

TRABAJO FIN DE GRADO

Detección de Células mediante Deep Learning en Secuencias de Vídeo-Microscopía

Escuela Técnica Superior de Ingeniería de Telecomunicación

Grado de Ingeniería en Sistemas Audiovisuales y Multimedia

Autor: Celia García Fernández

Tutor: Norberto Malpica González

Cotutor: Esteban Pardo Sánchez

Curso académico 2019 / 2020

Agradecimientos

En primer lugar, agradecer a mi tutor, Norberto, por darme esta oportunidad, y guiarme durante el desarrollo de este proyecto de principio a fin. Y a mi cotutor Esteban, de quien he aprendido mucho para iniciarme en el mundo del Deep Learning, y me ha enseñado con paciencia muchos de los conceptos presentados, y necesarios para la ejecución de este trabajo. Gracias por vuestra ayuda.

Este proyecto supone el fin de una larga etapa, llena de retos y obstáculos, pero también acompañada de mucho aprendizaje. Agradecer a mis compañeras con las que he compartido este camino desde el primer día, en especial a Marta, Patricia, Raquel, Sara, Vanesa.

Por supuesto, gracias a mi familia por apoyarme siempre. Especialmente a mi madre, Rosi, por confiar tanto en mí pese a las caídas, y celebrar conmigo cada logro conseguido durante estos años. Agradecerle todo el cariño incondicional y la energía que me transmite cada día y que me ha ayudado tanto a superar todos los retos.

Por otro lado, agradecer también a mi pareja, Adrián, por estar siempre tan orgulloso de mí, y contagiarme su energía tan positiva y necesaria cada vez que lo he necesitado.

Por último, gracias a mis compañeros del Laboratorio, que en los últimos meses me han empujado a conseguir esta meta, animándome y aguantándome en el último tirón.

Muchas gracias a todos.

Resumen

La capacidad de las células para ejercer fuerzas sobre su entorno y alterar su forma a medida que se mueven, es esencial para diversos procesos biológicos. Por ello, el análisis del movimiento celular mediante el microscopio es una herramienta clave para la investigación y el diagnóstico, que permite obtener medias cuantitativas de este proceso. El problema es que la segmentación y el seguimiento manual de células en movimiento en secuencias de vídeo es una tarea muy laboriosa, por lo que se hace necesario el desarrollo de técnicas automatizadas de análisis.

La mayoría de las técnicas estándar de segmentación y seguimiento no funcionan bien en condiciones de baja calidad de imagen. En imágenes de células vivas, para las que se utiliza a menudo microscopía de fluorescencia, la relación señal/ruido puede ser extremadamente baja, lo que hace que la detección automática de células sea una tarea aún más complicada, a la par que muy importante en muchas aplicaciones biomédicas.

El objetivo de este proyecto es implementar un sistema de segmentación para un conjunto de imágenes de microscopía celular proporcionado por Cell Tracking Challenge, una iniciativa dirigida a promover el desarrollo y la evaluación objetiva de segmentación celular y algoritmos de seguimiento.

Tras realizar una revisión de los últimos avances en Deep Learning (DL), segmentación de imágenes, Redes Neuronales Convolucionales y segmentación en imágenes biomédicas, se presenta el sistema implementado, que consiste en una RNC tipo encoder-decoder U-Net. Se presentan después los resultados y experimentos realizados sobre el conjunto de imágenes. Por último, el trabajo concluye con una revisión del trabajo realizado, las conclusiones obtenidas y futuras direcciones de investigación.

Summary

The ability of cells to exert forces on their environment and alter their shape as they move is essential for various biological processes. For this reason, the analysis of cell movement through the microscope is a key tool for research and diagnosis, which allows obtaining quantitative means of this process. The problem is that the manual segmentation and tracking of moving cells in video sequences is a very challenging task, which is why the development of automated analysis techniques is necessary.

Most of the standard segmentation and tracking methods do not work well under poor image quality conditions. In live cell imaging, for which fluorescence microscopy is often used, the signal-to-noise ratio can be extremely low, making automatic cell detection an even more complicated task, as well as very important in many biomedical applications.

The objective of this project is to implement a segmentation method for a set of cell microscopy images provided by the Cell Tracking Challenge, an ongoing initiative aimed at promoting the development and objective evaluation of cell segmentation and tracking algorithms.

After reviewing the current State Of The Art about Deep Learning (DL), image segmentation, Convolutional Neural Networks and segmentation in biomedical images, the implemented system is presented. It consists of a UNet CNN, an encoder-decoder network. The results and experiments carried out on the set images are then presented. Finally, the work concludes with a review of the work, the conclusions obtained and future research lines.

Índice general

Agradecimientos	3
Resumen	5
Summary	7
Índice de Figuras	11
Introducción	15
1.1. Contexto y Motivación	15
1.2. Objetivos	16
1.3. Estructura de la memoria	17
Estado del Arte	18
2.1 Introducción	18
2.2 Redes Neuronales Convolucionales (RNCs)	23
2.3 Segmentación de Imágenes con RNC	26
Infraestructura	29
3.1. Librerías y Frameworks	29
3.1.1. TensorFlow	29
3.1.2. Keras	31
3.1.3. Otros paquetes	38
3.2. Conjunto de Imágenes	39
Segmentación de Células con Deep Learning	42
4.1. Preprocesamiento de imágenes	43
4.2. Arquitectura del Modelo UNet	48
4.2.1. Capas	49
4.2.2. Estructura	54
4.3. Compilación del Modelo	57
4.3.1. Función de Pérdida	57
4.3.2. Optimizador	58
4.3.3. Métrica	59
4.4. Entrenamiento del Modelo	60
4.4.1. Hiper parámetros	61

4.4.2.	Subconjuntos de Datos	62
4.4.3.	Aumento de Datos	64
4.5.	Validación Cruzada	67
4.6.	Evaluación del Modelo	68
Resultados		69
5.1	Experimentos y evaluaciones	69
5.2	Pruebas con imágenes del Challenge	77
Conclusiones		79
6.1	Conclusiones	79
6.2	Líneas Futuras	80
Bibliografía		81

Índice de Figuras

Figura 1. Comparación de neurona biológica y su modelo artificial. Imagen obtenida de [7].	20
Figura 2. Arquitectura de una Red Neuronal Artificial básica.....	20
Figura 3. Comparación de red neuronal simple y red neuronal profunda.....	22
Figura 4. Ejemplo de una arquitectura de RNC.	23
Figura 5. Comparativa entre Red Neuronal tradicional y Red Neuronal Convolutacional. Imagen obtenida de [13].	24
Figura 6. Ejemplos de arquitecturas de RNC Fully Connected Layers [15] (a) y Fully Convolutional Network, SegNet [16] (b).	26
Figura 7. (a) Ejemplo de tensores. Formas estándar de representar datos en TensorFlow con Tensores. (b) Diferencia entre tensor y flujo. Imágenes obtenidas de [24].	30
Figura 8. Ejemplos de convolución 2D con relleno (izquierda) y convolución traspuesta 2D con relleno y strides (derecha). Imagen obtenida de [28].	35
Figura 9. Ejemplo de operación MaxPooling2D con pool_size (2,2).....	36
Figura 10. Dos muestras del conjunto de datos escogido para el proyecto. Fluo-C2DL-MSC	40
Figura 11 . 28 muestras [02] y sus correspondientes 28 anotaciones Estas últimas con los valores corregidos para poder visualizar las etiquetas en esta figura.	41
Figura 12. Diagrama de Flujo de la tarea de segmentación desarrollada.	43
Figura 13. 18 anotaciones para las 48 imágenes de la secuencia de vídeo 01.....	44
Figura 14. Imágenes y sus correspondientes anotaciones a su izquierda. Muestras leídas, sin pre procesar. Se adjuntan con su histograma de valores correspondiente.	44
Figura 15. Dos muestras de anotaciones después convertirlas en binarias, con sus histogramas de valores correspondientes.	45
Figura 16. Imagen reescalada a la derecha y su anotación reescalada correspondiente a la izquierda. Ambas con sus respectivos histogramas de valores.....	45

Figura 17. Ejemplo visual del resultado de categorizar las anotaciones.....	46
Figura 18. Función de distribución normal.....	47
Figura 19. Comparación de una muestra antes y después de la normalización, con su histograma de valores correspondiente.	47
Figura 20. Arquitectura UNet original. Imagen obtenida de [35]	48
Figura 21. Arquitectura del modelo UNet Keras implementado en este proyecto, presentada con la función de Keras <code>model.summary()</code>	50
Figura 22. Estructura de capa convolucional con varios mapas de características. Imagen obtenida de [36].....	51
Figura 23. Comparativa de las tres funciones de activación más populares.....	52
Figura 24. Comparativa de las funciones de activación Sigmoid y Softmax. Imagen obtenida de [38]	53
Figura 25. Ejemplo de operación MaxPooling	53
Figura 26. Definición de función de pérdida de entropía cruzada categórica.....	58
Figura 27. Rango de posibles valores de pérdida de entropía cruzada, dada una observación real. Imagen obtenida de [41].	58
Figura 28. Definición de función de métrica Coeficiente Dice.....	59
Figura 29. Curva de entrenamiento que muestra cómo se ajusta el modelo a los datos de entrenamiento.....	62
Figura 30. Ejemplo de subdivisión de un conjunto de 5 muestras.....	63
Figura 31. Ejemplos de algunas muestras generadas con Keras en el conjunto de datos utilizado.	65
Figura 32. Ejemplo de muestras demasiado deformadas, generadas con Aumento de Datos. Arriba las imágenes de células, abajo sus respectivas anotaciones.....	65
Figura 33. Representación de Validación Cruzada K-Fold.....	68
Figura 34. Función de pérdida (abajo) y métrica (arriba) para los datos de entrenamiento y validación. 100 epochs y sin aumento de datos (Azul: datos de entrenamiento, Rojo: datos de validación).....	70

Figura 35. Función de pérdida (abajo) y métrica (arriba) para los datos de entrenamiento y validación. 500 epochs y sin aumento de datos (Rojo: datos de entrenamiento, Azul: datos de validación).....	70
Figura 36. Función de pérdida (abajo) y métrica (arriba) para los datos de entrenamiento y validación. 700 epochs y sin aumento de datos (Rosa: datos de entrenamiento, Verde: datos de validación).....	71
Figura 37. Función de pérdida (abajo) y métrica (arriba) para los datos de validación. 100 epochs (Rojo), 500 epochs (Azul) y 700 epochs (Verde), sin aumento de datos.	71
Figura 38. Media de los resultados en las imágenes de prueba (test), utilizando validación cruzada.....	72
Figura 39. Varias Imágenes de segmentación predichas por los modelos entrenados con distintos epochs, comparadas con las anotaciones originales.....	72
Figura 40. Ejemplos de muestras generadas por el generados de datos de Keras. Arriba las imágenes deformadas y con niveles de gris modificados. Abajo sus respectivas anotaciones con la misma deformación aplicada.	73
Figura 41. Función de pérdida (abajo) y métrica (arriba) para los datos de entrenamiento y validación. 500 epochs con aumento de datos (Azul: datos de entrenamiento, Rojo: datos de validación)	74
Figura 42. Función de pérdida (abajo) y métrica (arriba) para los datos de entrenamiento y validación. 700 epochs con aumento de datos (Rosa: datos de entrenamiento, Verde: datos de validación).....	74
Figura 43. Función de pérdida (abajo) y métrica (arriba) para los datos de validación. 100 epochs (Azul oscuro), 500 epochs (Azul claro) y 700 epochs (Verde), con aumento de datos.	75
Figura 44. Función de pérdida (abajo) y métrica (arriba) para los datos de validación y entrenamiento. Aumento de datos (Rojo: datos de entrenamiento, Azul claro: Datos de validación) y sin aumento de datos (Azul claro: datos entrenamiento, Rosa: datos de validación) con 500 epochs.....	75
Figura 45. Función de pérdida (abajo) y métrica (arriba) para los datos de validación y entrenamiento. Aumento de datos (Azul: datos de entrenamiento, Rojo: Datos de validación)	

sin aumento de datos (Rosa: datos entrenamiento, Verde: datos de validación), con 700 epochs.....	76
Figura 46. Tabla con la media de los resultados en las imágenes de prueba (test), utilizando validación cruzada. Se comparan los modelos con aumento de datos y sin aumentos de datos, y según el número de epochs.	76
Figura 47. Varias Imágenes de segmentación predichas por los modelos entrenados con aumento de datos. Se compara con las anotaciones originales.	77
Figura 48. Resultado de las predicciones para dos muestras de la secuencia de vídeo de prueba 2.	78
Figura 49. Resultado de las predicciones para dos muestras de la secuencia de vídeo de prueba 1.	78

Capítulo 1

Introducción

La migración celular es un proceso esencial en el desarrollo normal de los tejidos, la reparación de tejidos y las enfermedades [1]. Su comprensión depende de una cuantificación precisa de la dinámica del movimiento celular y los cambios morfológicos que experimentan las células durante el movimiento, pero este proceso manual es una tarea muy laboriosa, por lo que cada vez depende más de técnicas automatizadas.

De aquí nace el objetivo general del proyecto, que consiste en desarrollar una técnica de segmentación de células en imágenes de vídeo microscopía (secuencias de vídeo time lapse en 2D), utilizando aprendizaje profundo a partir de una arquitectura de red neuronal convolucional, utilizando el repositorio de datos que ofrece Cell Tracking Challenge.

En este capítulo se situará el proyecto en su contexto general y se expondrán los objetivos concretos, además de un pequeño resumen de cómo se ha planificado y estructurado el trabajo.

1.1. Contexto y Motivación

La segmentación morfológica es una de las tareas básicas dentro de la imagen médica, que consume una gran parte del tiempo de un estudio clínico y además requiere de un experto para que se pueda llevar a cabo. Un software capaz de segmentar con éxito y rápidamente grandes cantidades de datos aumentaría la robustez y la calidad de los estudios y las investigaciones médicas.

La caracterización del movimiento de las células, mientras interactúan con su entorno es clave en la investigación biomédica y comprender la mecanobiología de la migración celular y sus múltiples implicaciones, tanto en el desarrollo normal de los tejidos como en muchas enfermedades [2] [3]. Segmentar y rastrear manualmente células en movimiento en secuencias de vídeo es una tarea muy laboriosa. Por lo tanto, el análisis de los experimentos de vídeo time-lapse depende cada vez más de las técnicas de procesamiento de imágenes automatizadas. La mayoría de las técnicas estándar de segmentación y seguimiento no

funcionan bien en condiciones de baja calidad (alta densidad celular, tinción no homogénea, cambios de linaje). Por lo que recientemente se han desarrollado nuevos algoritmos más robustos, convirtiéndose en la principal motivación de este trabajo.

En 2013 se lanzó la primera edición de Cell Tracking Challenge, como parte del Simposio Internacional de Imágenes Biomédicas (ISBI) del IEEE en San Francisco, para comparar y evaluar objetivamente métodos de segmentación y seguimiento de núcleos y células enteras de última generación utilizando videos de microscopía de time lapse reales (2D y 3D) de células y núcleos, junto con videos generados por computadora (2D y 3D) que simulan células y núcleos completos que se mueven en entornos realistas.

El potencial y el protagonismo que está tomando el Deep Learning para resolver problemas, especialmente en el campo de la Visión Artificial, nos presenta una oportunidad para trasladar esta tecnología al problema planteado.

1.2. Objetivos

El objetivo general es profundizar en técnicas de aprendizaje profundo y redes neuronales convolucionales, para desarrollar una herramienta capaz de segmentar células en imágenes de vídeo microscopía y alcanzar buenos resultados, teniendo de referencia los resultados de los diversos métodos publicados por los participantes de Cell Tracking Challenge.

Para lograr este objetivo general, el proyecto aborda subobjetivos más específicos:

- Análisis de las herramientas disponibles para trabajar con redes neuronales convolucionales, en concreto TensorFlow y Keras.
- Estudiar e investigar las redes neuronales convolucionales existentes, en concreto U-Net.
- Desarrollar, implementar y entrenar un modelo de red neuronal convolucional a partir de la estudiada en el punto anterior.
- Estudiar la posibilidad de introducir mejoras en la red implementada y su entrenamiento.
- Documentar, analizar y comparar el rendimiento de la arquitectura de red implementada, para su posterior conclusión.

1.3. Estructura de la memoria

Todos los hitos y el trabajo realizado quedan registrados en esta memoria con la estructura que se muestra a continuación:

Capítulo 1, Introducción. Introducción, objetivos y planificación. Sitúa el trabajo en su marco general, se establece la finalidad del mismo y las metas que se pretenden alcanzar.

Capítulo 2, Estado del Arte. Antecedentes bibliográficos de la inteligencia artificial y las redes neuronales convolucionales. Se realiza un estudio bibliográfico sobre las redes neuronales, especialmente en las redes neuronales convolucionales y en la aplicación de estas para la segmentación de imagen médica.

Capítulo 3, Infraestructura. Descripción de los materiales y métodos utilizados durante el desarrollo del proyecto. El software y hardware utilizado, y los conjuntos de datos empleados para el entrenamiento y evaluación de la red neuronal.

Capítulo 4, Segmentación de Células con Deep Learning. Descripción detallada de los pasos y las técnicas empleadas para abordar el desarrollo de la herramienta software que resuelve la segmentación de células.

Capítulo 5, Experimentos y Resultados. Se detallan los experimentos realizados durante el desarrollo del proyecto, que demuestran las técnicas empleadas para conseguir una red robusta. Resultados obtenidos con sus discusiones correspondientes.

Capítulo 6, Conclusiones. Argumentos obtenidos tras la consecución del proyecto, y posible plan de actuación futuro.

Capítulo 2

Estado del Arte

Para comprender el contexto en el que se ha desarrollado este proyecto, en este capítulo se describen los campos que han llevado al nacimiento de las redes neuronales convolucionales y cómo están relacionados entre sí. Profundizando en este tipo de redes neuronales, especialmente en su aplicación en el campo de la segmentación de imágenes médicas.

2.1 Introducción

Inteligencia Artificial (IA)

Desde que el término Inteligencia Artificial (IA) fue creado en la Conferencia de Dartmouth en 1956 [4] , este campo de la informática se ha desarrollado muy rápido.

La Inteligencia Artificial [5] es la inteligencia llevada a cabo por máquinas, es decir, la creación de algoritmos con el propósito de crear máquinas que presenten las mismas capacidades que el ser humano, como percibir el entorno y actuar en consecuencia. Por lo tanto, el ser humano ha logrado que las máquinas realicen tareas propias de las personas como automatizar el trabajo de rutina, entender el habla o las imágenes, detección de spam, hacer diagnósticos médicos, etc.

Machine Learning (ML)

El Aprendizaje Máquina o Aprendizaje Automático (Machine Learning, ML) [6] es una rama de la IA. Se trata de la ciencia de la programación de computadoras basada en la idea de que las máquinas pueden aprender de datos, identificar patrones y tomar decisiones con mínima intervención humana, sin ser programadas.

Según una cita atribuida a Arthur Samuel, es el "campo de estudio que le da a los ordenadores la capacidad de aprender sin estar explícitamente programado" [7] . Dada esta definición, se puede afirmar que Machine Learning es un subcampo de IA, porque los ordenadores capaces de aprender contienen un comportamiento inteligente, pero no toda la IA es ML porque mostrar un comportamiento inteligente no significa necesariamente aprender.

Ha existido durante décadas en algunas aplicaciones especializadas, como el reconocimiento óptico de caracteres, pero la primera aplicación de ML que realmente se convirtió en la corriente principal fue en la década de 1990: el filtro de spam.

Las tecnologías de aprendizaje automático han impulsado muchos aspectos de nuestra vida diaria durante años, como filtrado de contenido, búsqueda web, recomendaciones personalizadas, identificación de objetos en imágenes, etc.

Los algoritmos de ML se pueden dividir en dos grupos principales:

- Aprendizaje supervisado [8] : Los datos de entrenamiento que alimentan al algoritmo incluyen las soluciones/salidas deseadas, llamadas etiquetas. Es decir, hay supervisión humana.

