Trabajo de Fin de Grado
Grado en Ingeniería de las Tecnologías de Telecomunicación

Herramienta para segmentación de líneas de texto en imágenes basada en Deep Learning

Autor: José Luis Mendoza Sánchez

Tutor: Juan José Murillo Fuentes

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

Sevilla, 2020

Trabajo de Fin de Grado
Grado en Ingeniería de las Tecnologías de Telecomunicación

Herramienta para segmentación de líneas de texto en imágenes basada en Deep Learning

Autor:
José Luis Mendoza Sánchez

Tutor:
Juan José Murillo Fuentes
Catedrático de Universidad

Dpto. de Teoría de la Señal y Comunicaciones
Escuela Técnica Superior de Ingeniería
Universidad de Sevilla
Sevilla, 2020

Trabajo de Fin de Grado: Herramienta para segmentación de líneas de texto en imágenes basada en Deep Learning

Autor: José Luis Mendoza Sánchez

Tutor: Juan José Murillo Fuentes

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

Presidente:

Vocales:

Secretario:

Acuerdan otorgarle la calificación de:

Sevilla, 2020

El Secretario del Tribunal

A mi familia

A mis amigos

A mis maestros

Agradecimientos

En primer lugar, quería dar las gracias a mi familia, por apoyarme de la mejor forma que han sabido durante estos años de mi juventud. En especial, gracias a mi hermano por mostrarme un ejemplo de superación y resiliencia, y a mis padres por tener siempre una actitud positivista en la vida. Sin vosotros, nada de esto hubiera sido posible.

En segundo lugar, a mis amigos y a mi pareja, por comprenderme y soportarme en todo momento, por escucharme y por poder contar con todos en los peores momentos. Por las risas, los llantos, los momentos felices y todo el cariño que me dais, gracias.

Por supuesto, no me puedo olvidar de agradecer este proyecto a Juan José Murillo Fuentes, mi tutor, por la paciencia que ha tenido conmigo desde el primer momento, y que aceptó tutelarme sabiendo que yo no era ningún experto en el tema tratado. De usted me llevo la imagen de un magnífico tutor que ha sabido responder a todas mis dudas sin importarle las horas que le ha llevado. Muchas gracias por su dedicación y por su pasión por la enseñanza.

A mis compañeros de proyectos, Juan, Natalia y Andrés, que han conseguido hacer ameno el tiempo pasado en la escuela. Por ser un equipo de trabajo *perfecto* y acabar forjando una bonita amistad, gracias.

Para finalizar, me gustaría dar un agradecimiento a todos los profesores y profesoras que han puesto empeño, ganas y esfuerzo en mi educación, desde preescolar hasta la universidad, y que me han acompañado en este largo viaje.

A todos vosotros, muchas gracias

José Luis Mendoza Sánchez

20 de junio de 2020

Resumen

Este proyecto trata de utilizar las ventajas de la Inteligencia Artificial y el Deep Learning para la creación de una herramienta capaz de segmentar las líneas de un texto. La utilidad de esta herramienta radica principalmente en conformar la primera pieza de un programa mayor, como por ejemplo uno de reconocimiento óptico de caracteres, o cualquier otra en la que se necesite distinguir las líneas de un texto manuscrito. Estas herramientas tienen aplicación a nivel mundial en ámbitos que abarcan desde la digitalización de documentos antiguos hasta la traducción simultánea de texto manuscrito.

Para el código, hemos utilizado principalmente las bibliotecas Keras y TensorFlow 2.0 de Python, junto con muchas otras como Numpy, Matplotlib, Scipy o OpenCV2 para el tratamiento digital de imágenes. La base de datos utilizada para el entrenamiento ha sido la que se ofreció a los concursantes del ICDAR 2013 Handwriting Segmentation Contest.

Además, esta memoria intenta documentar el proceso seguido, de manera que el lector pueda tomarla como una guía para su propio proyecto de Machine Learning, entender los fallos que se han cometido y evitarlos a toda costa. También encontrará un capítulo entero donde encontrará un listado de referencias que han sido útiles para aprender esta tecnología partiendo de cero, además de otros conceptos que hemos preferido explicar por nosotros mismos.

Abstract

This project tries to take advantage of the Artificial Intelligence and Deep learning technologies for the creation of a tool to segment the lines of a text. The utility of this tool consists mainly in being the first piece of a bigger one, like an Optical Character Recognition software (OCR) or any other one which needs to distinguish the lines of a handwritten text. These are the tools used all over the world for different applications varying from old handwritten documents digitalization to simultaneous translation of texts.

For the code, we have used mainly the Keras and TensorFlow2.0 frameworks for Python, along with other libraries like Numpy, Matplotlib, Scipy and OpenCV2 for digital image processing. The dataset used for the training has been the one given to the participants of the ICDAR 2013 Handwriting Segmentation Contest.

Also, this memory tries to keep record of the process, so that the reader can take it as a guide for his or her own Machine Learning project, understand the mistakes made and avoid them at all costs. The reader will also find a whole chapter, the second one, where he or she would encounter a list of references useful for learning this technology from scratch, in addition to some other ideas we have preferred to explain ourselves.

Índice

ÍNDICE DE TABLAS

ÍNDICE DE FIGURAS

1 INTRODUCCIÓN

El campo de la inteligencia artificial y del tratamiento de textos manuscritos han ido de la mano casi desde el principio de sus días. Son muchas las herramientas que existen a nivel de usuario para el reconocimiento óptico de caracteres (OCR), algunas de ellas incluso con traducción simultánea. Además, hay también otras dirigidas a investigadores de todo el mundo que están destinadas al reconocimiento de textos antiguos, y se utilizan para digitalizar imágenes de archivos para su posterior estudio.

Estas tareas tan complejas necesitan ser apropiadamente divididas para que se puedan resolver por módulos. El primero de ellos con total seguridad siempre va a ser el de detectar cuáles son las líneas de texto que han de ser procesadas, y una distinción entre ellas. Es este primer problema el que intenta resolver este proyecto, intentando documentar todo el proceso y compartiendo todos los avances para que aquel que lo necesite pueda aprovecharse de la parte que más le beneficie.

Como alumno de la especialidad de Telemática, a fecha de comenzar este trabajo, todo lo que conocía acerca de la Inteligencia Artificial era el nombre, y no hablemos siquiera acerca del tratamiento digital de imágenes. Fueron mis ganas de aprender acerca del tema lo que me motivaron a elegir este trabajo y no otro en el cual profundizase más en cualquiera de los conocimientos que ya tenía. Este largo camino de aprendizaje me ha ayudado a ser más autodidacta y a aprender conceptos que desconocía totalmente.

Romper las barreras entre las especialidades de nuestro grado es visto por algunos alumnos como algo indeseable y cercano incluso al pecado. Sin embargo, desde mi punto de vista creo que este ejercicio nos ayuda a tener una perspectiva más amplia al finalizar la carrera. Por eso, recomiendo a todos los estudiantes a que se animen a investigar temas más allá de la etiqueta que ellos mismos se han impuesto como "área de especialización", y se embarquen en proyectos que supongan una oportunidad para aprender un área que nunca antes habían conocido.

Aprender por uno mismo el tema de Machine Learning está al alcance de cualquiera en estos días. Contamos con multitud de recursos en internet que están puestos a nuestra disposición de manera completamente gratuita y que son de especial utilidad. Tal es la cantidad de recursos, que en ocasiones pueden abrumar al aprendiz por no saber por dónde puede empezar. Por eso queremos dedicar una parte de este proyecto a esos alumnos que puedan verse beneficiados con la lectura de esta memoria, y que les ayude a empezar más alto en el tema de la Inteligencia Artificial y llegar aún más lejos en sus conocimientos.

1.1 Objetivos del Proyecto

El objetivo principal del proyecto es el de *segmentar las líneas de texto de un manuscrito*, implementando para ello el DeepLearning y utilizando el lenguaje de programación Python. Profundizaremos en los tipos de segmentación que existen, en algunas investigaciones que se han hecho sobre el tema, en los problemas que surgen en este tipo de proyectos y en cómo resolverlos.

El segundo objetivo es el de *documentar* todo el proceso de la mejor manera posible, dejando así constancia de los errores cometidos (que no han sido especialmente pocos) y cómo hemos podido superarlos, con la esperanza de que algún día esta memoria llegue a manos de alguien a quien le pueda servir de gran ayuda.

El tercer objetivo es la *reproducibilidad* del proyecto. Para ello, hemos dejado un repositorio, donde se podrá encontrar el código utilizado en el proyecto y también los datos de los que hace uso. El código está completamente en inglés y exhaustivamente comentado para que también lo puedan utilizar en todo el mundo como un ejemplo un poco más avanzado de uso de redes neuronales para el tratamiento de imágenes. Para mayor comodidad, hemos creado un Anexo que servirá como guía para indicar para qué sirve cada uno de los archivos que se encuentran en el repositorio.

El último objetivo ha sido el de *aprender* acerca del campo del Machine Learning y reflejar el aprendizaje para ayudar a quien lo deba hacer también. Durante el trayecto, hemos encontrado multitud de fuentes, algunas de mayor utilidad que otras, que han conseguido guiarnos para hacer este proyecto. Es por ello que en el punto 2.1 (y en general en toda la memoria) hemos intentado plasmar todas aquellas referencias que, en nuestra opinión, pueden ayudar mejor a aquellos que necesiten iniciarse en el tema.

1.2 Bibliotecas y frameworks utilizados

Todos los proyectos de Machine Learning pueden realizarse, con las bases matemáticas suficientes, directamente en código nativo de cualquier lenguaje de programación. Sin embargo, esta es una tarea tediosa que nadie hace, ya que existen diferentes librerías y *frameworks* que nos reducen inmensamente la cantidad de código y hacen muchísimas tareas por nosotros. Vamos a revisar brevemente las que hemos utilizado en este proyecto:

- **Keras**: es una librería de Python, *open-source*, que contiene implementaciones de los "**bloques**" más utilizados en redes neuronales, como por ejemplo las capas, funciones de activación, optimizadores… y también en redes neuronales convolucionales (usadas en este proyecto) como las capas de *pooling* o los filtros. Referencia [3].
- **TensorFlow**: es otra librería de código abierto creada por Google que ofrece un ***backend*** (principalmente de operaciones matemáticas) a Keras para su correcto funcionamiento. En un principio, TensorFlow y Keras nacieron como dos proyectos separados, soportando Keras a diferentes librerías que le hacían de *backend*. Sin embargo, con el paso de los años, la línea que divide a estos proyectos se ha ido difuminando, ya que la versión "multi-*backend*" de Keras se abandonó y nadie utiliza otro que no sea TensorFlow. Para que veamos un ejemplo de para qué sirve esta librería, cuando en keras utilizamos el bloque de convolución, el responsable de hacer realmente la operación matemática de la convolución es TensorFlow. Más información en: referencia [4].
- **Scikit-learn**: aunque no la hemos utilizado en este proyecto, la mencionamos aquí porque es una librería que complementa a las otras dos anteriores ofreciendo multitud de herramientas para el **procesado de datos** y también permite utilizar **modelos preprogramados** de redes neuronales. Con estos últimos, perdemos flexibilidad a la hora de configurar nuestros modelos (porque vienen "prefabricados"), pero reducimos al máximo el número de líneas necesarias para realizar tareas que quizás no requieren un alto grado de precisión.
- **Numpy**: archiconocida librería de Python de lenguaje matemático. En ella están definidos los objetos Numpy.Array (**matrices**) y multitud de funciones relacionadas. Documentación en la referencia [5].
- **Matplotlib**: utilizada para realizar gráficos (***plots***) con la opción de utilizar mapas de colores. [6].
- **PILLOW**: para cargar y guardar **imágenes** en múltiples formatos. Documentación en [7].
- **OpenCV2**: para el tratamiento digital de **imágenes** y operaciones **matemáticas**. Referencia [8].
- **Scipy**: librería **científica** con el mismo propósito que cv2, pero con funciones diferentes.

Además del trío formado por Keras - TensorFlow – Scikit-Learn, la compañía Facebook sacó otra librería para Python y C++ llamada **PyTorch**. La principal diferencia entre TensorFlow y PyTorch es que la primera utiliza computación de grafos de manera estática, lo que se traduce en que el modelo que quieras entrenar debe estar completamente creado antes del momento de su entrenamiento. En PyTorch, sin embargo, la computación es dinámica, lo que permite que el modelo pueda ser manipulado durante la ejecución, pudiendo por ejemplo variar el tamaño de la entrada, algo útil por ejemplo en redes neuronales recurrentes. Además, la comunidad de PyTorch es indiscutiblemente más reducida que la de TensorFlow.

Finalmente, decidimos utilizar Keras porque la solución que ofrece es más robusta (lleva más tiempo), y los recursos disponibles sobre los que basar el trabajo son más amplios. Además, nuestros motivos también fueron académicos, ya que parte del aprendizaje se hizo gracias a un libro llamado "Hands-On Machine Learning with Scikit-Learn, Keras, and TensorFlow – Aurélien Géron" (referencia [1]), que trata exclusivamente los *frameworks* que lleva en el título. Aún así, realmente merecería la pena valorar la implementación de PyTorch.

1.3 Estructura de la memoria

Esta memoria trata de reflejar el trabajo que se ha realizado durante la creación de una herramienta para la segmentación de líneas de texto en imágenes basada en Deep Learning, intentando reflejar los fallos cometidos y las estrategias adoptadas para solventarlos. De esta manera, la memoria podría considerarse una guía de los pasos que hemos seguido que, si bien han servido para resolver un problema muy específico, podrían asemejarse a los que se deben dar para cualquier proyecto de inteligencia artificial. Cada capítulo de este texto trata de describir cada uno de estos pasos.

Pasaremos primero por un par de nociones previas, dando una serie de referencias para asegurarnos de que el lector pueda estar informado acerca de los conceptos más básicos que hay detrás del campo de la inteligencia artificial en el caso de no estarlo. Además, explicaremos una serie de ideas más específicas que son aplicadas en el mundo de las imágenes digitales. Por último, abarcaremos el problema de la segmentación de texto manuscrito, los distintos acercamientos que se pueden tomar a la hora de atacar el problema, las bases de datos que existen y las soluciones que han sido publicadas en distintos artículos académicos acerca de este tema. Todo esto lo veremos en los **capítulos 2 y 3** de esta memoria.

Una vez hechos los pasos anteriores, nos fijamos un objetivo claro: intentar sacar una segmentación de las líneas del texto como podemos ver en la siguiente imagen:

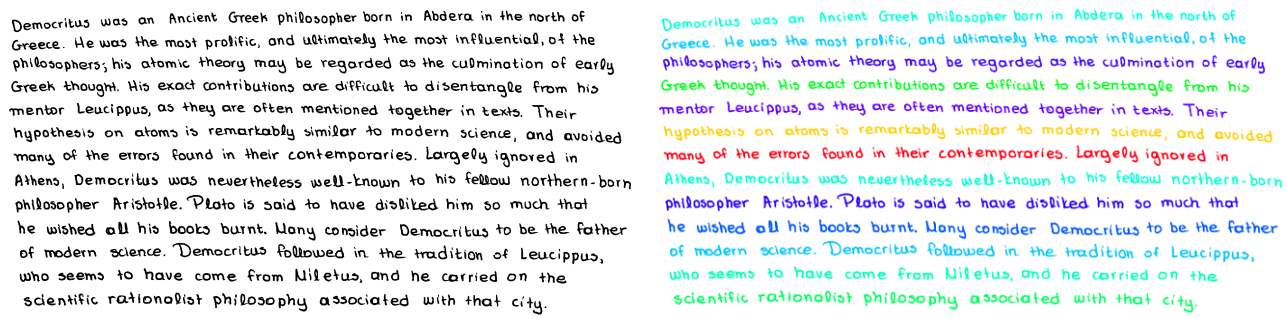


Figura 1-1. Un texto manuscrito y su segmentación

Donde:

- La imagen de la izquierda está representada en memoria como una matriz donde los píxeles de color blanco tienen el valor “0” y los de color negro tienen el valor “1”. Esta será la entrada de nuestro programa.
- La imagen de la derecha también es una matriz con los píxeles de color blanco de valor “0” (como en la original), mientras que los píxeles que antes eran negros ahora tendrán un valor “n” asociado a cada línea, es decir, que los píxeles de la línea 1 tienen el valor “1”, los de la línea 2 tienen el valor “2” y así sucesivamente. Aunque *a priori* haya líneas que tienen el mismo color (como por ejemplo la 3 y la 5), en realidad estos dos colores no son exactamente los mismos (no tienen el mismo código) y la similitud se debe a las limitaciones de los mapas de colores que hemos utilizado. Por ello, debemos tener presente en todo momento que los píxeles de colores que vamos a ver a lo largo de este texto representan celdas de matrices en memoria con valores numéricos distintos y consecutivos.

Aclaradas nuestras intenciones, comenzaremos un camino con el objetivo de alcanzar un algoritmo que sea capaz de crear una especie de *máscara* donde los gruesos de las líneas estén diferenciados. De esta manera, podremos poner la máscara encima de un texto, para que, comparando uno a uno los píxeles de la misma con los del texto, podamos obtener una segmentación como la que hemos visto. Acompañamos la explicación con un par de imágenes:

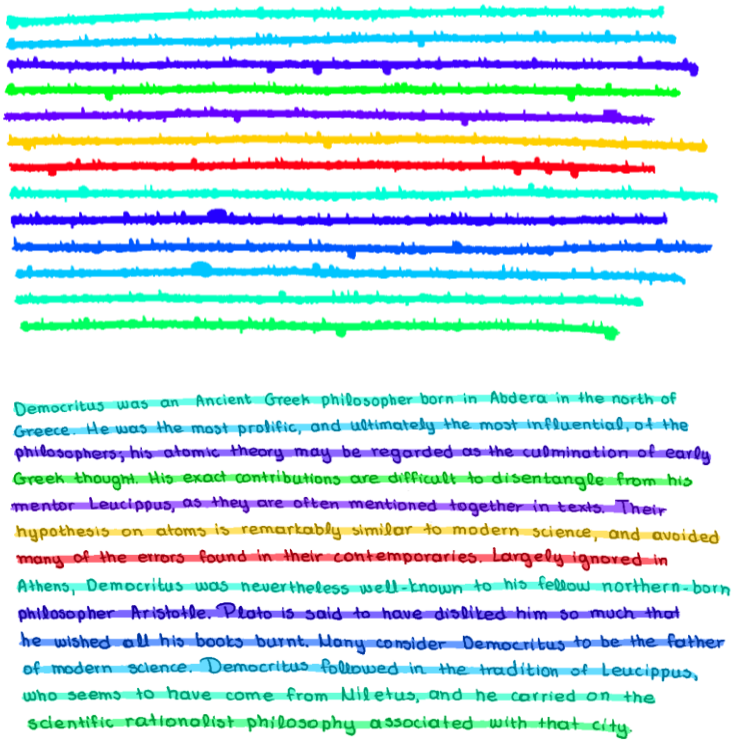


Figura 1-2. Máscara (arriba) y máscara encima del texto (abajo).

Así, si los píxeles que son "1" (negros, texto) en la imagen manuscrita original los igualaremos al valor que tenga la máscara en esas coordenadas, obteniendo así nuestra deseada segmentación descrita en la Figura 1-1.

El principal obstáculo que tenemos para la creación de la *máscara* es que en los textos manuscritos suele haber caracteres que se tocan entre sí, pero pertenecen a líneas distintas. A estos *caracteres conflictivos* se les puede separar de distintas formas, pero nosotros hemos decidido usar una que tiene el siguiente fundamento:

- Nos ayudaremos de 3 modelos distintos de redes neuronales, que se encargarán de extraer las partes superiores, las partes inferiores y los esqueletos de las líneas. Podemos ver las tres partes del texto anterior en la figura siguiente:

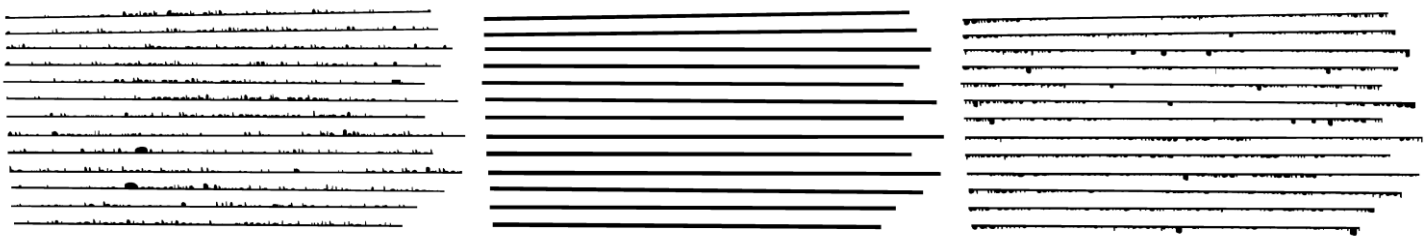


Figura 1-3. Partes superiores, esqueletos y partes inferiores de las líneas

- Mediante un algoritmo, haremos una relación univoca entre cada trio de partes superiores-medias-inferiores de cada línea, de manera que cada trío forme una línea de la máscara. De esta manera, al haber separado las líneas en tres partes, por mucho que el texto manuscrito presentase *caracteres conflictivos*, era muy baja la probabilidad de que dos partes superiores seguidas (o dos partes inferiores) reflejasen este defecto.

Para alimentar a los tres modelos de redes neuronales con pares de entradas – etiquetas, primero debemos crear varias modificaciones del dataset encontrado, que pasaremos a tratar en el **capítulo 4**, donde además hablaremos de un concepto muy extendido entre los modelos que tratan con imágenes: el *Data Augmentation*.

En el **capítulo 5**, veremos 4 soluciones que hemos tenido que implementar para solventar el problema de las exigencias de memoria que requería el entrenamiento de las redes.

Más tarde, en el **capítulo 6**, explicaremos el algoritmo que fusiona las partes superiores-medias-inferiores predichas por nuestras redes, puliendo los errores que presenta y realizando la segmentación.

Por último, en el **capítulo 7** analizaremos los resultados obtenidos, comparándolos con los proyectos de otros centros de investigación que concursaron en el ICDAR 2013 Handwriting Segmentation Contest.

2 REDES NEURONALES Y REDES NEURONALES CONVOLUCIONALES

En este capítulo vamos a realizar un breve repaso por algunos conceptos que usaremos a lo largo de toda esta memoria. Supondremos que el lector tiene claros ciertos conocimientos muy básicos relacionados con el Machine Learning y la Inteligencia Artificial en general. Si no es su caso, dejaremos una lista de conceptos y una serie de fuentes donde pueden ser consultados para que pueda seguir leyendo sin problemas esta memoria.

2.1 Conocimientos previos y dónde encontrarlos

Para poder seguir avanzando, el lector debe estar familiarizado con los siguientes conceptos:

- Qué es el Machine Learning y el Deep Learning
- Qué es una red neuronal completamente conectada (*fully connected*).
- Qué son las funciones de coste en una red neuronal
- Qué son las funciones de activación.
- Cómo funciona el algoritmo de *Backpropagation* y el del descenso del gradiente.
- El concepto de *overfitting*, *underfitting* y cómo prevenirlos.
- Diferencias entre el aprendizaje supervisado y no supervisado
- División entre conjuntos de entrenamiento, de validación y de test en los dataset.

Si desea explorar todos estos conceptos básicos, tenemos dos fuentes que nos han sido clave para la elaboración de este proyecto:

- [ESPAÑOL] El canal de youtube DotCSV, de Carlos Santana Vega, ingeniero informático y divulgador de conocimientos sobre inteligencia artificial. Debido a la multitud de vídeos que existen, hemos decidido dejar una lista de reproducción pública con los vídeos que creemos que son más importantes. Dicha lista de reproducción está en el siguiente link:

  https://www.youtube.com/playlist?list=PLx3EIRLhRAK6oovkZV8OjSI4CdRHaYo8H

  En concreto, los últimos 3 vídeos tratan aspectos un poco más avanzados y mencionaremos estos vídeos a lo largo de la memoria.
- [INGLÉS] El capítulo 1 del maravilloso libro de Aurélien Géron, que podemos encontrar en la referencia [1].

Además de estas nociones sobre inteligencia artificial, el lector también debe estar familiarizado con la manera en la que se guardan imágenes en un ordenador. A modo de resumen, una imagen es representada como una matriz donde cada entrada codifica la intensidad de luminosidad de un píxel. En imágenes en tonos de grises, este valor suele ser discreto, entre el 0 y el 255 en el caso de usar 8 bits por píxel. Estas imágenes en tonos de grises pueden volverse coloridas si utilizamos un *mapa de colores*, que relacionará cada valor asignado a un color guardado en forma de mapa. El problema es que limitamos el abanico de colores al número de colores de nuestro mapa. Por eso, si una imagen requiere un espectro más amplio, se suelen usar varias matrices para representar la descomposición del color de un píxel en una paleta de colores. Al número de matrices que se

utilizan para dar color a una misma imagen se le conoce como el *número de canales* que tiene dicha imagen. Habitualmente, para imágenes en color se utilizan 3 canales para determinar la intensidad del color Rojo – Verde – Azul (RGB por sus siglas en inglés) de cada píxel. Más información en la referencia [9].

Una vez que tenemos claros los fundamentos sobre inteligencia artificial, vamos a introducir una variante de la red completamente conectada (*fully connected*) que tiene especial utilidad cuando tratamos con imágenes.

2.2 Redes neuronales convolucionales

Cuando las entradas de una red neuronal son imágenes, se introducen nuevas herramientas para resolver el problema. La teoría nos dice que puedes crear una red neuronal convencional lo suficientemente grande como para que cada pixel de la entrada se conecte con una neurona y que recorra todo un camino de neuronas hasta llegar a la salida, creando así un modelo de red neuronal que funcione con imágenes. Sin embargo, en la práctica estos modelos son sumamente ineficientes y se dejan para casos donde las entradas con la que alimentamos nuestra red son inconexas entre sí.

Una imagen es todo lo contrario a un tipo de dato inconexo, ya que existen múltiples estructuras presentes en ella en base a las cuales podemos tomar decisiones para resolver nuestro problema. Estas estructuras se buscan gracias a filtros que realizan la operación de convolución en 2 dimensiones (que ahora pasaremos a ver con más detalle). Los pesos que tienen estos filtros en nuestra red, en general, se inician de manera completamente aleatoria, y será gracias las parejas de entradas – etiquetas con la que alimentemos a nuestra red que irán tomando forma hasta llegar a ser lo más óptimos posibles.

Además de la convolución, debido a la gran cantidad de valores que puede tener una imagen, también se utiliza otra operación llamada *pooling* o *downsampling*, que introduciremos más adelante.

2.2.1 Operación de convolución

Una convolución de 2 dimensiones en el área del procesamiento de imágenes se refiere a una operación matemática por la cual pasamos una matriz llamada **filtro** o *kernel* por encima de una imagen y realizamos el sumatorio de la multiplicación elemento a elemento del filtro con la imagen original. El filtro debe ser, por lo general, de dimensiones más pequeñas que la imagen, habiendo por lo tanto un número n de posiciones posibles donde aplicar el filtro. La operación de convolución implica obtener una matriz de n elementos resultado de aplicar la operación descrita anteriormente en las n posiciones posibles. Veámoslo con un ejemplo:

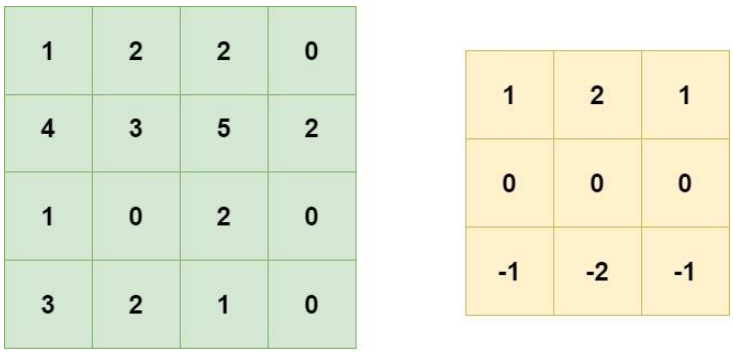


Figura 2-1. Ejemplo de convolución 2-D. A la izquierda la imagen y a la derecha el kernel

Tengamos una imagen de un solo canal de 4x4 pixeles de resolución y cuyas intensidades de píxeles son las representadas a la izquierda de la figura y un filtro o *kernel* como el representado a la derecha. Si intentamos superponer el filtro encima de la imagen, hay 4 posibles posiciones donde la podemos poner si queremos que todas las celdas del filtro toquen a una celda de la imagen. Esto nos da una idea del tamaño de la matriz resultante de realizar la convolución, que tendrá 4 elementos. Así, vamos a ver cómo quedaría el primer paso:

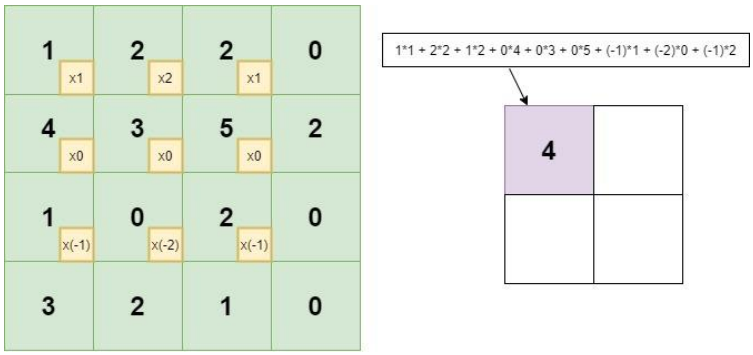


Figura 2-2. Primer paso para la convolución

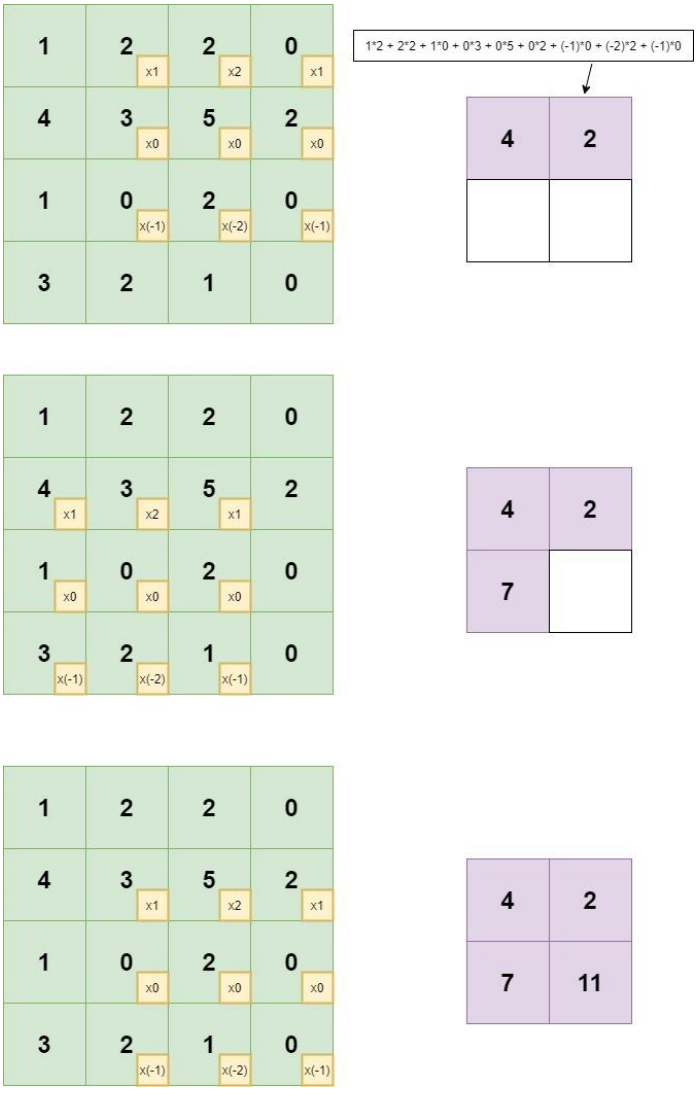


Figura 2-3. Resto de pasos para la convolución.

