xPath, y_path= yPath, zig_zag=False,
                               half_outline = 0,zig_zag_rotated= True,
data_augmentation=data_aug,IMAGES_HEIGHT = IMAGES_HEIGHT, IMAGES_WIDTH =
IMAGES_WIDTH,
                               DataBaseType = DataBaseType)
valid_gen_UD = DataGen.DataGen(list_validation, batch_size=batch_size,
x_path= xPath, y_path=yPath, zig_zag=False,
                               half_outline = 0,zig_zag_rotated= True,
data_augmentation=False,IMAGES_HEIGHT = IMAGES_HEIGHT, IMAGES_WIDTH =
IMAGES_WIDTH,
```

```
                                    DataBaseType = DataBaseType)

#  So we don't run out of input data
train_steps = len(list_training)//batch_size
valid_steps = len(list_validation)/batch_size


################################################################################
# 6.CHECK: Does the change we have made to obtain the rotated image with the
rotated
# label work? Does the random generation mode of these samples work? We can
check it
################################################################################
"""
#Input
x_train_UD = x_train_UD
print('Shape of x:', x_train_UD.shape)
plt.figure(figsize=(12,20))
plt.imshow(x_train_UD[1,:,:,0].astype('uint8'), cmap=mpl.cm.binary) #we get
the 0 degree image
plt.axis("off")

# Output (Label)
# y_train_UD = y_train_UD.astype('int8')
print('Shape of y:', y_train_UD.shape)
plt.figure(figsize=(12,20))
plt.imshow(y_train_UD[1,:,:,0].astype('uint8'), cmap=mpl.cm.binary)
plt.axis("off")

"""
# But when we introduce the patience, we got a problem, since we set
patience=3, we won't get the best model, but the model three epochs after the
best model.
earlyStoppingCallback = EarlyStopping(monitor='val_acc', mode='max',
patience=3)

# Solution: checkpoint to save the weights of the best

chkpoint_UD   = ModelCheckpoint(filepath=modelPathCP_ZZ  , monitor='val_acc',
mode='max', save_weights_only=True, save_best_only=True)

callbackList_UD   = [earlyStoppingCallback, chkpoint_UD]

# We save the model
model_json_UD = model_UD.to_json()
with open(modelPath_ZZ, "w") as json_file:
    json_file.write(model_json_UD)


################################################################################
# 7.We train the network with the training package
################################################################################

print('UD')
history_UD = model_UD.fit(train_gen_UD, validation_data=valid_gen_UD,
steps_per_epoch=train_steps, validation_steps=valid_steps, epochs=epochs,
verbose=1, callbacks=callbackList_UD)

################################################################################
# 8.We save the history
################################################################################
np.save(historyPath,history_UD.history)
```

```
##############################################################################
# 9.We show and save the figures
##############################################################################
history_UDload=np.load(historyPath,allow_pickle='TRUE').item()

#UD
# Plot training & validation accuracy values
plt.plot(history_UDload['acc'])
plt.plot(history_UDload['val_acc'])
plt.title('Model accuracy')
plt.ylabel('Accuracy')
plt.xlabel('Epoch')
plt.legend(['Train', 'Validation'], loc='upper left')
plt.show()

# Plot training & validation loss values
plt.plot(history_UDload['loss'])
plt.plot(history_UDload['val_loss'])
plt.title('Model loss')
plt.ylabel('Loss')
plt.xlabel('Epoch')
plt.legend(['Train', 'Validation'], loc='upper left')
plt.show()
##############################################################################
# 10.We analyze the training
##############################################################################
# Load the weights
model_UD.load_weights(modelPathCP_ZZ)
print('Best weights loaded')

batch_size_test = 3
list_test = list(test_indexes)
test_steps = len(list_test)//batch_size_test
print("Test list = " + str(list_test))
# We generate the test data
test_gen_UD = DataGen.DataGen(list_test, batch_size=batch_size_test,
x_path=xPath, y_path=yPath,
                                        zig_zag=False, half_outline = 0,
zig_zag_rotated= True, data_augmentation=False,
                                        IMAGES_HEIGHT = IMAGES_HEIGHT,
IMAGES_WIDTH = IMAGES_WIDTH,DataBaseType = DataBaseType)

score_UD = model_UD.evaluate(test_gen_UD, steps=test_steps,verbose=1)
print(score_UD)

##############################################################################
# 11.We check the result visually
##############################################################################
# UD
# Returns x,y, which are of type DataGen
x_test_UD, y_test_UD = test_gen_UD.load_single_image(91)

# Test Input
x_test_UD = x_test_UD.astype('int8')
mpl.image.imsave(ResultsPath + "UD_TESTX.png",
x_test_UD.reshape(int(IMAGES_HEIGHT/2), int(IMAGES_WIDTH/2)),
cmap=mpl.cm.binary)
print('Test: Input X')
plt.figure(figsize=(12,20))
plt.imshow(x_test_UD.reshape(int(IMAGES_HEIGHT/2), int(IMAGES_WIDTH/2)),
cmap=mpl.cm.binary)
plt.axis("off")
```

```
# Test Output
y_test_UD = y_test_UD.astype('int8')
mpl.image.imsave(ResultsPath + "UD_TESTY.png",
y_test_UD.reshape(int(IMAGES_HEIGHT/2), int(IMAGES_WIDTH/2)),
cmap=mpl.cm.binary)
print("Test: Y")
plt.figure(figsize=(12,20))
plt.imshow(y_test_UD.reshape(int(IMAGES_HEIGHT/2), int(IMAGES_WIDTH/2)),
cmap=mpl.cm.binary)
plt.axis("off")
#############################################################################
# 12.Now we get the final prediction:
#############################################################################

if DataBaseType:

  listPathsSegFus={"route_xPath": xPath,\
                           "route_shape": shapesPath,\
                           "route_modelPathCP_ZZ":modelPathCP_ZZ,\
                           "route_modelPathCP_TBB":modelPathCP_TBB,\
                           "route_modelPath_ZZ":modelPath_ZZ,\
                           "route_modelPath_TBB":modelPath_TBB,\
                           "route_resultsPath":segResultsPath}
  complete_list = np.arange(1, 349)
elif (DataBaseType == 0):

  listPathsSegFus={"route_xPath": xPath,\
                           "route_shape": shapesPath,\
                           "route_gt_page": gt_pagesPath,\
                           "route_modelPathCP_ZZ":modelPathCP_ZZ,\
                           "route_modelPathCP_TBB":modelPathCP_TBB,\
                           "route_modelPath_ZZ":modelPath_ZZ,\
                           "route_modelPath_TBB":modelPath_TBB,\
                           "route_resultsPath":segResultsPath}
  complete_list = np.arange(1, 121)



funSegFus(complete_list, listPathsSegFus,DataBaseType)

#############################################################################
# 13.We load some test image:
#############################################################################

# Load the image
image = Image.open(segResultsPath + '001-10x_segmentated.png')
plt.figure(figsize=(12,20))
plt.imshow(image)
plt.axis("off")
```