Algunos de los algoritmos de aprendizaje supervisado más importantes son: Regresión logística, árboles de decisión, SVM, redes neuronales, etc.

A pesar de que la comunidad científica mostró un gran interés por las redes neuronales, éstas no fueron muy utilizadas debido a la complejidad y el alto coste computacional que presentaban. Por ello, otros modelos de ML más simples, como SVM o árboles de decisión, han tenido un mayor éxito en la clasificación de datos, aunque no obtienen un buen resultado en la clasificación de imágenes.

- Aprendizaje no supervisado: Los datos de entrenamiento no están etiquetados, es decir, las entradas se proporcionan sin sus salidas, no hay supervisión humana. El objetivo de estos algoritmos es encontrar una función que describa alguna estructura subyacente en los datos.

Redes Neuronales Artificiales (RNA)

Uno de los algoritmos de ML son las Redes Neuronales Artificiales (RNA) [5] . Se trata de un modelo de inteligencia artificial, que trata de imitar computacionalmente la red neuronal biológica, el comportamiento de las neuronas en el cerebro del ser humano. Se fundamentan en la utilización de una serie de nodos, llamados neuronas artificiales, conectados con otros nodos de diferente capa. Cada neurona tiene varias entradas y una salida.

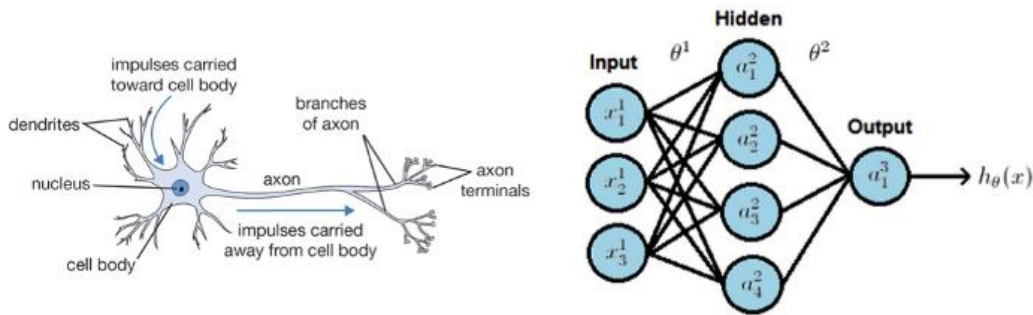


Figura 1. Comparación de neurona biológica y su modelo artificial. Imagen obtenida de [7].

En cada conexión entre neuronas artificiales se transmite información de forma unidireccional: la suma ponderada de señales de entrada aplicando una función de activación, lineal o no lineal (ReLU, Sigmoidal ...). La neurona receptora recibe las señales de una o más neuronas y las procesa para enviar una nueva señal procesada a las neuronas que están en una capa neuronal superior. Por lo tanto, se define una RNA como la conexión de neuronas de diferentes formas, dando lugar a un conjunto de capas interconectadas, que producen unos valores de salida. Este proyecto se centra en el tipo de red multicapa, que se compone de dos o más capas de neuronas, realizando un procesamiento no lineal de los datos que se presentan a la entrada.

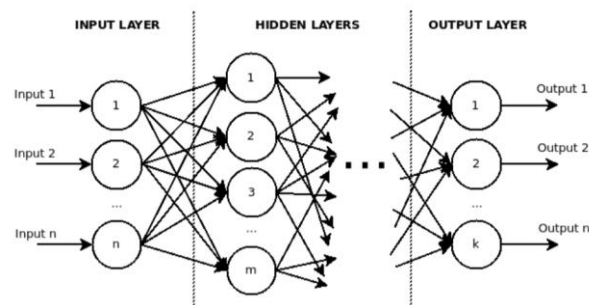


Figura 2. Arquitectura de una Red Neuronal Artificial básica.

Los problemas que frenaron el uso de redes neuronales artificiales, como la complejidad o el alto coste computacional, fueron solventados gracias tanto a una mejora en el hardware (utilización GPU, dada la posibilidad de poder llevar a cabo multi paralelización) como a una optimización de los algoritmos (uso de modelos neuronales pre entrenados, que han podido disminuir el tiempo de entrenamiento y el número de información necesaria para llevar a cabo el entrenamiento).

Existen muchos tipos de redes neuronales artificiales, pero la más exitosa hoy en día, gracias al Deep Learning, es la **Red Neuronal Convolutacional**, más comúnmente aplicada al análisis de imágenes, y en la que está basada este proyecto.

Deep Learning (DL)

Como ya se ha explicado, el ML se ocupa de dotar a las máquinas de capacidad de aprendizaje a través de la experiencia, sin programar reglas para las infinitas variables posibles que pueden aparecer en el mundo real [9] . Pero este aprendizaje automático tradicional estaba limitado, no pudiendo recibir datos sin procesar ni transformarlos en una representación adecuada sin intervención humana. Se necesitaba conocimiento experto usando características diseñadas a mano.

Impulsado por esta limitación y gracias al aumento de la potencia de cálculo y la gran abundancia de datos disponibles, ha brillado el Aprendizaje Profundo o Deep Learning (DL) [10] , logrando la capacidad de extraer patrones a partir de datos en bruto complejos.

Dentro del ML, y centrándonos en las redes neuronales apareció el concepto de DL, que arrancó principalmente en 2006 con Geoffrey Hinton y su equipo [11]. Se trata del enfoque actual del Estado del Arte con el que se explican nuevas arquitecturas de redes neuronales profundas que permiten a las computadoras ‘ver’ y distinguir objetos y texto en imágenes y videos.

Las redes neuronales artificiales aprenden de forma jerarquizada, por niveles de capas. Cuantas más capas tengamos, más complejos resultan los algoritmos de procesamiento. Este incremento en el número de capas y su complejidad es lo que hace que estos algoritmos sean conocidos como algoritmos de Deep Learning. Estas capas varían dependiendo de la función que se quiera realizar. Por ejemplo, en las tareas de clasificación, las capas de alto nivel amplifican los aspectos relevantes y descartan las variaciones menos importantes. En las imágenes, las primeras capas comienzan detectando bordes en ubicaciones particulares, las segundas detectan bordes independientemente de su ubicación, las terceras ensamblan estos bordes en combinaciones más grandes, y así sucesivamente.

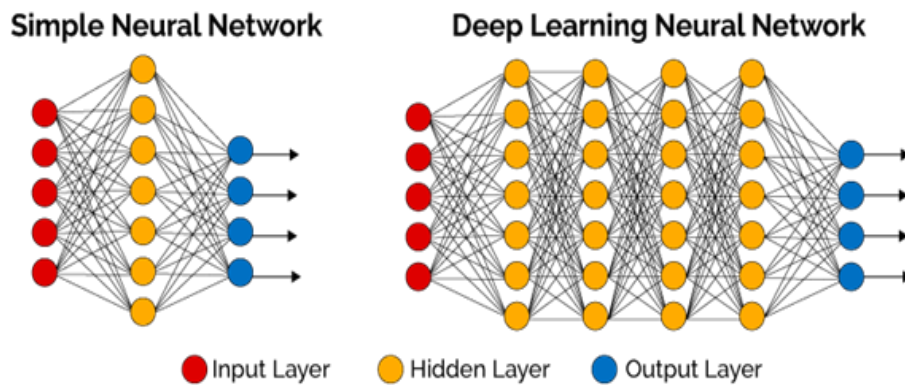


Figura 3. Comparación de red neuronal simple y red neuronal profunda.

Pero la clave de este sistema es que los pesos de estas capas de características se aprenden de los datos. Gracias a esto, el DL ha impulsado el avance en la computación, resolviendo problemas complejos a los que se resisten las técnicas tradicionales de Inteligencia Artificial.

El DL permite obtener el conocimiento de la máquina sin necesidad de realizar una extracción de características previa para representar los datos de entrada, sino que la propia red aprende automáticamente las funciones necesarias para realizar determinadas tareas.

Sus campos de aplicación se han expandido a muchos dominios de la ciencia y los negocios. Un campo particular que se ha beneficiado de este enfoque es la Visión Artificial. DL ha establecido nuevos estándares en campos como el procesamiento de imágenes médicas y biológicas, gracias a los avances en una de sus técnicas específicas, las redes neuronales convolucionales (RNC), que pretenden simular el funcionamiento del cerebro humano para establecer conclusiones sobre los datos introducidos a la red utilizando filtros convolucionales.

Visión Artificial (VA)

La IA abarca varios campos, entre ellos la Visión Artificial (VA) o Computer Vision, [5] cuyo propósito es que la máquina “entienda” una escena o las características de una imagen. Es decir, se trata de trasladar a una máquina la forma en que el ser humano usa sus ojos y su cerebro para comprender el mundo que le rodea, permitiendo tomar decisiones y actuar según la situación. Para ello, la aplicación de VA debe lidiar con la adquisición, procesamiento y análisis de imágenes, como veremos en este proyecto.

La VA se ha beneficiado de una gran evolución gracias al Deep Learning. Para ver la diferencia entre un sistema de VA tradicional y un sistema basado en ML y DL, pondremos como ejemplo la lectura de caracteres. El software de VA tradicional requiere una serie de bibliotecas de patrones y plantillas que contienen todos los posibles caracteres que se pueden reconocer en una foto, como la letra 'A'. Pero los algoritmos se vuelven poco eficientes cuando crecen las bibliotecas de excepciones y defectos. El sistema de VA con DL es capaz de leer una 'A' aunque en la imagen se encuentre doblada, medio borrada o con ruido y solo pueda verse un 30%. Aun así, será capaz de conceptualizar y generalizar la apariencia de los caracteres basado en sus características distintivas. Reconocerá correctamente la letra o el número, gracias a las capas de algoritmos integrados en su red neuronal que le permite sacar sus propias conclusiones.

Por lo tanto, disponer de un software de análisis de imagen basado en Deep Learning nos ofrece soluciones a desafíos complejos de VA, que anteriormente no podían realizarse con VA tradicional o que requerían una cantidad de recursos y dispositivos que no hacían viable la solución.

En este proyecto se demostrará cómo el sistema desarrollado con DL, basado en una red neuronal convolucional, es capaz de identificar y localizar dentro de la imagen un tipo de células, a base de aprender por sí mismo con una pequeña cantidad de imágenes.

2.2 Redes Neuronales Convolucionales (RNCs)

La técnica más común dentro del DL es Convolutional Neural Network (CNN) o Red Neuronal Convolucional (RNC) [12].

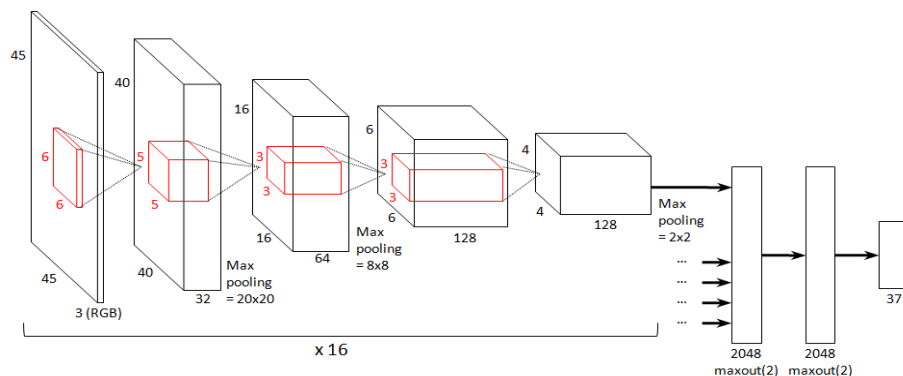


Figura 4. Ejemplo de una arquitectura de RNC.

Entre los diferentes tipos de redes neuronales, esta ha sido la que más éxito ha tenido especialmente en tareas de Visión Artificial, como en la clasificación y segmentación de imágenes, debido a que su aplicación es realizada en matrices bidimensionales, es decir, están diseñadas específicamente para procesar datos que tienen una topología similar a una cuadrícula, como imágenes y audio (detección de objetos o procesamiento del lenguaje natural).

La excelencia de este tipo de redes está en la reducción del uso computacional (campos receptivos locales) gracias al uso de la convolución, a diferencia de las redes neuronales artificiales tradicionales que usan la multiplicación matricial. Las RNA tradicionales (Fully Connected Neuronal Network) conectan cada elemento individual de la entrada con cada elemento oculto, generando enormes cantidades de parámetros sin ningún tipo de conciencia espacial, por lo que se hacen ineficientes para el procesamiento de imágenes. Sin embargo, en las RNC cada elemento/neurona de la capa oculta se asigna a un área específica de la entrada, con un método llamado campos receptivos locales, generando un número más pequeño pero suficiente de parámetros que tienen conciencia espacial.

Con esto el sistema adquiere invariancia de traducción, lo que significa que se detectará una determinada característica independientemente de dónde se encuentre en la imagen de entrada.

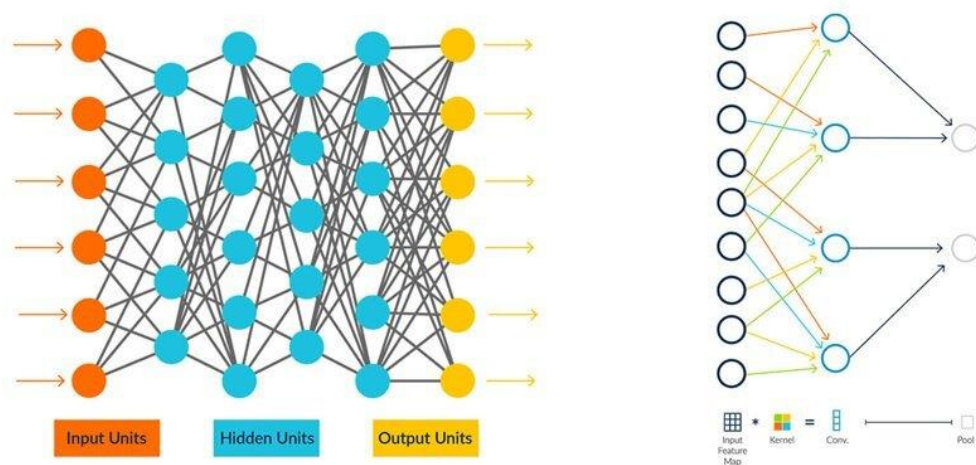


Figura 5. Comparativa entre Red Neuronal tradicional y Red Neuronal Convolutiva.

Imagen obtenida de [13].

Además, las capas de agrupación ‘max-pooling’ (función no lineal que reduce la resolución) se aplican para reducir la dependencia espacial de la característica detectada o, en otras

palabras, hacer que la red aprenda nuevas características sin que tengan que estar en una parte específica de una imagen. Con esto se consigue otra cualidad muy importante de las RNC, la capacidad de generalización (pesos compartidos), que se trata de la capacidad que tiene una red de identificar una muestra que no era conocida para ella. Esta capacidad de la red se evalúa con una base de datos específica (test) que se explicará más adelante.

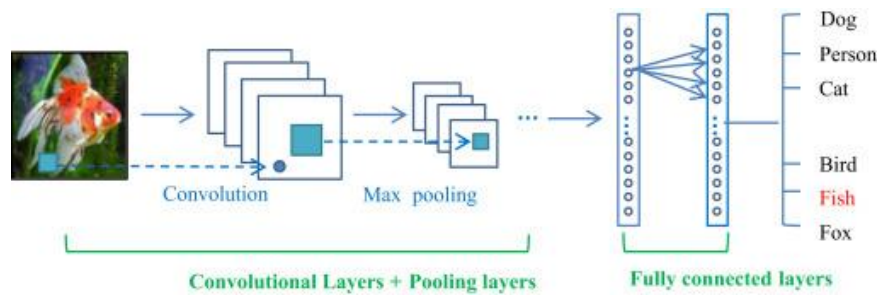
Basándonos en el patrón principal de estas redes (convoluciones y agrupación), se alcanzan dos tipos de arquitecturas de Redes Neuronales Convolucionales.

RNC Fully Connected Layers: Entrega una salida para toda la imagen, salida Fully Connected (todas conectadas con todas). Estas redes alternan capas convolucionales con capas max-pooling y al final se encuentran las capas completamente conectadas, seguidas de una capa final de clasificación (función de activación Softmax, que arroja un valor de probabilidad de 0 a 1 para cada una de las etiquetas de clasificación que el modelo intenta predecir). Con la salida Fully Connected se pierde toda la información espacial, por eso este tipo de redes se usan para la clasificación de imágenes.

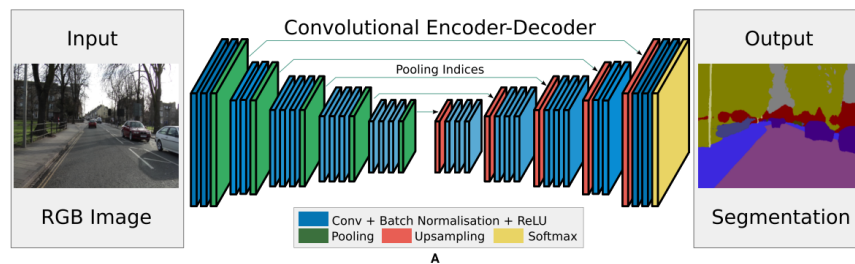
FCN (Fully Convolutional Network) [14]: El trabajo de Jonathan Long, Evan Shelhamer y Trevor Darrell en 2015, da a conocer por primera vez las Fully Convolutional Networks. Entregan una salida por píxel, por lo que se utilizan para la segmentación de imágenes. Es una arquitectura basada en encoder-decoder y será la utilizada para este proyecto (Consultar Sección 4.2 para más detalle de la arquitectura).

El uso de las técnicas de Deep Learning con Redes Neuronales Convolucionales ha ido en aumento gracias a la aceleración de la GPU, la expansión de frameworks y bibliotecas de DL, etc. Lo que hacen que la creación de prototipos sea fácil y rápido. Esto ha provocado la evolución de diversos campos como la escritura a mano, detección de caras, detección/categorización de objetos, reconocimiento de comportamiento...

Gracias a los resultados revolucionarios, las RNC han sido el enfoque dominante de la VA en los últimos tiempos, obteniendo un desempeño casi humano en algunas tareas.



(a)



(b)

Figura 6. Ejemplos de arquitecturas de RNC Fully Connected Layers [15] (a) y Fully Convolutional Network, SegNet [16] (b).

Durante la última década, las redes convolucionales han ganado popularidad, en parte, debido a la gran reducción de la tasa de error al aplicar redes de clasificación como AlexNet [17] en desafíos como ImageNet Large Scale Visual Recognition Challenge [18]. Desde que las redes convolucionales ganaron ImageNet en 2012, cada año se han desarrollado nuevos modelos y herramientas de análisis.

Uno de los primeros sucesores importantes de AlexNet es la red VGG [19] [20], que comenzó con el uso de las convoluciones pequeñas. Esta arquitectura mostró que la información contextual se puede recuperar mediante el uso de filtros pequeños y consecutivos 3x3, minimizando de este modo los campos receptivos.

2.3 Segmentación de Imágenes con RNC

Mediante el posprocesamiento de los mapas de características finales de las redes de clasificación, se puede recuperar la resolución espacial de la imagen de entrada. Esto permite el desarrollo de redes convolucionales que asignan una etiqueta diferente a cada píxel, un proceso generalmente conocido como segmentación semántica.

La salida de la red debe ser una máscara del mismo tamaño que la imagen original, donde el valor de cada píxel representa la clase a la que pertenece esa porción de la imagen. Es decir, predice una clase para cada píxel de la imagen de entrada. Ubica el objeto en el espacio [21].

De manera similar a los métodos de clasificación, la segmentación de imágenes tradicional se basaba en extractores de características codificadas que dividían la imagen en varias regiones. Uno de los métodos más antiguos de segmentación de imágenes es la creación de umbrales, una técnica sencilla en la que la intensidad de cada píxel se compara con un umbral. Otro algoritmo tradicional es el ajuste de contorno utilizando técnicas como los modelos de contorno activos.