Una vez sabemos cómo realizar convoluciones, hay 2 datos adicionales que debemos conocer:

- *Stride*: si queremos, podemos hacer que de un elemento a otro de la convolución en lugar de desplazarse 1 pixel a la derecha o hacia abajo se desplace dando un pequeño **salto**. Cuando ocurre esto, decimos que hay un *stride* (*paso* en español) diferente a 1.

- *Padding*: (***relleno*** en español) en ocasiones, deseamos que la imagen resultante tenga las mismas dimensiones que la original. Para conseguir eso, se pueden añadir 0 fuera de la imagen original hasta que las dimensiones sean tales que la convolución dé lugar a una imagen de igual resolución.

Hay multitud de filtros con valores y formas predeterminadas como podemos ver en la referencia [10]. Estos son útiles para por ejemplo localizar bordes, añadir ruido gaussiano… Sin embargo, los filtros que utilizamos en las redes neuronales convolucionales deberán *aprender* qué forma adoptar para ser útiles en nuestra resolución del problema.

Para profundizar más en el tema de la convolución de imágenes, consultar la referencia [11].

2.2.2 Operación de pooling o submuestreo

En la mayoría de redes convolucionales, además de capas de convolución también hacen falta capas que rebajen la resolución de las imágenes. A estas capas se les llaman capas de *pooling*, o de ***downsampling***, o de submuestreo. La operación de submuestrear consiste en coger tan solo 1 píxel de cada grupo de n x m pixeles de la imagen. El **criterio** para cogerlo variará según lo que queramos hacer. Así, lo más común es:

- Quedarnos con el máximo de los valores de esa región (*MaxPooling*).
- Quedarnos con el mínimo (*MinPooling*).
- Quedarnos con una media (*AvgPooling*).

Aquí dejamos un ejemplo de submuestreo con criterio de función el máximo.

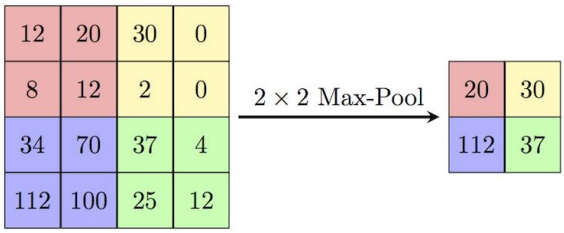


Figura 2-4. Ejemplo de MaxPooling sacado de la referencia [12].

Además de esta operación, también existe una "operación inversa" que sirve para aumentar la resolución de una imagen. El problema es que durante el submuestreo perdemos parte de la información, y por tanto la operación inversa ya no podrá restaurar la imagen como era originalmente. La operación inversa se llama ***upsampling***, y la intuición que hay detrás de esta operación es la de un algoritmo de interpolación que intenta predecir (por supuesto con mayor o menor precisión) aquellos píxeles nuevos que añadirle a una imagen.

2.2.3 Tipos de problemas que se resuelven con redes convolucionales

Hay multitud de problemas que tienen como entrada una imagen e intentan realizar algo con ella.

- **Clasificación**: Las aplicaciones más comunes son las que tienen como objetivo clasificar una imagen. Así, por ejemplo, el reconocimiento de objetos simples o el reconocimiento de caracteres manuscritos son dos tipos de problemas que encajan en esta definición. Las redes convolucionales que resuelven estos problemas se caracterizan por utilizar **capas de convolución** y de ***pooling*** hasta llegar a una dimensionalidad muy baja. Una vez que la tenemos, se pasa la imagen de 2 dimensiones a 1 (con la operación *flatten*) y se utiliza una red neuronal convencional para devolver un vector de categorías donde se determina la probabilidad de que la imagen pertenezca a cualquiera de las categorías que le hemos dicho que aprenda. Un ejemplo de este estilo es la clasificación de números manuscritos del 1 al 9 cuya implementación podemos encontrar en muchísimos sitios en internet. De todos ellos, hemos decidido destacar una aplicación web interactiva que permite visualizar el comportamiento interno de la red y que encontramos en la referencia [13].
- **Clasificación y localización**: además de devolver un vector de categorías, también devolverá de alguna manera las coordenadas que han influido en la decisión final.
- **Detección de objetos**: cuando en la imagen cabe la posibilidad de clasificar más de una categoría y

devolver las localizaciones de las mismas, decimos que el problema pasa a ser de detección de objetos. Como ejemplo de esta red podemos poner al algoritmo de YOLO (*You Only Look Once* por sus siglas en inglés). Podemos encontrar más información acerca de este en la referencia [14].

- **Segmentación de imágenes**: es una aplicación más específica de la detección de objetos puesto que también requiere que se devuelvan los píxeles exactos donde se encuentran los objetos.

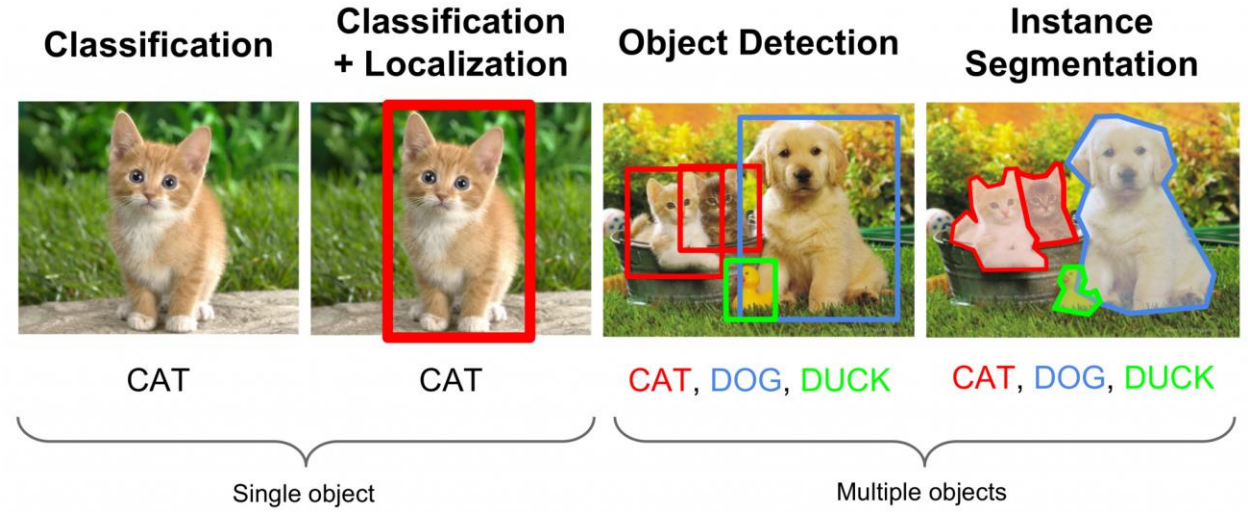


Figura 2-5. Distintos tipos de problemas donde se aplican redes convolucionales.

También existen proyectos que persiguen objetivos muy particulares como por ejemplo el de dar **color a imágenes en blanco y negro** [15], **aumentar la resolución** de imágenes (con métodos más precisos que la interpolación) [16], **aumentar la fluidez de un vídeo** [17] o incluso la **transferencia de estilos artísticos entre imágenes** [18].

Figura 2-6. Transferencia de estilo artístico entre imágenes

Si las explicaciones acerca de las redes convolucionales que hemos presentado en este capítulo han sabido a poco, se puede encontrar otro punto de vista por ejemplo en la referencia [19].

3 PASOS PREVIOS A LA ESCRITURA DEL CÓDIGO

Uno de los pasos más importantes que se debe dar a la hora de resolver un problema es el correcto análisis del mismo. Esta sección refleja el trabajo previo que se realizó para nuestro trabajo y que debería de servir de ejemplo para cualquier otro problema en el que estén involucrados los algoritmos de Deep Learning e Inteligencia Artificial.

3.1 Tipos de segmentación

Lo primero que debemos hacer es investigar las posibles soluciones que ya existen en el mercado ante nuestro problema y realizar un listado de las mismas. Así, descubrimos que en nuestro caso la "Segmentación de texto manuscrito" es un campo en el que se podrían ubicar distintas herramientas que resolverían un problema bien distinto las unas de las otras. Así, elaboramos esta lista con los posibles tipos de segmentación que podemos realizar ante un texto escaneado o fotografiado.

1. **Detección de los bordes de un folio**: es digno de mención este tipo de segmentación, aunque de manera rigurosa no forme parte del grupo de "Herramientas de segmentación de **líneas de texto**". Esta herramienta está ampliamente implementada en multitud de softwares para escanear documentos (sobre todo cuando hablamos de ciertas aplicaciones para móviles) y también existen sensores fotoeléctricos para impresoras y escáneres profesionales que permiten realizar esta tarea.

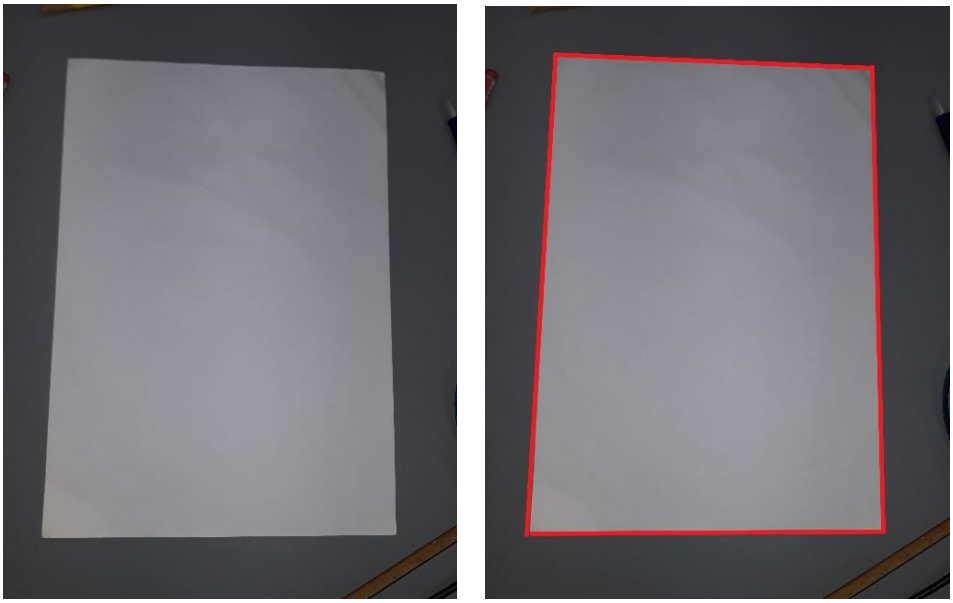


Figura 3-1. Detección de los bordes de un folio.

2. **Delimitación de bloques rectangulares de texto**: Cuando hablamos de texto "puro", nos estamos refiriendo a que no existen bloques con gráficos o figuras dentro del folio que queremos segmentar. El principal problema que tiene este método se da cuando diferentes líneas de texto dentro de un mismo folio tienen direcciones de escritura distinta. Además, en párrafos con una mezcla de líneas cortas y

largas, se seleccionaría un bloque rectangular con la longitud de la línea más larga a la hora de segmentar.

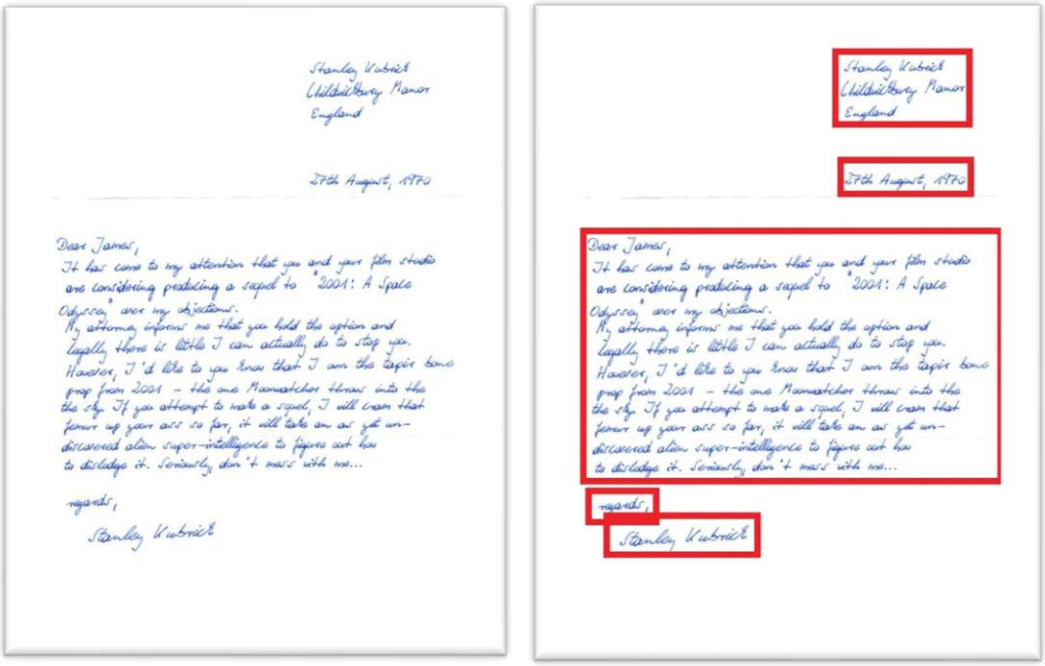


Figura 3-2. Delimitación de bloques rectangulares de texto

3. **Delimitación de bloques poligonales de texto**: Con el fin de eliminar el último problema que tenía el método anterior, se puede realizar una pequeña adaptación si en lugar de detectar bloques rigurosamente rectangulares, realizamos tantos trazos rectos como haga falta para delimitar el texto.

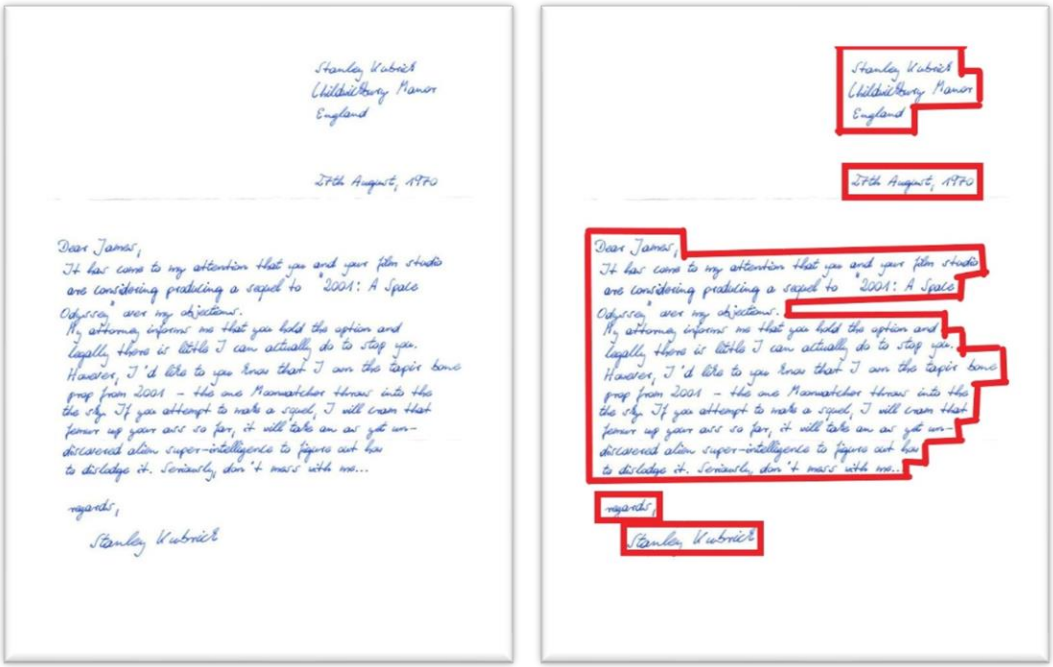


Figura 3-3.Delimitación bloques poligonales de texto.

4. **Segmentación rectangular de líneas de texto**: se trata de aplicar lo que vimos en el punto número 2 para cada línea de texto. Así, no encontraremos ningún problema si una línea de texto es más o menos larga que otra dentro de un mismo párrafo, ya que todas se segmentarían de forma individual. Existe un pequeño problema y es que en ocasiones un carácter de una línea puede llegar hasta otra, y el rectángulo estaría señalándolo a éste como parte de una línea que no le corresponde.

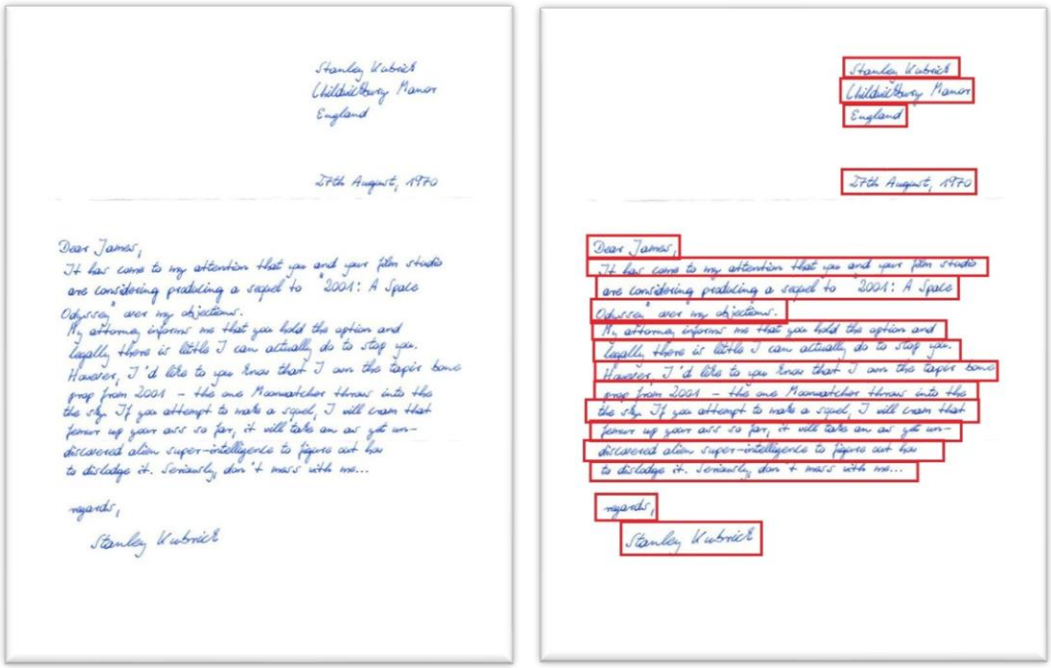


Figura 3-4: Segmentación rectangular de líneas de texto

5. **"Pixel segmentation"**: se trata de un método de segmentación que solo es posible realizar cuando la imagen está en color. Así, se pasaría de dicha imagen con 3 canales (RGB) a una sola en blanco y negro (1 solo canal de tipo "bool") donde se distinguirían las siluetas de las líneas de texto. En algunos artículos académicos, como el [20], este tipo de segmentación es un paso previo a la hora de realizar el resto del trabajo, el cual suele estar más orientado al reconocimiento óptico de caracteres (OCR).

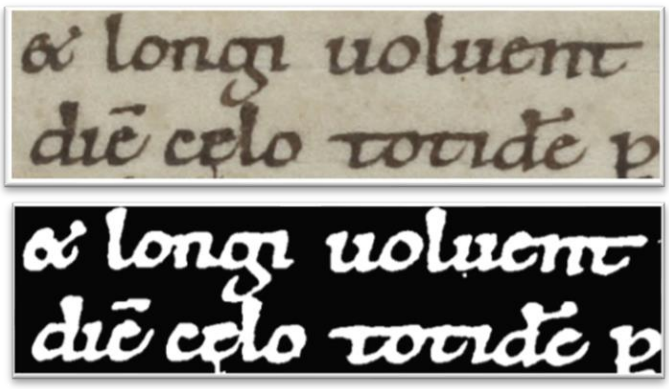


Figura 3-5. Pixel segmentation

6. **Detección de la espina dorsal de cada línea del texto**: Definimos "espina dorsal de una línea del texto" como una recta que cruza dicha línea y determina la dirección principal de la misma. Así, la idea es parecida a la que expresamos cuando hablamos de los bloques rectangulares, solo que en esta

ocasión con un ejercicio de post-procesado podríamos ser capaces de distinguir cuándo un carácter que inintencionadamente ha acabado en otra línea pertenece a una o a la otra.

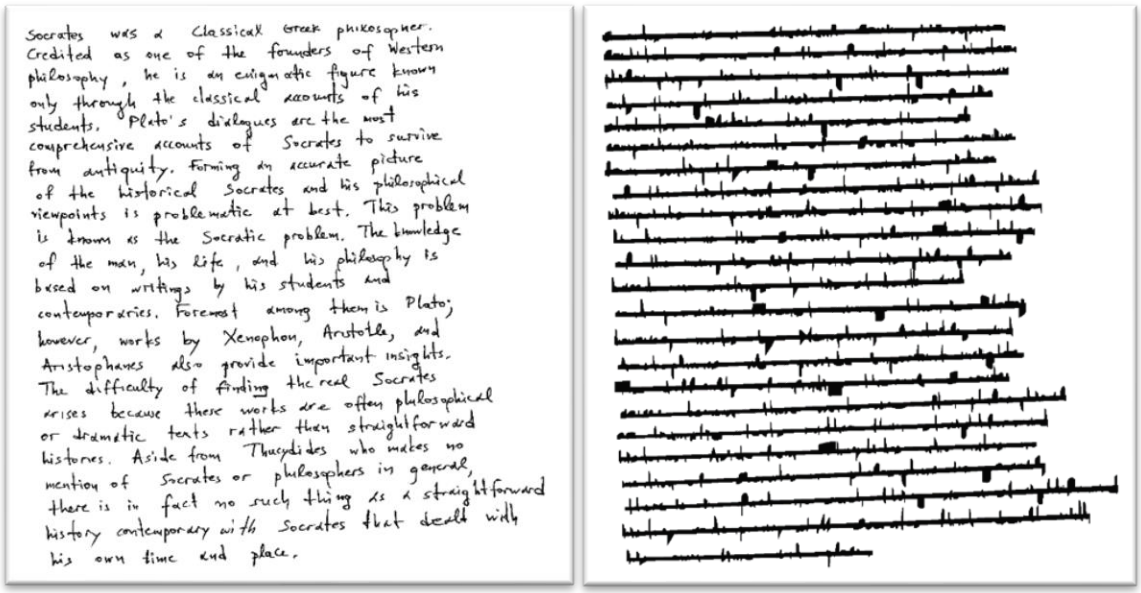


Figura 3-6: Detección de la espina dorsal de cada línea de texto

La idea utilizada proviene del artículo académico [2] que vamos a estudiar a fondo más adelante en esta sección, el cual viene hablando de un "mapa de calor" que muestra donde hay mayor posibilidad de encontrar texto escrito, como podemos ver en la siguiente imagen.

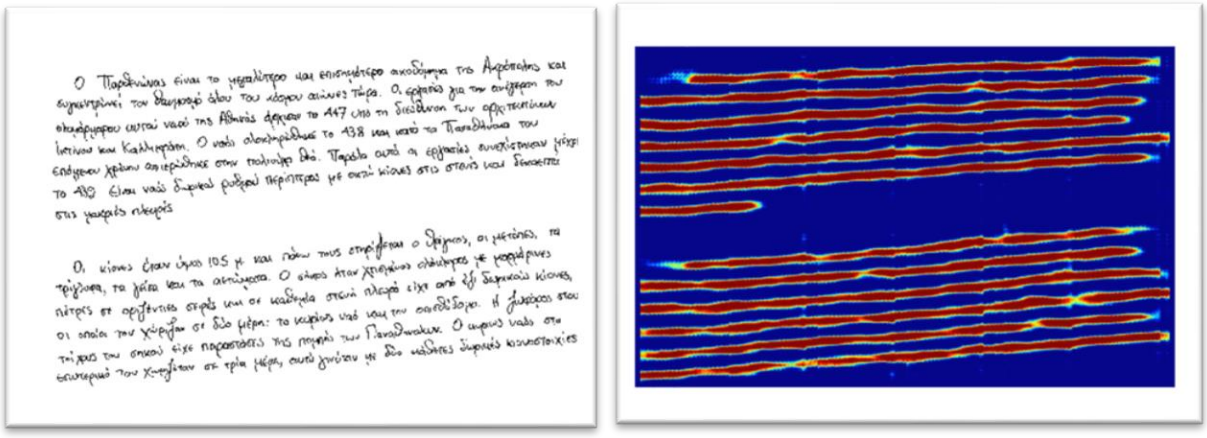


Figura 3-7: Mapa de calor estudiado en REFERENCIA

Para acabar, cabe mencionar que todos los métodos de segmentación que aquí hemos visto no están contemplando la posibilidad de que haya otro tipo de elementos en el folio a analizar más allá de texto puro. Es muy común encontrar textos que incluyan ilustraciones, gráficos, figuras, ecuaciones, filigranas al borde de la página, caracteres con formato distinto (por ejemplo, al iniciar el capítulo de un libro), comentarios, esquemas y un largo etcétera. Así, encontramos algunas herramientas que nos permiten reconocer distintas partes de un mismo texto, como por ejemplo la que podemos encontrar en el artículo académico de la Universitat Politècnica de València (Referencia [21]):

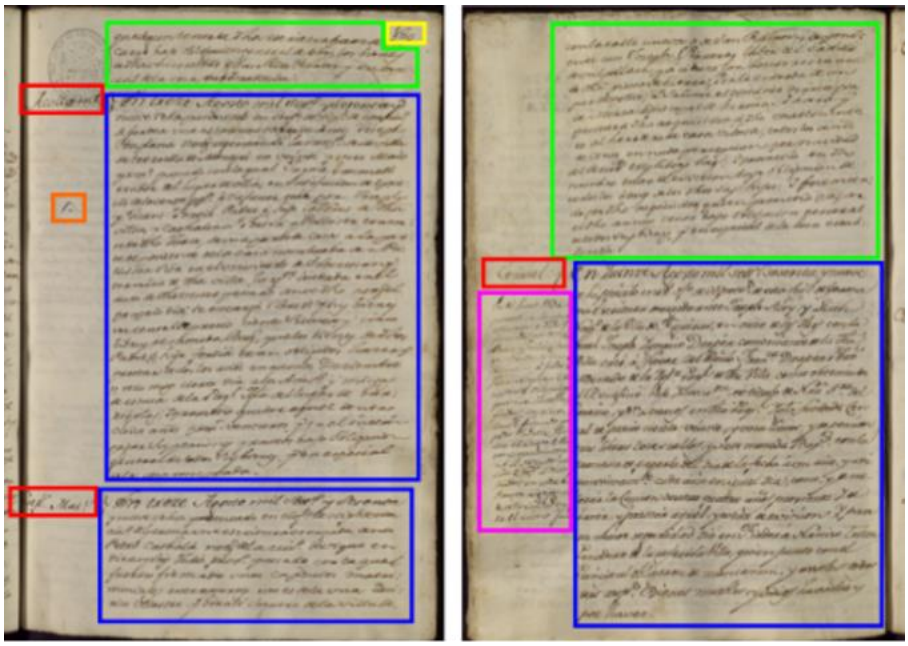


Figura 3-8. Análisis del diseño de página manuscrita. Referencia [21]

3.2 Búsqueda de datasets

Como ya sabemos, para los algoritmos de Deep Learning supervisados es necesario tener una inmensa cantidad de datos etiquetados, a los que llamamos **datasets**, que nos permitan entrenar a nuestra red. Sin embargo, la elaboración de dichos datasets es una tarea muy laboriosa y costosa (hasta el punto de ser prácticamente imposible su creación). Es por ello que debemos recurrir a la multitud de datasets que existen de manera gratuita en la red. Una gran parte de estos datasets han sido creados con el propósito de sentar las bases para un **concurso** donde equipos de investigación de todo el mundo compiten por ver quien puede crear la red neuronal más óptima para resolver un problema concreto.

Así, en el ámbito de la segmentación de líneas de texto, el concurso más famoso que existe precede a la **ICDAR**, la "International Conference on Document Analysis and Recognition". El ICDAR es un congreso celebrado cada 2 años desde 1991 y los artículos académicos derivados del mismo se pueden ver a través de la referencia [22]. Previa a dicha conferencia, los organizadores suelen establecer algunas competiciones relacionadas con el mundo del análisis de documentos. Así, en 2013 publicaron una nueva competición con la cual liberaban un dataset compuesto por un total de 350 imágenes manuscritas (sumando las partes de test y training) en inglés, griego y bengalí. Dicho dataset se puede encontrar en la referencia [23]. Como apunte adicional hemos de decir que los datos que aquí vemos son una ampliación del dataset de la competición de 2011 (donde solo estaban los manuscritos en inglés y griego).