Los enfoques tradicionales de aprendizaje automático han sido reemplazados por redes convolucionales para la segmentación semántica. Unos de los primeros intentos de realizar segmentación de imágenes mediante redes convolucionales fueron las redes de clasificación en parches de imágenes. Pero estos enfoques de segmentación pronto fueron reemplazados por redes totalmente convolucionales. Estas redes realizan una segmentación de imágenes de extremo a extremo mediante un muestreo superior iterativo de las características más profundas de las redes de clasificación (Figura 6.b).

A pesar del éxito de la arquitectura UNet en esta tarea, algunas arquitecturas sucesivas también han optado por transferir la mayoría de las operaciones convolucionales al lado del codificador. Una red popular que sigue este patrón es DeepLab [21], donde un extractor de características muy profundo como ResNet es seguido por convoluciones atroces, y luego las características intermedias se muestrean usando interpolación bilineal.

En la actualidad existen múltiples disciplinas que tratan de incluir esta tecnología en su ámbito de trabajo para obtener resultados de forma más rápida y precisa. Una de las disciplinas que más esfuerzos pone en la investigación de este campo es la medicina, cuyo principal objetivo es lograr diagnosticar y tratar, e incluso prevenir enfermedades como el cáncer.

En los últimos años la segmentación semántica en imágenes médicas ha sido uno de los principales campos de aplicación de las RNC. Además de las capacidades 2D utilizadas para delimitar órganos, malformaciones, etc., las RNC han impactado con sus capacidades 3D, ayudando también a procesar volúmenes de datos.

Antes de esto, los investigadores biomédicos seguían dos enfoques de segmentación:

- Clasificar la imagen en su conjunto (maligna o benigna).
- Dividir las imágenes en parches y clasificarlas.

La aplicación de parches fue mejor que la clasificación de una imagen completa, sin embargo, tenía inconvenientes, ya que se genera información redundante, y el tamaño de los parches podía producir pérdida de información contextual o alterar la localización. Según observaciones anteriores, una arquitectura de codificador-decodificador (como UNet) produce valores de intersección sobre unión (IoU) mucho más altos que la alimentación de cada píxel a una RNC para su clasificación.

Capítulo 3

Infraestructura

Este capítulo presenta el entorno computacional utilizado para desarrollar este proyecto. Las principales características de las bibliotecas y frameworks en los que se apoya el proyecto, y su justificación de uso. Además del software y hardware utilizado para desarrollar, entrenar y probar la red neuronal.

Las herramientas utilizadas son de código abierto. Esto es una gran ventaja, ya que el software de terceros se puede integrar fácilmente, y nos permite obtener resultados más rápidamente que si hubiera sido desarrollada desde cero.

Además, se presentan los conjuntos de imágenes de los que se dispone para realizar las distintas pruebas sobre la red neuronal.

3.1. Librerías y Frameworks

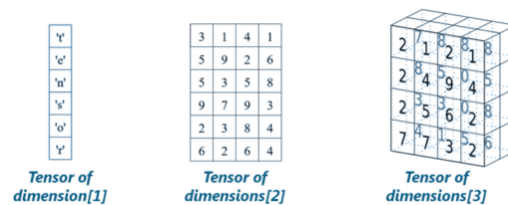
Esta sección contiene los antecedentes teóricos y las principales características de los paquetes, bibliotecas y frameworks más importantes que se han utilizado.

Python (Python 3.7.3) es el lenguaje de programación elegido para desarrollar el proyecto.

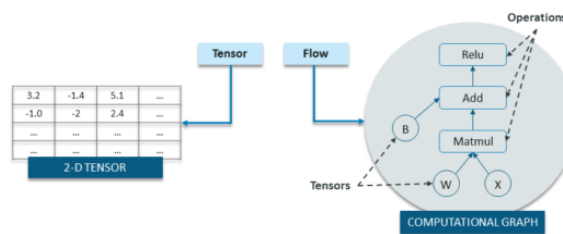
3.1.1. TensorFlow

TensorFlow [23] es una plataforma de código abierto de extremo a extremo, desarrollada por Google, cuya finalidad es extender el uso del Deep Learning a un rango de tareas muy amplio. Tiene un ecosistema integral y flexible de herramientas, bibliotecas y recursos comunitarios que permite a los investigadores impulsar el estado del arte en ML y a los desarrolladores crear e implementar fácilmente aplicaciones impulsadas por ML. Fue lanzado públicamente en noviembre 2015, bajo la licencia de código abierto Apache 2.0. Está escrito en C ++, Python y CUDA.

TensorFlow se ha seleccionado como biblioteca de Deep Learning para este proyecto porque admite la aceleración de la GPU (característica indispensable para acortar los tiempos de entrenamiento), porque tiene una comunidad muy extendida que facilita el aprendizaje desde cero y porque es el backend recomendado para Keras, la API principal utilizada para desarrollar este proyecto.



(a)



(b)

Figura 7. (a) Ejemplo de tensores. Formas estándar de representar datos en TensorFlow con Tensores. (b) Diferencia entre tensor y flujo. Imágenes obtenidas de [24].

En TensorFlow, el término Tensor se refiere a la representación de datos como una matriz multidimensional, mientras que el término Flow se refiere a la serie de operaciones (al flujo de operaciones) que se realizan con los tensores. TensorFlow permite crear redes neuronales a gran escala con muchas capas, basándose en el concepto de gráficos de flujo de datos, o gráfico computacional, para crear modelos. Estos gráficos constan de nodos (Nodes) y bordes (Edges), donde cada nodo representa una instancia de una operación matemática, y los bordes representan datos en forma de tensores, sobre los que se realizan las operaciones. En este proyecto se ha utilizado TensorFlow 2.0., aunque la versión actual en 2.3.

TensorBoard

Es una herramienta de TensorFlow que proporciona la visualización y las herramientas necesarias para alojar los resultados de nuestros experimentos de aprendizaje automático, y hacerles un seguimiento [25].

Permite seguir y visualizar métricas (como la pérdida y la métrica), ver histogramas dinámicos o mostrar imágenes, entre otros.

Hay dos formas de usar Tensorboard en Keras: A través del callback de Keras o a través `tf.summary` para escenarios más personalizados. En este caso, utilizaremos los dos métodos (ver Sección 3.1.2).

3.1.2. Keras

El desarrollo de Deep Learning comenzó siendo muy tedioso, pero actualmente TensorFlow incorpora módulos como Keras para facilitar el desarrollo.

Keras [26] es una API de Deep Learning escrita en Python, que se ejecuta sobre la plataforma de Machine Learning TensorFlow. Fue creada por François Chollet en 2015, no para ser un framework de aprendizaje automático independiente, sino para ser una interfaz de alto nivel sobre los backends de aprendizaje profundo más populares, y así permitir rápidas experimentaciones.

Keras se ha utilizado para implementar la arquitectura de red neuronal en este proyecto, ya que hemos valorado sus potentes ventajas, como:

- Modularidad: permite definir una red neuronal siguiendo un modelo, entendido como una secuencia o un grafo, pudiendo unir módulos totalmente configurables, como una pila lineal de capas. Esta es una característica útil ya que facilita unir las diferentes capas que componen la red neuronal, como las funciones de activación, capas convolucionales, etc.
- Extensibilidad: Aparte de los módulos ya creados, es fácil implementar y agregar nuevos módulos al modelo de red, lo que hace que Keras sea óptimo para investigaciones avanzadas.
- Python nativo. Keras trabaja en un entorno Python, lo que nos da la libertad de usar dependencias adicionales de Python.

En definitiva, permite un desarrollo rápido de redes neuronales, con sus capas predefinidas, devoluciones de llamada y funciones sencillas de usar.

La versión utilizada en este proyecto es Keras 2.3.0, aunque la más reciente sea 2.4.0.

A continuación, se analizan los principales elementos que componen una red neuronal construida con Keras, comenzando por su estructura de datos principal: el modelo.

Modelos

Cada red neuronal de Keras se define como un modelo. Los modelos son una forma de organizar capas. Hay dos tipos de modelos disponibles en Keras: El modelo secuencial, construido como una pila lineal de capas, donde cada capa tiene un tensor de entrada y uno de salida, y la clase de modelo utilizada con la API funcional, para arquitecturas más complejas y flexibles.

Ambos tipos de modelos tienen algunos de los métodos y atributos en común. A continuación, se nombran los que utilizamos en este proyecto:

.compile()

Cuando un modelo ya está definido, hay un paso más que se debe realizar antes de entrenar con él: configurar su proceso de aprendizaje. Esto se hace con el método de compilación y recibe tres argumentos principales (tiene más argumentos opcionales que no utilizamos):

- **Optimizador.** El optimizador minimiza las funciones deseadas, en nuestro caso la función de pérdida, actualizando los valores de los pesos del modelo durante el entrenamiento.

Este parámetro podría instanciar un optimizador definido por el usuario, pero Keras ya tiene una lista de optimizadores predefinidos, entre ellos el que implementa el algoritmo Adam [27], que es el que utilizaremos en este trabajo (Ver la Sección 4.3.2 para más detalle sobre el optimizador empleado).

- **Función de pérdida (loss).** Este parámetro especifica la función de pérdida que el optimizador intentará minimizar. Mide la diferencia entre las etiquetas predichas y las reales, y el objetivo es que sean lo más parecidas posibles, es decir, que el error sea mínimo.

Existen varias funciones de pérdida diferentes, concretamente en Keras hay 16 disponibles. En este proyecto, la función de pérdida que se calcula después de cada

lote es la entropía cruzada categórica, también conocida como pérdida de registro. Esta función calcula la pérdida de entropía cruzada entre las etiquetas originales y las predicciones. (Ver la Sección 4.3.1 para más detalle sobre la función de pérdida empleada).

- **Métrica.** Este parámetro especifica la función que se utiliza para evaluar el rendimiento del modelo durante el entrenamiento y su evaluación. Es similar a la función de pérdida (evalúa la diferencia entre las etiquetas predichas y las originales) a diferencia de que estos resultados no se utilizan para entrenar el modelo. Keras ofrece varias métricas ya definidas, pero en nuestro caso usaremos el Coeficiente Dice (Ver la Sección 4.3.3 para más detalle sobre esta métrica empleada).

.fit()

Es la función básica de entrenamiento en Keras. Se utiliza para entrenar el modelo para un número determinado de iteraciones en un conjunto de datos o epochs.

Sus argumentos son los siguientes:

- **x, y:** muestras de entrenamiento y sus correspondientes etiquetas. En nuestro proyecto introduciremos en este parámetro matrices Numpy. Excepto en los experimentos en los que utilicemos aumento de datos, que introduciremos el generador de datos ImageDataGenerator como argumento (Sección 4.4).
- **batch size:** número de muestras que se evalúan antes de actualizar los pesos. El valor predeterminado es 32.
- **epochs:** número de iteraciones que se ejecutarán en todo el conjunto de datos de entrenamiento x e y. El valor predeterminado es 10.
- **verbose:** este valor indica cómo queremos visualizar la información sobre el entrenamiento.
- **callbacks:** lista de callbacks que se aplicarán durante el entrenamiento.
- **validation_data o validation_steps:** datos de validación, como una matriz Numpy para el argumento de validation_data, o como una fracción de las muestras de entrenamiento mediante el argumento de validation_steps. Ambos argumentos son mutuamente excluyentes, por lo que solo se puede usar uno de ellos. En nuestro caso

vamos a indicar los datos de validación como una matriz Numpy, introduciéndose desde `validation_data`.

.evaluate()

Este método toma como parámetros un conjunto de muestras y etiquetas, como matrices Numpy, y evalúa el rendimiento del modelo ya entrenado, devolviendo una lista de las métricas previamente definidas.

.predict()

Toma un conjunto de muestras, como matriz Numpy, y devuelve la etiqueta predicha por el modelo ya entrenado.

.load_weights()

Carga todos los pesos del modelo ya entrenado, que han sido previamente guardados. En nuestro caso se cargan desde un archivo HDF5.

Capas

Las capas son los componentes básicos de las redes neuronales en Keras. Los modelos suelen construirse como una pila de capas, indicando el tipo de capa y sus parámetros particulares. Hay varios tipos de capas disponibles en Keras, pero sólo se describen las que se han utilizado en este proyecto (ver Sección 4.2.1 para más detalle sobre las capas implementadas en el modelo):

Capas convolucionales

Estas capas son las que convierten la red neuronal en una RNC. Keras proporciona diferentes tipos de capas convolucionales según las dimensiones de la entrada (Conv1D, Conv2D, Conv3D...). En nuestro caso usaremos Conv2D y Conv2DTranspose, para la convolución espacial de imágenes. Sus principales argumentos requeridos son:

- **Conv2D**

Esta capa crea un filtro de convolución que convoluciona con la entrada de la capa para producir un tensor de salida. Los parámetros son:

- **inputs:** En nuestro caso, la entrada será una matriz tridimensional definida por su ancho (número de columnas), alto (número de filas) y profundidad (número de canales).
- **filters:** número de filtros.
- **kernel size:** ancho y alto de los filtros.
- **strides:** pasos de la convolución a lo alto y lo ancho. Su valor predeterminado es 1.
- **padding:** puede ser 'valid' o 'same'. Si se establece 'valid', no se aplica ningún relleno, dando un tensor de salida menor que el de entrada. Sin embargo, si es 'same', se rellenará con ceros para producir una salida que conserva el tamaño de entrada. Por defecto es 'valid'. En nuestro caso utilizaremos 'same'.
- **activation:** Función de activación a utilizar.

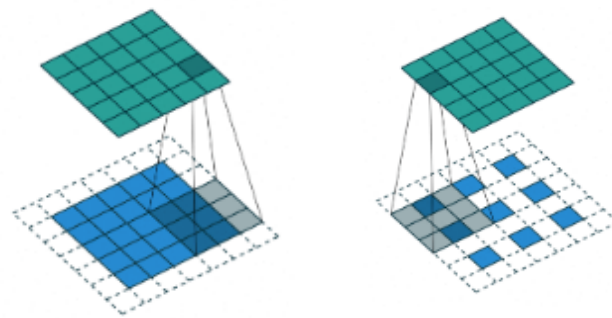


Figura 8. Ejemplos de convolución 2D con relleno (izquierda) y convolución traspuesta 2D con relleno y strides (derecha). Imagen obtenida de [28]

○ **Conv2DTranspose**

Capa de convolución traspuesta o deconvolución. La necesidad de convoluciones transpuestas generalmente surge del deseo de utilizar una transformación que vaya en la dirección opuesta a una convolución normal, es decir, de algo que tiene la forma de la salida de alguna convolución a algo que tiene la forma de su entrada mientras se mantiene un patrón de conectividad que sea compatible con dicha convolución. Los argumentos son muy similares a los del caso anterior.

Capas de agrupación

Desplaza una ventana a la matriz de entrada aplicando una operación, que devolverá una versión reducida de la misma. Keras proporciona diferentes capas de agrupación según las dimensiones de la entrada y la operación aplicada (MaxPooling2D, MaxPooling3D, AveragePooling1D...). En este proyecto se utiliza:

- **MaxPooling2D**

Operación de agrupación máxima para datos espaciales 2D. Reduce la representación de entrada tomando el valor máximo sobre la ventana definida por `pool_size` para cada dimensión a lo largo del eje de características. Los principales argumentos requeridos son:

- **pool_size**: tamaño de ventana sobre el que tomar el valor máximo. Por ejemplo (2, 2) tomará el valor máximo sobre una ventana de agrupación de 2x2.
- **strides**: Indica cuántos píxeles se mueve la ventana para cada paso de agrupación. Si es None será predeterminado por `pool_size`.

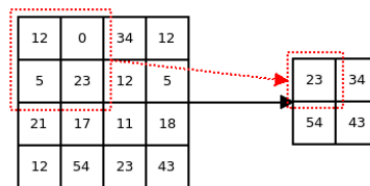


Figura 9. Ejemplo de operación MaxPooling2D con `pool_size (2,2)`

Capas merging

- **Concatenate**

Capa que concatena una lista de entradas. Toma como entrada una lista de tensores, todos de la misma forma, y el eje de concatenación. Devuelve un solo tensor que es la concatenación de todas las entradas.

Callbacks

Un callback, o devolución de llamada, es un objeto que puede realizar acciones en varias etapas del entrenamiento, por ejemplo, al comienzo o al final de una época. Se utilizan, por ejemplo, para monitorear métricas en TensorBoard o guardar periódicamente el modelo.

Keras proporciona la clase `Callback()` que se puede utilizar para crear callbacks definidas por el usuario, pero en este proyecto se introducen como argumento al método `.fit()` del modelo. Las que se han utilizado son:

ModelCheckpoint

Este callback se usa para guardar un modelo o pesos (en un archivo de punto de control) en algún intervalo del entrenamiento, por lo que el modelo o los pesos se pueden cargar en otro momento para continuar el entrenamiento desde el estado guardado.

Se puede configurar para sobrescribir el modelo solo si una determinada métrica ha mejorado con respecto al mejor resultado anterior, guardando la mejor versión del mismo. Estos son sus argumentos que usamos en nuestro caso:

- **filepath:** dónde guardar el modelo.
- **monitor:** el nombre de la métrica a monitorizar. En este caso introducimos 'loss' para monitorear la métrica total del modelo, y la prefijamos con 'val_loss' para monitorear las métricas de validación.
- **save_best_only:** booleano que indica que solo un modelo será guardado si es mejor que el modelo guardado actual.
- **mode:** {'auto', 'min', 'max'}. Si `save_best_only = True`, la decisión de sobrescribir el archivo guardado actual se toma en función de la maximización o minimización de la cantidad monitoreada. Para `val_loss` este argumento debe ser `min`.

TensorBoard

Este parámetro habilita la visualización en TensorBoard, registrando eventos que incluyen Gráficos de resumen de métricas, Visualización de gráficos de entrenamiento, etc.

Preprocesamiento de datos

Keras ofrece utilidades de preprocesamiento de conjuntos de datos, que ayudan a procesar los datos que se le pasan a un modelo para su entrenamiento. Nosotros nos centraremos en las de preprocesamiento de datos de imagen.

El aumento de datos consiste en coger las imágenes originales del conjunto de datos y aplicarles transformaciones para agregar más variabilidad a las muestras, lo que lleva a una mejor generalización (ver Sección 4.4.3).

Esta funcionalidad está incluida en Keras gracias al método `ImageDataGenerator`.

ImageDataGenerator

Genera lotes de datos de imágenes (tensores) con aumento de datos en tiempo real. Aplica aleatoriamente las transformaciones deseadas a muestras aleatorias del conjunto de datos proporcionadas por el usuario. Transformaciones como la rotación, el desplazamiento y el zoom se pasan como argumentos. Incluso se le puede pasar como argumento una función definida por el usuario.

El conjunto de datos (`y`) y el tamaño de lote se definen mediante el método `.flow()`. Durante el entrenamiento, el generador se repetirá hasta que el número de imágenes por época y el número de epochs establecidas por el usuario se cumplen.

Utilidades

to_categorical()

Convierte un vector de clase (enteros) en una matriz de clase binaria. En este proyecto utilizamos esta clase en el preprocesamiento de imágenes, ya que utilizamos la función de pérdida entropía cruzada categórica y necesitamos variables categóricas. Sus argumentos son:

- **y**: vector de clase a convertir en una matriz (enteros de 0 a `num_classes`).
- **num_classes**: número total de clases. Si es `None`, sería igual al número más grande en `y` + 1.

Devuelve una representación matricial binaria de la entrada, colocando el eje de clases en último lugar (ver Sección 4.1).

3.1.3. Otros paquetes

Numpy: [29] es una biblioteca para Python que da soporte para crear vectores y matrices multidimensionales, junto con una gran colección de funciones matemáticas de alto nivel

para operar con ellas. Algunas funcionalidades de NumPy utilizadas en este proyecto son también los generadores de números aleatorios.

Scikit-image (skimage): [30] Es una colección de algoritmos para el procesamiento de imágenes en Python, de software libre creado por una comunidad de voluntarios. Algunos de los módulos que se usan en este proyecto son: io (para leer, guardar o visualizar imágenes), transform (para realizar transformaciones geométricas y otras, a las imágenes), exposure, etc.