La competición tenía dos modalidades. La primera consistía en crear una herramienta que pudiese segmentar cada pixel de las imágenes y asignarle un valor numérico de manera que todos los píxeles pertenecientes a texto de una misma **línea** tuviesen el mismo valor. La segunda modalidad era básicamente lo mismo, pero aplicando este concepto a cada **palabra** en lugar de a cada línea de texto. Así pues, para verificar la precisión que tenía cada modelo presentado, se añadió en la sección de recursos un **software** que permitía visualizar cada imagen con el ground-truth generado por ellos mismos y el generado por la herramienta del participante. La herramienta software de la que hablamos tiene el siguiente aspecto:

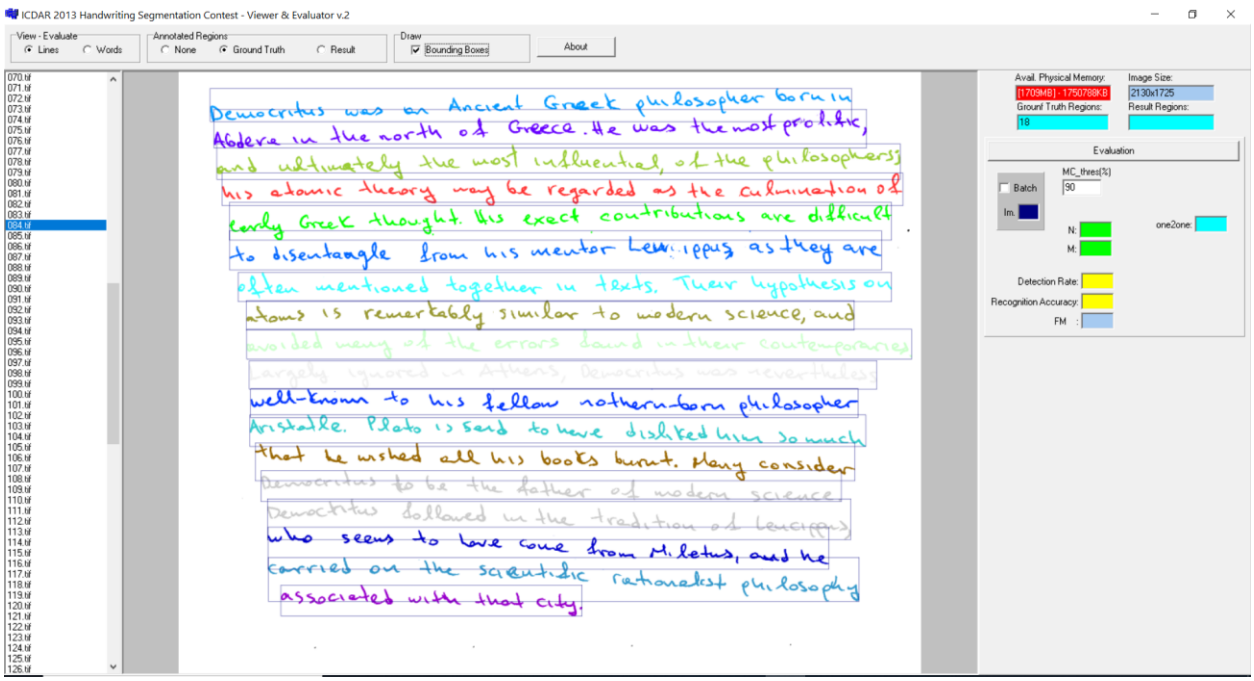


Figura 3-9. Captura de pantalla del software incluido en la competición del ICDAR2013

En nuestro caso, para el proyecto pensamos que debería ser más útil el ground-truth perteneciente a la modalidad de segmentación de líneas de texto. Este dataset prometía ser un candidato perfecto para cumplir el objetivo de nuestro proyecto. El primer problema que tuvimos que afrontar fue el de abrir los archivos del proyecto en python, que, aunque a posteriori parezca una tarea trivial, en el momento fue todo un reto debido a la codificación de las imágenes.

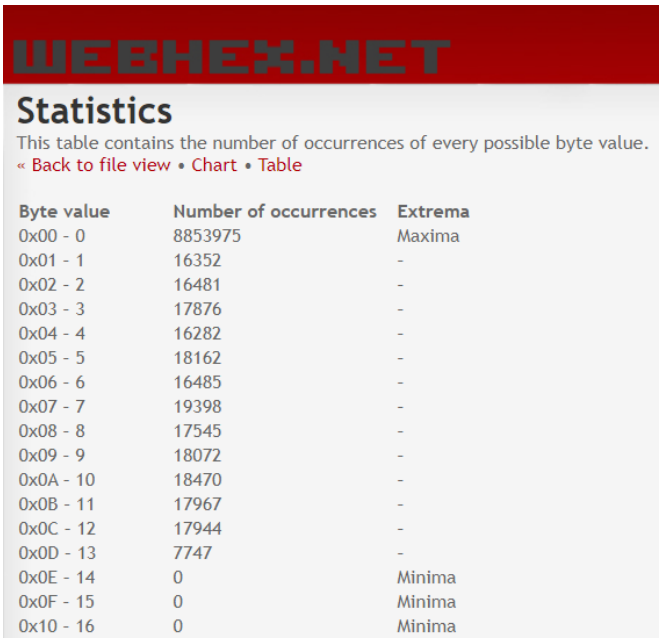
La estructura que tiene el dataset es la siguiente:

- Por un lado, tenemos una colección de imágenes numeradas del 001 al 350 (contando las 200 pertenecientes a lo que ellos usaban como training y las 150 que usaban como test). Las imágenes están guardadas en un formato ".tif", el cuál puede ser abierto por la mayoría de reproductores de imágenes. La distribución de dichas imágenes es:
    - 250 imágenes en griego y en inglés a partes iguales intercalando una en griego y otra en ingles desde las imágenes 001 – 150 (para ellos, las de training) y de la 201 a la 300 (para ellos, las de test).
    - 100 imágenes en bengalí desde las imágenes 151 – 200 (para ellos, las de training) y de la 301 a la 350 (para ellos, las de test).
- Por otro lado (y aquí vino el reto), tenemos una colección de archivos guardados en una carpeta llamada "gt_lines" (que se refería al ground-truth de la modalidad de "líneas") que siguen la misma enumeración de las imágenes mencionadas anteriormente. Dichos archivos tienen la extensión ".tif.dat", de la cual no teníamos ninguna referencia o documentación.

Queremos dejar constancia del proceso de análisis que seguimos para abrir el archivo por si alguien que pueda leer esta memoria se encuentra con una situación similar. El documento 001.tif.dat debería de tener codificado de alguna manera las líneas que tenía la imagen correspondiente 001.tif y tenía las siguientes características:

- El archivo tenía un tamaño total de 9.072.756 bytes, mientras que la imagen tenía una resolución de 2079 x 1091 píxeles. Eso nos hacía pensar que el documento codificada cada píxel de la imagen mediante un valor de 4 bytes (4 x 2079 x 1091 nos da en total los 9.072.756 bytes de tamaño).
- Al abrirlo con un editor de texto, inmediatamente nos dimos cuenta de que la información del archivo estaba codificada de manera binaria y que contenía principalmente espacios en blanco.

- Al saber que estaba en binario, decidimos utilizar una herramienta de análisis de archivos binarios llamada "Webhex" y que podemos encontrar online en la referencia [24]. El análisis estadístico que permite hacer esta herramienta nos dio la siguiente información:



WEBHEX.NET

Statistics

This table contains the number of occurrences of every possible byte value.
« Back to file view • Chart • Table

Byte value	Number of occurrences	Extrema
0x00 - 0	8853975	Maxima
0x01 - 1	16352	-
0x02 - 2	16481	-
0x03 - 3	17876	-
0x04 - 4	16282	-
0x05 - 5	18162	-
0x06 - 6	16485	-
0x07 - 7	19398	-
0x08 - 8	17545	-
0x09 - 9	18072	-
0x0A - 10	18470	-
0x0B - 11	17967	-
0x0C - 12	17944	-
0x0D - 13	7747	-
0x0E - 14	0	Minima
0x0F - 15	0	Minima
0x10 - 16	0	Minima

Figura 3-10. Análisis frecuencial del archivo 001.tif.dat.

La imagen contenía un total de 13 líneas por lo que entendimos que el valor 0 (el más utilizado) correspondía al "background" (el fondo blanco de la imagen) y que cada número estaba asignado a definir el píxel de una línea de texto.

- Por último, en la página de la competición había una nota que decía cómo abrir los archivos en lenguaje C++. Dicha nota se puede encontrar en la referencia [25] y es la siguiente:

```
int Ix; //Image Width (x=0...Ix-1)
int Iy; //Image Height (y=0...Iy-1)
unsigned int *IM_SegmResult; //Pointer to store raw data
AnsiString SegmResultName; //File that contains the raw data
FILE *f1;

IM_SegmResult = (unsigned int *) calloc (Ix*Iy,sizeof(int));
f1 = fopen(SegmResultName.c_str(),"rb");
fread(IM_SegmResult,Ix*Iy,sizeof(int),f1);
fclose(f1);
```

Figura 3-11. Nota para abrir el archivo para la competición del ICDAR2013

Comprendimos entonces que la información que nos aportaba ese archivo era una ristra de bytes donde se almacenaban los píxeles de la imagen guardados de 4 en 4 bytes (sizeof(int)) guardados en formato *little endian*. Es por ello que nos pusimos manos a la obra para realizar esa conversión a mano mediante bucles que recorrieran el archivo para guardarlos en una lista. Nuestro trabajo fue en vano cuando descubrimos que esta función ya está implementada de manera eficiente en la archiconocida librería **NumPy** de Python. Además, para cargar la imagen en sí era suficiente con usar un método implementado en la librería **Pillow**. Por ello, para cargar cada imagen (x) con su ground-truth (y) solo teníamos que hacer:

```
import numpy as np
from PIL import Image

x001 = Image.open(route_image_001)

y001 = np.fromfile(route_lines_001, dtype='int32')
y001 = np.reshape(y001, x001.shape)
```

Figura 3-12. Código utilizado para abrir una imagen del ICDAR2013 en Python

Al realizar un gráfico con la librería **Matplotlyb**, asignando un mapa de color aleatorio (a cada número se le asigna un color distinto), el resultado es el siguiente:

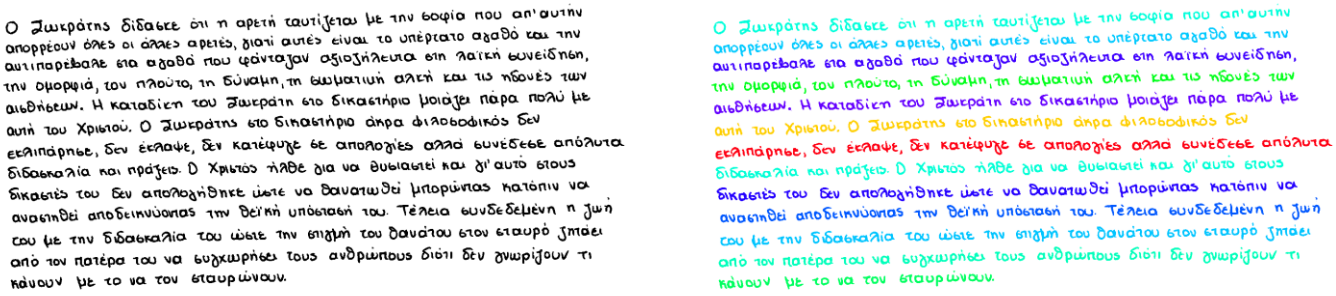


Figura 3-13. Representación de la imagen 001 y su ground-truth.

En memoria, las imágenes son una matriz compuesta únicamente por valores "0" (blanco – *fondo*) y "1" (negro – texto) y hemos usado el mapa de colores *matplotlib.pyplot.cm.binary* para representarla. Los *ground-truth* en cambio, se guardan en memoria como matrices donde el valor "0" sigue representando el blanco – fondo, pero el resto de píxeles que antes eran negros ahora tienen un valor "n" asociado a cada línea. Así, los pixeles pertenecientes a la línea 1 tienen el valor "1" en memoria, los de la línea 2 tienen el valor "2" y así sucesivamente. Para representarlo, hemos usado una variación aleatoria del *matplotlib.pyplot.cm.gist_rainbow*, forzando a que el valor 0 se corresponda con el blanco.

Como comentario adicional queda añadir que las imágenes en bengalí (imágenes número 150–200 y 300–350) tienen una disposición diferente, teniendo los píxeles de la primera línea el valor "51", los de la segunda, "52", etcétera. Por eso lo primero que hicimos (y debería hacer cualquiera que fuese a utilizar este dataset) fue pasar los ground-truth de las imágenes en bengalí a los mismos valores que tenían los demás.

Una vez que hemos hecho este ejercicio de abrir e inspeccionar todo el dataset, ya podemos decir que es el que vamos a utilizar. No obstante, la manipulación del dataset no acaba aquí, y se realizará un análisis más en profundidad en el apartado 4 de esta memoria.

3.3 Análisis de textos académicos relacionados

Otra de las tareas que debemos realizar antes de comenzar nuestro proyecto es la lectura de artículos académicos (los llamados *papers*) relacionados con el tema que vamos a tratar. En concreto, nuestro problema ya había sido resuelto de antemano de diferentes maneras. Por ello vamos a hacer un breve repaso por dos artículos académicos que nos resultaron de especial utilidad para el proyecto.

3.3.1 Artículo 1: Text line segmentation using a fully convolutional network in handwritten document images – Quang Nhat vo, Soo Hyung Kim *et al*. Referencia [2]

Este artículo fue publicado en 2017 por unos investigadores de la Universidad Nacional de Chonman en Corea del Sur. El objetivo principal del artículo es dar una solución que involucre redes neuronales convolucionales y que pudiera haber competido en el concurso del ICDAR 2013 del que hemos sacado el dataset. Para ello,

dividen su solución en dos fases:

1) La obtención de la espina dorsal de las líneas del texto.
2) La separación de las letras que por error de escritura están en contacto, pero pertenecen a líneas distintas.

Una vez se tienen estas dos piezas, la segmentación de las líneas de texto es una tarea trivial. Vamos a explicar cómo resuelven cada una de estas tareas:

1. **Obtención de la espina dorsal de las líneas del texto**

Primero de todo, para aclarar el término "espina dorsal de las líneas del texto" veamos la siguiente figura:

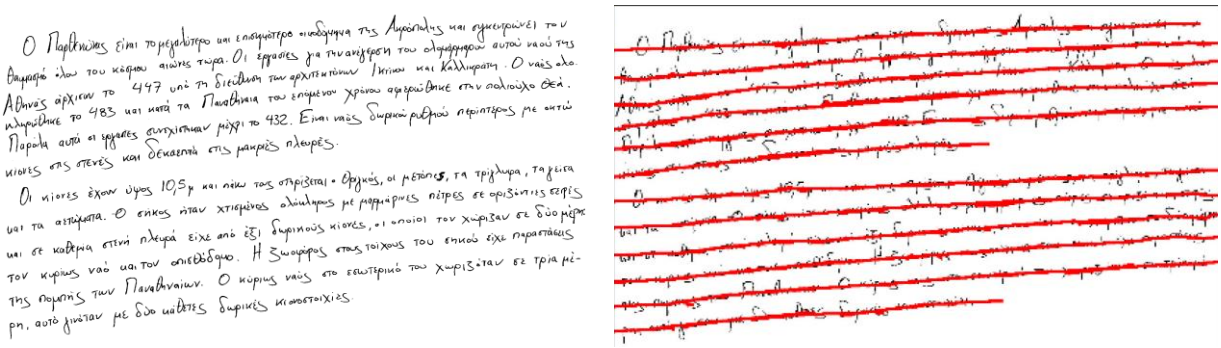


Figura 3-14. Imagen del ICDAR 2013 dataset junto con sus "espinas dorsales"

Ahora que entendemos a lo que se refieren, veamos el proceso de extracción que ellos siguen. El objetivo de este resumen es el de aprender la "intuición"[1] que hay detrás de este proceso sin entrar en detalle con los detalles técnicos.

A) El primer bloque que encuentra la imagen es una red completamente convolucional (FCN) con 2 salidas diferentes. Estas dos salidas las denominan "Line map" y "Background map", y se tratan de dos mapas que tratan de predecir la posibilidad de que un determinado píxel del espacio pertenezca al texto o que pertenezca al fondo.

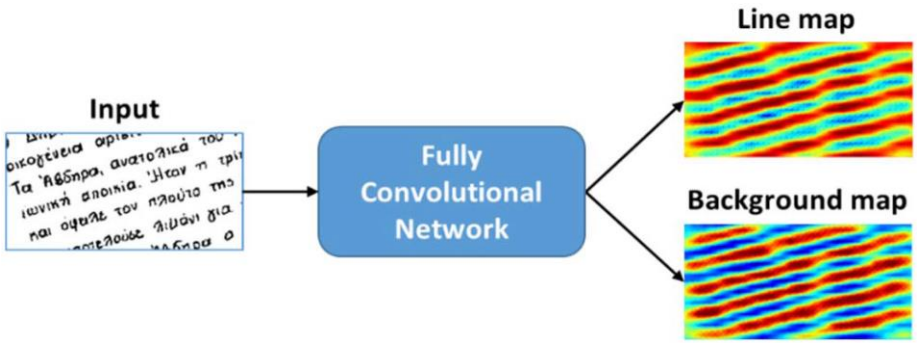


Figura 3-15. Entrada y salida de la FCN del articulo 1

B) El segundo paso consiste en deslizar estos dos mapas a lo largo de la imagen e integrar usando una función que muestran en su artículo. Esta operación matemática nos da como resultado un nuevo mapa. Las líneas del texto estarán localizadas en las zonas de mayor energía de este mapa.

C) Lo siguiente que hacen es utilizar un "*thresholding algorithm*" (algoritmo con umbral) que permite binarizar el mapa de energía. Esto nos da como resultado una nueva imagen donde se puede ver la dirección general principal del texto.

[1] El término intuición es ampliamente usado en inglés (*gain insights*) por Ian Goodfellow en su libro "Deep Learning" (referencia []) y por Carlos Santana Vega en los vídeos que sube en su canal de youtube.

D) Por último, se debe de sacar el esqueleto o espina dorsal de cada línea de texto. En esto hay un pequeño problema a la hora de discernir entre aquellos mapas de energía binarizados que tienen dos líneas diferentes conectadas entre sí por un error de escritura. Por ello se ejecuta un algoritmo que viene definido por la ecuación (4) del artículo y que permite, tras varias iteraciones, extraer y fusionar las espinas dorsales.

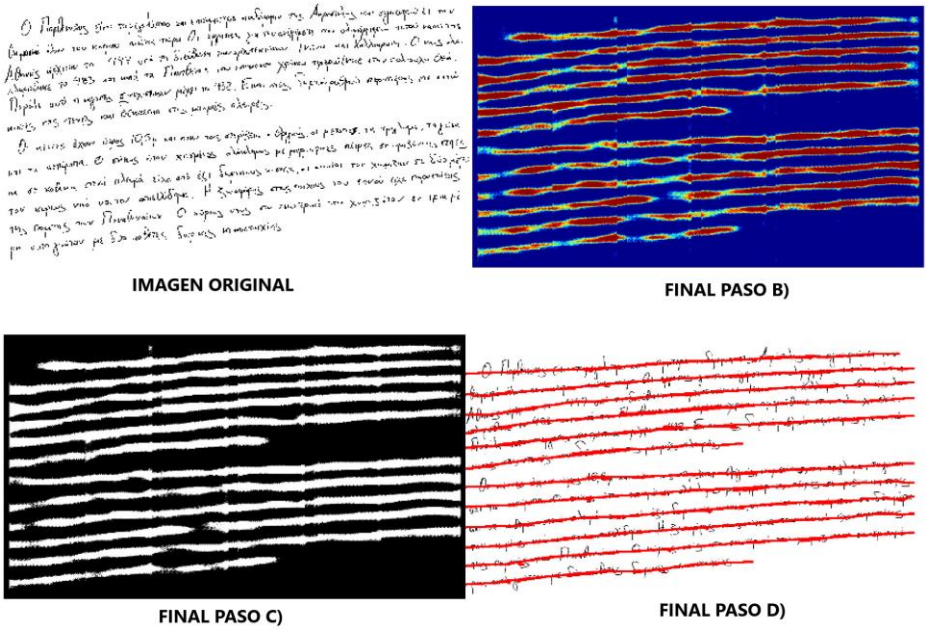


Figura 3-16. Representación de los resultados después de cada paso.

En cuanto a la arquitectura utilizada para el paso A), barajaron la posibilidad de usar la familia de redes FCN-pool. Esta familia tiene 3 miembros: la FCN-pool2, la FCN-pool3 y la FCN-pool4. Todas estas utilizan filtros de 3x3 para aprender los "aspectos tempranos" (*early features*). La diferencia entre las 3 es que estos aspectos tempranos son compuestos por capas de pooling diferentes (la segunda capa de pooling en la FCN-pool2, la tercera en FCN-pool3 y la cuarta en la FCN-pool4).

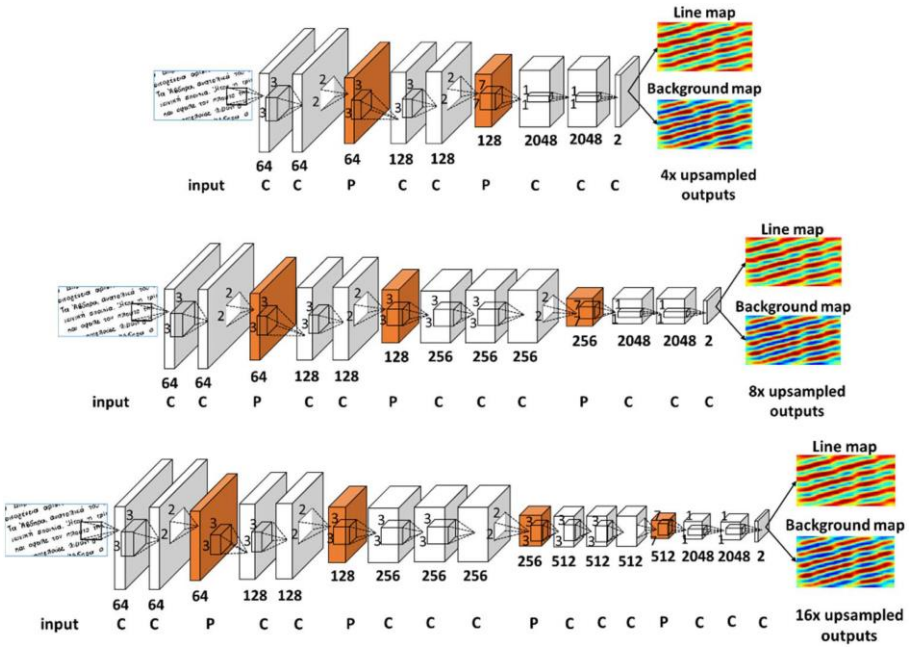


Figura 3-17. De arriba abajo: FCN-pool2, FCN-pool3 y FCN-pool4

A la luz de los resultados, en la versión final del proyecto utilizaron la FCN-pool3.

2. **Separación de caracteres conflictivos**

Llamamos **caracteres conflictivos** a aquellos que por un error durante la escritura del documento se tocan entre sí pero que pertenecen a dos líneas diferentes. Una vez que se tiene la espina dorsal de las dos líneas que están en conflicto la tarea se hace un poco más sencilla. Así, utilizan la línea que surge en el medio de las dos espinas dorsales para ayudar en su algoritmo de separación.

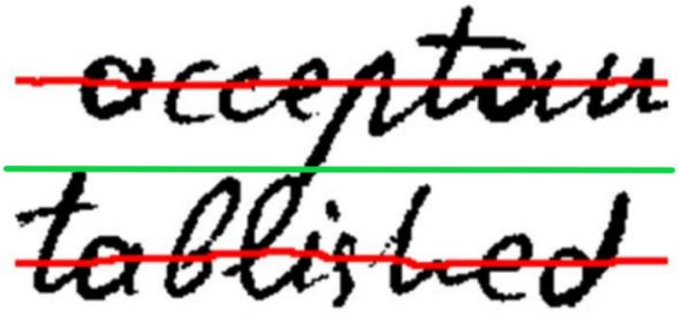


Figura 3-18. Ejemplo de caracteres conflictivos.

El algoritmo de separación que emplean se apoya en la teoría de grafos y en la tecnología *LAG – Line adjacency graphs* (grafos de líneas adyacentes). Según palabras del propio artículo, para crear un LAG a partir de un carácter de la imagen se sigue la conocida codificación *RLE – Run-Length Encoding*. El concepto se visualiza de una manera más clara cuando se ve el siguiente ejemplo de creación del grafo para una letra "e".

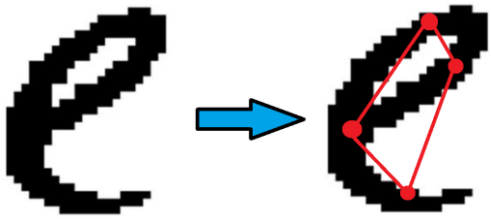


Figura 3-19. LAG de tiradas verticales (grafo rojo)

El proceso de creación no es tan relevante desde nuestro punto de vista. Sin embargo, cabe mencionar que hay 2 tipos de LAG y es que a la hora de crearlos se pueden realizar tiradas verticales o tiradas horizontales y el resultado variará de uno a otro. En el artículo se utilizan ambos sin especificar el criterio, pero enfatizan que a veces se debe usar uno y no el otro. Cuando surge el problema de los caracteres conflictivos, este es transferido al problema de la partición de grafos, donde se mide la mínima distancia euclídea desde cada nodo del grafo hasta la espina dorsal más cercana. El proceso es más complicado de entender, pero con la siguiente figura podemos hacernos una idea.

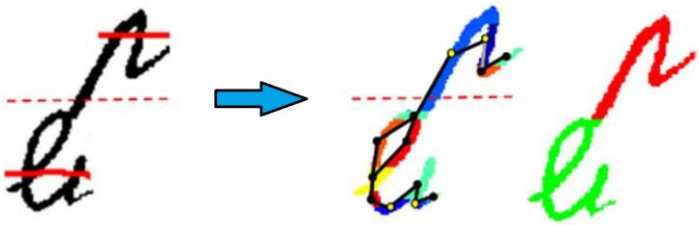


Figura 3-20. Proceso de separación de caracteres conflictivos

Una vez que tenemos las dos piezas, realizar la segmentación es muy sencilla. El objetivo era dar un valor numérico distinto a los píxeles de cada línea. Para ello, se les da un mismo valor a todo el conjunto de píxeles

negros que tienen algún camino de píxeles negros hasta una espina dorsal. Sabemos que un pixel negro nunca podrá llegar a dos espinas dorsales distintas porque habremos cortado esa ruta gracias a la tecnología LAG.

Los autores del artículo utilizaron el sistema de medición de la precisión de la red que se usó en el concurso de 2013 y descubrieron que el sistema que utilizaron tenía casi el mismo rendimiento que el ganador del concurso.

3.3.2 Artículo 2: U-Net: Convolutional Networks for Biomedical Image Segmentation – Olaf Ronneberger, Philipp Fischer, and Thomas Brox. Referencia [26]

Este artículo fue publicado en 2015 por la Universidad de Freiburg, en Alemania. Las redes de tipo U-Net son ampliamente utilizadas en el campo de la visión artificial. De hecho, hay multitud de artículos académicos que tratan este tipo de red. El motivo por el que hemos escogido este en especial es porque un usuario de GitHub consiguió recrearlo usando las librerías de Keras y TensorFlow. El código de este usuario está en la referencia [27].

En el artículo hablan sobre el auge de las redes neuronales convolucionales gracias a la aceleración por GPU. Hablan del ganador de la competición de segmentación de imágenes de microscopio electrónico del ISBI 2012. Dicen que la red del ganador se fijaba en un punto de la imagen e inspeccionaba un "círculo" de pixeles alrededor, lo cual es ineficiente y lento en términos de los autores. Es por ello que deciden recrear una nueva red con la tecnología U-Net para compararla con la ganadora.