Scikit-learn: [31] es una biblioteca para aprendizaje automático, en Python, de software libre. En este caso la utilizamos para realizar la validación cruzada, gracias a la clase KFold, que nos permite dividir el conjunto de datos en K pliegues consecutivos.

SciPy: SciPy [32] es una biblioteca libre y de código abierto para Python. Se compone de herramientas y algoritmos matemáticos. En nuestro caso la utilizamos para interpolación y aplicar filtros en el caso de las deformaciones elásticas.

3.2. Conjunto de Imágenes

Para realizar el entrenamiento y la evaluación de la red, es necesario disponer de un conjunto de datos. Recordando el objetivo de este proyecto, segmentación morfológica de células en vídeo microscopía, disponemos de un conjunto de imágenes facilitado por la organización Cell Tracking Challenge [2] . Ofrecen diversos conjuntos de datos que consisten en secuencias de video time lapse en 2D y 3D de núcleos o células contrateñidos con fluorescencia que se mueven en la parte superior o sumergidas en un sustrato, junto con videos de microscopía de campo brillante 2D, contraste de fase y contraste de interferencia diferencial (DIC) de células que se mueven en un sustrato plano. Los videos cubren una amplia gama de tipos de células y calidad (resolución espacial y temporal, niveles de ruido, etc.). Además, proporcionan vídeos en 2D y 3D de células sintéticas teñidas con fluorescencia y núcleos de diferentes formas y patrones de movimiento.

Para el entrenamiento y la evaluación de la red neuronal, se ha escogido el dataset 2D + Time Dataset: **Fluo-C2DL-MS**C. Células madre mesenquimales de rata transfectadas con GFP en un sustrato plano de poliacrilamida, adquiridas usando un microscopio confocal de disco giratorio Perkin Elmer UltraVIEW ERS (cortesía del Dr. F. Prósper, Laboratorio de Terapia Celular, Centro de Investigaciones Médicas Aplicadas, CIMA-ES, Pamplona). La

dificultad del conjunto de datos es alta debido a la baja relación señal-ruido y la presencia de áreas protuberantes similares a filamentos causadas por el estiramiento de las células, que a veces aparecen como extensiones discontinuas de las células. Para complicar aún más el análisis de las escenas, estas protuberancias a menudo entran en contacto con otras células [1] [3].

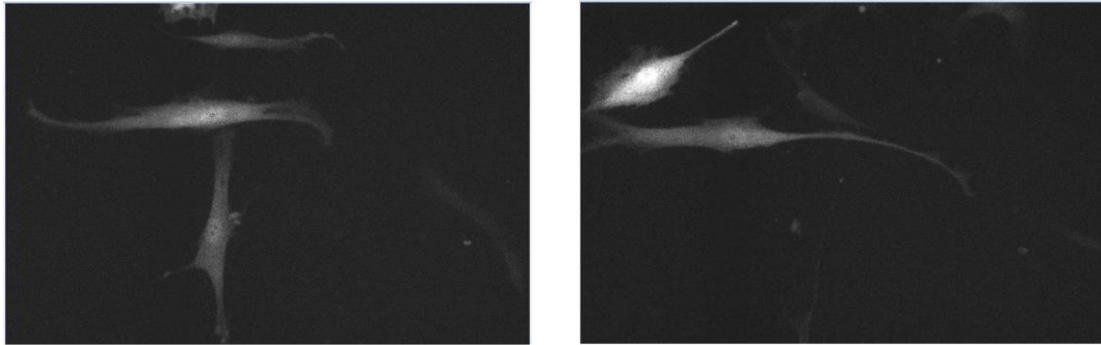


Figura 10. Dos muestras del conjunto de datos escogido para el proyecto. Fluo-C2DL-MS

El conjunto de datos consta de dos secuencias de vídeo:

Secuencia de vídeo 1:

- **[01]**: 48 imágenes de células (832x992 píxeles). Paso de tiempo 20.
- **[01_GT][SEG]**, Gold-standard corpus (gold truth): 18 anotaciones de células de referencia (832x992 píxeles) para evaluar el rendimiento de la segmentación, hechas por humanos y obtenidas como opinión consensuada o mayoritaria de varios expertos humanos. Las anotaciones dan información de clase para cada pixel, con tantas clases como células haya en la imagen, incluyendo el fondo.
- **[01_ST][SEG]**, Silver-standard corpus (silver truth): 48 anotaciones de referencia (832x992 píxeles) generadas por ordenador, obtenidas como opinión mayoritaria sobre los resultados de varios métodos de análisis automático, presentados por ex participantes de este desafío.

Secuencia de vídeo 2:

- **[02]**: 48 imágenes de células (1200x782 píxeles). Paso de tiempo 30.
- **[02_GT][SEG]**, Gold-standard corpus (gold truth): 33 anotaciones de células de referencia (1200x782 píxeles) para evaluar el rendimiento de la segmentación, hechas por humanos y obtenidas como opinión consensuada o mayoritaria de varios

expertos humanos. Las anotaciones dan información de clase para cada píxel, con tantas clases como células haya en la imagen, incluyendo el fondo.

- **[02_ST][SEG]**, Silver-standard corpus (silver truth): 48 anotaciones de referencia (1200x782 píxeles) generadas por ordenador, obtenidas como opinión mayoritaria sobre los resultados de varios métodos de análisis automático, presentados por ex participantes de este desafío.



Figura 11 . 28 muestras [02] y sus correspondientes 28 anotaciones Estas últimas con los valores corregidos para poder visualizar las etiquetas en esta figura.

Adicionalmente, también se presentan dos secuencias de vídeo extras, pero únicamente con las imágenes de células, sin aportar las anotaciones. Estas imágenes serán utilizadas en la última sección del proyecto, para evaluar, de forma visual, la segmentación final de nuestros modelos entrenados.

Capítulo 4

Segmentación de Células con Deep Learning

En la segmentación de imágenes biomédicas es muy importante obtener una etiqueta de clase para cada píxel de la imagen, pero el mayor problema en este tipo de tareas es que no son fácilmente accesibles miles de imágenes etiquetadas para un buen entrenamiento de la red.

En este proyecto disponemos de pocas imágenes etiquetadas, por ello se desarrolla una estrategia para utilizar las que tenemos disponibles de manera más eficiente e intentar conseguir una segmentación precisa.

Se ha elaborado una herramienta en Python que permite la segmentación de clase, en este caso clasificar todos los píxeles de una imagen en dos clases: célula o fondo, mediante una red neuronal generada en Keras sobre Tensorflow. Además, dicha red ha sido mejorada gracias a un estudio sobre las variantes posibles aplicadas.

El resumen de la tarea de segmentación sigue la estructura principal que se describe en el diagrama de la Figura 12. En este capítulo se explica cada bloque del diagrama.

En primer lugar, las imágenes y anotaciones con las que se realiza el entrenamiento son cargadas y pre procesadas para adaptarlas a la entrada de la red neuronal. Posteriormente, el modelo propuesto es entrenado, con los diferentes parámetros de entrenamiento y compilación predefinidos. Una vez entrenado, se le introducen una o varias imágenes de prueba para su evaluación, adquiriendo los valores de métricas resultantes, y una matriz Numpy con las máscaras binarias de la imagen de prueba como resultado de la red.

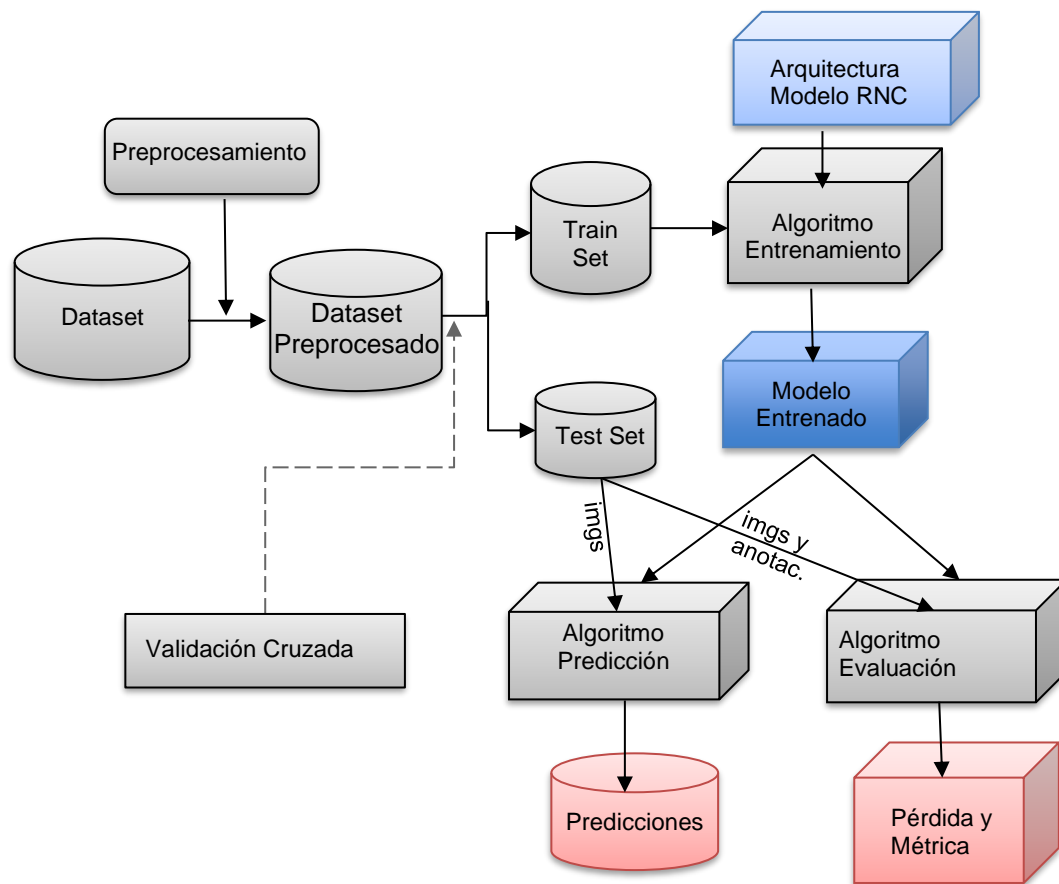


Figura 12. Diagrama de Flujo de la tarea de segmentación desarrollada.

4.1. Preprocesamiento de imágenes

El conjunto de datos utilizado para entrenar y evaluar la red está especificado en la Sección 3.2. Las imágenes se procesan con un script de Python llamado ‘data.py’, donde serán cargadas y adaptadas, con el objetivo de ser aceptadas por la arquitectura y de lograr mayor robustez en el resultado final. A continuación, se detallan todos los pasos de pre procesamiento de las imágenes y sus anotaciones.

Carga de imágenes y anotaciones.

```
img = imread(os.path.join(data_path, image_name_data), as_gray=True)
img_mask = imread(os.path.join(mask_path, image_name_mask))
```

El conjunto de imágenes utilizado para entrenar y evaluar la red queda limitado a un número de imágenes que contienen su anotación completa. Ya que, como se puede observar en la

Figura 13, no todas incluyen anotación, y no todas las anotaciones son completas. Al leer las imágenes, se convierten en formato binario NumPy (.npy).



Figura 13. 18 anotaciones para las 48 imágenes de la secuencia de vídeo 01.

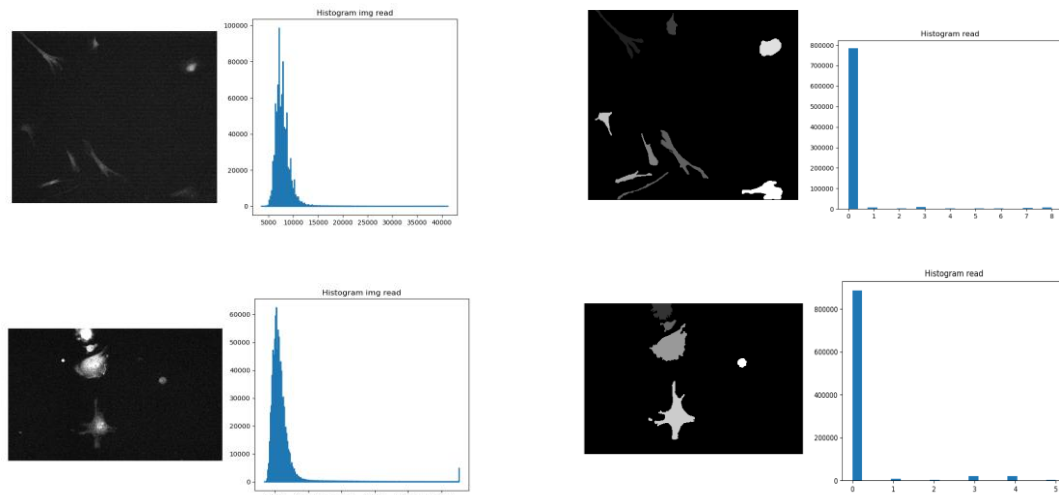


Figura 14. Imágenes y sus correspondientes anotaciones a su izquierda. Muestras leídas, sin pre procesar. Se adjuntan con su histograma de valores correspondiente.

Binarización de anotaciones.

```
img_mask = 1.0 * (img_mask > 0.2)
```

Este trabajo busca la segmentación de clases, es decir, queremos clasificar todos los píxeles de una imagen en dos clases: célula o fondo. En las imágenes de nuestro conjunto de datos aparece más de una célula, y sus anotaciones aplican una clase distinta a cada célula, es decir, cada píxel es un número entero indicando la célula a la que corresponde, o el fondo. Las células están etiquetadas. Para este proyecto, sólo interesa saber si el píxel corresponde a una célula, no a qué célula. Por lo tanto, se binariza la etiqueta para mostrar 'célula [1]' o 'fondo [0]'.

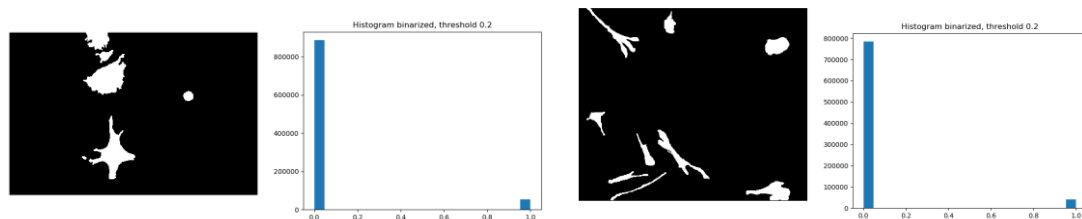


Figura 15. Dos muestras de anotaciones después convertirlas en binarias, con sus histogramas de valores correspondientes.

Cambio de tamaño de imágenes y anotaciones.

Se modifica el tamaño de los datos originales a 256x256 píxeles.

```
img = resize(img, (image_rows, image_cols), preserve_range=True)
img_mask = resize(img_mask, (image_rows, image_cols), preserve_range=True)
```



Figura 16. Imagen reescalada a la derecha y su anotación reescalada correspondiente a la izquierda. Ambas con sus respectivos histogramas de valores.

Cambio de canales de las anotaciones.

```
categorical_labels = to_categorical(img_mask, num_classes = None)
```

Para alimentar el modelo de Keras, las anotaciones deben ser remodeladas en distribuciones de probabilidad, es decir, una matriz en la que cada elemento representa la probabilidad de ocurrencia de las dos etiquetas (célula o fondo) de cada píxel. Por ejemplo, si un píxel corresponde a una célula y tiene valor [1], en la matriz remodelada será [0, 1], indicado que tiene 0 probabilidad de pertenecer a fondo, y 1 de probabilidad de pertenecer a célula. Esta conversión se logra usando el método incorporado de Keras `utils.to_categorical()`. Esto nos devuelve la matriz con forma (filas, columnas, canales), lo que quedaría (256, 256, 2).

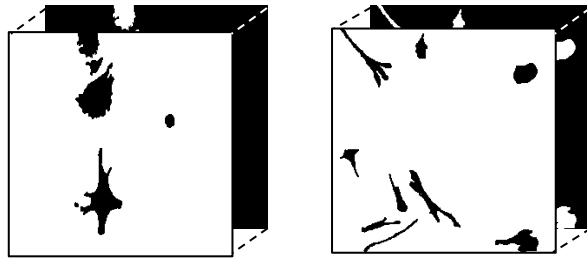


Figura 17. Ejemplo visual del resultado de categorizar las anotaciones.

Cambio de canales de las imágenes.

```
img = np.reshape(img, (image_rows, image_cols, 1))
```

Para alimentar el modelo de Keras, el número de canales de las muestras debe declararse explícitamente, por lo que el conjunto de datos debe ser remodelado para indicar dichos canales. Pasamos de (256, 256) a (256, 256, 1). En este caso, las imágenes originales son en escala de grises, lo que implica que el número de canales es igual a 1 (si fuese RGB, el número de canales sería 3). El orden en el que se deben declarar las dimensiones depende del Backend, en este caso Tensorflow: (filas, columnas, canales).

Para el caso de las anotaciones ya tenemos la forma correcta, indicada en el paso anterior.

Normalización de las imágenes. Estandarización de características.

```
mean = np.mean(imgs)
std = np.std(imgs)
imgs -= mean
imgs /= std
```

Por último, se realiza la normalización de las imágenes, una técnica que se aplica a menudo como parte del procesamiento de datos en el aprendizaje automático [33]. El método más común para la normalización es la estandarización de características, que se refiere a la configuración de los datos para que tengan una media cero y un mismo rango de variación. El objetivo de la normalización es cambiar los valores del conjunto de datos a una escala común, sin distorsionar las diferencias en los rangos de valores. Nos aseguramos de que las diferentes características adopten rangos de valores similares para que los descensos de gradiente puedan converger más rápidamente.

$$z = \frac{x - \mu}{\sigma}$$

μ = Mean
 σ = Standard Deviation

Figura 18. Función de distribución normal.

Esta técnica sólo se aplica en las imágenes, no en las anotaciones, obteniendo una distribución normal, como se puede observar en el histograma de valores de la Figura 19.

- **Centrar, media 0:** Centramos los valores de todas las imágenes de entrada en cero, restando la media. Se aplica a la matriz de todas las imágenes para que estén centradas en 0.
- **Normalizar, desviación típica 1:** Dividimos cada imagen por la desviación típica, obteniendo una desviación típica igual a 1. Se obtiene una normalización de sus valores originales para que todas las imágenes se encuentren en los mismos rangos de valores normalizados.

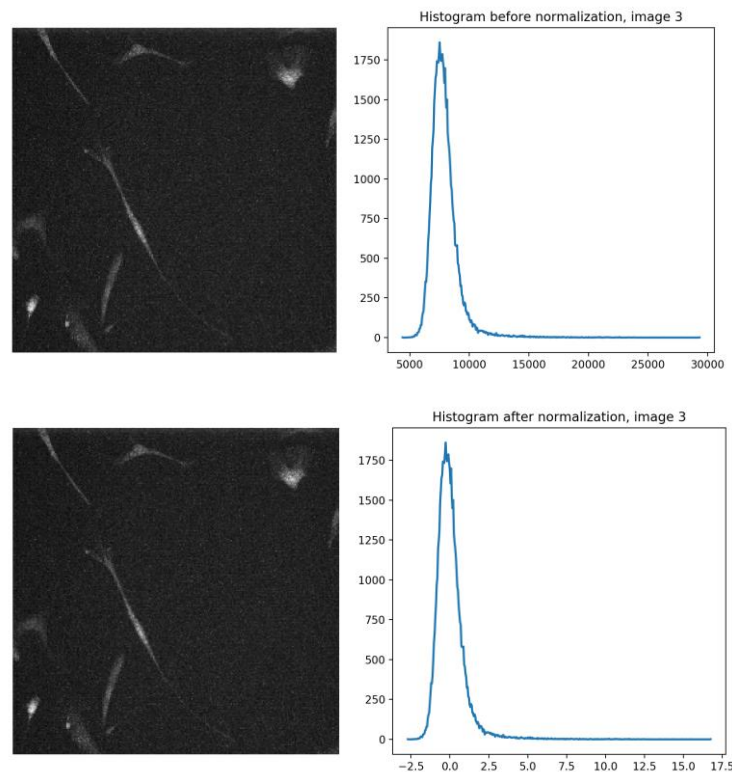


Figura 19. Comparación de una muestra antes y después de la normalización, con su histograma de valores correspondiente.