Lo primero que debemos comentar es acerca del dataset. El dataset de la competición del ISBI 2012 se compone de una serie de imágenes tomadas a neuronas bajo microscopio electrónico. El objetivo de la competición es automatizar la binarización de esas imágenes que están llenas de ruido para que puedan ser estudiadas con mayor facilidad. Aquí podemos ver un ejemplo:

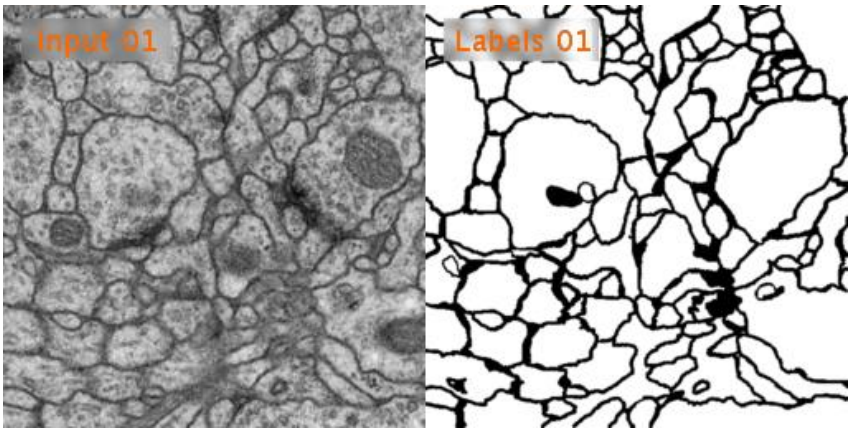


Figura 3-21. Ejemplo del dataset de la competición del ISBI 2012 [28]

El dataset se puede encontrar en el enlace que se encuentra en la referencia [28]. Como podemos observar, el objetivo del concurso guarda cierto parecido con la tarea que queremos realizar. Por ello vamos a analizar la parte que más nos interesa del artículo: la red y cómo se implementa. Veamos un esquema de la red:

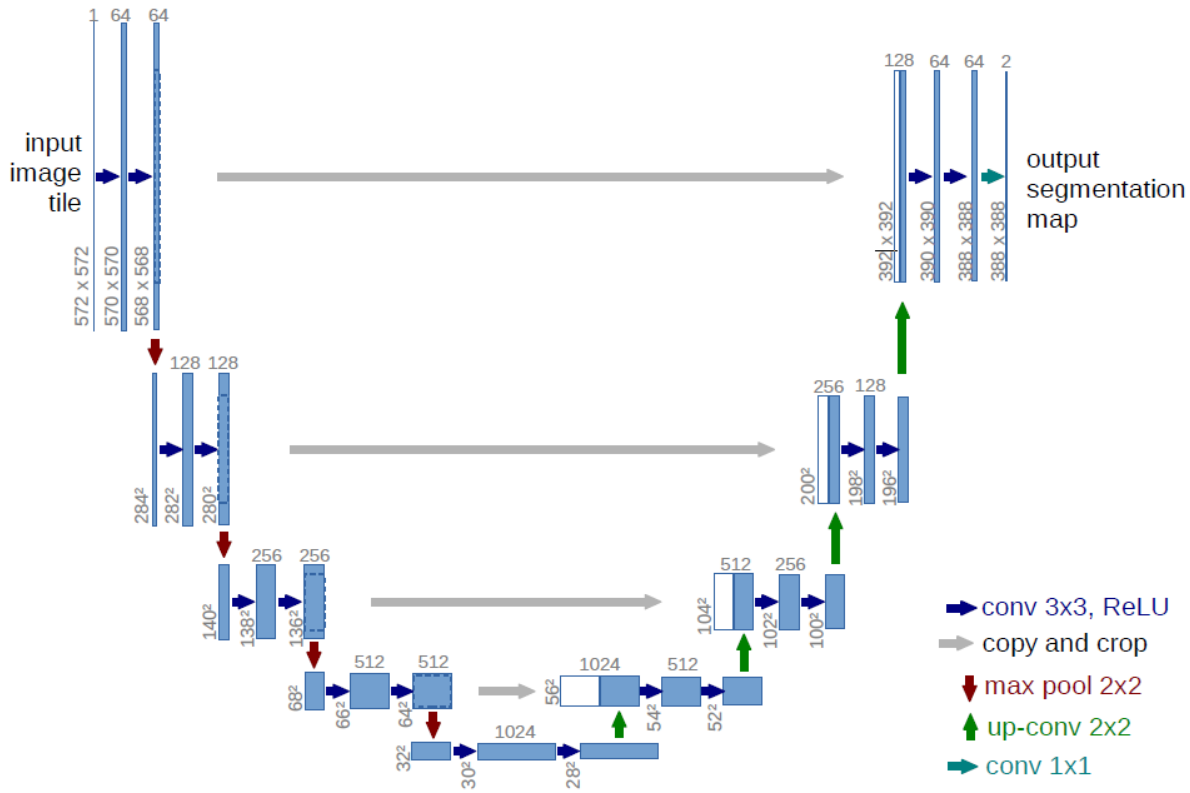


Figura 3-22. Arquitectura tipo U-Net utilizada por el artículo 2.

Las características principales de las redes U-Net son:

1) Los aspectos de la imagen se van sacando gracias a filtros convolucionales, en este caso de tamaño 3x3.
2) La dimensionalidad de las imágenes se va reduciendo después de cada sesión de filtrado gracias a muestreos por parte de unas capas de Pooling. Esto ocurre hasta que llega un punto en el que la imagen vuelve a subir de dimensionalidad para finalmente terminar llegando a la resolución objetivo (que en muchos casos es similar a la original).
3) Para esta subida en la resolución se utiliza información de las capas anteriores. Es esto lo que le da a esta red el característico aspecto de "U".

Este tipo de redes se utiliza con mucha frecuencia cuando la entrada es una imagen y también lo es la salida. Adelantamos que la solución que vamos a aportar para nuestro proyecto involucrará una red de tipo U-Net y es por ello que hemos decidido hablar acerca de este artículo. El resto del mismo no tiene mayor relevancia para nuestro proyecto.

4 ESTUDIO Y MANIPULACIÓN DEL DATASET

Como ya adelantamos en el capítulo anterior, la base de datos que hemos utilizado es la que se utilizó para la competición de segmentación de texto manuscrito en el ICDAR de 2013 [23]. Recordamos que está compuesta por un total de 250 imágenes en griego e inglés a partes iguales y 100 más en idioma bengalí, además de un ground truth donde se tiene una segmentación de palabras por palabras y de líneas por líneas.

A lo largo de este capítulo realizaremos una serie de modificaciones al ground truth de este dataset para ir acercándonos a la solución final. Algunas de las versiones tratadas están en el repositorio del proyecto en GitHub (ver Anexo). Como ya adelantamos en el punto 1.3, la idea final es la de conseguir que, de la imagen de un manuscrito, asignemos el valor "1" a los píxeles que pertenecen a la primera línea del texto, el valor "2" a los que pertenecen a la línea 2, etcétera. Así, si representamos una imagen utilizando un mapa de color en el que el "0" sea blanco y el resto de números se correspondan con un color "aleatorio" justo al lado de la original, tendríamos la Figura 3-13 que la hemos traído aquí para una mejor aclaración:

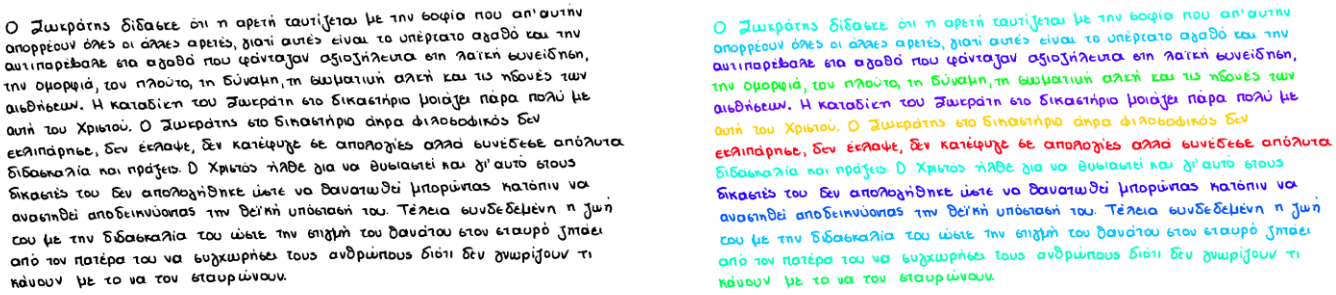


Figura 4-1. Representación de la imagen 001 y su ground-truth.

La tarea de realizar esta segmentación es demasiado compleja como para que una red neuronal aprenda a hacerla sin pasos intermedios. Por ello, decidimos hacer lo siguiente:

El paso intermedio sería el de obtener una especie de *máscara* donde los gruesos de las líneas estén diferenciados. Así, en la máscara (Figura 4-2 a la izquierda) aparecerá una mancha por cada línea, que se asemeja al contorno de las letras que la forman. Obtendríamos así la silueta o perfil de cada renglón, diferenciando uno de otro y asignándole el valor que le corresponde ("1" para la línea 1, "2" para la línea 2...). De esta manera, podremos poner la máscara encima de un texto (Figura 4-2 a la derecha), para que, comparando uno a uno los píxeles, podamos obtener una segmentación como la que deseamos.

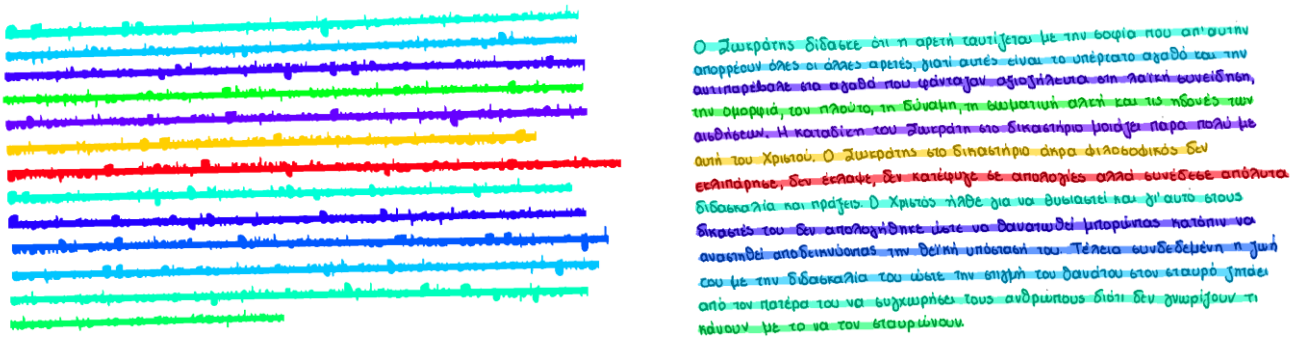


Figura 4-2. Máscara 001 y máscara encima del texto.

Sin embargo, el método presenta un pequeño problema. En la mayoría de los casos, los textos manuscritos presentan los denominados *caracteres conflictivos*, que son letras que se han escrito en una línea, pero están tocando a otras que pertenecen a una distinta. En ese caso, en la máscara se pueden apreciar líneas que se tocan entre sí, lo cuál hace casi imposible una segmentación adecuada. Lo peor es que el problema de los *caracteres conflictivos* es demasiado frecuente como para obviarlo, y por lo tanto debíamos averiguar una manera de, al menos, reducir el impacto que produce.

Fue por ello que decidimos entrenar 3 modelos de redes neuronales distintos, uno para que aprendiese a predecir las mitades superiores de cada línea, otro para las mitades inferiores, y otro para que predijera los esqueletos de cada línea.

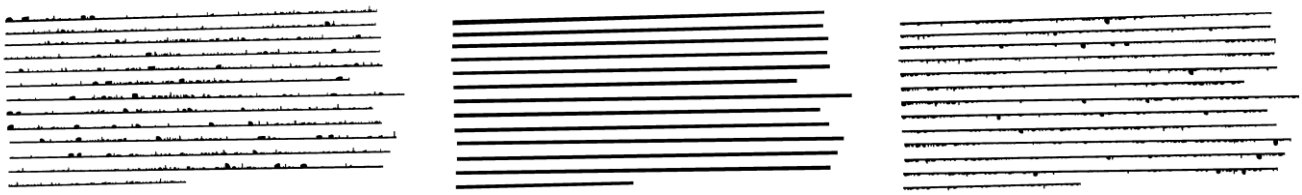


Figura 4-3. Partes superiores, esqueletos y partes inferiores de las líneas

Así, una vez tuviésemos las 3 piezas del rompecabezas, la *máscara* podría ser conseguida haciendo una relación unívoca entre cada trío superior-media-inferior predicho por las redes.

Una vez marcado nuestro objetivo, nos encontramos con un choque de realidad: el dataset solo nos da un ground truth como el que vemos en la Figura 4-1, el cuál no nos sirve para alimentar a las 3 redes descritas. Por ello, durante este capítulo haremos modificaciones al ground truth que nos dan hasta conseguir 4 versiones, y la manera en que la conseguimos. También hablaremos de la división en grupos para los modelos y de un concepto muy extendido en el campo de la Inteligencia Artificial: el *Data Augmentation*.

4.1 Modificaciones realizadas al ground truth

4.1.1 Espina dorsal en forma de recta de regresión (*Linear Regression Backbone*)

La idea detrás de esta primera modificación es la de sustituir cada línea del texto por una única recta de regresión por mínimos cuadrados que se ajuste de la mejor manera posible a la posición que ocupaban los píxeles de la línea sustituida.

Para realizar esta modificación (y todas las demás) vamos a diseñar una función en python que obtenga como parámetro una matriz y la procese.

Para llegar a esta solución, vamos a pasar por una intermedia. A esta la llamaremos “espina dorsal en términos de pixeles verticales” (*Pixel-wise Backbone*). Consistirá en extraer una por una cada columna de píxeles de la imagen y extraer la posición media de cada línea en esa columna. Así, de la imagen de ejemplo saldrá la siguiente “*Pixel-wise Backbone*”:

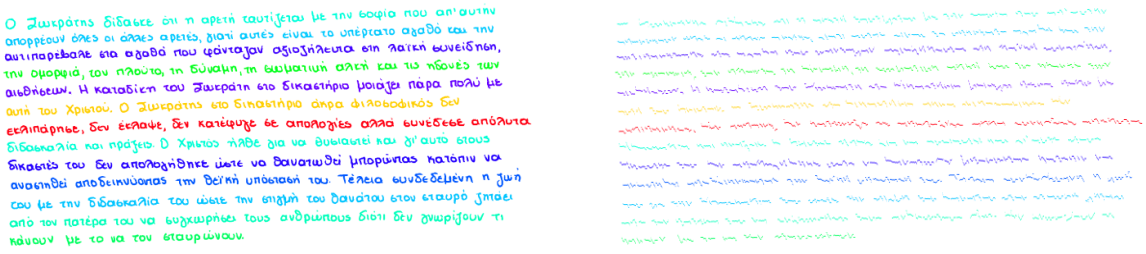


Figura 4-4. A la derecha, el Pixel-wise Backbone procedente de la imagen 001, que está a la izquierda.

Esta nueva imagen será la que se utilice para crear la ya mencionada espina dorsal en forma de recta de regresión. Para su creación, lo que se hará será utilizar la función polyfit de Numpy con el tercer argumento (grado del polinomio) igualado a 1. Esto devuelve la pendiente y la ordenada en el origen de la recta de mejor ajuste a la nube de puntos dada. Una vez que tengamos montado el polinomio que es la recta, solo nos quedará evaluarlo desde el principio de la línea hasta el final para alcanzar nuestra espina dorsal en forma de línea.

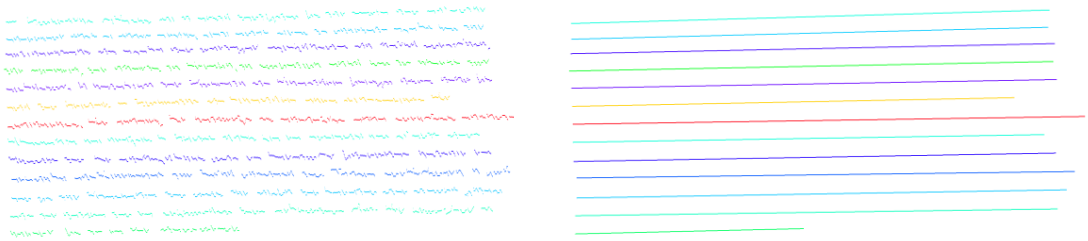


Figura 4-5. Izquierda: "Pixel-wise Backbone" de 001. Derecha: "Linear Regression Backbone" de 001

Así es como tenemos terminada nuestra primera modificación en el dataset. La función que hemos creado (*createLinearRegressionBackbone*) nos servirá de gran ayuda para las demás modificaciones.

4.1.2 Espina dorsal gruesa (*Thick Backbone*)

Para una red neuronal es más difícil aprender a sacar una línea delgada que una línea gruesa. Esto conformaría lo que al principio del capítulo hemos descrito como el esqueleto o "partes medias" de las líneas, y sería al final el ground truth con el que alimentaríamos a uno de nuestros tres modelos objetivo.

Para crear esta modificación, volveremos a inspeccionar la imagen por columnas. En ellas, cogeremos el pixel más alto y el más bajo de cada línea. A continuación, pasaremos estos dos conjuntos de píxeles por la función *createLinearRegressionBackbone* que creamos para la anterior modificación. Esto nos dará como resultado dos bandas que estarán aproximadamente a la altura del máximo y del mínimo de cada línea. Por último, rellenamos todos los pixeles que hay entre estas dos rectas, creando así un trapecio que se asemeja bastante a lo que queremos conseguir.

Si pasamos ahora el resultado a tipo Boolean, estaríamos creando lo mismo que lo que se habla en la primera parte del articulo 1 (sección 3.3.1, [2]), pero habiéndonos saltado la parte del mapa de calor, la binarización y todo lo demás.

De la primera imagen sale el siguiente resultado:

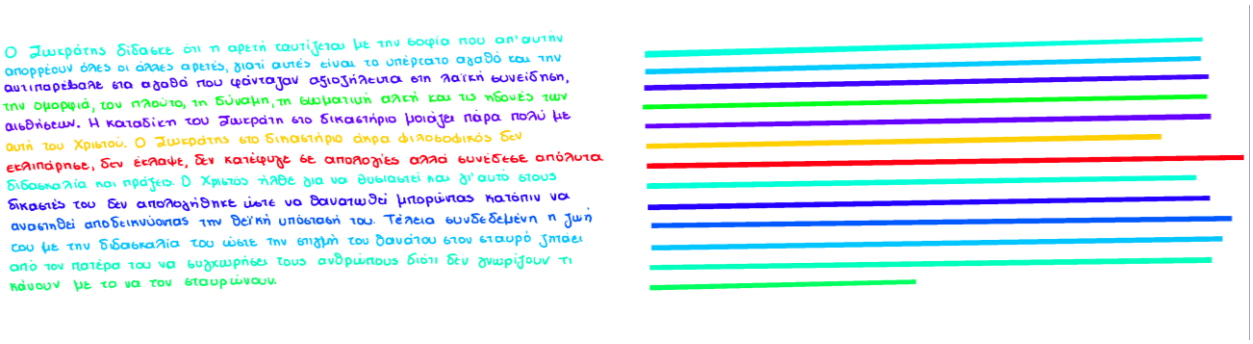


Figura 4-6. A la derecha, la espina dorsal gruesa de la imagen 001, que está a la izquierda

4.1.3 Contorno de cada línea (*Outline Backbone*)

En esta nueva versión del ground truth, queríamos realizar precisamente esa máscara de la que hablamos al principio del capítulo. Dicha máscara no tendría por qué ser demasiado precisa, y lo importante sería que una red neuronal pudiera ser entrenada para que dada una imagen del dataset, fuese capaz de averiguar el contorno de sus líneas.

Partimos de la anterior modificación. En ella, estamos dejando atrás muchísimos píxeles de cada línea que no estarían dentro del trapecio mencionado. Por eso, volvemos a sacar esas dos rectas de regresión de los píxeles más bajos y más altos de cada línea y empezaremos a rellenar desde la posición más alta que haya entre la recta de regresión de los máximos y el pixel más alto de la columna y lo haremos hasta la posición más baja que haya entre la recta de regresión de los mínimos y el píxel más bajo de la columna. Así, de la primera imagen nos saldría el siguiente contorno:

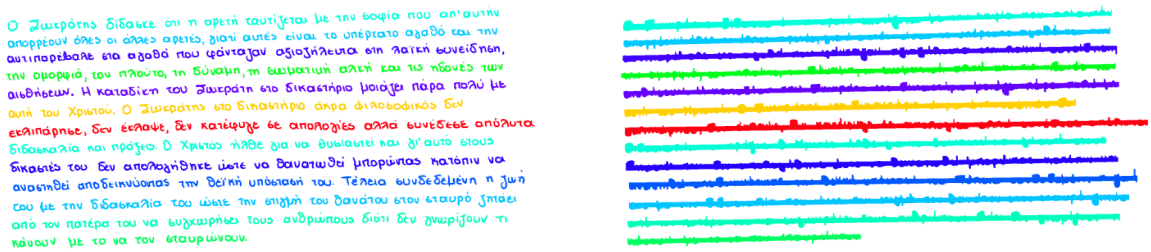


Figura 4-7. A la derecha, el contorno de cada línea procedente de la imagen 001, que está a la izquierda.

Si pasamos esta nueva imagen a tipo Boolean, estaríamos colapsando todos los contornos para que los pudiésemos distinguir en contraposición al fondo, pero sin diferenciar el contorno de una línea con la de otra. En una imagen con las líneas suficientemente separadas, si tuviésemos el contorno de cada línea de esta imagen, ya seríamos capaz de realizar la segmentación, puesto que a los píxeles negros de la imagen que estén debajo de un mismo contorno se les podría asignar un mismo valor numérico y por lo tanto estaríamos distinguiendo una línea de otra.

Sin embargo, como ya dijimos al principio del capítulo, una red que aprendiese a sacar el contorno de las líneas de un texto tendría el problema de tener que diferenciar aquellos *caracteres conflictivos,* que son aquellos que, por descuido al escribir a mano, pertenecen a una línea, pero tocan a otra. Fue por eso que este ground truth no fue el que utilizamos para realizar el proyecto, pero merecía la pena contarlo aquí porque sentó las bases de la siguiente modificación.

4.1.4 Contorno de cada línea en forma de zigzag (*ZigZag Backbone*)

El principal problema que se encuentra en el párrafo anterior es el mismo con el que se encontraron los investigadores coreanos del artículo 1 (sección 3.3.1, [2]): hay ocasiones en las que, por descuido a la hora de escribir, un carácter de una línea toca a un carácter de otra. En ese caso, la solución anterior ya no funciona y hay que averiguar un método para separar esos casos. Nosotros no hemos querido recrear la tecnología LAG de la que hablaban en el artículo debido a su dificultad.

Como hemos ido introduciendo, la solución que hemos adoptado tiene un enfoque diferente. En nuestro caso, decidimos reinventar la modificación anterior para que, en lugar de tener el contorno de unas líneas de texto, tuviésemos por un lado el contorno de las mitades superiores de las líneas de texto y por otro lado el contorno de las mitades inferiores de las líneas de texto. De esta forma, una vez tuviésemos esta información, podríamos aplicar esta máscara encima de una imagen normal y determinar que los píxeles negros que pertenecieran a una misma línea serían aquellos que estuviesen bajo un par mitad superior – mitad inferior de contorno de línea. Así, sabríamos que cuando topásemos con la siguiente mitad superior sería momento de parar de segmentar y habríamos impedido que aquellos caracteres que están en contacto, pero no pertenecen a una misma línea acabasen siendo mal segmentados.

A la hora de crear esta modificación, reutilizamos todas las funciones que habíamos creado anteriormente. Así, lo que hacemos es crear por un lado el contorno de las líneas de una imagen y también la espina dorsal en forma de línea de esa imagen con las funciones que ya habíamos programado. A continuación, podemos hacer que, dado un número de contorno de línea, se sustituyan los valores de ese contorno que están por debajo de la espina dorsal en forma de línea. Para mayor comodidad, decidimos hacer que para un número de línea "n", la mitad superior de su contorno estuviese marcado con el valor "n" y la mitad inferior con el valor "-n".

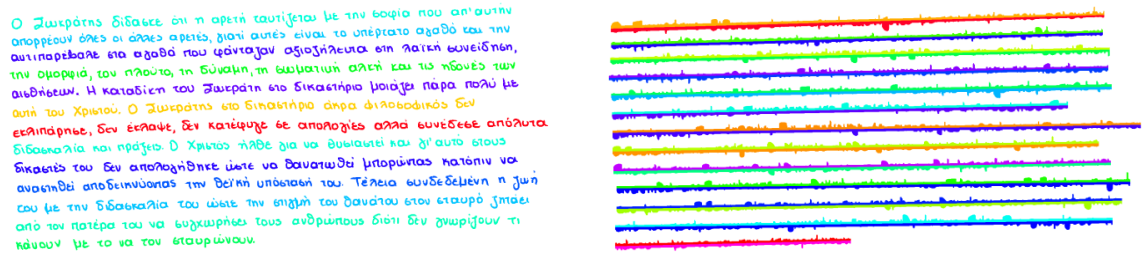


Figura 4-8. A la derecha, el ZigZag Backbone de la imagen 001, que es la de la izquierda

Por supuesto, sería una locura para una red neuronal aprender a hilar tan fino. Por eso, hacer una simplificación de este ground truth es tan sencillo como colapsar todos los valores positivos (mitades superiores de contornos) en el valor +1 y todos los valores negativos (mitades inferiores de contornos) en el valor -1. Así, el resultado sería el siguiente:

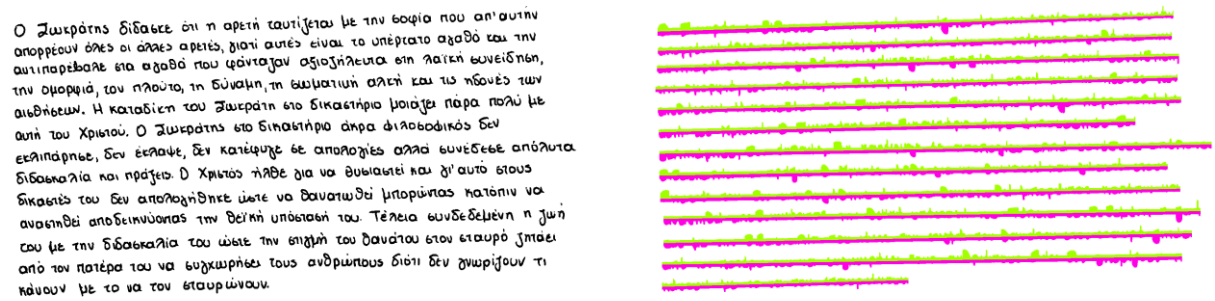


Figura 4-9. A la derecha, el ZigZag Backbone simplificado de la imagen 001, que es la de la izquierda

Nótese que esto, junto con la espina dorsal del punto 4.1.2, es lo que se intentará que aprenda a sacar la red neuronal a partir de la imagen del texto.

4.2 División del dataset en los conjuntos de *entrenamiento, test* y *validación*

Una vez realizada las pertinentes modificaciones a las 350 imágenes que tenía el dataset lo lógico hubiese sido dividirlo para utilizarlo en la red neuronal que íbamos a programar. Las recomendaciones que podemos encontrar en libros como por ejemplo el de A. Géron [1], o el de I. Goodfellow [29] nos dicen que el 20% del dataset debe de formar parte del conjunto de test y que el restante debe ir destinado para el entrenamiento. Además, para tener una medición no sesgada de las métricas que elegimos durante el entrenamiento, parte de ese mismo conjunto debería de formar parte de un nuevo conjunto llamado conjunto de validación. Por ello, la división más lógica hubiera quedado así

De las 350 imágenes:

- 210 para el conjunto de entrenamiento.
- 70 para el conjunto de validación.

- 70 para el conjunto de test.

Además, una de las cosas en las que más hacen hincapié los textos arriba mencionados es que estos 3 conjuntos deben seguir la misma distribución. La mayor diferencia que hay entre las imágenes son los idiomas en los que están escritos, habiendo muchísima más diferencia entre las imágenes en bengalí y las otras de idiomas occidentales. Es por ello que, como 100 de las 350 imágenes estaban escritas en bengalí, habría que asegurarse de que esa misma proporción se cumpla en los tres conjuntos.

Hemos dejado constancia de cómo hubiese sido una división sensata de las imágenes que disponíamos para que sirva de ejemplo para aquel que lo pueda necesitar. Sin embargo, para nuestro proyecto decidimos utilizar otro criterio. Resulta que, en el concurso de 2013, de donde salió originalmente el dataset, se especificó a los concursantes una serie de pautas a seguir para presentar sus proyectos. Entre otras, se encontraba la restricción de destinar como conjunto de entrenamiento las 200 primeras imágenes del dataset, guardando las restantes (150) para el conjunto de test. Quisimos ser fieles a las condiciones del concurso para poder comparar el rendimiento de nuestra red con el del resto de participantes, y, por lo tanto, teníamos que hacerlo bajo las mismas condiciones que habían tenido en 2013. Entonces, de las 350 imágenes:

- 160 para el conjunto de entrenamiento.
- 40 para el conjunto de validación.
- 150 para el conjunto de test.

4.3 *Data augmentation*

Precisamente ahora que habíamos decidido que íbamos a seguir las bases del concurso de 2013, nos dimos cuenta de que tener solamente 160 imágenes para entrenar podría reducir significativamente el rendimiento de nuestra red.

En el campo del Machine Learning, se conoce bajo el nombre de "*data augmentation*" a una serie de operaciones que nos permiten multiplicar las muestras con las que alimentamos a nuestra red. En el caso de imágenes, como las que tenemos en nuestro caso, podemos aumentar el número de las mismas utilizando numerosas técnicas propias del procesamiento de imágenes, tales como la rotación, la inversión de la imagen o algo tan sencillo como darle la vuelta. Sin embargo, no todo vale como un data augmentation válido. Como bien dicen en la referencia [30], si nuestra red estuviera aprendiendo a diferenciar instancias de gatos en una imagen, resultaría perfecto alimentar a la red con copias de las imágenes originales reflejadas sobre su eje vertical, ya que esto sería un escenario habitual si la cámara con la que se tomase una nueva imagen tuviese activado por defecto esta manera de tomar fotografías. Sin embargo, poco sentido tendría que aprendiese esas mismas imágenes reflejadas sobre el eje horizontal, ya que este escenario no estaría contemplado salvo error humano y podría llevar a un peor aprendizaje de la red.

Dicho esto, finalmente decidimos triplicar nuestro conjunto de entrenamiento, rotando cada imagen (y sus ground truth) 5º en sentido horario y 5º en sentido antihorario utilizando la función *rotate* de la librería PILLOW de Python.

5 ESCASEZ DE MEMORIA Y SOLUCIÓN

Una vez que teníamos nuestro dataset modificado nos pusimos manos a la obra en la implementación de una red neuronal que pudiese aprender a sacar, al menos, los contornos de las imágenes. Siempre tuvimos el objetivo final de que pudiéramos crear una *máscara* como la descrita en el capítulo anterior para que al compararla con el texto original pudiésemos sacar una segmentación del mismo.

Ya introdujimos las arquitecturas de tipo U-Net en el punto 3.3.2, y dado que históricamente han servido para hacer tareas parecidas a la nuestra (como la segmentación de imágenes biomédicas del Artículo 2 [26]), decidimos que era buena idea utilizarla en nuestro proyecto. Por eso, seguimos los pasos del usuario de GitHub que dejó la implementación de la red en Keras y TensorFlow [27] y adaptamos el código a nuestras necesidades.

Las redes neuronales de tipo U-Net no admiten que haya resoluciones diferentes entre las entradas. Eso suponía que debíamos hacer que todas las imágenes tuviesen la misma resolución. Si hubiésemos realizado un ajuste de cada imagen a una resolución determinada, estaríamos perdiendo completamente el *aspect ratio* de las mismas, por eso decidimos rellenar el fondo de las imágenes más pequeñas hasta llegar a los anchos y altos más grandes que se encontraban en el dataset. Tocaba entonces averiguar cuál era la resolución más grande que había en nuestro dataset, por lo que hicimos un script en python y nos salieron los siguientes resultados (decidimos analizar también el dato del número de líneas por simple curiosidad):

	MÍNIMO	MEDIO	MÁXIMO
ALTO (PX)	651	2088.19	4496
ANCHO (PX)	1427	2285.86	3227
NÚMERO DE LÍNEAS	6	18.22	36

Tabla 5-1. Estadísticos acerca del dataset

Entonces agrandamos el fondo de todas las imágenes hasta tener una resolución de 4496 x 3227 píxeles. Esta resolución tan extraña (la factorización en números primos de 3227 es 7 × 461) nos dio problemas a la hora de ser implementada en una red de tipo U-Net, ya que esta se encarga de rebajar la resolución (parte izquierda de la U-Net) en factores de 2 y luego volver a restaurarla (parte derecha de la U-Net), y por mucho que ajustásemos las opciones de *padding* del modelo, siempre daba un error.

Por eso, haciendo un análisis de la situación, caímos en la cuenta de que los valores máximos que habíamos obtenido eran los que presentaban las imágenes 224.tif y 327.tif. Por eso decidimos eliminarlas del dataset, con lo cual los valores máximos se vieron drásticamente mermados. El análisis estadístico nuevo nos dio estos resultados:

	MÍNIMO	MEDIO	MÁXIMO
ALTO (PX)	651	2082.52	3652
ANCHO (PX)	1427	2283.48	2534
NÚMERO DE LÍNEAS	6	18.22	36

Tabla 5-2. Estadísticos del dataset quitando la 224 y la 327

Una vez eliminadas estas dos imágenes del conjunto de entrenamiento, decidimos fijar como resolución de entrada a la 4096 x 3072. Escogimos esta por estar cerca de los valores máximos y por sus factorizaciones (4096 es 2^{12} y 3072 es $3*2^{10}$). Desde ese momento, no volvimos a tener problemas con las resoluciones en la U-Net.

5.1 El problema de la memoria

El método convencional de entrenamiento de redes neuronales en Keras consiste en crear un modelo y ejecutar su método *fit*. Con esto, se carga todo el conjunto de entrenamiento y de validación en la memoria RAM y empieza a entrenar. Además, el tipo de datos que se carga debe ser forzosamente de tipo *float* (no deja con ningún otro tipo), ya que hacen falta realizar operaciones lineales para ejecutar el algoritmo del descenso del gradiente y el redondeo si se utilizasen datos de tipo *int* no es una de ellas.

Como era de esperar, cargar 600 imágenes (las 200 * 3 del data augmentation) de 4096 x 3072 píxeles de resolución con un tipo de dato de *float32* más su ground truth (que pesaría lo mismo) ocuparía aproximadamente 28 GB en total. Quizás para un servidor potente fuese algo viable, pero para un ordenador convencional, que además tenía que correr el intérprete de python, el sistema operativo y tener espacio suficiente para realizar todas las operaciones involucradas en el entrenamiento, esta cantidad de memoria sobrepasaba con creces sus límites. Teníamos un problema de memoria.

Este problema es intrínseco al entrenamiento de redes neuronales, especialmente cuando hablamos de aquellas que tratan volúmenes importantes de imágenes. Por ello hemos querido dejar constancia de todas las soluciones que implementamos ya que pueden ser de utilidad en proyectos futuros.

5.1.1 Submuestreo de las imágenes (*Downsampling*)

Esta fue la primera solución que implementamos para solucionar la falta de memoria. Si nos fijamos en las imágenes que tenemos, en realidad podemos apreciar que para cometer nuestra tarea de crear esa especie de máscara que podamos utilizar para segmentar las líneas del texto no era necesario contar con una calidad tan grande como las que veíamos en las imágenes.

Por ello decidimos reducir esta calidad realizando un submuestreo de la imagen. Usamos el método block_reduce del paquete skimage.measure. Se muestrearon las imágenes 3 veces usando cuadrados de 2x2 cada vez. En cuanto al criterio que utilizamos para ver con qué pixel quedarnos del recuadro 2x2 cada vez, decidimos tras varias pruebas usar una combinación: quedarse con el máximo valor en el primer y tercer submuestreo y con el mínimo en el segundo (hacer un *Max-pooling,* luego un *Min-pooling* y por último otro *Max-pooling*). En las imágenes, 1 significaba "pixel negro" y 0 "fondo", por lo que si usábamos la función Máximo en los 3 submuestreos las letras quedaban como una especie de mancha y no se distinguían bien, mientras que si usábamos la función Mínimo en los 3 submuestreos las letras se hacían demasiado finas.

El resultado con la combinación elegida eran imágenes de 512 x 384 píxeles de resolución que tenían el siguiente aspecto:

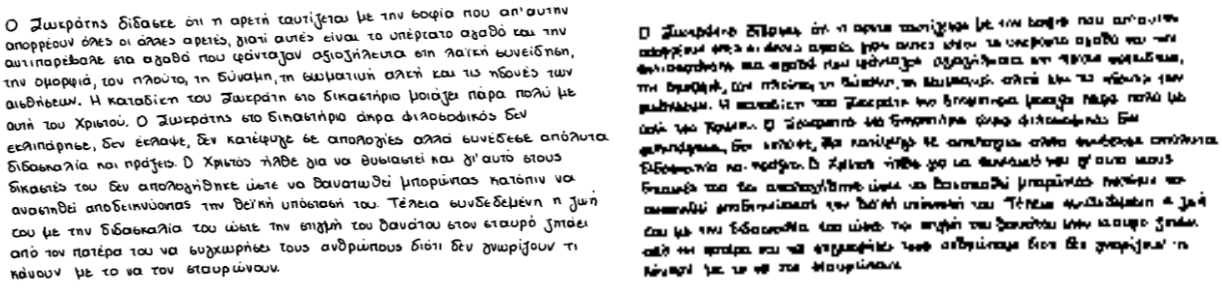


Figura 5-1. A la derecha, la versión submuestreada de la imagen 001, que es la de la izquierda

Con la versión submuestreada ya no se puede comprender bien el texto, pero sí que se puede entrenar a una red para que nos saque el contorno de las líneas de texto del mismo. Habíamos conseguido reducir el alto y el ancho de la imagen por un factor de 8 en cada dimensión.

Podríamos haber parado aquí, pero quisimos aprender nuevas formas para optimizar el gasto de memoria y por ello continuamos investigando.

5.1.2 Entrenamiento "*on-line*" o entrenamiento por *mini-batches*

Ya hemos hablado acerca del método convencional de entrenamiento, basado en cargar todo el dataset en memoria y entrenar a la red. En determinados problemas, como el nuestro, es especialmente pesado cargar todo un dataset de golpe en memoria. Es por ello que existe el llamado *entrenamiento on-line* o *entrenamiento por mini-batches,* que consiste en alimentar a la red neuronal con grupos de entradas pequeños (*mini-batches*) de manera que no se sature la memoria del sistema.

Para usar el entrenamiento *on-line* en Keras es necesario utilizar una clase llamada *DataGenerator* y en lugar de utilizar el método *fit* o *evaluate* convencional en el modelo a entrenar, deberemos usar los métodos *fit_generator* y el *evaluate_generator*, que tomarán como argumento a la clase anterior. Esta clase DataGenerator heredará de la clase keras.utils.Sequence y deberá implementar obligatoriamente 3 métodos (*__init__*, *__len__*, *__getitem__*). El método *getitem* será el que se llame en cada paso durante el entrenamiento, por lo que podemos programar lo que queramos aquí.

Es más, si queremos realizar algún ligero cambio en el dataset sin tener que procesarlo entero de nuevo, siempre podemos programar el cambio para que se realice a la hora de llamar al método *getitem*. Por ejemplo, en nuestro caso el submuestreo de las imágenes está implementado en este método y no es una modificación del propio dataset.

Para conocer más acerca de los *DataGenerator* recomendamos acudir a la referencia [31] o mirar el código del proyecto en el repositorio de GitHub (ver Anexo).

5.1.3 Aceleración por GPU

La explicación resumida de por qué las tarjetas gráficas son buenas para los proyectos de inteligencia artificial es que las mismas son capaces de realizar cálculos básicos en paralelo de una manera más óptima que los procesadores. Si tuviésemos que quedarnos con una idea, es que las CPUs son capaces de hacer cálculos muy potentes en serie mientras que las GPUs son expertas en realizar cálculos más básicos en paralelo. Se puede encontrar una explicación más detallado en el vídeo de la siguiente referencia [32].

Habilitar la aceleración por GPU es una tarea altamente tediosa ya que requiere instalar una serie de librerías cuyas versiones deben de ser compatibles entre ellas y con la versión de python, keras y tensorflow que estemos utilizando, además de con la versión de Anaconda que utilicemos como IDE. Por ello nuestra mejor recomendación es la de seguir las instrucciones del Dr. Jeff Heaton (profesor en la Universidad de Washington), quien tiene un vídeo explicando muy bien cómo instalarlo todo y que funcione de maravilla. El vídeo se puede encontrar en la referencia [33].

5.1.4 Google Colab

La última solución que implementamos fue también la más poderosa de todas. Google Colab o Google Colaboratory es un entorno gratuito en el que puedes programar un Jupyter Notebook y ejecutarlo en uno de los servidores de Google destinados a este proyecto. Así, si cambiamos el tipo de entorno de ejecución y activamos la aceleración por GPU, nos dedicarán una tarjeta gráfica para nuestro proyecto. Para hacernos una idea de la calidad del servicio que aportan los señores de Google, esta es la tarjeta gráfica que nos otorgaron todas las veces que ejecutamos nuestro programa:

```
!nvidia-smi

Fri May  1 10:41:42 2020
+-----------------------------------------------------------------------------+
| NVIDIA-SMI 440.64.00    Driver Version: 418.67       CUDA Version: 10.1     |
|-------------------------------+----------------------+----------------------+
| GPU  Name        Persistence-M| Bus-Id        Disp.A | Volatile Uncorr. ECC |
| Fan  Temp  Perf  Pwr:Usage/Cap|         Memory-Usage | GPU-Util  Compute M. |
|===============================+======================+======================|
|   0  Tesla P100-PCIE...  Off  | 00000000:00:04.0 Off |                    0 |
| N/A   37C    P0    25W / 250W |      0MiB / 16280MiB |      0%      Default |
+-------------------------------+----------------------+----------------------+

+-----------------------------------------------------------------------------+
| Processes:                                                       GPU Memory |
|  GPU       PID   Type   Process name                             Usage      |
|=============================================================================|
|  No running processes found                                                 |
+-----------------------------------------------------------------------------+
```

Figura 5-2. Especificaciones de la tarjeta gráfica otorgada en Google Colab

Esta gráfica a día de la escritura de esta documentación cuesta aproximadamente 2400€ si quisiéramos comprarla en Amazon a día de escribir este texto. Esto nos da una idea de la calidad de los recursos que ponen a nuestra disposición los cuales, en la mayoría de los casos, estarían fuera de nuestro alcance de no ser por esta herramienta.

Además, por si no fuese suficiente con los recursos de RAM y GPU que nos están dando, en el momento en el que agotamos la memoria del entorno otorgado, nos aparece un cartelito avisándonos de que hemos acabado con todos los recursos disponibles y nos pregunta si queremos que nos pasen a un entorno de alto rendimiento. Nótese que en ocasiones se puede forzar el agotamiento de la memoria con un simple bucle while infinito en el que se vayan añadiendo términos a una lista indefinidamente para disfrutar de una mayor calidad del entorno.

Para que se puedan acceder a los archivos almacenados en drive, hay que saber que la ruta absoluta de la raíz de nuestro Google Drive es "/content/gdrive/My Drive". Recomendamos añadir la ruta donde estén nuestros ficheros para ser importados a la variable "PATH" del entorno de ejecución y así podamos utilizarlo con la mayor comodidad. Montar la unidad de Google Drive y añadir al path se puede hacer con estas líneas:

```
from google.colab import drive
drive.mount('/content/gdrive')
import sys
sys.path.append('/content/gdrive/My Drive/Colab Notebooks')
```

Figura 5-3. Código para usar recursos de disco de Google Drive.

Por supuesto, sustituyendo la ruta por donde tengamos nuestros ficheros.

5.2 Creación de los tensores y alimentación de la red

Hasta ahora, hemos llegado hasta este punto de la memoria sin hablar del formato de las imágenes con el cuál debemos alimentar a las redes neuronales, y por ello consideramos este punto un buen momento para hacerlo.

Un tensor es una matriz de más de 2 dimensiones. En el campo del Machine Learning, las imágenes siempre deben de estar en forma de tensor, siendo las tres dimensiones de tamaños: (alto, ancho, num_canales), donde el número de canales es lo que define añade esta dimensión adicional y con él se refiere a que habrá tantas matrices de 2 dimensiones como canales de color (3 en caso de formato RGB y 1 en caso monocolor).

Como ya hemos dicho en el anterior punto, decidimos crear unos tensores con un *shape* = (4092,3076,1). Aquí hemos de hacer una pequeña pausa y recordar que lo que queremos que haga la red es aprender a extraer:

- Por un lado, los esqueletos o espinas dorsales de las líneas, que podemos ver en el punto 4.1.2.
- Por otro lado, las partes superiores e inferiores de las líneas de texto (lo que llamamos "ZigZag

Backbone" y podemos encontrar en el punto 4.1.4).

Como era tedioso rellenar las imágenes cada vez que la íbamos a usar, decidimos crear los tensores con los que alimentamos a la red de una sola vez. Así, nos ayudamos de "thickBacboneTensorsGenerator.py" y "zigZagTensorsGenerator.py", que crearon estos tensores en variables de tipo ".npy", las cuales pueden ser cargadas únicamente usando la función "load" de Numpy.

Ahora que tenemos guardadas en unas variables las variables con las que entrenamos a la red, el funcionamiento es el siguiente:

1. El DataGenerator (punto 5.1.2) coge una mini-batch de tensores.
2. Realiza el submuestreo (punto 5.1.1) para bajar la resolución.
3. Rota las imágenes para realizar el data augmentation (punto 4.3).
4. Alimenta a la red.

Todo esto usando Google Colab (punto 5.1.4) y aceleración por GPU (punto 5.1.3).

Vamos a realizar un poco de *spoilers* aquí, diciendo que, tras las muchas pruebas de las que se hablan en el punto 6, descubrimos que para el mini-batch más pequeño (1 sola imagen), el máximo *shape* que admitía la red para que no se quedara sin memoria era (2048,1536, 1), que se conseguía haciendo un único submuestreo a las imágenes.

6 ENTRENAMIENTO Y SEGMENTACIÓN

Una vez habíamos solventado el problema de la memoria, ahora quedaba entrenar a la red y seleccionar aquellos ground truth que más nos acercaran a nuestra tarea de segmentar el texto. Descubrimos que el Data Generator podía ser aún más "multiusos", ya que, añadiéndole las líneas de código apropiadas, podíamos usarlo o bien para que nos diese la mitad superior del ZigZag Backbone, o bien para que nos diese la mitad inferior, o bien para que nos diese ambas en un mismo tensor, teniendo así, en definitiva, el Outline Backbone sin tener que generar nuevos tensores.

Probamos distintas configuraciones de redes tipo U-Net y distintas resoluciones de entrada. En este capitulo resaltaremos los primeros fallos que descubrimos y más tarde hablaremos de la solución final que acabamos adoptando.

Veremos entonces un primer acercamiento, donde enseñaremos cómo fracasamos en el intento de que el modelo aprendiera a extraer el contorno de las líneas de un texto, encontrándonos de frente con el problema de los *caracteres conflictivos* (aquellos que, al escribir a mano, pertenecen a una línea, pero conectan con otra).

Pasamos después a comentar una nueva estrategia que también fracasó, que la red neuronal sacase a la misma vez el contorno superior de las líneas y el contorno inferior (lo que hemos llamado la "ZigZagBackbone" y podemos comprobar en el punto 4.1.2). Después de analizar por qué no se puede hacer con una única red neuronal, decidimos hacer que se encarguen dos, de manera que una saque el contorno superior y otra el contorno inferior por separado. Para suplir los problemas derivados de fusionar estos dos contornos que han sido extraídos por separado, implementamos un tercer modelo que extrae el esqueleto o espina dorsal de las líneas.

Por último, hablaremos del proceso de fusión de cada trío de parte superior-media-inferior para crear una *máscara* que como ya hemos hablado en la introducción del capítulo 4, nos sirva para realizar la segmentación de las líneas del texto.

6.1 Entrenamientos realizados

6.1.1 Primer acercamiento: Entrenamiento con el Outline Backbone

El **primer contacto** que tenemos con la red nos sirvió para probar que funcionaba. Nótese que ni este entrenamiento ni esta red fueron las que usamos en la versión final, pero los mencionamos como un punto de partida y aprovechamos para mencionar algunos conceptos que hemos usado con el resto de entrenamientos.

La estructura de la red utilizada es la siguiente:

```
Model: "model_1"
__________________________________________________________________________________________________
Layer (type)                    Output Shape         Param #     Connected to
==================================================================================================
input_1 (InputLayer)            (None, 512, 384, 1)  0
__________________________________________________________________________________________________
conv2d_1 (Conv2D)               (None, 512, 384, 16) 160         input_1[0][0]
__________________________________________________________________________________________________
conv2d_2 (Conv2D)               (None, 512, 384, 16) 2320        conv2d_1[0][0]
__________________________________________________________________________________________________
max_pooling2d_1 (MaxPooling2D)  (None, 256, 192, 16) 0           conv2d_2[0][0]
__________________________________________________________________________________________________
conv2d_3 (Conv2D)               (None, 256, 192, 32) 4640        max_pooling2d_1[0][0]
__________________________________________________________________________________________________
conv2d_4 (Conv2D)               (None, 256, 192, 32) 9248        conv2d_3[0][0]
__________________________________________________________________________________________________
max_pooling2d_2 (MaxPooling2D)  (None, 128, 96, 32)  0           conv2d_4[0][0]
__________________________________________________________________________________________________
conv2d_5 (Conv2D)               (None, 128, 96, 64)  18496       max_pooling2d_2[0][0]
__________________________________________________________________________________________________
conv2d_6 (Conv2D)               (None, 128, 96, 64)  36928       conv2d_5[0][0]
__________________________________________________________________________________________________
up_sampling2d_1 (UpSampling2D)  (None, 256, 192, 64) 0           conv2d_6[0][0]
__________________________________________________________________________________________________
concatenate_1 (Concatenate)     (None, 256, 192, 96) 0           up_sampling2d_1[0][0]
                                                                 conv2d_4[0][0]
__________________________________________________________________________________________________
conv2d_7 (Conv2D)               (None, 256, 192, 32) 27680       concatenate_1[0][0]
__________________________________________________________________________________________________
conv2d_8 (Conv2D)               (None, 256, 192, 32) 9248        conv2d_7[0][0]
__________________________________________________________________________________________________
up_sampling2d_2 (UpSampling2D)  (None, 512, 384, 32) 0           conv2d_8[0][0]
__________________________________________________________________________________________________
concatenate_2 (Concatenate)     (None, 512, 384, 48) 0           up_sampling2d_2[0][0]
                                                                 conv2d_2[0][0]
__________________________________________________________________________________________________
conv2d_9 (Conv2D)               (None, 512, 384, 16) 6928        concatenate_2[0][0]
__________________________________________________________________________________________________
conv2d_10 (Conv2D)              (None, 512, 384, 16) 2320        conv2d_9[0][0]
__________________________________________________________________________________________________
conv2d_11 (Conv2D)              (None, 512, 384, 1)  17          conv2d_10[0][0]
==================================================================================================
Total params: 117,985
Trainable params: 117,985
Non-trainable params: 0
```

Figura 6-1. Estructura de la red usada. Resultado del model.summary()

Si representamos esta estructura con la herramienta online de la referencia [34], tendríamos un diagrama como el siguiente:

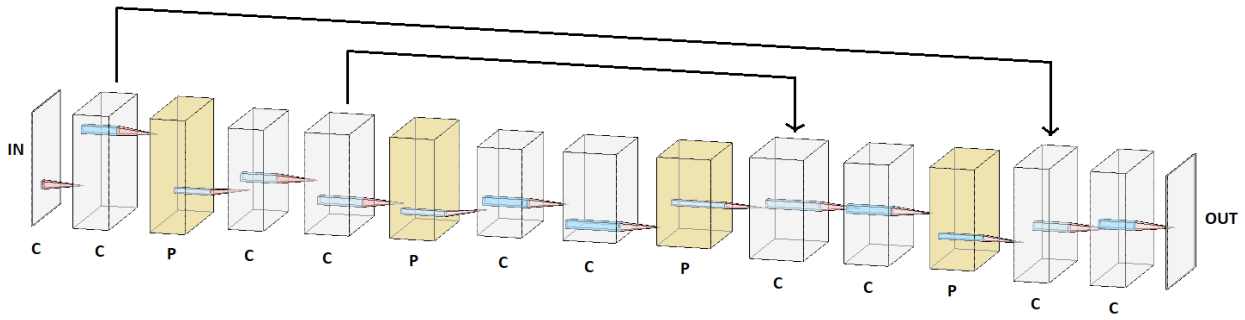


Figura 6-2. Diagrama de la red usada

Veamos ahora un ejemplo de la entrada y el ground-truth usado en este entrenamiento.

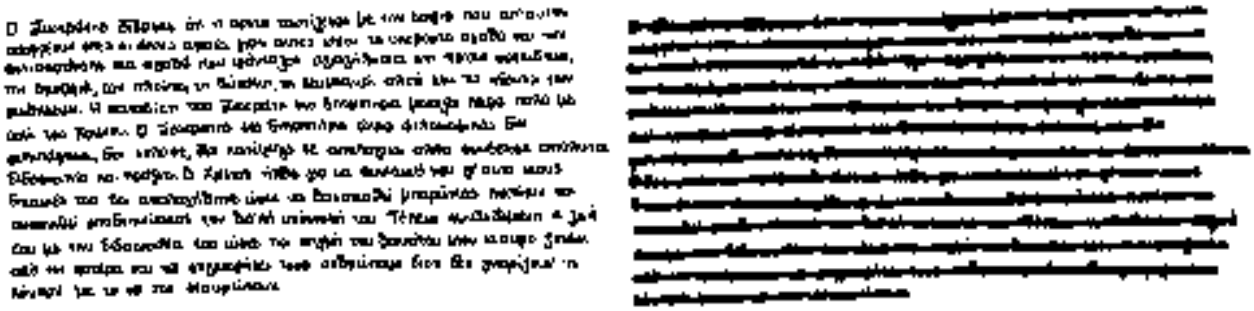


Figura 6-3. Entrada y ground-truth usados

Como podemos observar, el *shape* de entrada (512, 384, 1) y el aspecto de la imagen nos da la pista de que hemos usado las versiones submuestreadas. Para llegar a este *shape* hemos submuestreado 3 veces las imágenes tal y como se explicaba en el punto 5.1.1. Aunque este no sea el entrenamiento final, hay algunos conceptos que explicar que son comunes a los demás. Por eso, hay que fijarse en que la entrada es monocanal (cosa que siempre ha sido porque los colores se sacaban de un *colormap*), pero además observamos que está aprendiendo a dibujar el contorno de las líneas con un solo valor (1 – negro, 0 – blanco). Se podría llegar a pensar... ¿Y por qué no le hemos dado como ground-truth la versión colorida, aquella en la que los contornos tenían los valores "1", "2", "3" ... asociados a cada número de la línea? La respuesta es que ojalá se pudiera, pero las pruebas han determinado que es imposible que un modelo como el que tenemos pueda aprender directamente y sin mayor ayuda a realizar esa tarea.

La función de activación que utilizaban las capas intermedias era la función ReLU, mientras que la última capa de todas utilizaba la función logística. Así, los resultados acababan siendo cercanos a 0 o a 1, justo lo que necesitábamos.

Como ya sabemos, uno de los principales problemas que encontramos en el campo del Machine Learning es el *overfitting*. Este se puede prevenir utilizando un grupo de control que es el conjunto de validación (del que ya hablamos en el punto 4.2) y midiendo el rendimiento de este a cada época de entrenamiento. Cuando la época anterior tenga mejores resultados que la nueva obtenida, deberemos parar el entrenamiento porque habremos llegado al punto más óptimo. A esta técnica se le conoce como "*Early stopping*" y es ampliamente utilizada. Durante el entrenamiento, siempre hemos usado esta técnica. En cuanto a las métricas del modelo, decidimos elegir dos métricas muy estándares: como función de costes la precisión (*accuracy*) y como función de pérdidas la entropía binaria cruzada (*binary cross-entropy*). Pararemos cuando la precisión en el grupo de

validación sea máxima. Dejamos a continuación una gráfica que representa el entrenamiento:

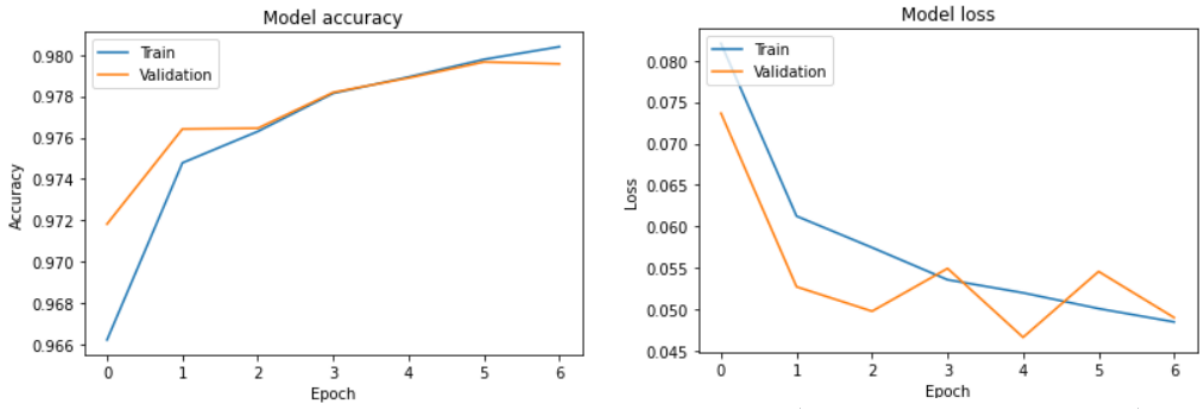


Figura 6-4. Entrenamiento de la red con Outline Backbone y formato de entrada (512,384)

Al ejecutar el test, nos sale una pérdida de 0.06502 y una precisión del 97,8789 %.

Estos resultados son muy esperanzadores para ser la primera vez, pero hay una serie de apuntes que debemos realizar:

1. Estamos entrenando a la red con imágenes submuestreadas. Esto podría dar (y, de hecho, daba) problemas a la hora de segmentar las líneas de texto, ya que al elevar la resolución de las predicciones perderíamos información.
2. Saber el contorno de las líneas es útil, pero como ya hemos dicho en numerosas ocasiones, existe el problema de los *caracteres conflictivos* y por lo tanto no es suficiente con esto. Por eso el objetivo final era sacar la parte superior, la parte inferior y la espina dorsal de los contornos de las líneas para realizar una relación unívoca que sortease el problema de los caracteres conflictivos.
3. El dato de la precisión del 97,8789% era espectacular. Sin embargo, dada la naturaleza de las imágenes, la mayoría de píxeles eran blancos y por lo tanto era "difícil" para la red no acertar el valor de un píxel. Por eso, mejor que fijarnos en el dato de la precisión debíamos entenderlo en conjunto con la función de pérdidas usada, la entropía binaria cruzada, que muestra también un cómputo que tiene en cuenta las veces que nos hemos equivocado.

Ahora que tenemos la certeza de que funcionan, vamos a intentar forzar un poco más la máquina.

6.1.2 Entrenamiento con la ZigZag Backbone

6.1.2.1 Primer acercamiento

Queríamos que aprendiera a extraer la ZigZag Backbone simplificada de un texto.

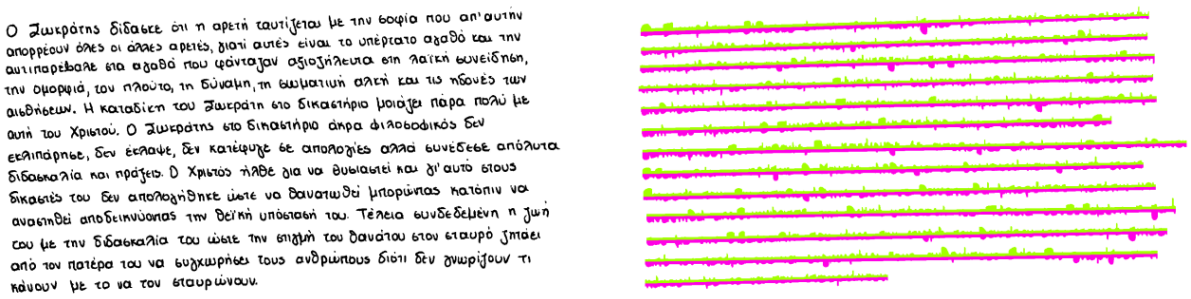


Figura 6-5. Entrada y ground-truth utilizados.

Como podemos recordar, el formato que presentaba la ZigZag Backbone simplificada (Figura 6-5, derecha) es una matriz de dos dimensiones que representa una imagen monocanal (como siempre). En esta, los píxeles del fondo tienen el valor "0", los contornos superiores de las líneas tienen el valor "1" y los contornos inferiores el valor "-1". Para realizar la gráfica de la Figura 6-5, hemos utilizado un *colormap* que asocia el valor 0 al blanco, el 1 al verde y el -1 al rosa.

Nuestro siguiente intento fue por la siguiente línea:

- Subimos la resolución de entrada hasta un *shape* = (2048, 1536). El entorno proporcionado por Google Colab se caía en algunos de los entrenamientos que efectuamos con esta resolución incluso cogiendo mini-batches de tamaño 1 (se carga solo 1 imagen en memoria cada vez, más 2 extra por el *data augmentation*).
- Cambiamos todas las funciones de activación, si queríamos que predijera una imagen con valores acotados entre -1 y 1, no nos servían las ReLU que habíamos utilizado. Por ello cambiamos todas las funciones de activación por la sigmoide hiperbólica para que los resultados estuviesen acotados entre estos valores.

El resultado fue un completo desastre. Para empezar, al haber cambiado la función de activación por una de complejidad computacional muy superior y haber aumentado la resolución, el entrenamiento duró muchísimo más de lo que tardó en el intento anterior. Además, la precisión cayó por debajo del 90%, que a priori se puede interpretar como algo más o menos positivo, pero las pérdidas estaban por encima de 16. La explicación es que la precisión mide cuántos píxeles se han acertado, lo cual no es difícil de acertar porque la mayoría tienen valor 0 (fondo), mientras que la entropía binaria cruzada es una medida más compleja que guarda relación con cuántas veces hemos dado "falsos positivos" y "falsos negativos" (es decir, cuándo nos hemos equivocado).