4.2. Arquitectura del Modelo UNet

La segmentación de imágenes es útil en muchos campos, pero uno de los más importante es en el campo de la imagen médica. Las sutilezas de las imágenes médicas son muy complejas, incluso para médicos muy capacitados. Una máquina que pueda comprender estos matices y pueda identificar y segmentar las áreas necesarias puede tener un gran impacto [34].

Ahí es donde UNet entra en escena. UNet [35] se diseñó por primera vez especialmente para la segmentación de imágenes médicas, por Olaf Ronneberger, Phillip Fischer y Thomas Brox en 2015. Mostró tan buenos resultados que se utilizó en muchos otros campos después.

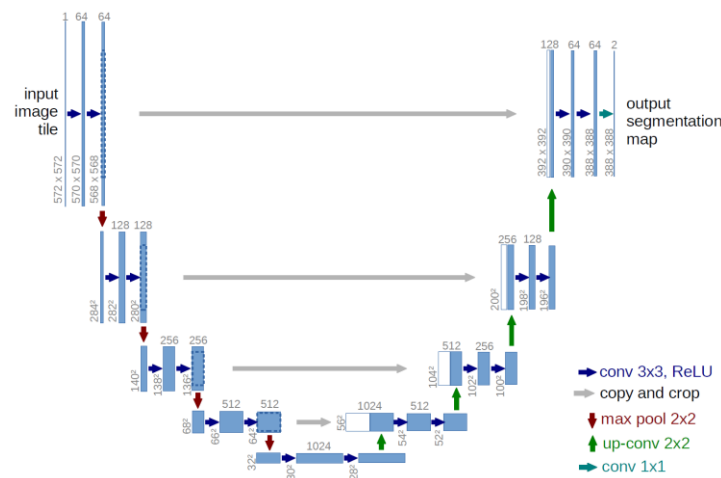


Figura 20. Arquitectura UNet original. Imagen obtenida de [35]

UNet es la arquitectura de red neuronal convolucional utilizada para entrenar y evaluar en este proyecto, debido a que consigue una segmentación de imágenes rápida y precisa y es utilizada para segmentación de imágenes biomédicas. Una red profunda de este tipo se puede entrenar de extremo a extremo con pocas imágenes.

Para lograr la salida deseada (máscara de segmentación, con las mismas dimensiones que la imagen de entrada, que indique qué píxeles corresponden a célula o fondo) era necesario usar una arquitectura codificador-decodificador. Consiste en una ruta de contracción que codifica la información de entrada capturando el contexto (información semántica) y una ruta de expansión simétrica que decodifica la información hasta que alcanza una dimensión similar a la de la entrada y permite una localización precisa recuperando información espacial.

U-Net debe su nombre a su forma simétrica, que es diferente de otras variantes de FCN. En comparación con FCN-8, UNet al ser simétrica, combina la información de ubicación de la ruta de muestreo descendente con la información contextual en la ruta de muestreo ascendente para obtener finalmente una información general que combina la localización y el contexto, que es necesaria para predecir un buen mapa de segmentación.

No utiliza ninguna capa completamente conectada. Como consecuencia, el número de parámetros del modelo se reduce y se puede entrenar con un pequeño conjunto de datos etiquetado (utilizando el aumento de datos apropiado).

La arquitectura implementada en este proyecto, mostrada en la Figura 21, sigue una estructura similar a la del documento original.

4.2.1. Capas

Antes de explicar la estructura de la red, vamos a enumerar las diferentes capas que la componen y los antecedentes teóricos de cada una de ellas. Todas estas capas forman patrones repetitivos que implementan los modelos de U-Net, como se explicará en la siguiente sección 4.2.2.

Capas Convolucionales 2D (*Conv2D*)

Como su propio nombre indica, la base de esta arquitectura es la convolución, que consiste en filtrar una imagen usando un filtro o máscara.

En la convolución, cada píxel de salida es una combinación lineal de los píxeles de entrada. Ventana deslizante. Los filtros representan la conectividad entre las capas sucesivas. Cada filtro genera un mapa de características una vez que se aplica a la entrada completa. Se crean varios mapas de características, uno de cada filtro en las capas convolucionales [36]. Los pesos de la red neuronal son valores de los filtros, es lo que se calcula durante el entrenamiento.

Model: "model_1"

Layer (type)	Output Shape	Param #	Connected to
input_1 (InputLayer)	(None, 256, 256, 1)	0	
conv2d_1 (Conv2D)	(None, 256, 256, 32)	320	input_1[0][0]
conv2d_2 (Conv2D)	(None, 256, 256, 32)	9248	conv2d_1[0][0]
max_pooling2d_1 (MaxPooling2D)	(None, 128, 128, 32)	0	conv2d_2[0][0]
conv2d_3 (Conv2D)	(None, 128, 128, 64)	18496	max_pooling2d_1[0][0]
conv2d_4 (Conv2D)	(None, 128, 128, 64)	36928	conv2d_3[0][0]
max_pooling2d_2 (MaxPooling2D)	(None, 64, 64, 64)	0	conv2d_4[0][0]
conv2d_5 (Conv2D)	(None, 64, 64, 128)	73856	max_pooling2d_2[0][0]
conv2d_6 (Conv2D)	(None, 64, 64, 128)	147584	conv2d_5[0][0]
max_pooling2d_3 (MaxPooling2D)	(None, 32, 32, 128)	0	conv2d_6[0][0]
conv2d_7 (Conv2D)	(None, 32, 32, 256)	295168	max_pooling2d_3[0][0]
conv2d_8 (Conv2D)	(None, 32, 32, 256)	590080	conv2d_7[0][0]
max_pooling2d_4 (MaxPooling2D)	(None, 16, 16, 256)	0	conv2d_8[0][0]
conv2d_9 (Conv2D)	(None, 16, 16, 512)	1180160	max_pooling2d_4[0][0]
conv2d_10 (Conv2D)	(None, 16, 16, 512)	2359808	conv2d_9[0][0]
conv2d_transpose_1 (Conv2DTrans	(None, 32, 32, 256)	524544	conv2d_10[0][0]
concatenate_1 (Concatenate)	(None, 32, 32, 512)	0	conv2d_transpose_1[0][0] conv2d_8[0][0]
conv2d_11 (Conv2D)	(None, 32, 32, 256)	1179904	concatenate_1[0][0]
conv2d_12 (Conv2D)	(None, 32, 32, 256)	590080	conv2d_11[0][0]
conv2d_transpose_2 (Conv2DTrans	(None, 64, 64, 128)	131200	conv2d_12[0][0]
concatenate_2 (Concatenate)	(None, 64, 64, 256)	0	conv2d_transpose_2[0][0] conv2d_6[0][0]
conv2d_13 (Conv2D)	(None, 64, 64, 128)	295040	concatenate_2[0][0]
conv2d_14 (Conv2D)	(None, 64, 64, 128)	147584	conv2d_13[0][0]
conv2d_transpose_3 (Conv2DTrans	(None, 128, 128, 64)	32832	conv2d_14[0][0]
concatenate_3 (Concatenate)	(None, 128, 128, 128)	0	conv2d_transpose_3[0][0] conv2d_4[0][0]
conv2d_15 (Conv2D)	(None, 128, 128, 64)	73792	concatenate_3[0][0]
conv2d_16 (Conv2D)	(None, 128, 128, 64)	36928	conv2d_15[0][0]
conv2d_transpose_4 (Conv2DTrans	(None, 256, 256, 32)	8224	conv2d_16[0][0]
concatenate_4 (Concatenate)	(None, 256, 256, 64)	0	conv2d_transpose_4[0][0] conv2d_2[0][0]
conv2d_17 (Conv2D)	(None, 256, 256, 32)	18464	concatenate_4[0][0]
conv2d_18 (Conv2D)	(None, 256, 256, 32)	9248	conv2d_17[0][0]
conv2d_19 (Conv2D)	(None, 256, 256, 2)	66	conv2d_18[0][0]

Figura 21. Arquitectura del modelo UNet Keras implementado en este proyecto, presentada con la función de Keras model.summary().

En la red U-Net implementada en este trabajo, y de la misma manera que la del artículo original, las convoluciones (3×3) se aplican repetidamente a la imagen. Además, se usa la convolución con relleno 'same', es decir, que la salida tiene la misma longitud que la entrada.

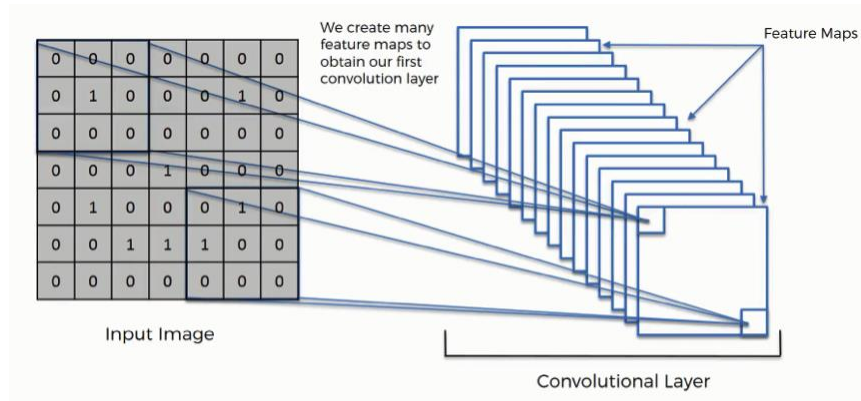


Figura 22. Estructura de capa convolucional con varios mapas de características. Imagen obtenida de [36].

```
conv1 = Conv2D(32, (3, 3), activation='relu', padding='same')(inputs)
conv1 = Conv2D(32, (3, 3), activation='relu', padding='same')(conv1)
...
conv2 = Conv2D(64, (3, 3), activation='relu', padding='same')(pool1)
conv2 = Conv2D(64, (3, 3), activation='relu', padding='same')(conv2)
...
```

Función de Activación

La salida que nos da la convolución es una transformación lineal de la entrada, pero necesitamos añadir no linealidad a la red para poder resolver problemas que no sean exclusivamente linealmente separables. Esta no linealidad la introducimos a través de funciones de activación, que se aplica después de cada convolución.

Tanto en las redes neuronales artificiales como biológicas, una neurona no sólo transmite la entrada que recibe. Existe un paso adicional, una función de activación, que es análoga a la tasa de potencial de acción disparando en el cerebro. La función de activación se encarga de devolver una salida a partir de un valor de entrada [37].

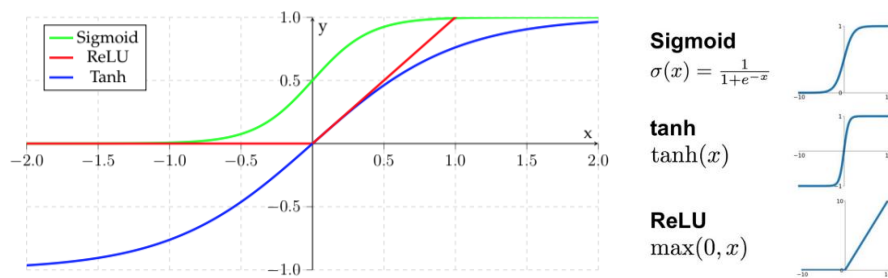


Figura 23. Comparativa de las tres funciones de activación más populares.

Hay varios tipos de función de activación, pero en este caso hemos usado ReLU, excepto en la última capa de salida que se ha usado la función Softmax. Se explican sus características para justificar su uso [5]:

- **ReLU:** La función ReLU transforma los valores introducidos anulando los valores negativos y dejando los positivos tal y como entran (sólo se activan si son positivos). No está acotada, se comporta muy bien con imágenes y tiene muy buen desempeño en redes neuronales convolucionales.

```
conv1 = Conv2D(32, (3, 3), activation='relu', padding='same')(inputs)
```

- **Softmax:** La función Softmax es una generalización de Sigmoid, pero existen diferencias entre ellas. Generalmente, usamos la activación de Softmax, en lugar de Sigmoid, con la función de pérdida de entropía cruzada (sección 4.4.1) porque la activación de Softmax distribuye la probabilidad en cada nodo de salida, para la clasificación de varias clases.

Su principal ventaja es que el rango de probabilidades de salida es de 0 a 1. Transforma las salidas a una representación en forma de probabilidad, de tal manera que el sumatorio de todas las probabilidades de las salidas es 1. En este caso, devuelve las probabilidades de cada clase (célula o fondo), para cada píxel. La suma de probabilidades de las dos clases para ese píxel es 1, y la clase objeto tendrá la probabilidad más alta que la otra.

Esta función de activación solo se hemos aplicado en la última capa para conseguir un vector discreto de distribución de probabilidad en las dos capas de la imagen de salida. Actúa como un normalizador tipo multiclase.

```
conv10 = Conv2D(2, (1, 1), activation='softmax')(conv9)
```

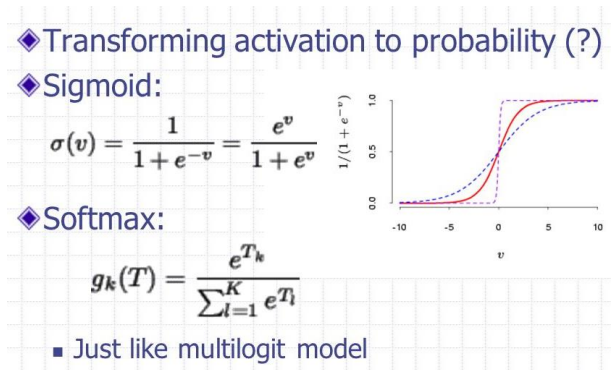


Figura 24. Comparativa de las funciones de activación Sigmoid y Softmax. Imagen obtenida de [38]

Capas de Submuestreo (MaxPooling)

El submuestreo, o capas de "agrupación" se suelen colocar después de las capas convolucionales, principalmente para reducir la dimensionalidad del mapa de características y así mejorar la eficiencia computacional, que a su vez puede mejorar el rendimiento real. El tipo principal de capa de agrupación en uso hoy en día es una capa de "agrupación máxima", en la que el mapa de características se minimiza de tal manera que se mantiene la respuesta máxima de la característica dentro de un tamaño de muestra determinado [39]. Esto contrasta con la agrupación promedio, en la que básicamente reduce la resolución promediando un grupo de píxeles.

La agrupación máxima tiende a funcionar mejor porque responde más a los núcleos que están "encendidos" o responden a patrones detectados en los datos.

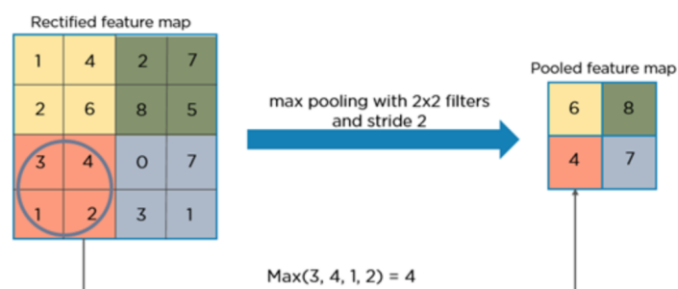


Figura 25. Ejemplo de operación MaxPooling

En términos generales, cuando buscamos una característica en una imagen, la posición absoluta no es relevante, siendo lo más importante la posición relativa a otras características. Este razonamiento está detrás de las capas de agrupación. La agrupación reduce el tamaño espacial de la representación, haciendo que el número de parámetros que se aprendan sea menor, así como el costo computacional.

```
pool1 = MaxPooling2D(pool_size=(2, 2))(conv1)
```

Capas de Convolución Traspuesta (Conv2DTranspose)

Es como una capa que combina las capas UpSampling2D y Conv2D en una sola capa. Tiene como objetivo principal duplicar la resolución de los datos de entrada y reducir a la mitad el número de mapas de características. Este tipo de capas se aplican en la ruta expansiva.

```
up6 = concatenate([Conv2DTranspose(256, (2, 2), strides=(2, 2), padding='same')(conv5),  
conv4], axis=3)
```

Capas de Concatenación

La capa de concatenación une dos mapas de características por el eje dado. Como veremos en la sección de Estructura 4.2.2, estas capas las utilizamos en la parte decodificadora o de expansión, con el fin de ir concatenando las características obtenidas antes de la pérdida de información producida por la contracción y las más procesadas después de varias convoluciones y activaciones.

4.2.2. Estructura

Como ya se ha citado, U-Net se parece a una 'U', que justifica su nombre, utiliza un modelo de red totalmente convolucional FNC y se trata de una arquitectura encoder-decoder. Este tipo de arquitecturas consta de dos bloques: ruta de contracción y ruta de expansión, dónde se van repitiendo los pasos mencionados en la sección anterior (convolución, agrupación, convolución traspuesta y concatenación) [35] .

Ruta de contracción

La ruta de contracción se utiliza para extraer e interpretar el contexto de la imagen de entrada (qué). Comprime la información. Esta información contextual general se transferirá a la ruta de expansión a través de concatenaciones.

La ruta de contracción sigue la arquitectura típica de una red convolucional. Consiste en aplicar repetidamente dos convoluciones 3x3, cada una seguida de una función de activación ReLU y una operación de agrupación máxima 2x2 para reducir la resolución [40] .

- 3x3 Capa de convolución + función de activación ReLU
- 3x3 Capa de convolución + función de activación ReLU
- 2x2 Capa de Agrupación. Agrupación máxima 2x2

Tanto en la red UNet original, como en nuestra implementación, este patrón se repite cuatro veces. Con cada uno de estos bloques se reduce la resolución y se duplica el mapa de características, para que la arquitectura pueda aprender estructuras más complejas de forma efectiva.

Cómo vemos, comienza con la imagen de entrada [256, 256, 1] y finaliza con un vector [16, 16, 256]. Por lo tanto, en esta ruta se reduce el tamaño de la imagen y aumenta el número de filtros en cada capa convolucional.

La ruta de contracción funciona de forma similar a un codificador, por eso existe un tipo de RNC, denominada red de codificador, formada únicamente por esta sección. Este tipo de redes tiene como objetivo sintetizar la información de la imagen de entrada en un vector unidimensional, como por ejemplo un sistema que toma como entrada una imagen que representa un número y la salida es el número representado.

En nuestro caso no queremos saber qué hay en nuestra imagen, sino dónde está situado. Queremos como salida una imagen segmentada y para ello continuamos con la ruta de expansión.

Ruta Central

Es el “valle” de la red, se encuentra entre las rutas de contracción y expansión. Es decir, la entrada a este bloque llega de la ruta de contracción, y la salida de este bloque es el inicio de la ruta de expansión. En este paso se realizan únicamente dos convoluciones iguales a las de la ruta de contracción, seguidas de la función de activación ReLU.

- 3x3 Capa convolucional + función de activación ReLU
- 3x3 Capa convolucional + función de activación ReLU

Aquí se duplican por última vez el número de mapas de características, acabando con 512, en nuestro caso. El vector resultado, como entrada a la red de expansión quedaría [16, 16, 512]

Ruta de expansión simétrica

La ruta de expansión simétrica es similar a un decodificador, que permite una localización precisa (dónde). Este paso se realiza para conservar la información de límites (información espacial) a pesar del submuestreo realizado en la etapa del codificador. Una parte de la red decoder (concatenación) se centra únicamente en utilizar las características extraídas por el encoder, para generar la segmentación más precisa posible. Con esto conseguimos una localización precisa combinada con la información contextual de la ruta de contracción.

Esta ruta consiste en aplicar repetidamente, en mi caso, una convolución traspuesta que duplica la resolución de la imagen y además reduce a la mitad el número de mapas de características. Seguido de una concatenación con los mapas de características del mismo nivel de la ruta de contracción, lo que vuelve a duplicar el número de mapas de características, y dos convoluciones 3x3 seguidas de ReLU.