Intentamos de todo para hacer que el modelo funcionase. Nuestro segundo intento fue en camino de eliminar esta nueva función de activación que habíamos introducido. Para ello, lo que hicimos fue que el DataGenerator dividiera la imagen entre 2 y sumase 0,5 a la misma. Así, habríamos trasladado los valores de antes (-1 – mitad inferior, 0 – fondo, +1 – mitad superior) a: 0 – mitad inferior, 0,5 – fondo, 1 – mitad superior. Lo que conseguíamos ahora era prescindir de los valores negativos para que pudiésemos volver a usar las funciones ReLU y logística. El tiempo de entrenamiento se vio drásticamente reducido, pero los resultados seguían siendo pésimos.

6.1.2.2 Versión final

Descartamos así totalmente la posibilidad de que la red aprendiese a realizar el ZigZag Backbone simplificado de un texto manuscrito de una sola pasada. Entonces nos conformamos con la idea de entrenar **dos redes**, una que aprendiera a discernir la mitad superior de una línea y otra la mitad inferior. Así, las imágenes con las que alimentaríamos a la red tendrían la siguiente forma:

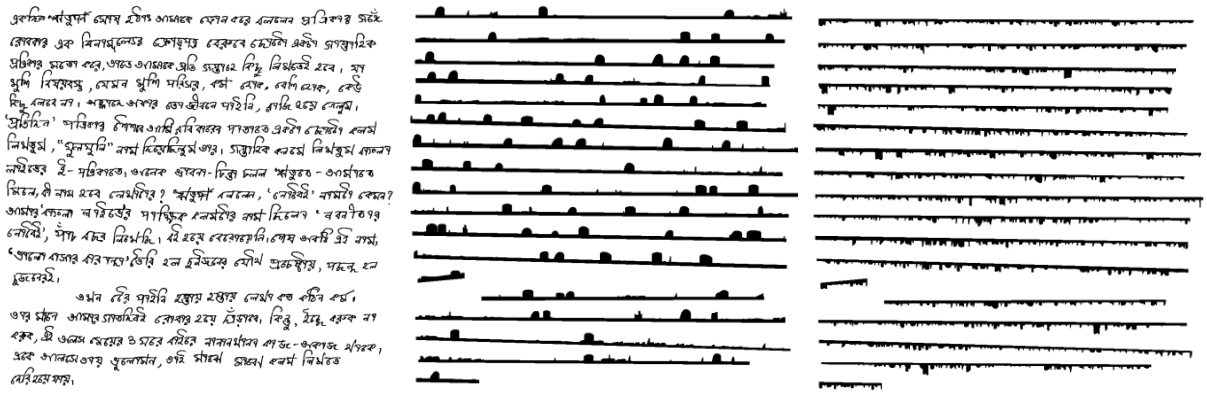


Figura 6-6. Entrada y ground truth de la primera red neuronal y de la segunda.

Entonces la red aprendería a segmentar por un lado el contorno superior y por otro lado el inferior y después

los fusionaríamos.

El problema era que la precisión que lográbamos seguía siendo más bien baja, por lo que lo último que quedaba por ajustar era la propia red. Aquí vino una pregunta que surge siempre que tenemos una red de tipo UNet y queremos mejorar el rendimiento del modelo, ¿Debíamos aumentar el tamaño de los filtros o debíamos aumentar la profundidad de la red? La conclusión a la que se ha llegado ha sido muy clara: la red debe crecer en profundidad, no en anchura (referencias [35] y [36]).

La nueva red tiene más capas que la de antes, repitiendo el mismo patrón que podemos observar en la anterior, y el número de parámetros había crecido bastante (de aproximadamente 120.000 parámetros a casi 2.000.000). Aquí podemos ver la estructura del nuevo modelo que utilizaríamos tanto para la red neuronal de las partes superiores como para la de las partes inferiores:

```
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 6-7. Estructura de la nueva red. Resultado del model.summary()

Crearíamos ahora en realidad 2 modelos, pero gracias a la versatilidad que tiene, podíamos simplemente cambiar algunos parámetros del *Data Generator* y usarlo para que sirviese tanto para dar los ground-truth de los contornos superiores como los de los inferiores. Ajustamos el parámetro de la paciencia del Early Stopping para que diese un resultado mejor (más información en la referencia [37]. Así, los resultados del entrenamiento para el modelo de la mitad superior fueron:

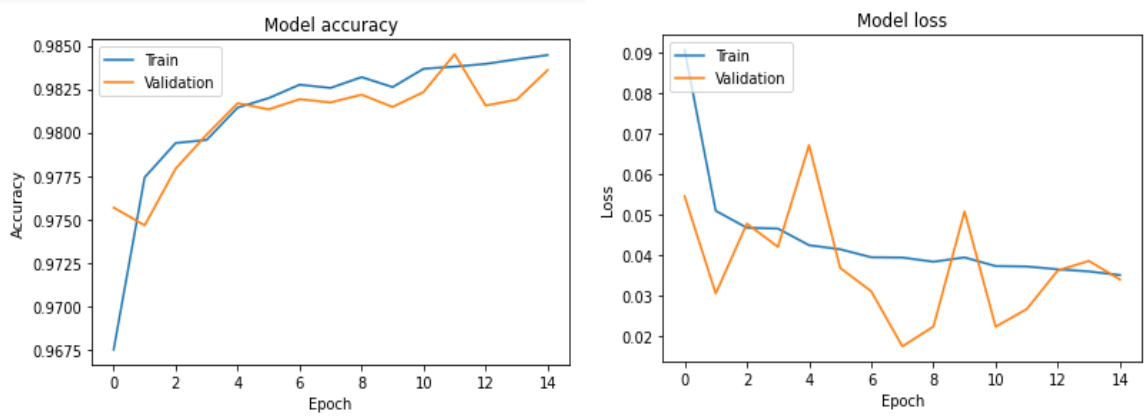


Figura 6-8. Resultados del entrenamiento para la mitad superior.

Además, tras varios intentos nos quedamos con los dos mejores modelos, que obtuvieron estos resultados en el test:

- Para la mitad superior, el modelo alcanzó una precisión de 97.9613% y una entropía binaria de 0.0453
- Para la mitad inferior, el modelo alcanzó una precisión de 97.8768% y una entropía binaria de 0.0325

Vamos a enseñar una predicción de las dos redes para la entrada que vemos en la Figura 6-6:

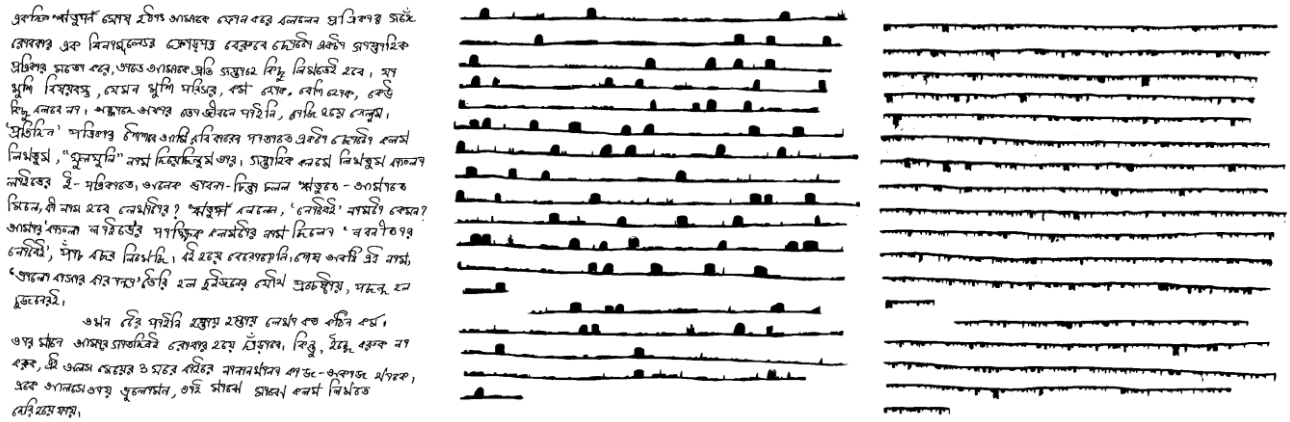


Figura 6-9. Predicciones de los contornos de la mitad superior e inferior de la imagen 314.

Los resultados que aquí vemos son bastante decentes, y con estos ya podríamos hacer una segmentación más o menos apropiada de la siguiente manera:

1. Etiquetamos las líneas de la mitad superior (píxeles negros de la primera línea a 1, píxeles negros de la segunda línea a 2…) y hacemos lo mismo las de la mitad inferior.
2. Unimos ambas imágenes correctamente etiquetadas y la usamos como máscara para ponerla encima del texto original tal y como dijimos que se haría en el principio del capítulo 4.

La teoría es muy bonita, pero cuando unimos ambas imágenes descubrimos que en la mayoría de casos no coinciden a la perfección. Por eso teníamos que ayudarnos de una predicción más que nos ayudaría a realizar una fusión más precisa: la Thick Backbone.

6.1.3 Entrenamiento con la Thick Backbone

La Thick Backbone es la última pieza que nos falta para resolver este rompecabezas. Recordemos que la Thick Backbone de un texto es una imagen donde hay una línea gruesa en la posición de cada línea, que muestra el esqueleto o espina dorsal de la misma. Como ya hemos dicho en el párrafo anterior, el motivo por el cuál queremos predecir la Thick Backbone es porque esto nos ayudaría a fusionar las partes superiores e inferiores predichas en las dos anteriores redes y así conseguir la *máscara* de la que hablamos en el capítulo 4 con la que realizar la segmentación.

Por eso, utilizando la misma estructura de la UNet de la Figura 6-7, decidimos entrenar un nuevo modelo para extraer la Thick Backbone. Además, los entrenamientos no tendrían por qué comenzar desde cero, ya que podíamos cargar un modelo entrenado anterior y comenzaríamos desde un punto más alto que el simple "comienzo aleatorio". Como ya hemos hablado suficiente de los entrenamientos de las otras redes y este es uno muy parecido, vamos a enseñar solamente un ejemplo de predicción del modelo:

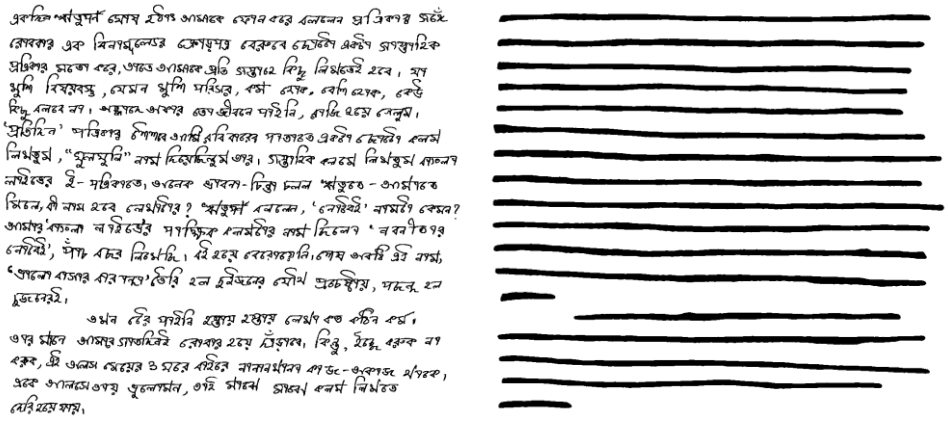


Figura 6-10. A la izquierda, la imagen 314. A la derecha, la predicción de su Thick Backbone.

La predicción del modelo en el conjunto de test es de 97,3070% y una entropía binaria cruzada de 0,0498.

6.2 Proceso de fusión y segmentación

Los 3 modelos de Deep Learning que tenemos son capaces de sacarnos: los contornos superiores, los contornos inferiores y la espina dorsal gruesa de una línea de texto. Como ya sabemos, andamos buscando una máscara que poner encima del texto original que tenga un número diferente por cada línea (representado por un color gracias a la utilización de un *colormap*). Aún a riesgo de parecer repetitivos, vamos a rememorar el proceso final que queremos conseguir:

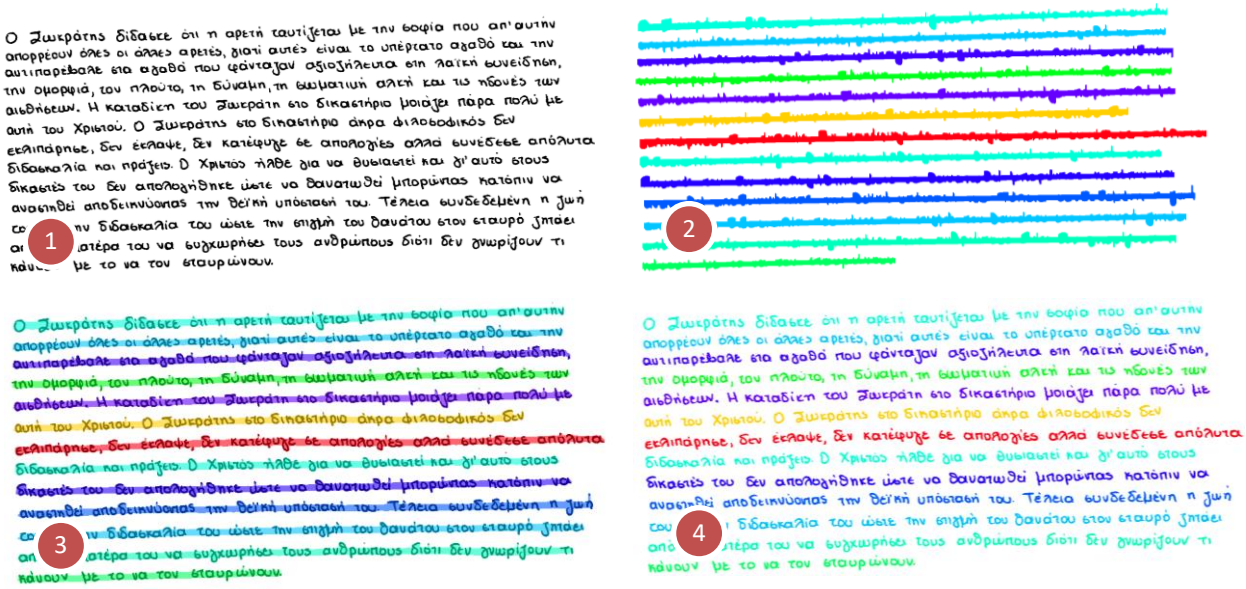


Figura 6-11. Proceso de segmentación una vez conseguida la máscara

Como podemos observar, partimos de la imagen original (1) y de la máscara una vez calculada (2). A continuación, compararíamos los valores píxel a pixel de la una con la otra, lo cuál equivale a "ponerla encima" (3). Crearíamos entonces una imagen nueva (4) donde los píxeles que valían "1" (negros – texto) en la imagen original tomarían como nuevo valor el que tuviera la máscara en esas coordenadas.

Para conseguir esta máscara tendremos que fusionar apropiadamente las 3 piezas que tenemos, que son: la predicción de las partes superiores de las líneas, la de las partes inferiores y la espina dorsal. A este proceso le hemos llamado **proceso de fusión**, denotando una carencia completa de imaginación y originalidad.

Al proceso que hemos visto en la Figura 6-11 le hemos llamado (como era de esperar) **proceso de segmentación**.

A continuación, vamos a ver ambos procesos y los errores que hemos corregido en ambos.

6.2.1 Proceso de fusión

Nota para el lector: El mapa de colores que utilizamos es completamente aleatorio y se consigue extrayendo "n+1" (de las "n" líneas) colores aleatorios del *colormap* "*rainbow*" de la librería matplotlib, forzando a que el primero de todos siempre sea blanco para que corresponda con el 0 (fondo) en las imágenes. Es por la naturaleza de la aleatorización y la limitación de colores que tiene el mapa que en ocasiones puede parecer que haya 2 colores parecidos, pero que en realidad, son distintos.

Antes de comenzar a explicar lo que vamos a hacer con las 3 predicciones de nuestros modelos, debemos definir un concepto que hasta ahora no habíamos mencionado: los *Connected Components*.

Un Connected Component, al que nos referiremos ahora por sus siglas, CC, es un concepto que hereda de la teoría de grafos y que representa una región cerrada de la imagen cuyos píxeles están conectados y tienen el mismo valor numérico. Se diferencian así dos regiones, la región interior, cuyos píxeles están totalmente rodeados (en cualquiera de las 8 posiciones posibles) por píxeles de su mismo valor, y la región exterior, cuyos píxeles están rodeados por al menos un píxel de su mismo valor y al menos uno de un valor diferente. Para etiquetar CC, utilizaremos la función scipy.ndimage.label (referencia [38]). Esta función puede tomar como parámetro una matriz 3x3 que representará una máscara para segmentar los píxeles de la región exterior, ya que en ocasiones nos basta con que tan solo algunos de los 8 píxeles de su alrededor sean de distinto valor para segmentarlo como un nuevo CC. Por ejemplo, el valor por defecto que tiene esta matriz es en forma de cruz, por lo que, si dibujásemos una línea de 1 píxel de grosor y tuviese un escalón en forma de diagonal, la línea sería segmentada en 2 partes con la máscara por defecto (ver documentación para explicación más detallada). Lo que hace esta función lo podemos intuir en la siguiente imagen sacada de la referencia [39]:

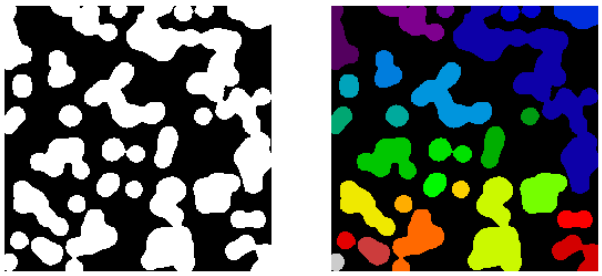


Figura 6-12. Funcionamiento de scipy.ndimage.label. Referencia [39]

La manera de proceder para obtener la deseada máscara va a ser la siguiente:

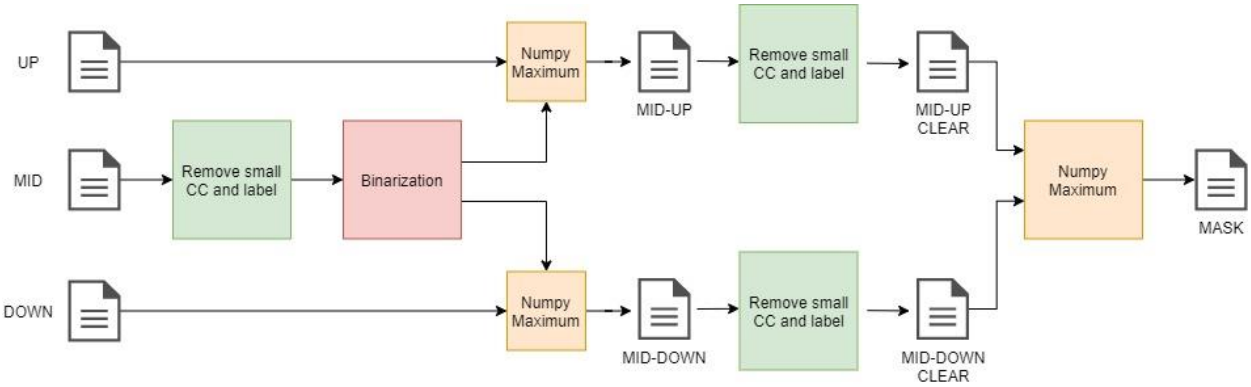


Figura 6-13. Proceso de creación de la máscara

Donde "Up" es la predicción de los contornos superiores de las líneas, "Mid" la predicción de la espina dorsal del texto y "Down" la predicción de los contornos inferiores de las líneas.

Pasamos a explicar el diagrama:

- Lo primero que haremos será pasar la predicción de la Thick Backbone por una función llamada *RemoveSmallCC* (la mencionaremos muchas veces en lo que queda de la memoria) que se encargará de etiquetar los CC y eliminar aquellos cuyo tamaño sea menor que un cierto número de píxeles. Tras varias pruebas, determinamos que esta tolerancia debía ser de aproximadamente 2000 - 3500 píxeles (con esta resolución de 4096x3072 después de haber sobremuestreado para revertir el cambio que se hizo al alimentar a la red, equivale a una pequeña mancha en la imagen que es prácticamente imposible que sea una espina dorsal). Ponemos un ejemplo gráfico de lo que hace esta función:

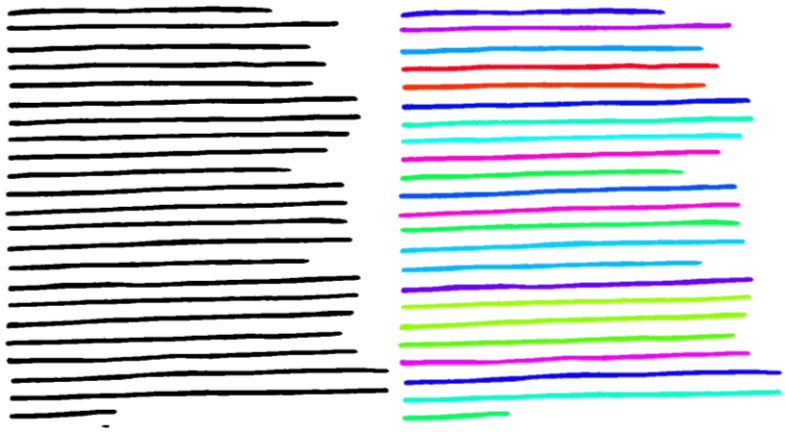


Figura 6-14. Función RemoveSmallCC con la Thick Backbone de la imagen 032

Podemos observar como un pequeño fragmento que apareció cerca de la última línea se ha eliminado. Además, los CC de la imagen han sido etiquetados y se les ha puesto: a los píxeles de la línea 1, el valor numérico 1, a los de la línea 2, el 2…

- Una vez que hemos hecho esto, binarizamos el resultado (volvemos a quedarnos con la imagen con todos los pixeles negros a 1 y el fondo blanco a 0) y cogemos el máximo elemento a elemento entre la matriz de predicción de la parte de arriba y la que acabamos de binarizar. Hacemos lo mismo con la parte de abajo. Lo que conseguimos con esta función es tener un contorno superior e inferior mucho más delimitado, libre de huecos como podría tener en caso de no hacer esto. Dejamos aquí un ejemplo gráfico:

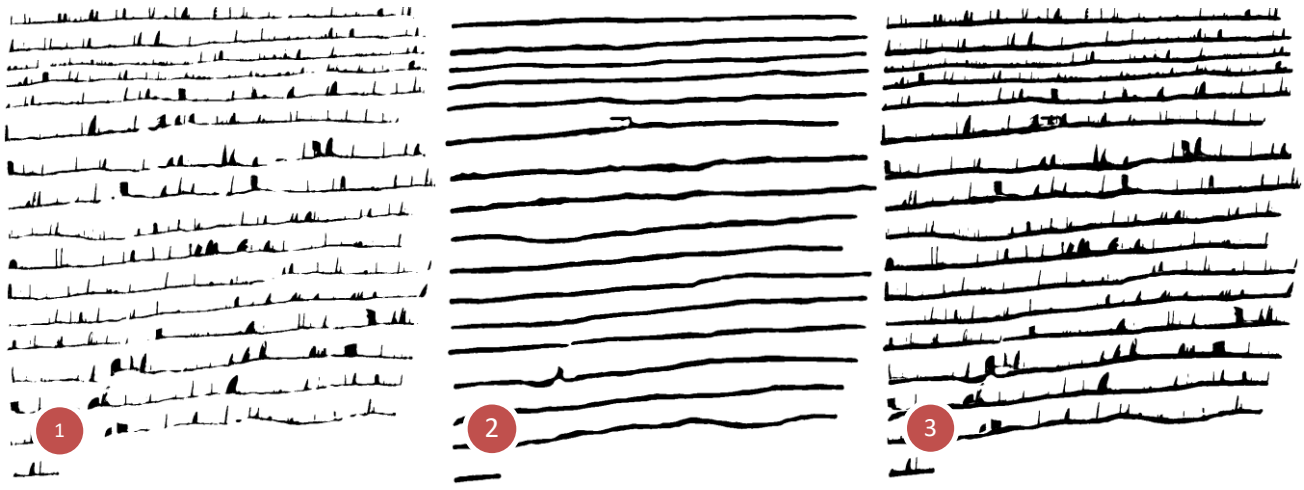


Figura 6-15. Construcción de midUp (3) a partir de Up (1) y Mid (2)

- Las dos imágenes que hemos conseguido en el paso anterior (llamadas *mid-up* y *mid-down*) las pasamos por la función *RemoveSmallCC* para limpiarla al igual que hicimos anteriormente con la otra, quedando así los CC correctamente etiquetados.
- Una vez que tenemos las dos imágenes solo nos queda coger el máximo elemento a elemento de

nuevo en lo que llamamos. De esta manera, la máscara quedará justo como la queremos:

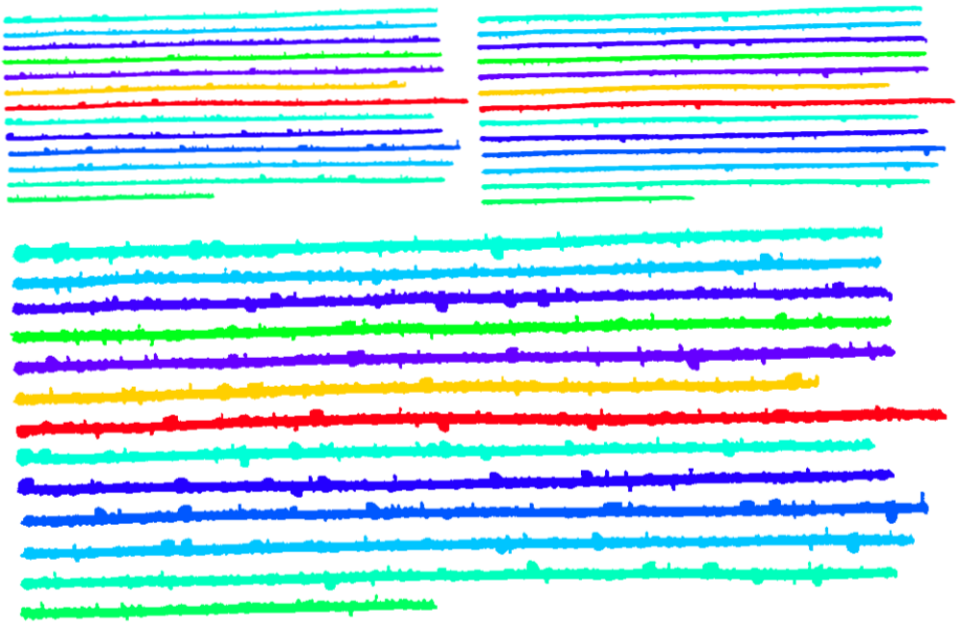


Figura 6-16. Arriba, el mid-up y mid-down y abajo, la máscara de la 001

Con esto y poniéndolo encima de la imagen original, somos perfectamente capaces de realizar la segmentación (se ve más en el 6.2.2 Proceso de segmentación). Los resultados además son sorprendentemente precisos, pero aún hay margen de mejora debido principalmente a que surgen dos problemas:

1. El primer problema es que hay píxeles que no entran dentro de la máscara que acabamos de crear, creando así una situación un poco absurda, puesto que sabemos que estos píxeles representan texto (tienen valor 1 en la imagen original), pero no los asignamos a ninguna línea. *Este problema no pertenece al proceso de fusión y por ello lo trataremos en el siguiente punto: 6.2.2 Proceso de segmentación*.
2. El segundo problema es que en ocasiones el número de CC que se detectan en la imagen *midUp* y los que se detectan en *midDown* no coinciden, lo que crea una máscara completamente errónea. Este problema sí que pertenece al proceso de fusión y va a ser tratado de inmediato.

6.2.1.1 Errores en la fusión.

Para la creación de la máscara es requisito indispensable que la mitad superior (midUp) y la mitad inferior (midDown) de las segmentaciones coincidan en el número de CC y en las posiciones que guardan. De las 350 imágenes que existen en el dataset, alrededor de unas 70 de ellas presentaban problemas en la segmentación. Inspeccionando estas imágenes descubrimos que existía un patrón entre algunas, y que, por lo tanto, los fallos se podían "categorizar". Vamos a tratar por lo tanto 4 subsecciones a continuación: la primera servirá para describir estas categorías, la segunda tratará acerca de como se procede a la detección de esos casos, la tercera, sobre cómo corregir los fallos y la última, sobre cómo cambian las imágenes tras los cambios introducidos.

6.2.1.1.1 Descripción de los casos de error

CASO "A"

Recordemos por un momento el diagrama de la figura 6-13. Al fusionar la predicción de la Thick Backbone con la predicción de cualquiera de las dos mitades, puede pasar que dos líneas acaben juntas por error debido a la imperfección de la predicción. Cuando esto ocurre en una mitad de las líneas, pero en la otra no (que además suele ser lo más habitual), la enumeración de las líneas no coincide, dando resultados desastrosos como estos:

Figura 6-17. Caso "A" en la imagen 031.

Debemos fijarnos aquí en que: la midUp (1), fruto de unir las predicciones de Thick Backbone y la del contorno superior, no presenta problemas de ningún tipo. La midDown (2), fruto de unir las predicciones de Thick Backbone y la de contorno inferior, presenta un problema rodeado con un círculo. Este problema, que hemos denominado *Caso "A"*, se materializa en que las líneas 10 y 11 de midDown han sido predichas como "juntas" y por culpa de eso, la máscara (3 – fusión de ambas) ha juntado valores de líneas en un lado con valores de línea en otro, dando lugar a que a partir de la línea 10, toda la máscara esté mal hecha porque la enumeración no concuerda.

CASO "B"

El caso "B" se da mayoritariamente cuando una imagen presenta un hueco en una de sus líneas, aunque también puede ocurrir cuando varios caracteres están muy separados entre sí. Para que se dé el caso "B", una de las mitades (midUp o midDown) tiene que haber detectado el hueco mientras que la otra no. De este modo, en una de ellas la línea tendrá 2 partes (lo que significa que hay 2 CC) y en la otra solo 1. Esto hace que la enumeración de las líneas no coincida a la hora de fabricar la máscara y vuelva a haber un fallo en la segmentación. Veamos gráficamente cómo se manifiesta:

Figura 6-18. Caso "B" en la imagen 089.

La situación descrita se puede ver en la línea 14 de esta ilustración. En la midUp (1), se ha detectado que había un hueco en el texto, mientras que en la midDown (2) no. Por ello, la máscara fusión de ambas (3) ha tenido una mala enumeración a partir de esa línea. Además, debemos fijarnos en otro detalle: en la imagen del centro (midDown) no sólo ha ocurrido el caso "B", sino que, además, la línea 14 y la 15 están juntas, lo cuál hemos definido más arriba como caso "A". Esto significa que nos podemos encontrar situaciones mixtas donde convivan diferentes casos.

CASO "C"

El caso "C" ocurre bajo las mismas condiciones del caso anterior: que haya un pequeño hueco en el texto manuscrito. Sin embargo, mientras el caso "B" se daba cuando una mitad detectaba el hueco y otra no, el caso "C" se da cuando ambas mitades detectan el hueco. Así, una misma línea tendrá dos partes en el resultado final. En el concurso del ICDAR 2013, esto se considera una segmentación errónea. Sin embargo, en nuestra opinión, hay algunos textos donde el caso "C" está más que justificado, siendo la unión de esas líneas algo demasiado artificial, que puede no corresponderse con el objetivo de este proyecto.

Aquí tenemos un caso "C" representado de forma gráfica, conviviendo por cierto con un caso "A":

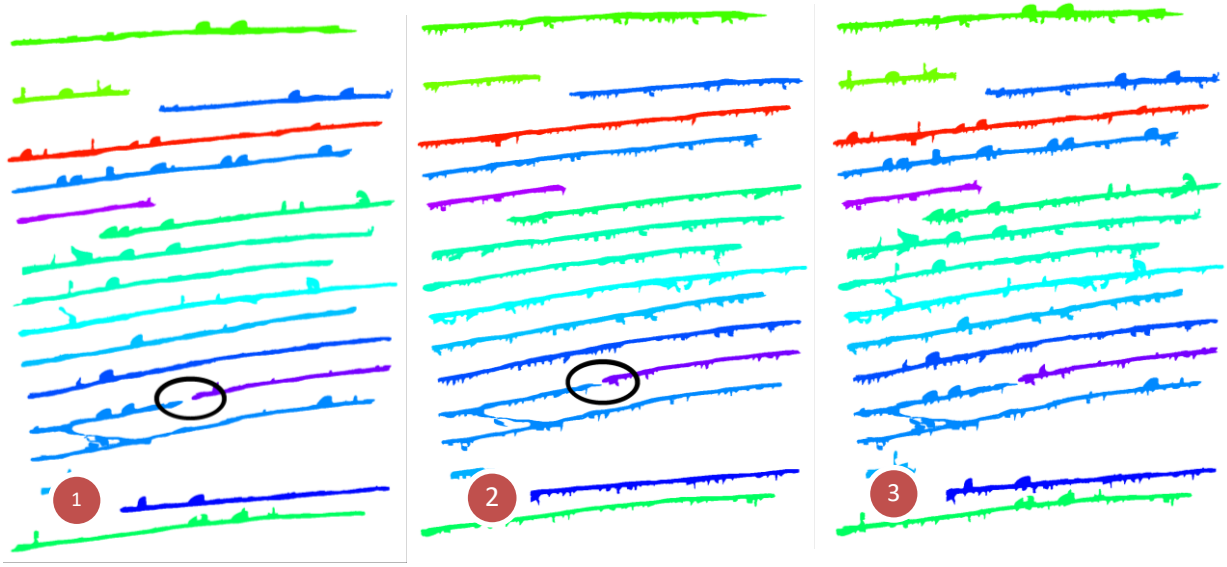


Figura 6-19. Caso "C" en la imagen 155

Aquí vemos que en la midUp (1) se ha detectado un hueco en la línea y en la midDown (2) también. Por haberse detectado en ambos, la máscara (3) fusión de ambas no ha sufrido desperfectos.

CASO "D"

El caso "D" es el que se da con menor frecuencia de todos, y es una derivación del caso anterior. La función que utilizamos para etiquetar los CC, la scipy.ndimage.label, les pone valores consecutivos a los CC que se va encontrando de arriba abajo y de izquierdas a derechas. De esta manera, si casualmente un hueco ha dividido una línea en 2 partes y en la midUp (1) se ha enumerado primero la izquierda y luego la derecha mientras que en la midDown (4) ha ocurrido lo contrario, estaremos ante un caso "D" y tendremos mal la máscara (3) pero sólo en esa línea.

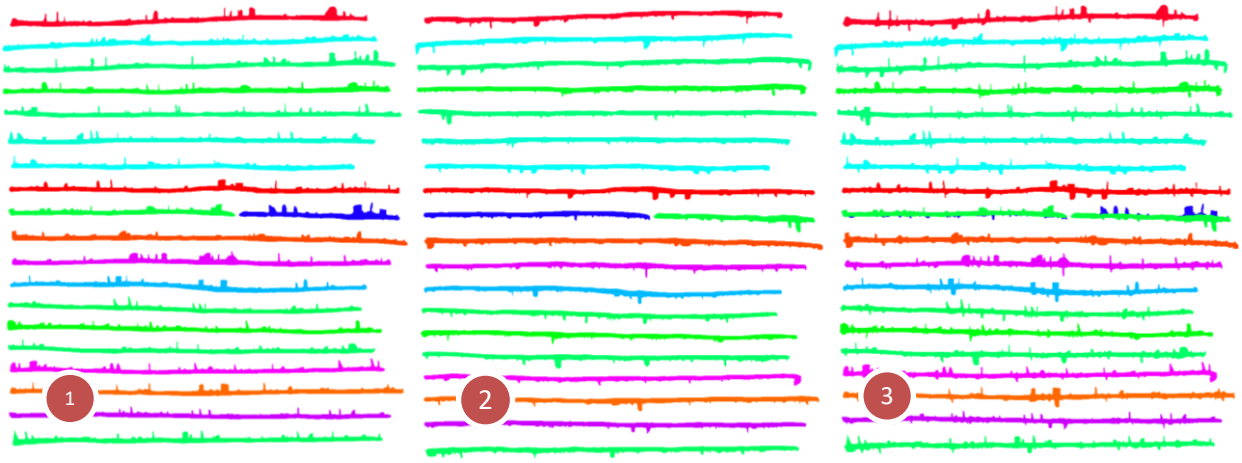


Figura 6-20. Caso "D" en la imagen 155.

De estos 4 problemas, podemos ver claramente que los casos "A" y "B" son mucho más destructivos que los "C" y "D", ya que mientras que los dos primeros arruinan por completo las líneas posteriores a donde sucede el problema, los dos últimos cometen un error que es local y no se expande más allá de la línea donde surge el problema. Es por ello que vamos a tratar exclusivamente a los casos "A" y "B", dejando el tratamiento de los otros dos como una posible mejora. Vamos a pasar ahora a ver cómo detectamos el caso "A" y el caso "B" y después explicaremos cómo lo hemos arreglado.

6.2.1.1.2 Detección de los casos de error

El programa de segmentación comprobará después de pasar a las imágenes midUp, mid y midDown por la función "removeSmallCC" que las 3 tienen el mismo número de CC. De esta manera, a no ser que ocurra un número par de errores o una situación muy extraña, lo más posible es que estemos ante un caso "A" o un caso "B" (o incluso más de uno). Entonces, el programa creará un diccionario donde recoja las dimensiones de cada CC de las imágenes midUp y midDown, creará un archivo de debug con el contenido de ese diccionario, y pasará el hilo de ejecución a una función auxiliar que se encuentra en otro módulo del proyecto. A esta función le pasará las tres imágenes y el diccionario, con la esperanza de que le devuelva dichas imágenes correctamente arregladas o bien al menos se intente, pero se siga hacia adelante sabiendo que la segmentación puede albergar errores.

Dentro de nuestra función auxiliar, jugaremos con algunos conceptos estadísticos para intentar averiguar qué línea es diferente a las demás y así identificar cuales tienen errores y de qué casos se tratan.

En concreto, estudiaremos dos variables aleatorias que vamos a definir ahora mismo y que son:

> X_midUp ~ variable aleatoria que representa el tamaño en pixeles de las líneas de midUp.
> $X\_midDown$ ~ variable aleatoria que representa el tamaño en pixeles de las líneas de midDown.

Dentro del diccionario que construimos encontramos las listas de las muestras que tienen estas variables aleatorias, y podemos hallar, con funciones básicas de Numpy, su media y su desviación típica. Una vez tengamos la media y la desviación típica, podemos examinar uno por uno el tamaño de las líneas de midUp y de midDown para ver cuales son aquellas líneas que están resaltando por encima de las demás. Veamos ahora cómo detectar:

CASO "A"

Para que se dé un caso "A", un CC tiene que tener el tamaño de la suma de dos líneas (un número muy grande dentro del conjunto de tamaños de CC). El criterio que hemos seguido para que una línea sea detectada como caso "A" es que su tamaño sea superior a la media de los tamaños de las líneas de su imagen sumándole 2,3 veces la desviación típica. Cuando se detecta un Caso "A", se procede a pasar el mando a otra función auxiliar que arreglará el error y devolverá la imagen parcheada. Hablaremos más delante de dicha función.

CASO "B"

Para que se encuentre un caso "B", lo primero que debemos encontrar es una línea cuyo tamaño sea inferior que la media de tamaños. Sin embargo, esta condición es necesaria pero no suficiente para determinar un caso "B", ya que, en la gran mayoría de manuscritos, la última línea de un párrafo no suele llegar al final, lo cuál hace que su tamaño sea bastante inferior con respecto a la de los demás. Necesitamos entonces una manera de diferenciar una línea pequeña de un caso "B".

Si recordamos la descripción de los casos "B", notaremos que para que exista un error así, la línea homóloga a la que tenemos en la imagen complementaria debe ser muy grande en comparación con la de la otra. Así, como tenemos las variables X_midUp y X_midDown, podemos suponer que son variables normales y sacar la variable aleatoria "$Y= X\_midUp - X\_midDown$", que con la aproximación mencionada, podemos sacar su media y la desviación típica. De esta manera, cuando sospechemos de una línea pequeña, podremos restarle su homóloga y comparar el resultado con la media y la desviación típica de esta nueva variable aleatoria. En concreto:

1. Se considera "línea pequeña" si sus dimensiones están por debajo de la media menos 1,2 veces la desviación típica de los tamaños de las líneas.

2. Una "línea pequeña" se considera un caso "B" si la resta con su homóloga (siempre resta del midUp – el midDown) está fuera del intervalo definido por la media de la variable Y ± 0.6 veces su desviación típica.

6.2.1.1.3 Corrección de los casos de error

Una vez que tenemos un caso "A" o un caso "B" detectado y conocemos la línea donde se encuentran los errores, le pasamos el hilo de ejecución a una función diferente ("*fixCaseA*" o "*fixCaseB*"), que reciben como parámetros las imágenes a arreglar y la línea errónea, e intentan corregir el error. Si no lo consiguen, devolverán la imagen como la tengan de arreglada (el mejor intento) y un código de error para que el programa principal sepa que se ha hecho todo lo posible, pero la imagen puede seguir teniendo errores. Se contempla que las imágenes tengan múltiples errores, de manera que estos se irán corrigiendo uno por uno. Cuando se arregla un error, se vuelve a analizar desde el principio, actualizando todos los valores cambiados para que no haya ningún problema, hasta un número máximo de intentos para evitar bucles infinitos. La manera de corregir los fallos es la siguiente:

CASO "A"

1. Extraemos la línea que supuestamente ha sido fusionada con la siguiente.
2. Hacemos la recta de regresión (reutilizando la función "createLinearRegressionBackbone del apartado 4.1.1") y con suerte dicha recta cortará las dos líneas fusionadas por la mitad.
3. Ponemos a 0 (fondo) los píxeles de la recta.
4. Pasamos el resultado por la función "removeSmallCC" de la que hemos hablado al principio de este apartado.
5. Si el resultado son 2 CC diferenciados, hemos hecho bien e insertaremos las líneas en su sitio desplazando los números de las líneas siguiente. Si el resultado no son 2 CC, algo hemos hecho mal y devolvemos la imagen al programa principal tal y como está, con un código de error.

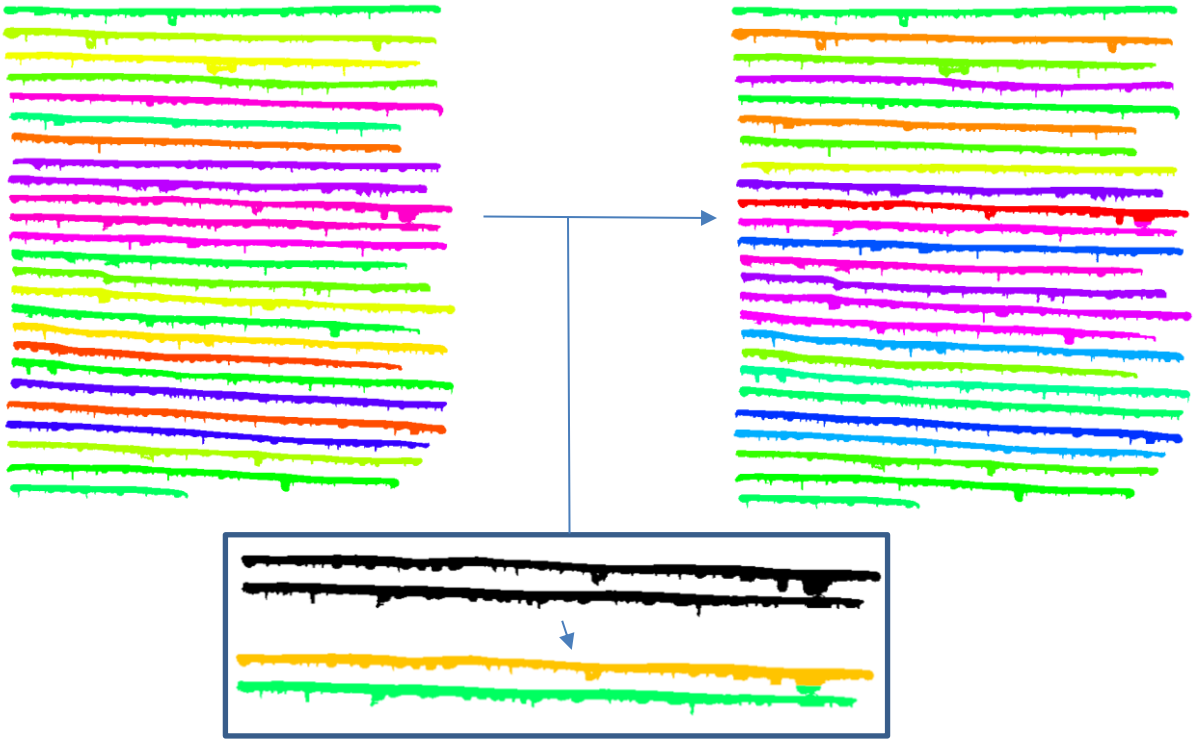


Figura 6-21. Proceso de corrección case "A" en la midDown de la imagen 031.

CASO "B"

1. Extraemos la línea donde hemos detectado el caso "B" y también extraemos a la siguiente para fusionarlas.
2. Sacamos la parametrización de la recta de regresión de ambas líneas.
3. Sacamos una media de las posiciones donde comienzan y donde acaban las líneas de la imagen, y evaluamos ambas rectas de regresión con 3 píxeles de ancho a lo largo de esos dos valores.
4. Pasamos el resultado por la función "removeSmallCC" de la que hemos hablado a principios de este apartado
5. Si el resultado son 2 CC diferenciados, quiere decir que esas dos líneas estaban destinadas a estar juntas, pero no se juntaron por equivocación, por lo que desplazamos los números de las líneas posteriores e insertamos el resultado. Si por el contrario no son 2 CC diferenciados, las dos líneas nunca debieron estar juntas y por lo tanto devolvemos la imagen al programa principal tal y como lo tenemos, con un código de error.

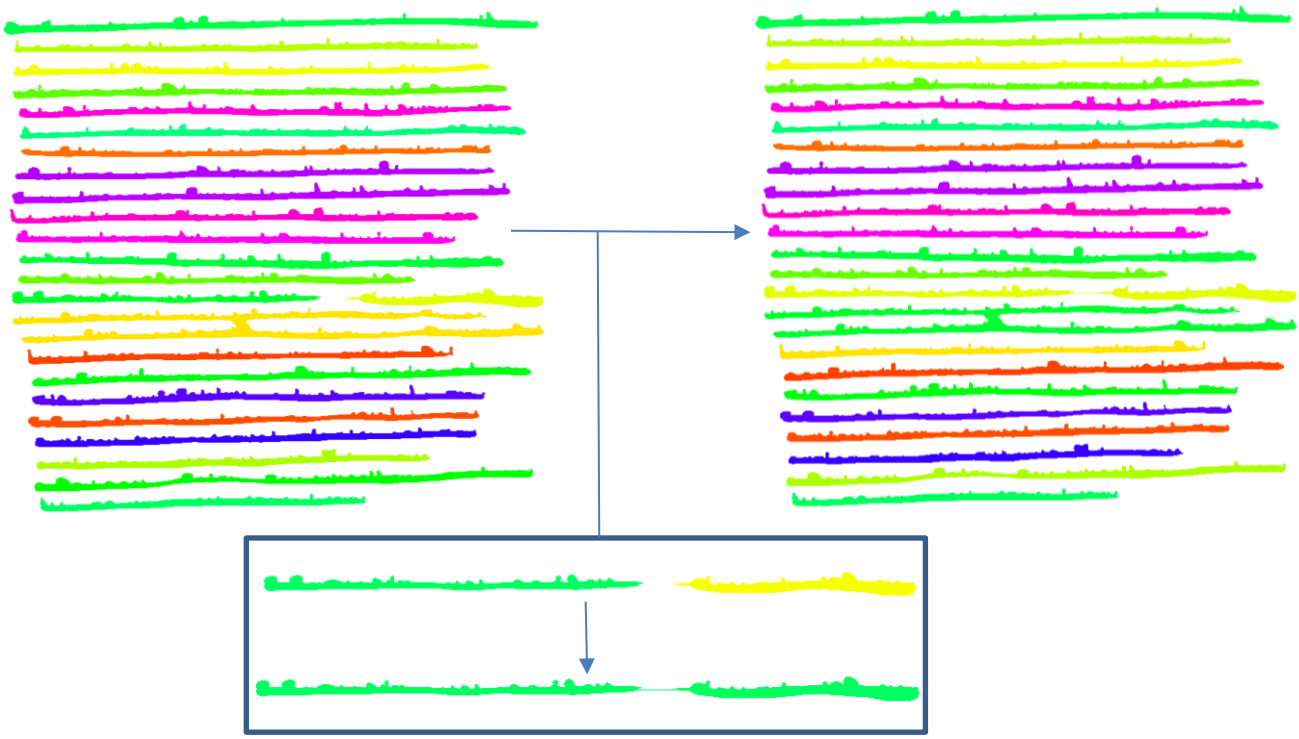


Figura 6-22. Proceso de corrección case "B" en la midUp de la imagen 089.

6.2.1.2 Resultados de la corrección

Gracias a estas correcciones que acabamos de ver, de las 67 imágenes que se han detectado como posibles errores, encontramos:

- 19 casos "C" y "D" que, como comentamos en el punto 6.2.1.1.1, decidimos dejar sin corregir porque estos casos solo afectaban de manera local en la construcción de la máscara.
- 48 casos "A" y "B", que tras pasar por el programa que aplica los algoritmos aquí contados, hemos conseguido arreglar completamente 28 de ellos. De entre los 20 restantes, la mayoría de ellas han sido parcialmente arregladas, algunas no han sido arregladas en absoluto y otras en realidad se detectaron como errores sin serlo ("falsas alarmas").

6.2.2 Proceso de segmentación