- 2x2 Capa convolucional traspuesta, con stride 2
- Concatenación con el mapa de características correspondiente de la ruta de contracción
- 3x3 Capa convolucional + función de activación ReLU
- 3x3 Capa convolucional + función de activación ReLU

Igual que en la ruta de contracción, este patrón se repite cuatro veces. Pero al contrario de esa ruta, con cada bloque se duplica la resolución y se reducen a la mitad los mapas de características, con el objetivo de llegar a una imagen de salida con la misma resolución que la de entrada.

Después de estas cuatro repeticiones, en la capa final se aplica una convolución 1x1 para mapear cada vector de características de 32 componentes, al número deseado de clases, en mi caso dos (célula y fondo).

- 1x1 Capa convolucional + función de activación Softmax

En la sección 4.2.1 se explica con detalle por qué se utiliza la función Sotmax en esta última convolución.

4.3. Compilación del Modelo

Una vez definida la arquitectura de red neuronal convolucional y su funcionamiento, se llevará a cabo un proceso de aprendizaje para enseñar a la red cómo resolver la segmentación.

En esta sección vamos a detallar la configuración de este proceso de aprendizaje antes de entrenarlo. En la Sección 3.1.2 se explica el método compile aplicado al modelo de Keras, ahora vamos a explicar cómo actúan esas funciones en mi red, entendiendo su funcionamiento y aplicación.

```
model.compile(optimizer='Adam',loss='categorical_crossentropy',metrics=[dice_coef])
```

4.3.1. Función de Pérdida

La función de pérdida devuelve la diferencia, el error, entre la predicción (máscara que ha generado la red) y la verdad fundamental (máscara proporcionada en el conjunto de imágenes) en cada lote. Recordemos que estamos en un entorno de aprendizaje supervisado y disponemos de la máscara/etiqueta que nos indica el valor esperado.

Queremos que estas dos máscaras se parezcan, que nuestro error sea cero, por lo que queremos que el modelo minimice el valor de esta función. Un modelo perfecto tendría un valor de pérdida 0. A medida que se entrena el modelo, se van ajustando y optimizando, con el optimizador, los parámetros de la red (los pesos de las interconexiones de las neuronas, en este caso los valores de los filtros de las convoluciones) de manera automática, con el objetivo de reducir esta función hasta obtener buenas predicciones.

Existen varias funciones de pérdida diferentes, para este proyecto la función de pérdida que se calcula después de cada lote es la entropía cruzada categórica. Se define en la Figura 26, donde c denota el índice de clase, N es el número de clases, y_c es 1 para clase anotada y 0 en caso contrario, y z_c es la salida de la red para cada clase C .

$$Loss(y, z) = - \sum_{c=1}^N y_c \log\left(\frac{e^{z_c}}{\sum_{c=1}^N e^{z_c}}\right)$$

Figura 26. Definición de función de pérdida de entropía cruzada categórica.

La entropía cruzada categórica, es una medida de precisión para variables categóricas, como es el caso de este proyecto, en el que en la salida tenemos dos etiquetas posibles, dos probabilidades, célula y fondo. Variables binarias.

La función de pérdida de entropía cruzada mide el rendimiento de un modelo de clasificación cuyo resultado es un valor de probabilidad entre 0 y 1 [41] . En nuestro caso esta clasificación se aplica a cada píxel, por ello previamente usamos *to_categorical*, en el preprocesamiento, para darle dos valores a cada píxel y conseguir las probabilidades de que cada píxel pertenezca a una etiqueta (célula o fondo) con la función de activación Softmax de nuestra arquitectura.

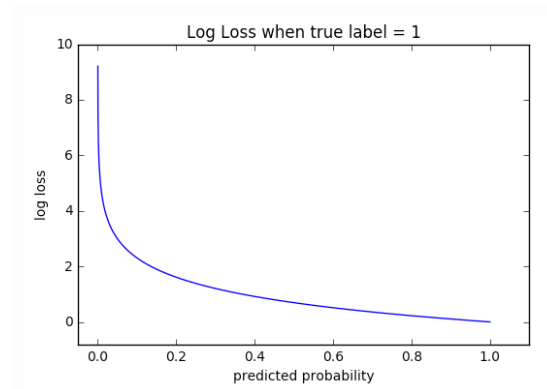


Figura 27. Rango de posibles valores de pérdida de entropía cruzada, dada una observación real. Imagen obtenida de [41].

Aunque estemos realizando una segmentación, cada píxel debe permanecer a una categoría/clase, así que hemos convertido un problema de segmentación en uno de clasificación binaria. Esto es lo que se llama segmentación semántica.

4.3.2. Optimizador

El optimizador se encarga de minimizar el error (valor que nos devuelve la función de pérdida) en cada iteración, actualizando los valores de los filtros convolucionales.

Inicialmente, cuando se intentan procesar imágenes utilizando una red que no ha sido entrenada, se obtendrán resultados aleatorios, ya que los valores de los filtros convolucionales

no están configurados. Durante el entrenamiento, cada imagen se enviará a la red convolucional, que la transformará en una predicción. Luego, esta salida se comparará con la anotación proporcionada con la muestra de entrada utilizando una función de pérdida. Una vez que la salida de la red se compara con la deseada, los parámetros deben actualizarse para que la próxima vez que los datos de entrada se envíen a la red, la salida esté más cerca del objetivo.

En este caso utilizamos el optimizador Adam [27], algoritmo de optimización estocástica. Es computacionalmente eficiente, tiene pocos requisitos de memoria y adecuado para problemas grandes en términos de datos y/o parámetros. También adecuado en estrategias de entrenamiento tipo estocásticas o por lotes, como en el caso de Redes Neuronales Profundas [42] .

La optimización de Adam es un método de descenso de gradiente estocástico que se basa en la estimación adaptativa de momentos de primer y segundo orden [26].

4.3.3. Métrica

Una métrica es una función que se utiliza para evaluar el rendimiento del modelo durante el entrenamiento y las pruebas. Es similar a una función de pérdida, a diferencia que los resultados de evaluar una métrica no se utilizan para entrenar el modelo. Es decir, los valores que devuelve esta función no son relevantes para actualizar y optimizar los filtros convolucionales.

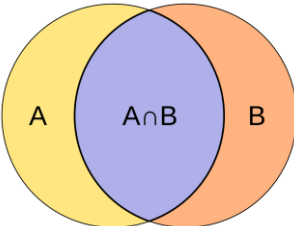
$$QS = \frac{2C}{A+B} = \frac{2|A \cap B|}{|A| + |B|}$$


Figura 28. Definición de función de métrica Coeficiente Dice.

En este caso usamos el **Coeficiente Dice**, para evaluar cuánto se parece el resultado predicho y el resultado real. Es decir, cuanto se solapa la máscara que ha creado la red neuronal y la máscara original del conjunto de datos.

Esta métrica se calcula comparando la concordancia de píxeles entre la verdad básica (A) y su segmentación predicha correspondiente (B).

```
def dice_coef(y_true, y_pred):
    y_true = y_true[:, :, 1]
    y_pred = y_pred[:, :, 1]

    y_true_f = K.flatten(y_true)
    y_pred_f = K.flatten(y_pred)
    intersection = K.sum(y_true_f * y_pred_f)

    return (2. * intersection + smooth) / (K.sum(y_true_f) + K.sum(y_pred_f) + smooth)
```

4.4. Entrenamiento del Modelo

Una vez se ha definido la arquitectura de red y su configuración para el proceso de aprendizaje, procedemos a iniciar el entrenamiento. En esta sección vamos a explicar algunos conceptos importantes a la hora de iniciar el entrenamiento de un modelo, como son los hiper parámetros, o el cómo voy a utilizar los datos de entrada.

La red se entrena gracias al método `.fit()`, que ya se ha descrito en la Sección 3.1.2. El uso de este método se puede ver en el siguiente código.

Durante el entrenamiento el callback ‘ModelCheckPoint’ genera un archivo que contiene los pesos del modelo, guardando los mejores pesos obtenidos.

```
model_checkpoint = ModelCheckpoint('weights.h5', monitor='val_loss',
save_best_only=True, mode='min')

model.fit(x=imgs_train, y=imgs_mask_train,
          batch_size=BATCH_SIZE,
          epochs=NB_EPOCH,
          verbose=2,
          callbacks=[model_checkpoint, tensorboard],
          validation_data=(imgs_validation, imgs_mask_validation))
```

4.4.1. Hiper parámetros

El entrenamiento de la red consiste en la acción de minimizar la función de pérdida con el optimizador, ambos conceptos definidos en la sección anterior. El modelo propuesto UNet lo entrenaremos desde cero.

Para optimizar el entrenamiento se usan los parámetros estándar e hiper parámetros. Los parámetros estándar (los pesos) se calculan a durante el entrenamiento del modelo. Mientras que los hiper parámetros se diferencian porque definen conceptos de nivel superior del modelo y su valor se establece antes de que comience el proceso de aprendizaje no pudiendo ser modificados automáticamente durante dicho proceso, es decir, están predefinidos. Se emplean para parametrizar el proceso de instanciación del modelo, es decir, describir la configuración del modelo. Influyen en la capacidad y características de aprendizaje, aunque no se usan para modelar los datos directamente [43].

El modo de decidir los valores de los hiper parámetros es mediante prueba-error, eligiendo el que muestre mejores resultados. Si el modelo no se adapta a los datos de entrada se modifican los hiper parámetros, entrenando nuevamente y volviendo a evaluar hasta obtener los resultados deseados.

Estos valores ya predefinidos, se le pasan en la función de entrenamiento `.fit()`:

Batch size: El tamaño del lote. Se trata del número de imágenes que pasan por la red antes de actualizar los pesos. Se pueden particionar los datos de entrenamiento en mini lotes para pasarlos por la red.

Se puede entrenar con un tamaño de lote igual o menor al número total de imágenes de entrenamiento. Es recomendable elegir un tamaño de lote que sea tan grande como sea posible sin salir de la memoria.

Epochs: Una época es cuando el conjunto de imágenes de entrenamiento completo se pasa hacia adelante y hacia atrás a través de la red , sólo una vez. Si tenemos 20 imágenes de entrenamiento y un batch size de 10, se necesitarán dos iteraciones para completar una época.

Al final de cada época se realiza una evaluación del modelo con los datos de validación, y además existe la posibilidad de ejecutar callbacks.

Se utiliza para separar el entrenamiento en distintas fases, lo que es útil para el registro y la evaluación periódica.

Actualizar los pesos con una sola pasada o una época no es suficiente (underfitting), igual que demasiadas epochs también pueden producir un efecto negativo (overfitting), como se observa en la Figura 29.

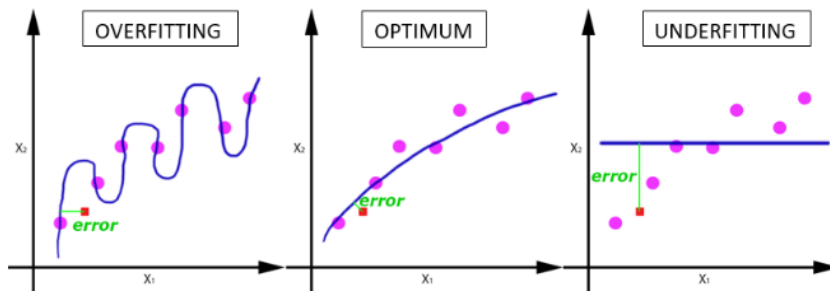


Figura 29. Curva de entrenamiento que muestra cómo se ajusta el modelo a los datos de entrenamiento.

Para detectar cuál es el número óptimo de epochs a definir, una buena pista es incrementar el número de epochs hasta que la pérdida (loss) empiece a aumentar con los datos de validación, incluso cuando la pérdida de los datos de entrenamiento sigue decreciendo. Aquí se estaría detectando un overfitting. Este caso se tratará en el Capítulo 5 de Experimentos.

4.4.2. Subconjuntos de Datos

Para que el modelo pueda aprender y hacer predicciones sobre los datos, se deben seguir varios pasos durante el entrenamiento y la prueba, antes de que esté disponible para su uso.

- Paso 1: Hacer que el modelo examine las imágenes de muestra.
- Paso 2: Hacer que el modelo aprenda de sus errores.
- Paso 3: Llegar a una conclusión sobre el rendimiento del modelo.

Dado que estos pasos son bastante diferentes, los datos en cada uno de ellos se tratarán de manera diferente. Por lo tanto, vamos a subdividir el conjunto de datos en tres subconjuntos:

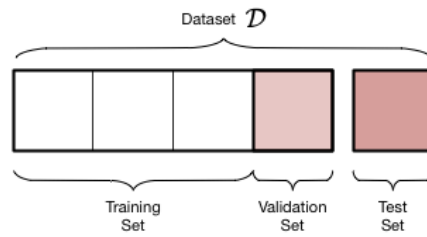


Figura 30. Ejemplo de subdivisión de un conjunto de 5 muestras

- **Train set.** Subconjunto de datos de entrenamiento.

Este conjunto de datos corresponde al Paso 1. Incluye el conjunto de imágenes de entrada a las que el modelo se ajustará, ajustando los parámetros (los pesos). Son las imágenes con las que se construye el modelo.

- **Validation set.** Subconjunto de datos validación.

El modelo se va evaluando periódicamente (Paso 2) durante el entrenamiento, después de cada época, para seguir ajustando los pesos correctamente. Dicha evaluación periódica se realiza con las imágenes de validación.

Se calcula el error producido en las imágenes de validación (son imágenes nuevas, con las que no se ha entrenado), así se sabe cómo de preciso es el modelo y se ajustan los pesos en función de este resultado.

La validación se emplea generalmente para detener el entrenamiento cuando se cumplen ciertos criterios, evitando el sobreajuste. Me dice donde tengo que parar de aprender.

- **Test set.** Subconjuntos de datos de prueba.

La forma de evaluar la precisión del modelo es ejecutarlo en un conjunto de imágenes ‘nuevas’, que no haya visto antes, una vez que ya ha finalizado el entrenamiento. Lo que corresponde a una evaluación final (Paso 3). Este paso es crítico para probar la generalización del modelo.

Al no exponer el modelo a este conjunto de pruebas hasta que finalice la fase de entrenamiento, podemos considerar que la medida de precisión final es fiable. Permite estimar el desempeño de la red en imágenes futuras.

Para este proyecto se dispone de un conjunto de datos con un número muy limitado de imágenes de entrenamiento (ver Sección 3.2). el cual no contiene un conjunto de datos de prueba específica. Por lo tanto, dentro del conjunto se realiza la subdivisión de la siguiente forma: Imágenes de entrenamiento (70%), Imágenes de validación (20%), Imágenes de test (10%). Al disponer de pocas imágenes, optamos por escoger pocas imágenes de test y realizar la validación cruzada (ver Sección 4.5).

4.4.3. Aumento de Datos

El preprocesamiento de imágenes es un factor clave en todas las aplicaciones de visión artificial y especialmente, se puede usar para evitar el sobreajuste mediante el aumento de datos [44] .

En Deep Learning, está demostrado que cuanto mayor es el tamaño de los datos de entrenamiento, mejor será el rendimiento de la red neuronal. Pero en ocasiones, no se dispone de tal cantidad de imágenes, como es el contexto de la imagen biomédica. Dicha asequibilidad requiere tiempo, dinero y hardware. No es fácil recopilar grandes cantidades de imágenes, además el proceso de etiquetado requiere mucha experiencia, ya que se necesita información de clase para cada píxel.

Al entrenar con un conjunto de datos tan pequeño, la red puede acabar memorizando las imágenes de entrada si estamos entrenando demasiado. provocando sobreajuste, cuando el modelo, expuesto a muy pocos ejemplos, aprende patrones que no se generalizan a nuevas imágenes.

La regularización es muy utilizada en Deep Learning, se trata introducir métodos/modificaciones durante el entrenamiento del modelo para evitar que se ajuste demasiado a las imágenes de entrenamiento y que no generalice lo suficientemente bien.

Existen varios métodos de regularización. En este proyecto se utiliza el Aumento de Datos, como solución a este problema. Consiste en generar más datos de entrenamiento aplicando transformaciones/deformaciones de forma aleatoria a las imágenes del set de entrenamiento originales, cada vez que se introducen en la red. Así, se enseña a la red las propiedades de invarianza y robustez que necesitamos.

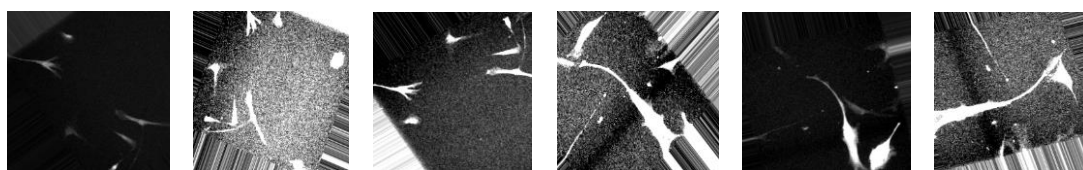


Figura 31. Ejemplos de algunas muestras generadas con Keras en el conjunto de datos utilizado.

Para este proyecto, aplicamos el aumento de datos mediante el generador proporcionado por Keras con la clase `ImageDataGenerator` (ver Sección 3.1.2), que genera lotes de datos aplicando las transformaciones deseadas, a tiempo real durante el entrenamiento. Aplicando la misma deformación a las imágenes y a sus máscaras correspondientes. El generador se repite hasta que el número de imágenes por época y el número de epochs establecidas se cumplen.

Este generador permite aplicar diversas modificaciones a la imagen. Pero hay que estudiar qué deformaciones y técnicas son óptimas para este caso. Hay que buscar un balance que no perturbe demasiado las imágenes para obtener, al menos, los mismos resultados de validación que sin el aumento [45] .

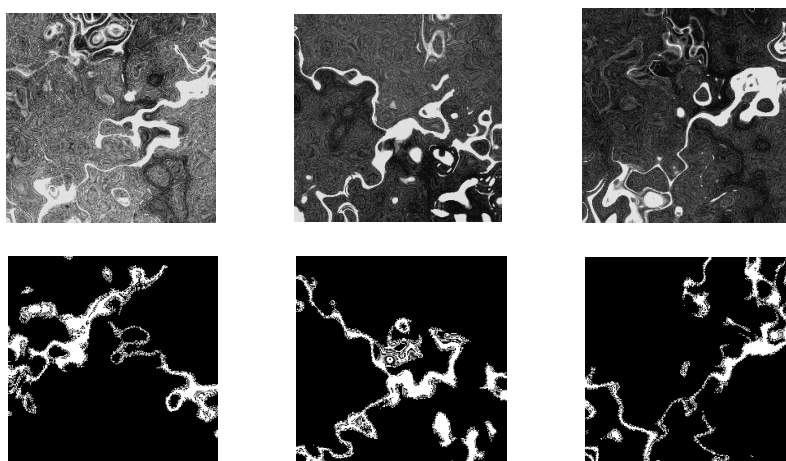


Figura 32. Ejemplo de muestras demasiado deformadas, generadas con Aumento de Datos. Arriba las imágenes de células, abajo sus respectivas anotaciones.