Ahora que ya hemos conseguido la máscara, toca realizar el proceso descrito Figura 6-11, que consiste en comparar elemento a elemento la matriz de la máscara con la imagen original y crear una imagen nueva donde aparezcan los mismos píxeles de fondo que en la matriz original, pero con los píxeles de texto igualados al valor que tenga la máscara en esa posición. Para ilustrar el proceso, vamos a traer una copia de la figura aquí.

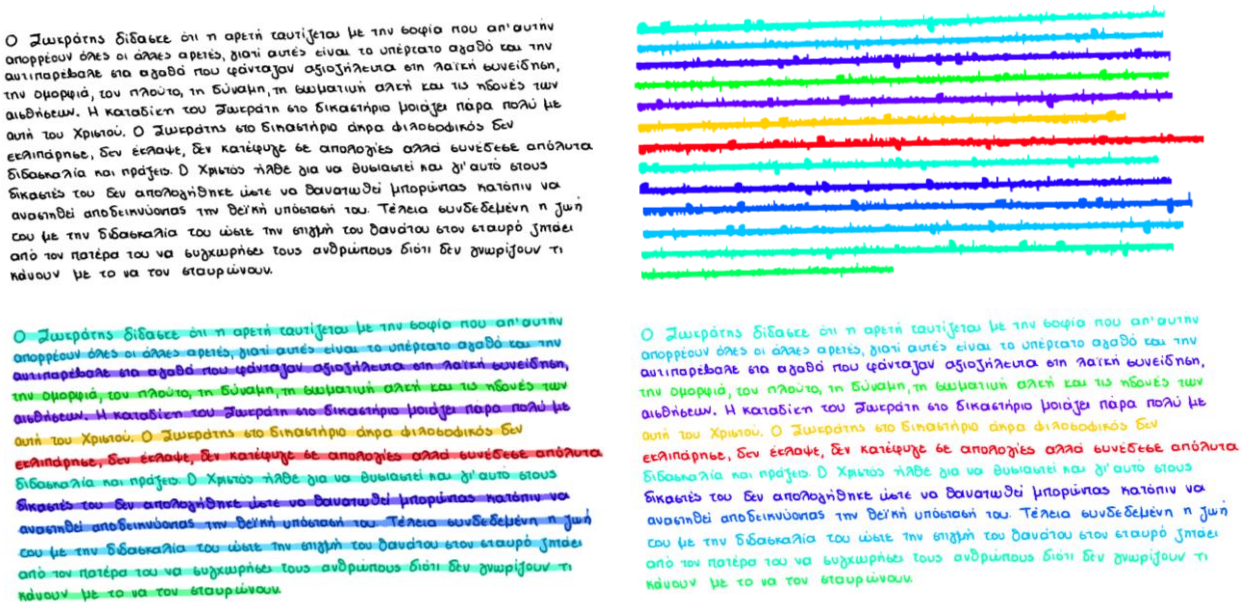


Figura 6-23. Proceso de segmentación. Copia de la figura 6-11

Como ya avanzamos antes de comenzar la sección 6.2.1.1, hay un error común en las segmentaciones de los textos y es que hay ciertos píxeles que no están segmentados.

Los píxeles no segmentados a los que nos referimos son aquellos que tienen su valor a 1 en la imagen original (son negros) pero no tienen valor en la imagen segmentada. Podríamos simplemente pensar que se les podría asignar el valor de la máscara más cercana que tengan, de manera que, si están muy cerca de la línea con valor 7, se les asigne automáticamente este número de línea, ya que es probable que se nos haya pasado incorporarlos pero efectivamente pertenezcan allí. Sin embargo, la tarea no es tan simple porque, al igual que vimos en el Artículo 1 (punto 3.3.1), hay ciertos caracteres que llamamos "conflictivos" cuyo principal problema es que se encuentran muy cerca de una línea, pero realmente no pertenecen a la misma. En dicho artículo, usaban teoría de grafos y LAG para separar el conflicto, pero nosotros hemos optado por una solución más simple, aunque por supuesto menos precisa (ver capítulo 7 – resultados). Veamos los dos casos de píxeles no segmentados que existen:

Figura 6-24. Recreación del final de la línea 24 de la imagen 055 segmentada

Figura 6-25. Recreación de parte de las líneas 10 y 11 de la imagen 032 segmentada

Podemos ver aquí dos ejemplos de lo que acabamos de decir. El error se puede encontrar al principio o al final de una línea (Figura 6-24) de manera que claramente debería de ser segmentada con su CC más cercano, mientras que en el caso de que el error se encuentre de manera vertical (Figura 6-25), la tarea se vuelve más complicada. Para resolver este problema, ideamos un algoritmo de búsqueda que comienza a partir de aquellos píxeles no segmentados y tendría las siguientes características:

1. El algoritmo iría buscando en la máscara un valor numérico cercano distinto a 0 (0 – blanco – fondo) y una vez lo encontrase, se lo asignaría a sí mismo.
2. El píxel debería poder buscar con mayor facilidad en la componente horizontal, ya que, si encuentra un valor en dicha componente, lo más probable es que hubiese quedado suelto al principio o al final de una línea.
3. El algoritmo tendría que tener un límite. Si un pixel no segmentado encontrase el primer valor numérico distinto a 0 a 8000 píxeles de distancia, probablemente no sería correcto asignarlo.
4. El algoritmo tendría que ser iterativo. En la primera iteración, añadiríamos a tantos píxeles como valores se encontraran, pero no actualizaríamos el valor de dichos píxeles hasta que no pasásemos a la siguiente iteración. De esta manera, no sufriríamos el problema de que, por culpa de segmentar un píxel con un valor muy lejano (cercano al límite), segmentásemos erróneamente a todos los píxeles cercanos en una especie de "efecto dominó".

El patrón que seguía el algoritmo de búsqueda acabó teniendo el siguiente aspecto:

0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
0	0	109	107	105	103	101	100	102	104	106	108	110	0	0
0	0	87	85	83	81	79	78	80	82	84	86	88	0	0
0	0	65	63	61	59	57	56	58	60	62	64	66	0	0
0	0	43	41	39	37	35	34	36	38	40	42	44	0	0
0	0	21	19	17	15	13	12	14	16	18	20	22	0	0
0	0	10	8	6	4	2	1	3	5	7	9	11	0	0
0	0	32	30	28	26	24	23	25	27	29	31	33	0	0
0	0	54	52	50	48	46	45	47	49	51	53	55	0	0
0	0	76	74	72	70	68	67	69	71	73	75	77	0	0
0	0	98	96	94	92	90	89	91	93	95	97	99	0	0
0	0	120	118	116	114	112	111	113	115	117	119	121	0	0
0	0	0	0	0	0	0	0	0	0	0	0	0	0	0

Figura 6-26. Patrón del algoritmo de búsqueda de valores para los píxeles no segmentados

Representamos en la figura el orden donde se va a buscar el valor, ubicando el 1 en la posición donde se encuentra el píxel no segmentado. Como podemos observar, empezará buscando de izquierdas a derechas e irá subiendo y bajando poco a poco, inspeccionando antes la componente horizontal que la vertical. Las dimensiones del rectángulo de búsqueda son variables, de manera que podemos ajustarlas hasta dar con aquellas que nos adaptemos lo mejor posible (finalmente acabaron en 200 píxeles de ancho por 60 de alto). Como el proceso es iterativo, debemos prevenir bucles infinitos, que suceden siempre que en una iteración no se asigna el valor a ningún píxel (si en una iteración no hemos incorporado nuevos píxeles a la máscara, en la siguiente tampoco si no cambiamos nada, y se va a generar un bucle infinito). Los resultados no están del todo mal, aunque por supuesto la técnica compleja del LAG que utilizan en el Artículo 1 consigue unos mejores.

7 RESULTADOS Y POSIBLES MEJORAS

Como en todo proyecto que se realice en cualquier ámbito del mundo de la Ciencia, tras realizar los convenientes experimentos hay que analizar los resultados. Por suerte, gracias a que tomamos los datos del concurso del ICDAR 2013 Handwritting Segmentation Contest, podemos realizar una comparación de los algoritmos que participaron en el concurso de aquel año. En el artículo de la referencia [40], podemos encontrar la puntuación que lograron y en esta sección vamos a calcular la que hubiéramos obtenido nosotros.

7.1 Resultados finales del proyecto.

Hemos decidido seguir la misma medida para el rendimiento de nuestro proyecto que la que usaban en las bases del concurso del ICDAR 2013 de donde sacamos el dataset. Estas medidas establecen un cómputo del número de entidades detectadas por el proyecto y el número de entidades del ground truth. Así, para empezar, usan una tabla llamada "MatchScore" que sirve para determinar la similitud que tiene una región del ground truth con una de nuestros resultados. Así, la tabla MatchScore(i,j) se calcula de la siguiente manera:

$$MatchScore\ (i,j) = \frac{T\ (\ G_j\ \cap\ R_i\ \cap\ I)}{T\ (\ (G_j\ \cap\ R_i)\ \cap\ I)}$$

Donde:

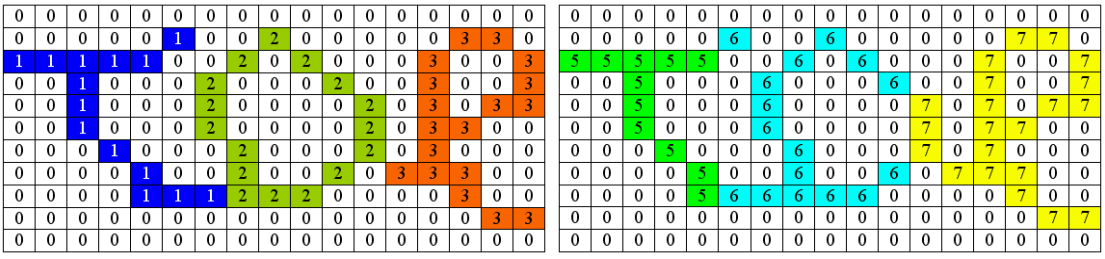
T(x) es una función que cuenta los píxeles que hay en un conjunto "x" (**total**).

G_j son todos los píxeles dentro de una región j del **ground truth**.

R_i son todos los píxeles dentro de una región i de nuestros **resultados**.

I son todos los píxeles de la **imagen**.

Como la explicación es un poco complicada, en la página web del concurso acompañan la ecuación con una imagen de ejemplo para el cálculo:



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

Figura 7-1. Arriba, el ground truth y los resultados de ejemplo, abajo la tabla Matchscore obtenida

Como podemos ver, la tabla MatchScore lo que hace es calcular la similitud de una cierta región del ground truth con una de nuestros resultados. Después, compara estos valores con un umbral, T_a que depende del evaluador, y si alguno de los valores de la tabla está por encima de dicho umbral, se considera que se ha hecho una segmentación correcta de esa zona, o lo que llaman un "one-to-one". En el 2013, el umbral T_a que se escogió fue del 95% para el caso del concurso de las líneas.

Después de calcular los "one-to-one" que hay, se calculan estas dos medidas:

$$DR = \frac{o2o}{N}, \qquad RA = \frac{o2o}{M}$$

Donde:

DR es el **Detection Rate** (tasa de detección). Se expresa en tanto por ciento.

RA es el **Region Accuracy** (precisión de la región). Se expresa en tanto por ciento.

o2o es el número total de **one-to-one** que tenemos.

N es el número total de elementos del ground truth.

M es el número total de elementos de nuestros resultados.

Finalmente, una vez calculamos DR y RA, la métrica que verdaderamente medirá el rendimiento de nuestro algoritmo estará dado por la siguiente fórmula:

$$FM = \frac{2 * DR * RA}{DR + RA}$$

Esta medida, FM (**Final Measurement**), se calcula para cada imagen del dataset y se hace la media entre las 350 imágenes. Según los organizadores del concurso, esta métrica es robusta, ya que se había utilizado con anterioridad, y tan solo depende del T_a.

Por suerte, el programa que venía con los recursos de la competición calcula por nosotros el DR, RA y FM medio de nuestro algoritmo.

En nuestro caso, dicha puntuación fue de: DR = 95,99%, RA = 95,24% y FM = 95,61% como se puede ver en la captura del programa.

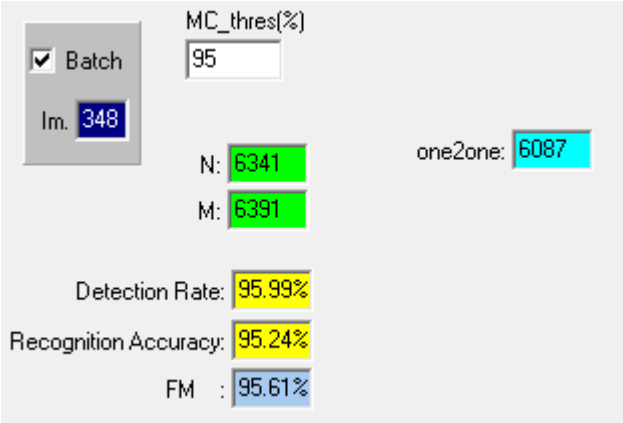


Figura 7-2. Resultados en la captura del programa de la competición.

Aquí dejamos además de algunas capturas de pantalla que muestran la segmentación que se han hecho en algunas imágenes para que las veamos como ejemplo:

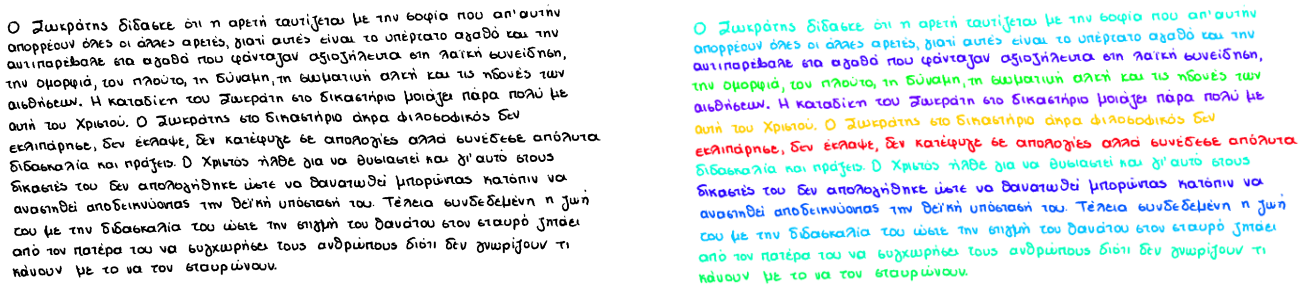


Figura 7-3. Imagen 001 y su segmentación.

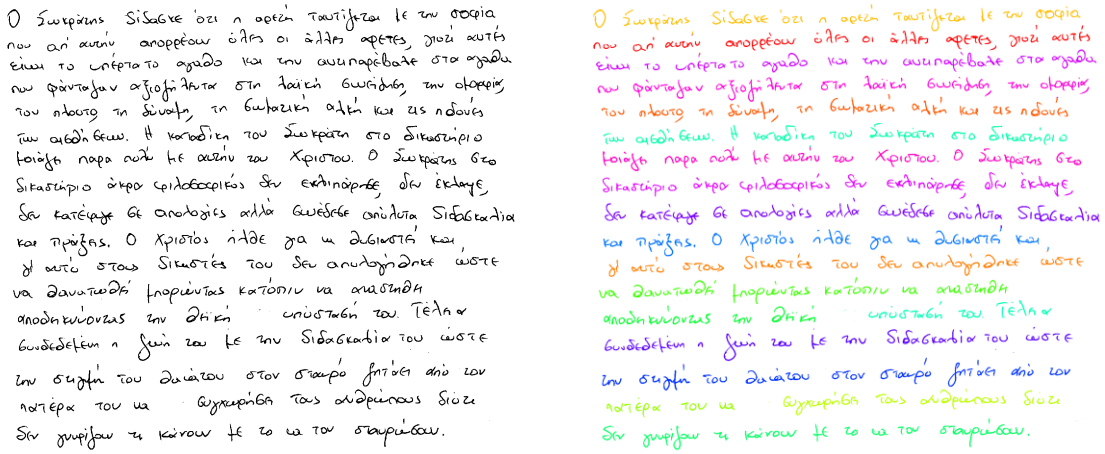


Figura 7-4. Imagen 041 y su segmentación.

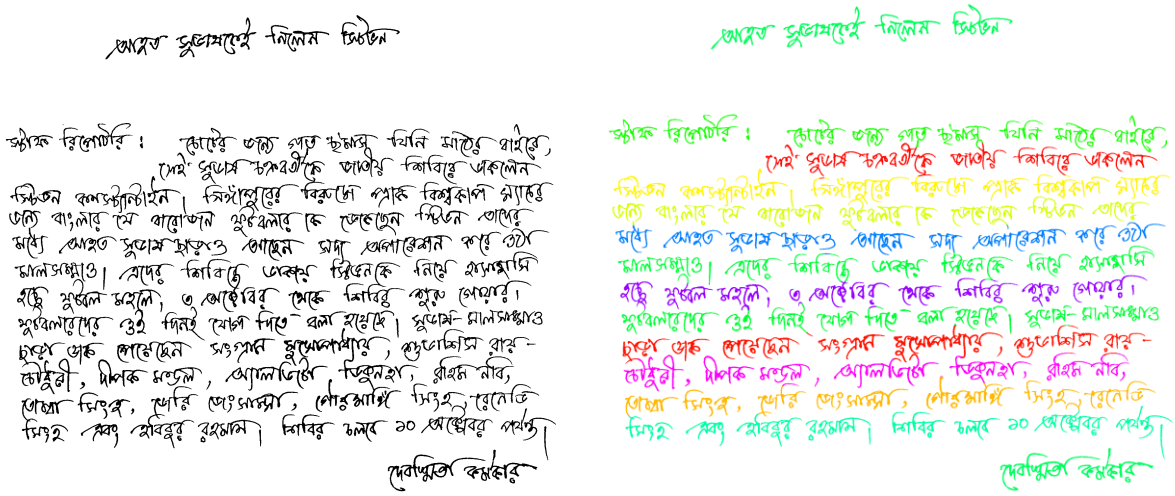


Figura 7-5. Imagen 197 y su segmentación.

Para ver una comparativa con el resto de algoritmos que participaron en el concurso, este es el gráfico que recoge las puntuaciones de los que presentaron sus algoritmos a la sección de segmentación de líneas de texto.

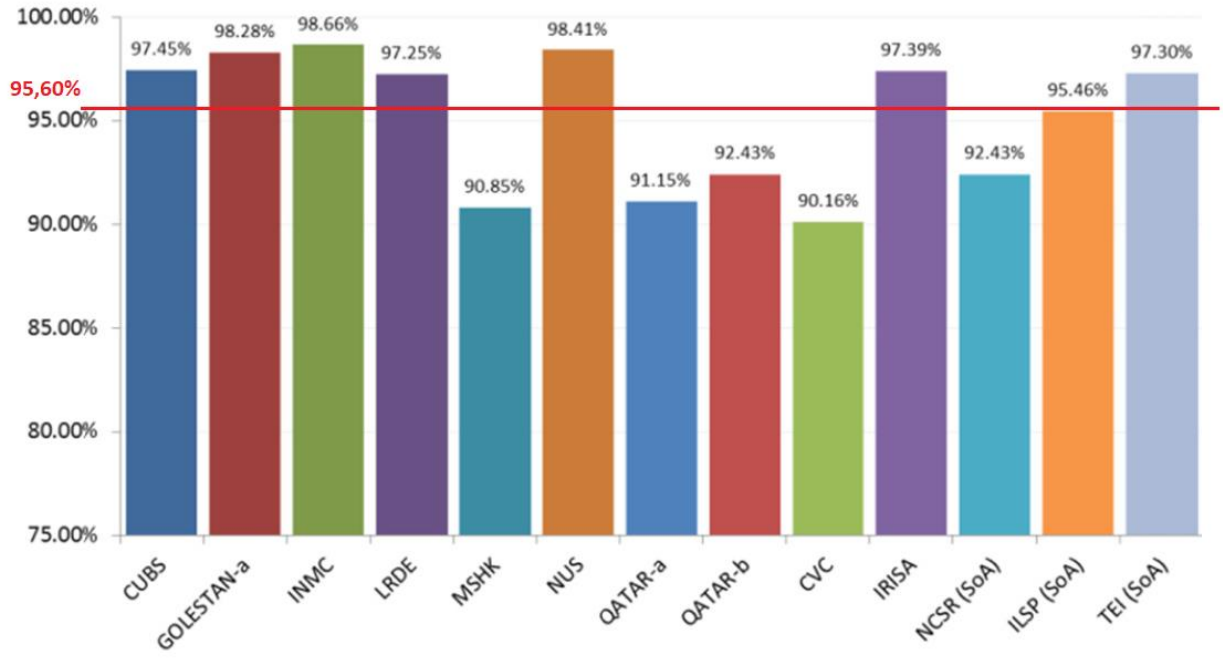


Figura 7-6. Competidores en el concurso de ICDAR 2013 en la sección de líneas de texto.

Como podemos ver, aunque el concurso se realizó en 2013 y hemos usado una tecnología mucho más avanzada que la que había por entonces, nuestra red neuronal hubiese quedado en el octavo puesto, dejando 6 algoritmos por debajo. Por eso, estamos muy contentos con los resultados.

El algoritmo que describían en el Artículo 1 (sección 3.3.1, referencia [2]), que ha sido una gran referencia para nosotros, tiene una puntuación calculada de FM=98.56%, por lo que podemos intuir que su técnica del LAG es más precisa que la nuestra.

7.2 Posibles mejoras para el proyecto

Por supuestísimo, el proyecto tiene ciertas carencias y vamos a dedicar este apartado a aquellas ideas que, por falta de tiempo o preparación, no hemos implementado.

7.2.1 Primera mejora: cambiar las tres redes por una sola

Como hemos hablado a lo largo de toda la memoria y, en especial, en la sección 6.1.2.1, nuestra primera intención no era tener 3 modelos diferentes y realizar un algoritmo que fusionase sus predicciones, sino que lo que queríamos desde un principio era que una única red aprendiera a predecir conjuntamente las partes superior e inferior de los contornos de las líneas de un texto. Sin embargo, cuando hicimos el intento, esto no fue posible y tuvimos que cambiar a la solución que aquí recogemos.

Por ello, quizás una línea de mejora consista en superar este reto y comparar los resultados.

7.2.2 Segunda mejora: máscara más precisa

La máscara que se genera en ocasiones tiene una precisión que deja que desear. Esto se debe a que, en lugar de haber utilizado la tecnología "LAG" como hicieron en el Artículo 1, hemos optado por implementar el concepto de segmentar por un lado las partes de arriba y por otro las de abajo. Con esto hemos hecho un código más sencillo, pero hemos perdido precisión, dejando muchas segmentaciones donde el resultado tiene segmentadas algunas letras que no pertenecen a la línea segmentada. Observemos la "g" de "general" y de "thing" en la siguiente figura:

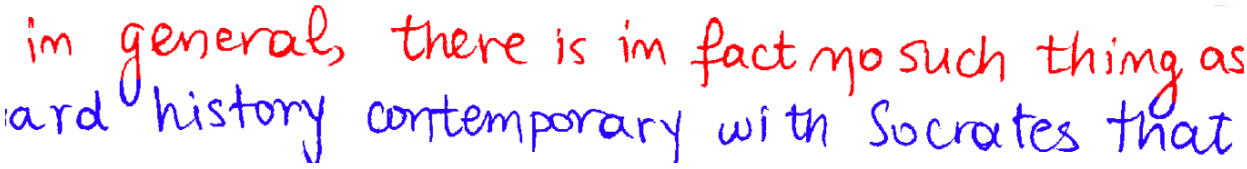


Figura 7-7. Extracto de la imagen 026 segmentada.

7.2.3 Tercera mejora: contemplar el caso "C"

Como bien dijimos, el caso "C" no es detectado ni arreglado por nuestro programa. Esto tiene un peso bastante grande en la puntuación conseguida con las métricas del concurso, ya que no son pocas las imágenes que tienen un pequeño hueco en el centro y la red neuronal detecta la línea en dos partes.

7.2.4 Cuarta mejora: mejorar la corrección de errores

Quizás esta sea la parte que mejor se pueda refinar de todas. Para empezar, el algoritmo de detección de errores no es robusto (aunque sí bastante preciso), ya que la condición para que una imagen sea tratada por dicho algoritmo es que el número de CC no coincida entre las imágenes de mid-midUp-midDown. Esto carece de sentido en ciertos casos como, por ejemplo:

- En una de las imágenes hay un caso "A" (que disminuye en 1 el número de CC) y un caso "B" a la vez (que aumenta en 1 el número de CC). En el caso de que hubiera un número par de errores, la imagen nunca sería procesada y por lo tanto pasarían desapercibidos estos errores.
- En una de las imágenes se ha eliminado un CC cuya área sea menor que la tolerancia establecida, pero en otro no, lo que da lugar a una "falsa alarma". El problema está en que la eliminación de CC pequeños reside únicamente en una medida de sus áreas, pero existen líneas que son muy pequeñas y quizás podríamos mejorar la misma red neuronal para que cuando predijera una línea, acompañara su predicción con un porcentaje que representase la seguridad que tiene la red de haber detectado una línea o no.

Además, la misma corrección de los errores sigue un proceso bastante simple (hallar rectas de regresión, básicamente) y podría ser mejorada. Por ejemplo, un algoritmo más complejo donde tuviera en cuenta la posición donde es más probable encontrar una línea (en su espina dorsal), o utilizar las predicciones superiores e inferiores para saber donde empiezan y acaban las líneas.

7.2.5 Quinta mejora: segmentación de palabras

El concurso tenía dos modalidades: la segmentación de líneas y la segmentación de palabras. Los algoritmos que se presentaron servían en su mayoría para ambas modalidades, mientras que el nuestro solo para la primera.

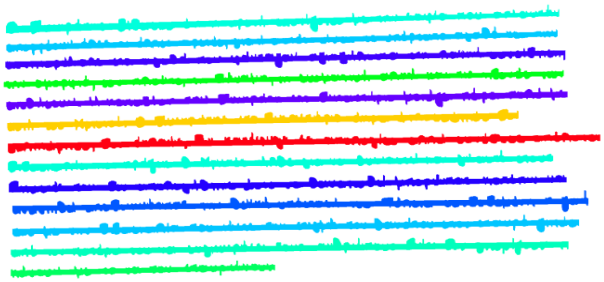
ANEXO A: EL CÓDIGO DE GITHUB

Parte del código que hemos utilizado para nuestro proyecto ha sido subido a un repositorio de GitHub, al que podemos acceder a través de la siguiente URL: https://github.com/jlmendo11/textLinesSegmDL. En este anexo vamos a explicar qué hemos subido y la estructura que tiene.

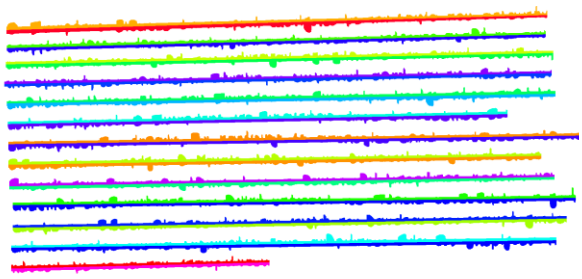
Directorio "datasets"

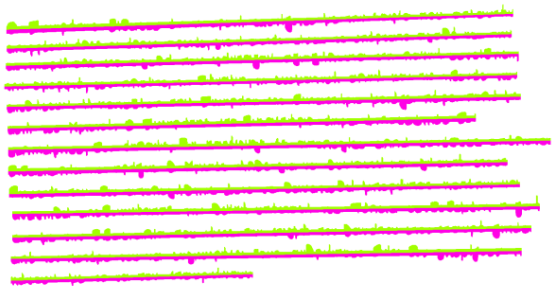
Contiene 3 **versiones** modificadas del dataset en formato comprimido, y son:

- Outline Backbone.

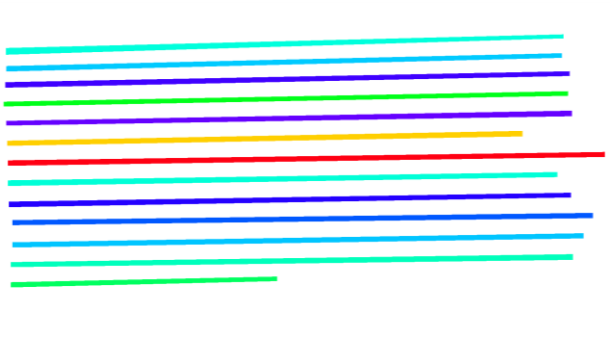


- ZigZag Backbone en versión normal y simplificada.





- Thick Backbone.

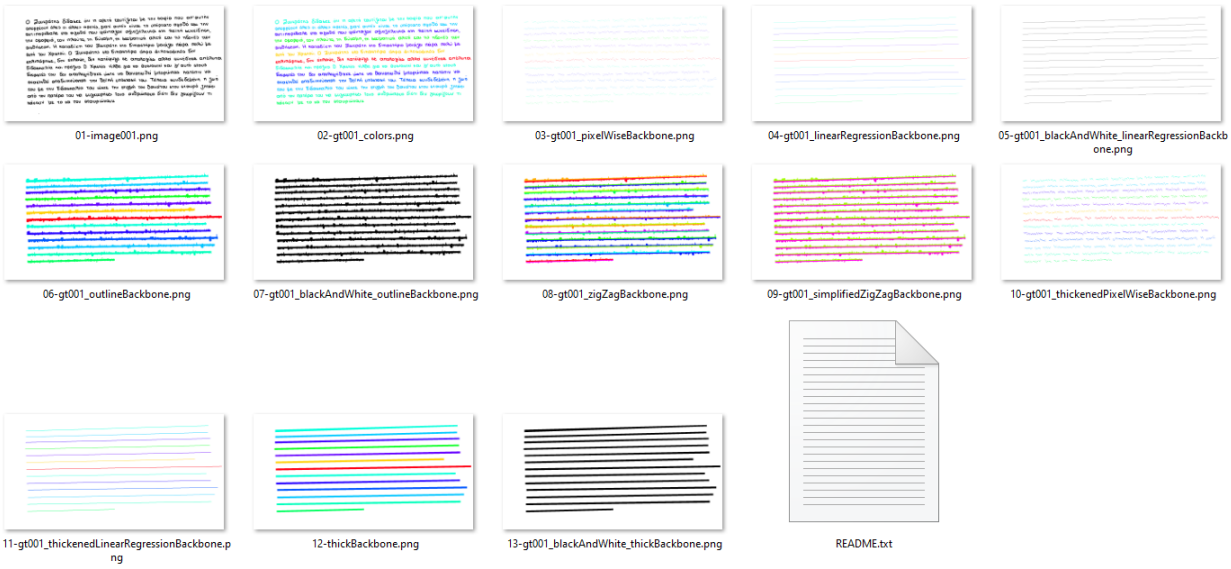


En cada una de las subcarpetas de este directorio podemos encontrar un archivo "README.txt" con una descripción del la modificación y el link de los autores originales. Aquí, vemos los ground-truth modificados, pero no vemos las imágenes originales. Para encontrarlas, tenemos que irnos al sitio original [23].

Directorio "datasetHandling"

Contiene ficheros que serán de utilidad para manipular el dataset. Así, tenemos:

- **datasetHandlingExample.py**: es un fichero donde se puede ver cómo se carga una imagen del dataset y se crean las distintas versiones de la misma. Para la imagen 001, generará un álbum de imágenes como el que podemos ver en la carpeta "**images_version**" y que podemos visualizar en la imágen a continuación:



- **groundTruthFunctions.py**: contiene todas las funciones que realizan la conversión del ground-truth original a cada una de sus versiones. Es utilizado en el fichero anterior.
- **nLinesAndShapeAnalyzer.py**: es el fichero que genera los estadísticos que vimos en las tablas 5-1 y 5-2 de esta memoria. Estas mismas tablas están reflejadas en **nLinesAndShapeAnalysis.txt**.
- **thickBackboneGroundTruthGenerator.py** y **zigZagGroundTruthGenerator.py**: generan las respectivas versiones del dataset. Sus resultados es lo que hemos adjuntado en el directorio "**dataset**"
- **thickBackboneTensorsGenerator.py** y **zigZagTensorsGenerator.py**: generan los tensores con los que luego alimentamos a las redes neuronales.

Directorio "training"

Contiene los ficheros con lo que hemos conseguido el entrenamiento con solo uno de los tres modelos:

- **DataGen_2048_dataAug.py**: contiene el código del *Data Generator* que hemos estado usando para alimentar los modelos de Machine Learning (ver sección 5.1.2). Una versión reducida está en el fichero complementario **DataGen.py**.
- **Unet_2048.py**: la arquitectura de la red que hemos utilizado.
- **CNN_thickBackbone_2048**: el *Jupyter Notebook* para cargarlo en Google Colab junto con los tensores y el resto de ficheros de la carpeta.

Directorio "models"

El modelo resultado de un entrenamiento con la ThickBackbone en un **.json** y sus parámetros en un **.h5**

REFERENCIAS

[1] Géron, Aurélien. 2019. Hands-On Machine Learning with Scikit-Learn, Keras, and TensorFlow, 2nd Edition. 2. ISBN-13: 978-1492032649.

[2] Q. N. Vo, S. H. Kim, H. J. Yang and G. S. Lee, "Text line segmentation using a fully convolutional network in handwritten document images," in IET Image Processing, vol. 12, no. 3, pp. 438-446, 3 2018, doi: 10.1049/iet-ipr.2017.0083. Disponible en https://ieeexplore.ieee.org/document/8302745.

[3] Keras Team. Keras: the Python deep learning API. Keras.io [en línea]. Disponible en: https://keras.io/.

[4] Why does Keras need TensorFlow as backend?. Data Science Stack Exchange. 2020 [en línea]. Disponible en: https://datascience.stackexchange.com/questions/65736/why-does-keras-need-tensorflow-as-backend.

[5] NumPy v1.18 Manual. numpy.org 2020 [en línea]. Disponible en: https://numpy.org/doc/stable/.

[6] Matplotlib v3.2.2 Documentation. matplotlib.org. 2020 [en línea]. Disponible en: https://matplotlib.org/contents.html.

[7] Pillow (PIL Fork) 7.1.2 documentation. Pillow.readthedocs.io. 2020 [en línea]. Disponible en: https://pillow.readthedocs.io/en/stable/.

[8] OpenCV 2.4.8. Opencv.org. 2020 [en línea]. Disponible en: https://opencv.org/opencv-2-4-8/.

[9] Raster graphics. En.wikipedia.org. 2020 [en línea]. Disponible en: https://en.wikipedia.org/wiki/Raster_graphics.

[10] Sinha, U. Convolutions: Image convolution examples - AI Shack. Aishack.in. 2020 [en línea]. Disponible en: https://aishack.in/tutorials/image-convolution-examples/.

[11] Brownlee, J. A Gentle Introduction to Padding and Stride for Convolutional Neural Networks. Machine Learning Mastery. 2020 [en línea]. Disponible en: https://machinelearningmastery.com/padding-and-stride-for-convolutional-neural-networks/.

[12] Ander Fernández. Cómo crear una red neuronal convolucional en Keras. 2020 [en línea]. Disponible en: https://anderfernandez.com/que-es-una-red-neuronal-convolucional-y-como-crearlaen-keras/.

[13] A. W. Harley. "An Interactive Node-Link Visualization of Convolutional Neural Networks". 2020 [en línea]. Disponible en: https://www.cs.ryerson.ca/~aharley/vis/conv/.

[14] REDMON, Joseph, et al. You only look once: Unified, real-time object detection. En Proceedings of the IEEE conference on computer vision and pattern recognition. 2016. p. 779-788. Disponible en: https://arxiv.org/abs/1506.02640.

[15] jantic. DeOldify – GitHub. 2018 [en línea]. Disponible en: https://github.com/jantic/DeOldify.

[16] thunil. TecoGAN – GitHub. 2019 [en línea]. Disponible en: https://github.com/thunil/TecoGAN.

[17] baowenbo. DAIN – GithHub. 2020 [en línea]. Disponible en: https://github.com/baowenbo/DAIN.

[18] GATYS, Leon A., et al. A neural algorithm of artistic style. arXiv 2015. arXiv preprint arXiv:1508.06576, 2015. Disponible en: https://arxiv.org/abs/1508.06576.

[19] ¿Cómo funcionan las Convolutional Neural Networks? Visión por Ordenador. Aprende Machine Learning. 2020 [en línea]. Disponible en: https://www.aprendemachinelearning.com/como-funcionan-las-convolutional-neural-networks-vision-por-ordenador/.

[20] ALBERTI, Michele, et al. Labeling, cutting, grouping: an efficient text line segmentation method for medieval manuscripts. En 2019 International Conference on Document Analysis and Recognition (ICDAR). IEEE, 2019. p. 1200-1206. Disponible en: https://arxiv.org/abs/1906.11894.

[21] QUIRÓS, Lorenzo. Multi-task handwritten document layout analysis. arXiv preprint arXiv:1806.08852, 2018. Disponible en: https://arxiv.org/abs/1806.08852.

[22] Listado de artículos ICDAR. dblp.org. 2020 [en línea]. Disponible en https://dblp.org/db/conf/icdar/.

[23] ICDAR 2013 Handwriting Segmentation Contest - Resources. Users.iit.demokritos.gr. 2013 [en línea]. Disponible en: https://users.iit.demokritos.gr/~nstam/ICDAR2013HandSegmCont/resources.html

[24] BEMMERL, M. 2020. webhex.net - Online Hex Viewer. en.webhex.net [en línea] Disponible en: https://en.webhex.net/.

[25] ICDAR 2013 Handwriting Segmentation Contest - Protocol. Users.iit.demokritos.gr. 2013 [en línea]. Disponible en: https://users.iit.demokritos.gr/~nstam/ICDAR2013HandSegmCont/Protocol.html.

[26] RONNEBERGER, Olaf; FISCHER, Philipp; BROX, Thomas. U-net: Convolutional networks for biomedical image segmentation. En International Conference on Medical image computing and computer-assisted intervention. Springer, Cham, 2015. p. 234-241. Disponible en: https://arxiv.org/abs/1505.04597.

[27] nikhilroxtomar. UNet Segmentation in Keras and TensorFlow – GitHub. 2019 [en línea]. Disponible en: https://github.com/nikhilroxtomar/UNet-Segmentation-in-Keras-TensorFlow.

[28] ISBI Challenge: Segmentation of neuronal structures in EM stacks. Brainiac2.mit.edu. 2012 [en línea]. Disponible en: http://brainiac2.mit.edu/isbi_challenge/.

[29] Ian Goodfellow and Yoshua Bengio and Aaron Courville. Deep Learning. MIT Press - 2016. Disponible en línea en: http://www.deeplearningbook.org/.

[30] BROWNLEE, J. 2019. How to Configure Image Data Augmentation in Keras. Machine Learning Mastery [en línea]. Disponible en: https://machinelearningmastery.com/how-to-configure-image-data-augmentation-when-training-deep-learning-neural-networks/.

[31] BROWNLEE, J. How to Configure Image Data Augmentation in Keras. Machine Learning Mastery. 2019 [en línea]. Disponible en: https://machinelearningmastery.com/how-to-configure-image-data-augmentation-when-training-deep-learning-neural-networks/.

[32] SANTANA VEGA, C. (DotCSV). ¿Por qué las GPUs son buenas para la IA? [en línea]. video. 2019. S.l.: s.n. Disponible en: https://www.youtube.com/watch?v=C_wSHKG8_fg.

[33] HEATON, J. 2020, TensorFlow 2.0 GPU (CUDA), Keras, & Python 3.7 in Windows 10 [en línea]. video. 2020. S.l.: s.n. Disponible en: https://www.youtube.com/watch?v=qrkEYf-YDyI.

[34] Herramienta online para la creación de diagramas de arquitectura de redes neuronales convolucionales. alexlenail.me. [en linea]. Disponible en: http://alexlenail.me/NN-SVG/AlexNet.html.

[35] Arnault. About Convolutional Layer and Convolution Kernel. 2018 [en línea]. Disponible en: https://www.sicara.ai/blog/2019-10-31-convolutional-layer-convolution-kernel.

[36] ANTOGNINI, J. 2017. Why are neural networks becoming deeper, but not wider?. stackexchange [en línea]. Disponible en: https://stats.stackexchange.com/questions/222883/why-are-neural-networks-becoming-deeper-but-not-wider.

[37] BROWNLEE, J. 2018. Use Early Stopping to Halt the Training of Neural Networks At the Right Time. Machine Learning Mastery [en línea]. Disponible en: https://machinelearningmastery.com/how-to-stop-training-deep-neural-networks-at-the-right-time-using-early-stopping/.

[38] scipy.ndimage.label — SciPy v1.4.1 Reference Guide. Docs.scipy.org. [en línea]. Disponible en: https://docs.scipy.org/doc/scipy/reference/generated/scipy.ndimage.label.html.

[39] 3.3.9.8. Labelling connected components of an image — Scipy lecture notes. Scipy-lectures.org [en línea]. Disponible en: https://docs.scipy.org/doc/scipy/reference/generated/scipy.ndimage.label.html.

[40] N. Stamatopoulos, B. Gatos, G. Louloudis, U. Pal and A. Alaei, "ICDAR 2013 Handwriting Segmentation Contest," 2013 12th International Conference on Document Analysis and Recognition, Washington, DC, 2013, pp. 1402-1406, doi: 10.1109/ICDAR.2013.283. Disponible en: https://ieeexplore.ieee.org/document/6628844