En el caso de imágenes microscópicas, se necesita fundamentalmente la invariancia de desplazamiento y rotación, así como la robustez de las deformaciones y las variaciones de los valores de gris. Particularmente las deformaciones elásticas de las imágenes de entrenamiento

parecen ser el concepto clave para formar una red de segmentación con muy pocas imágenes anotadas [40]. Además, la deformación es la variación más común en el mundo de la imagen biomédica y el tejido real, y puede ser simulado fácilmente mediante transformaciones. Teniendo esto en cuenta, las deformaciones que se aplican para este proyecto son:

- Rotación aleatoria
- Desplazamiento horizontal
- Desplazamiento vertical
- Volteo
- Zoom
- Variaciones de valores de gris (esto sólo se le aplica a la imagen, no a la notación)

Las muestras generadas por el siguiente código se pueden ver en la Figura 31.

```
model.fit ( data_generator_keras ( x = imgs_train, y = imgs_mask_train,  
batch_size=BATCH_SIZE, full_dataset = imgs, flip=True),  
epochs=NB_EPOCH,  
steps_per_epoch=2,  
verbose=2,  
callbacks=[model_checkpoint, tensorboard],  
validation_data=(imgs_validation,imgs_mask_validation))
```

```
def data_generator_keras(x, y, batch_size, full_dataset, flip):  
    datagen_x = ImageDataGenerator(  
        featurewise_center=True,  
        featurewise_std_normalization=True,  
        rotation_range=90,  
        width_shift_range=0.2,  
        height_shift_range=0.2,  
        shear_range=0.2,  
        zoom_range=0.2,  
        horizontal_flip=flip,  
        vertical_flip=flip,  
        fill_mode='nearest',  
        preprocessing_function=equal_hist)
```

```

datagen_y = ImageDataGenerator(
    rotation_range=90,
    width_shift_range=0.2,
    height_shift_range=0.2,
    shear_range=0.2,
    zoom_range=0.2,
    horizontal_flip=flip,
    vertical_flip=flip,
    fill_mode='nearest')
if full_dataset is not None:
    datagen_x.fit(full_dataset)
for batch in itertools.zip_longest(
    datagen_x.flow(x, batch_size=batch_size, seed=1337),
    datagen_y.flow(y, batch_size=batch_size, seed=1337)
):
    yield batch

```

4.5. Validación Cruzada

Hay que tener en cuenta que si solo se realiza una única partición de datos train-test, es decir, evalúo el modelo ya entrenado únicamente con un subconjunto de imágenes test, se puede estar produciendo un mal entrenamiento sin ser detectado, por ejemplo, un sobreajuste.

Por ello, utilizamos la técnica de Validación Cruzada, para evaluar (ver Sección 4.6) la calidad del modelo. Evalúa los resultados de las imágenes test y garantiza que son independientes de la partición entre las imágenes de train-test. Consiste en repetir y calcular la media aritmética obtenida de las medidas de evaluación sobre diferentes particiones (diferentes subconjuntos de imágenes test) [46].

Es una manera de predecir el ajuste de un modelo a un hipotético conjunto de datos test cuando no se dispone de ese conjunto explícito.

Si el resultado de la media es similar al de cada iteración, significa que se está produciendo un entrenamiento óptimo.

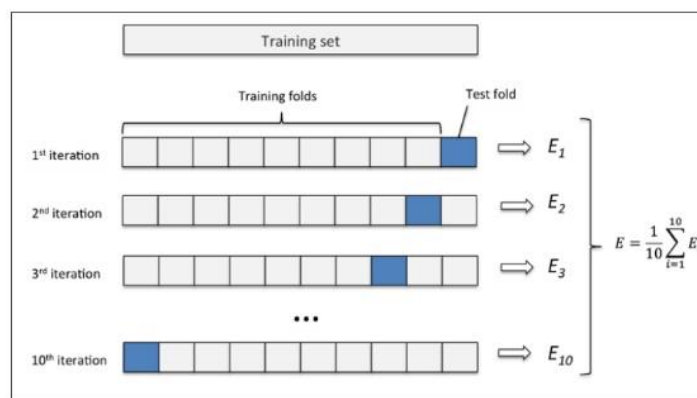


Figura 33. Representación de Validación Cruzada K-Fold

Para este proyecto se pone en práctica la Validación Cruzada de K iteraciones, escogiendo en cada iteración una o varias imágenes distintas de test. Por lo tanto, si por ejemplo tenemos un total de 11 imágenes, se entrenan 11 modelos desde cero, y cada modelo será evaluado con una imagen test diferente.

```
from sklearn.model_selection import KFold

kf = KFold(n_splits=len(imgs))

for train_index, test_index in kf.split(imgs):
    ...
    ...
```

4.6. Evaluación del Modelo

Una vez que se ha entrenado el modelo y sus pesos quedan almacenados en el archivo 'weights.h5', se puede evaluar el rendimiento del modelo con los métodos que ofrece Keras (ver Sección 3.1.2). El método `.evaluate()` toma el conjunto de datos de prueba (test) y calcula la pérdida de registro y la precisión. El método `.predict()` devuelve la etiqueta predicha por el modelo para las imágenes de prueba.

```
model.load_weights('weights.h5')

imgs_mask_test_evaluate = model.evaluate(imgs_test, imgs_mask_test, verbose=1)

imgs_mask_test_predict = model.predict(imgs_test, verbose=1)
```

Capítulo 5

Resultados

Este capítulo muestra y analiza los resultados obtenidos siguiendo los experimentos realizados durante el desarrollo del proyecto. Además, resume el entorno utilizado y la metodología que se ha seguido para el entrenamiento y evaluación de la red.

La red se ha desarrollado utilizando Keras como núcleo principal, con el apoyo de Tensorflow (ver Capítulo 2). El conjunto de datos utilizado para realizar el entrenamiento y los experimentos ha sido Fluo-C2DL-MSC (ver Sección 3.2), y se ha dividido al azar en dos conjuntos, uno para el entrenamiento y otro para la validación, que representan el 80% y el 20% del conjunto de datos original, respectivamente (ver Sección 4.4.2). En todo el proceso de entrenamiento se ha utilizado la entropía cruzada categórica como función de pérdida (ver Sección 4.3.1), Coeficiente Dice como métrica (ver Sección 4.3.3) y el optimizador Adam para actualizar los pesos del modelo (ver Sección 4.3.2). El tamaño de lote utilizado en todos los experimentos ha sido 15.

5.1 Experimentos y evaluaciones

En primer lugar, se entrenaron varios modelos en las mismas condiciones (30 imágenes de entrenamiento sin aumento de datos, 6 imágenes de validación y 3 imagen de prueba, con validación cruzada para la evaluación), pero modificando el número de epochs de entrenamiento.

Se observa cómo con pocas epochs (100), el modelo no ha entrenado lo suficiente, porque los resultados no han dejado de mejorar, por lo que se han seguido aumentando el número de epochs a 500 y 700.

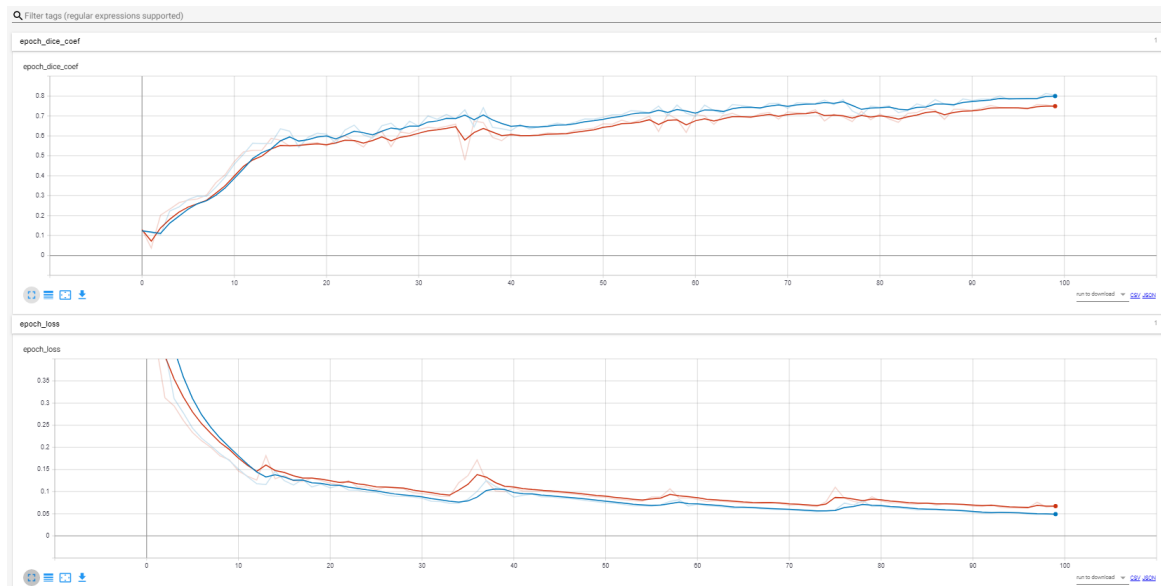


Figura 34. Función de pérdida (abajo) y métrica (arriba) para los datos de entrenamiento y validación. 100 epochs y sin aumento de datos (Azul: datos de entrenamiento, Rojo: datos de validación).

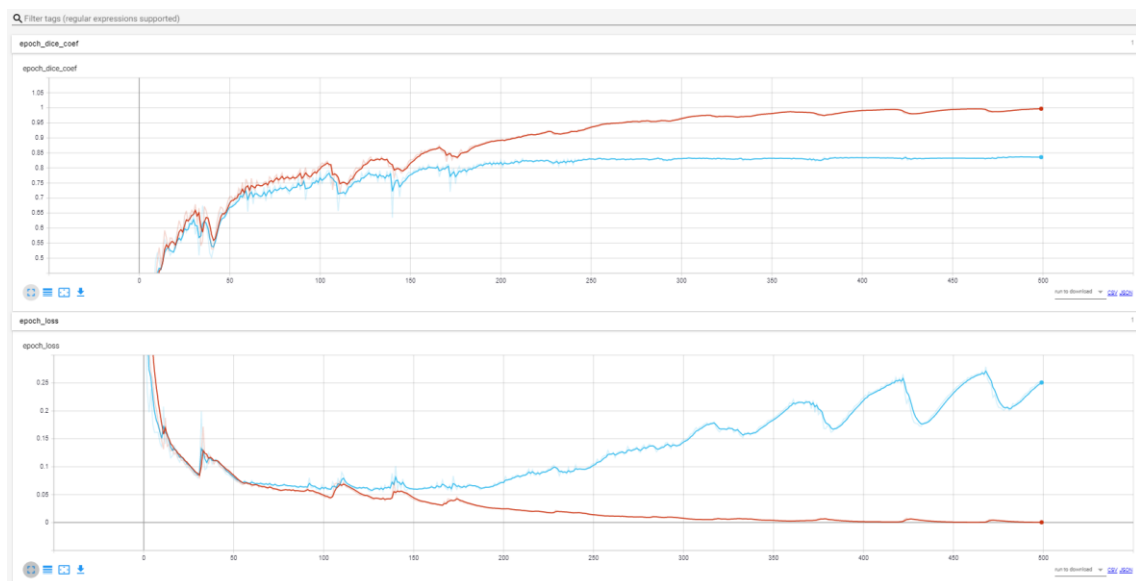


Figura 35. Función de pérdida (abajo) y métrica (arriba) para los datos de entrenamiento y validación. 500 epochs y sin aumento de datos (Rojo: datos de entrenamiento, Azul: datos de validación).

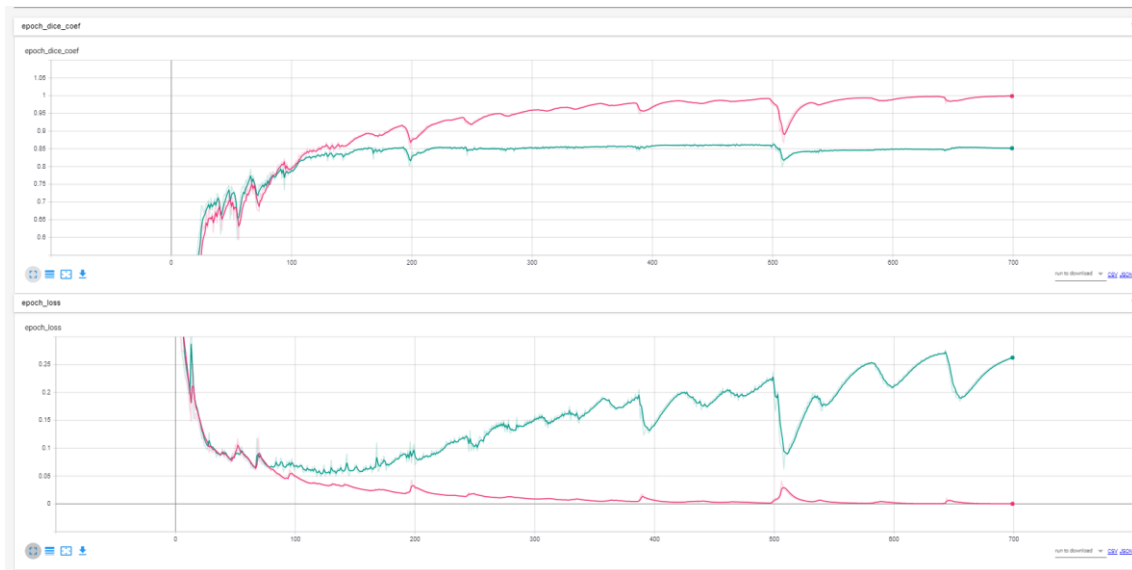


Figura 36. Función de pérdida (abajo) y métrica (arriba) para los datos de entrenamiento y validación. 700 epochs y sin aumento de datos (Rosa: datos de entrenamiento, Verde: datos de validación)

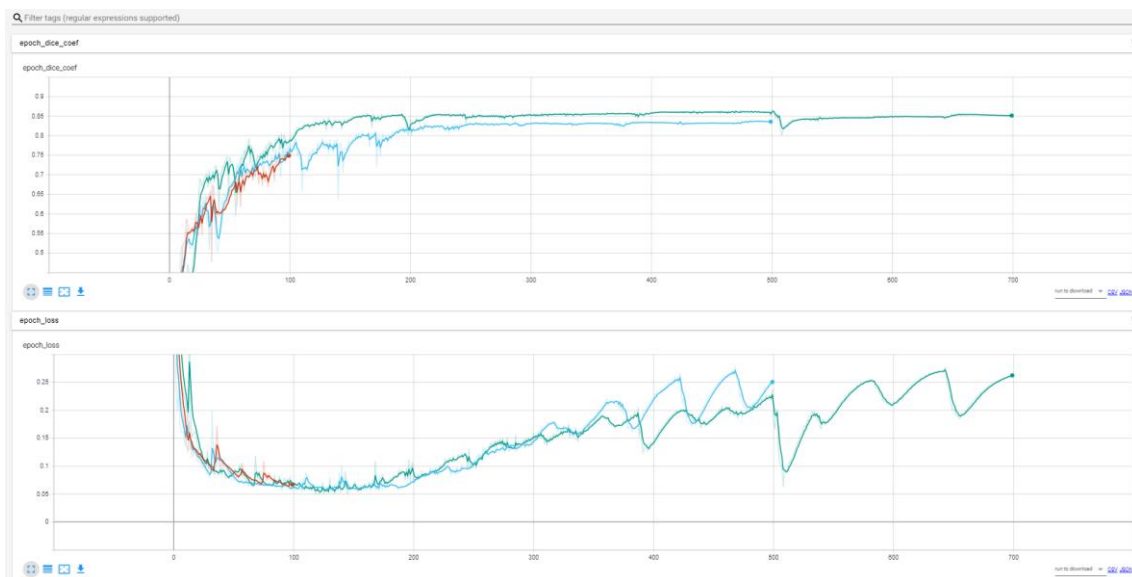


Figura 37. Función de pérdida (abajo) y métrica (arriba) para los datos de validación. 100 epochs (Rojo), 500 epochs (Azul) y 700 epochs (Verde), sin aumento de datos.

En las figuras 35 y 36 se puede ver cómo a partir de la epoch número 100-150, aproximadamente, el modelo comienza a sobreajustarse. Los resultados con las imágenes de entrenamiento no dejan de mejorar, pero con las imágenes de validación se mantienen o empeoran. Los pesos se ajustan demasiado a las imágenes de entrenamiento.

Este comportamiento no es un problema en este caso, ya que las imágenes de entrenamiento, validación y pruebas son muy parecidas. Por lo que, si los pesos se ajustan mucho, también nos dará buenos resultados en la evaluación con las imágenes de prueba como se ve en la Figura 38.

	Loss (Mean)	Coef. Dice (Mean)
100 epoch	0,0691	0,7565
500 epoch	0,0853	0,7665
700 epoch	0,0707	0,7983

Figura 38. Media de los resultados en las imágenes de prueba (test), utilizando validación cruzada.

En la Figura 39 se presentan algunas predicciones generadas por los distintos modelos, y se puede ver cómo a mayor número de epochs, más se acercan a las máscaras originales de nuestro conjunto de datos.

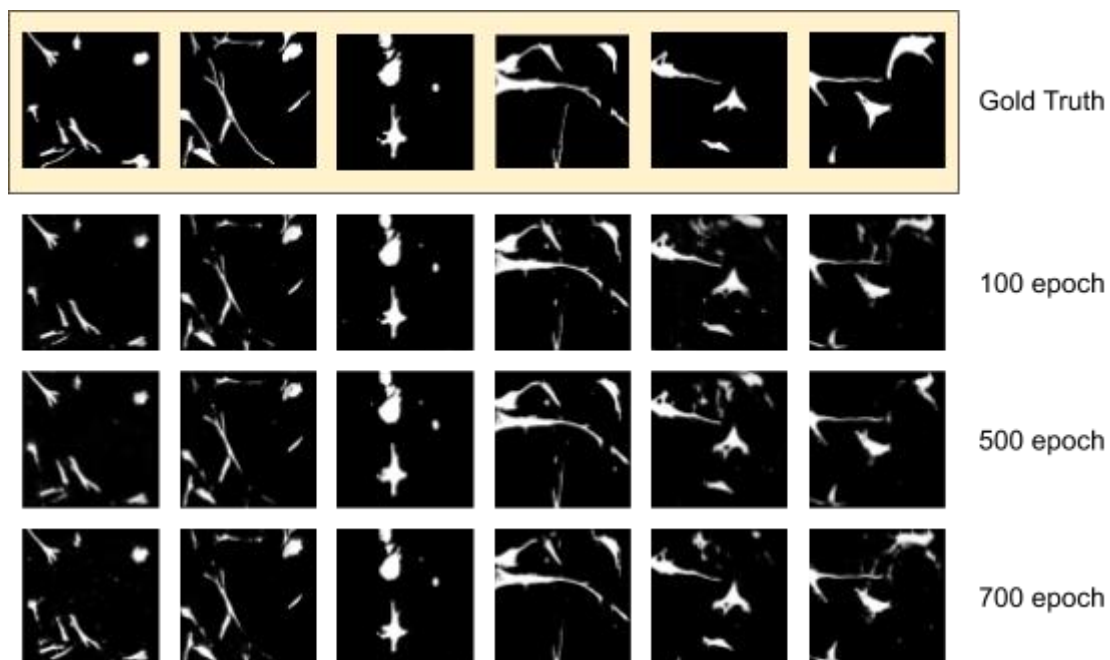


Figura 39. Varias Imágenes de segmentación predichas por los modelos entrenados con distintos epochs, comparadas con las anotaciones originales.

Hasta ahora se ha entrenado únicamente con las imágenes proporcionadas en el conjunto de datos, utilizando 30 de ellas para entrenar y 3 para las pruebas. Los resultados son buenos porque las imágenes de prueba son muy parecidas a las de entrenamiento y se ajustan muy bien los pesos.

El segundo lote de experimentos se hará con el fin de permitir a la red la capacidad de generalizar, aunque no sea necesaria, ya que todas las imágenes son parecidas. Pero sí debemos preparar el modelo y asegurarnos buenos resultados para secuencias de vídeo diferentes a las de entrenamiento, como se puede ver en la Sección 5.3. Para ello se introduce el aumento de datos (ver Sección 4.4.3), deformando los datos de entrenamiento, como se muestra en la Figura 40, y se entrena de nuevo con validación cruzada, utilizando distintos números de epochs.

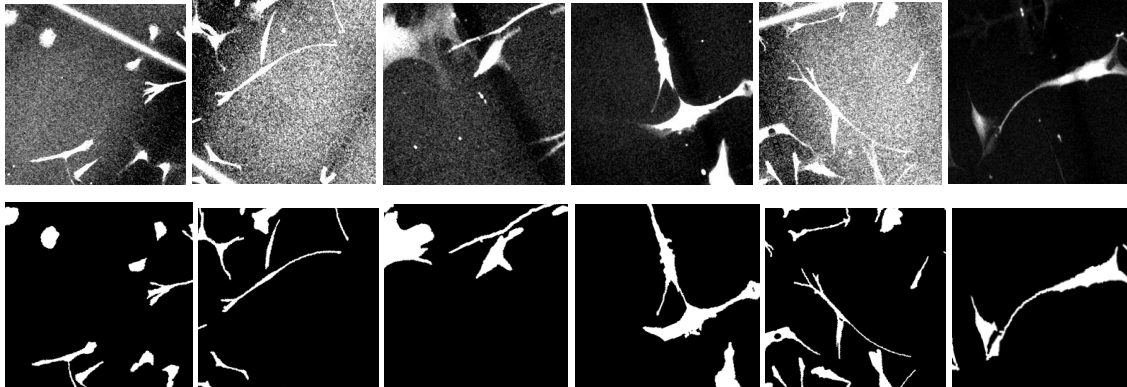


Figura 40. Ejemplos de muestras generadas por el generados de datos de Keras. Arriba las imágenes deformadas y con niveles de gris modificados. Abajo sus respectivas anotaciones con la misma deformación aplicada.

En las Figuras 41 y 42 se puede apreciar irregularidad en los resultados con aumento de datos. Esto es debido a la generación de imágenes aleatorias, lo que consigue que el modelo no se ajuste demasiado a los datos de entrenamiento, evitando el sobreajuste.

Las deformaciones aplicadas no pueden perturbar demasiado los datos. Deben mejorar o, al menos mantener los mismos resultados de validación que sin utilizar aumento de datos. En la Figuras 44 y 45 se puede ver esta comparativa.

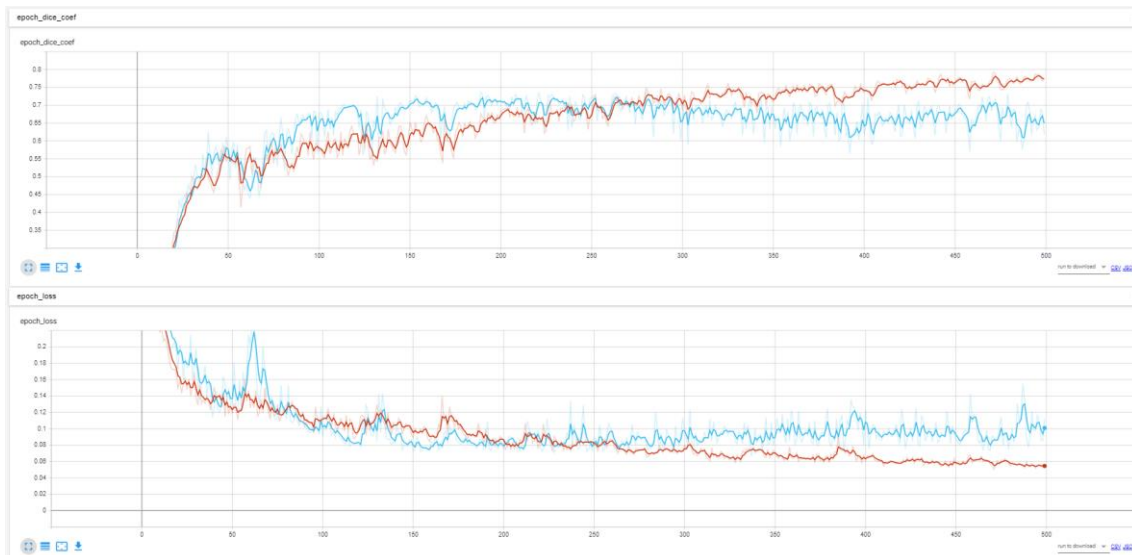


Figura 41. Función de pérdida (abajo) y métrica (arriba) para los datos de entrenamiento y validación. 500 epochs con aumento de datos (Azul: datos de entrenamiento, Rojo: datos de validación)

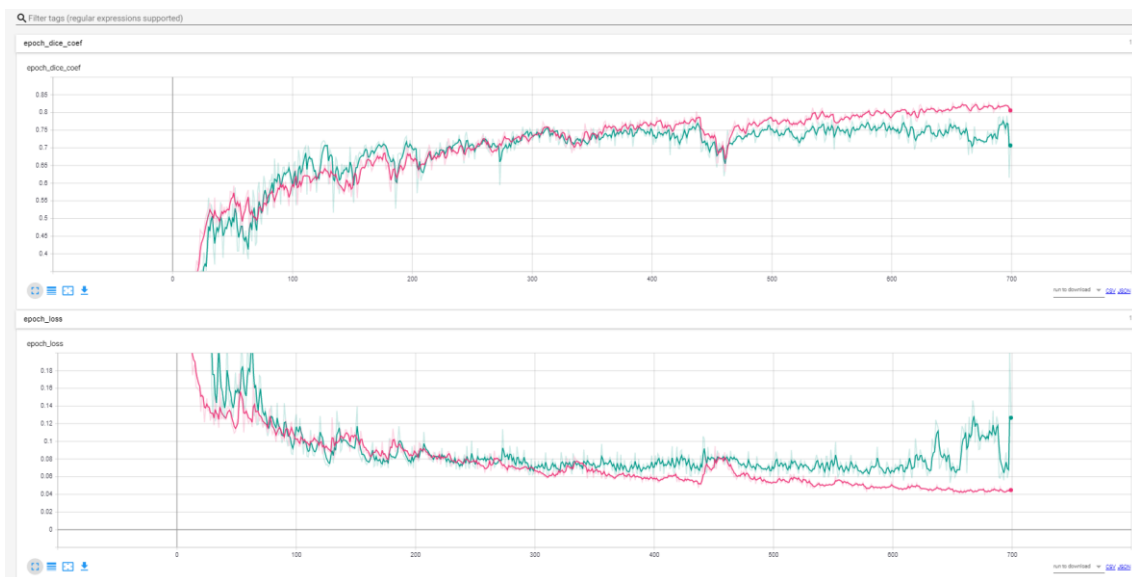


Figura 42. Función de pérdida (abajo) y métrica (arriba) para los datos de entrenamiento y validación. 700 epochs con aumento de datos (Rosa: datos de entrenamiento, Verde: datos de validación)

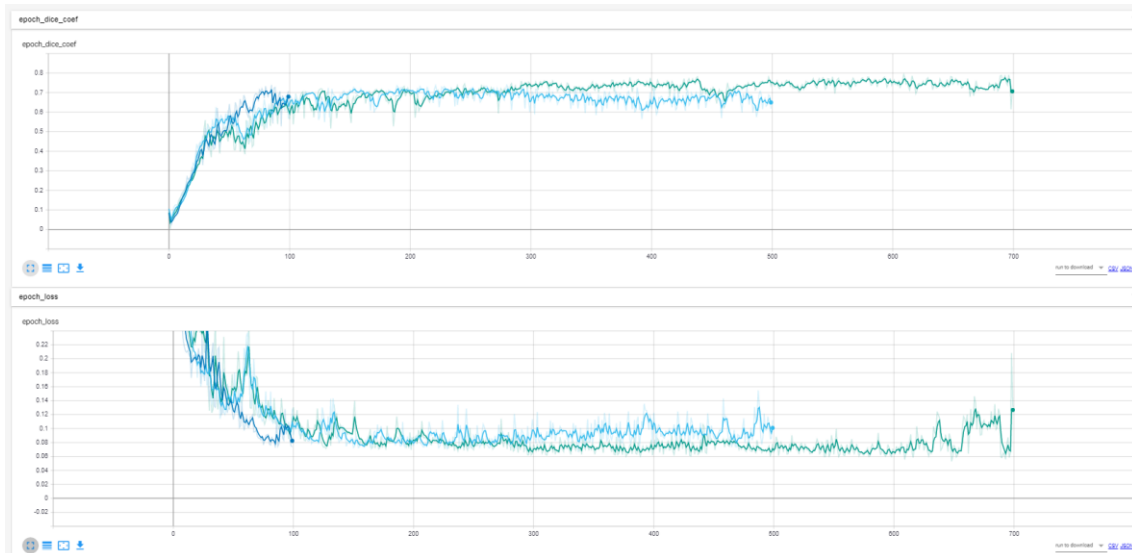


Figura 43. Función de pérdida (abajo) y métrica (arriba) para los datos de validación. 100 epochs (Azul oscuro), 500 epochs (Azul claro) y 700 epochs (Verde), con aumento de datos.

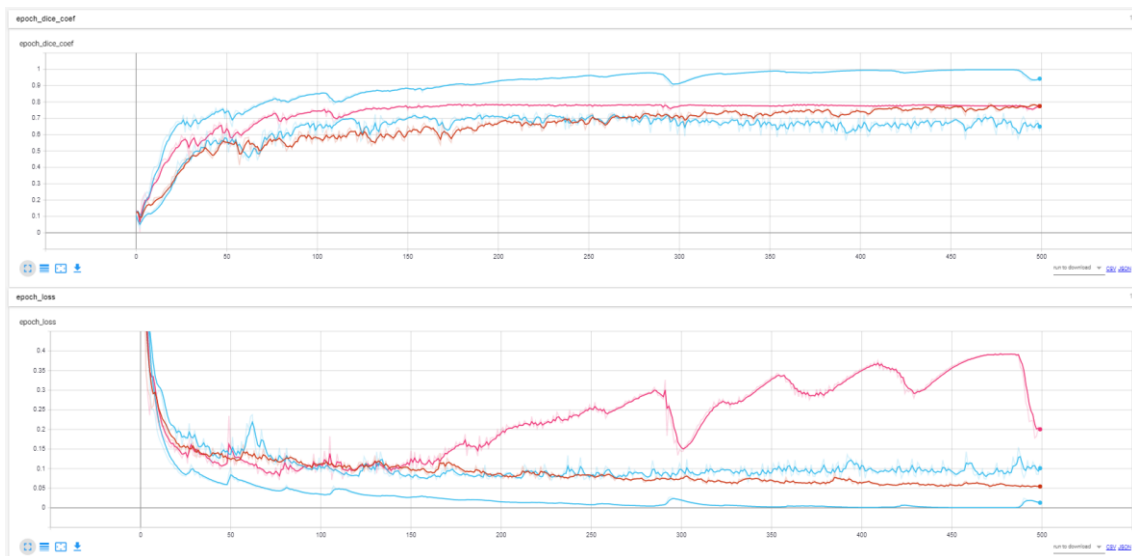


Figura 44. Función de pérdida (abajo) y métrica (arriba) para los datos de validación y entrenamiento. Aumento de datos (Rojo: datos de entrenamiento, Azul claro: Datos de validación) y sin aumento de datos (Azul claro: datos entrenamiento, Rosa: datos de validación) con 500 epochs.



Figura 45. Función de pérdida (abajo) y métrica (arriba) para los datos de validación y entrenamiento. Aumento de datos (Azul: datos de entrenamiento, Rojo: Datos de validación) sin aumento de datos (Rosa: datos entrenamiento, Verde: datos de validación), con 700 epochs.

	Con Aumento Datos		Sin Aumento Datos	
	Loss (Mean)	Coef. Dice (Mean)	Loss (Mean)	Coef. Dice (Mean)
100 epoch	0,1209	0,6096	0,0691	0,7565
500 epoch	0,0901	0,6807	0,0853	0,7665
700 epoch	0,0909	0,7068	0,0707	0,7983

Figura 46. Tabla con la media de los resultados en las imágenes de prueba (test), utilizando validación cruzada. Se comparan los modelos con aumento de datos y sin aumentos de datos, y según el número de epochs.

Utilizando aumento de datos también se consiguen mejores resultados con mayor número de epochs, sin embargo, los resultados de pérdida y métrica con las imágenes de prueba no son tan buenos como entrenando sin aumento de datos (Figura 46). Esto puede ser posible a que se están deformando demasiado los datos, pero gracias a su capacidad de generalizar, puede ofrecer mejores resultados para conjuntos de datos diferentes. En la siguiente sección se realiza este experimento.

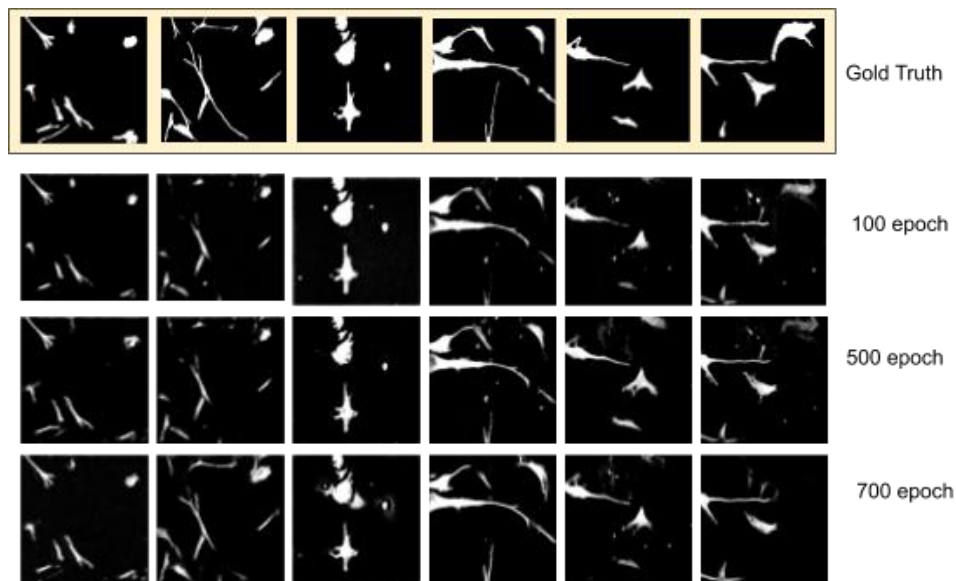


Figura 47. Varias Imágenes de segmentación predichas por los modelos entrenados con aumento de datos. Se compara con las anotaciones originales.

5.2 Pruebas con imágenes del Challenge

Las pruebas descritas en esta sección se realizan con las secuencias de vídeo adicionales del conjunto de datos que no contienen anotaciones, lo que quiere decir que no se han utilizado para entrenar el modelo.

El entrenamiento se ha realizado con el conjunto de datos de entrenamiento completo, quedando 39 imágenes de entrenamiento y 6 imágenes de validación, con la misma función de pérdida, métrica y optimizador que en la sección anterior, utilizando el mismo batch size y la misma configuración para las deformaciones en el aumento de datos.

Los resultados obtenidos se muestran en las siguientes Figuras, comparando las imágenes originales con las máscaras predichas por los distintos modelos entrenados.

Analizando los resultados obtenidos, subjetivamente, se observa como el aumento de datos sí mejora los resultados, especialmente en la secuencia de vídeo 2, Figura 49.

Por lo tanto, aunque en los experimentos anteriores (Sección 5.1) los resultados de las métricas con aumento de datos eran inferiores, en este caso se puede demostrar la capacidad de generalización de la red gracias a este método de regularización.

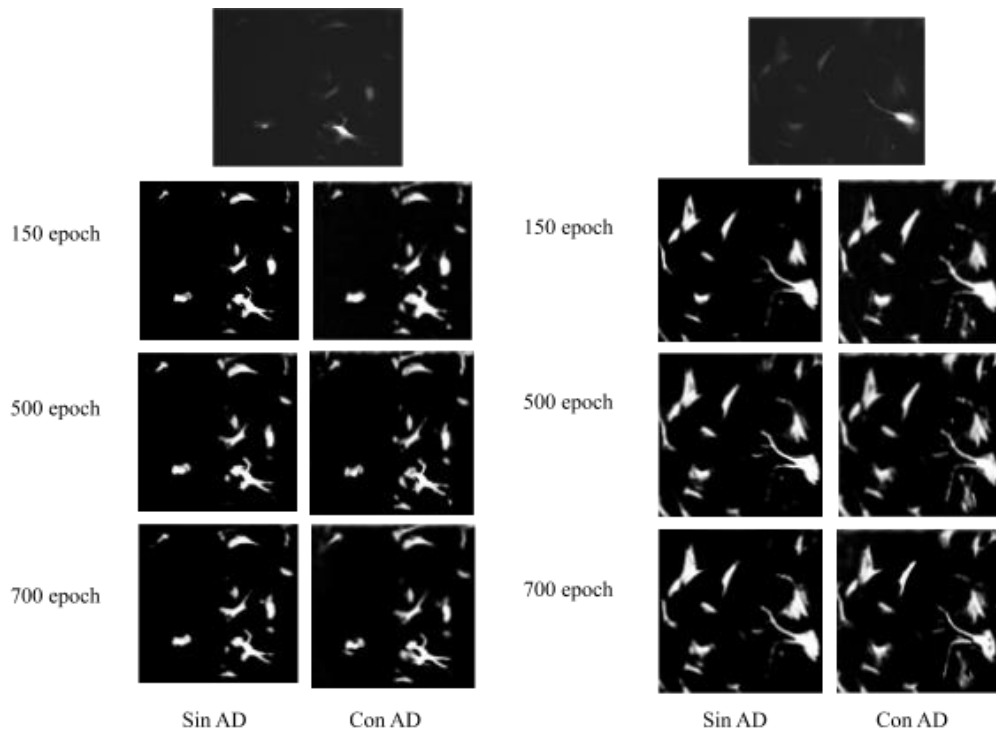


Figura 48. Resultado de las predicciones para dos muestras de la secuencia de vídeo de prueba 2.

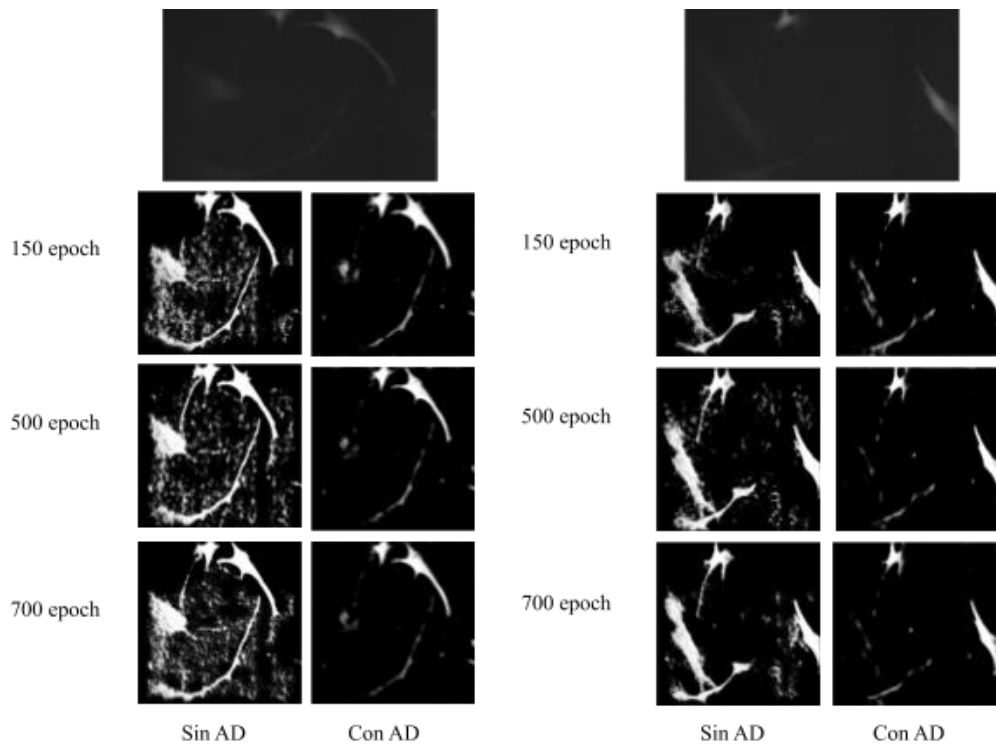


Figura 49. Resultado de las predicciones para dos muestras de la secuencia de vídeo de prueba 1.

Capítulo 6

Conclusiones

Este último capítulo presenta las conclusiones del desarrollo de este proyecto. Ofrece un resumen final con los objetivos más destacados alcanzados y las competencias adquiridas. Así como las posibles líneas futuras para continuar el trabajo realizado.

6.1 Conclusiones

En este trabajo se ha implementado un sistema capaz de segmentar células vivas en imágenes de vídeo microscopía, cumpliendo el objetivo principal inicial. El sistema está compuesto por una red neuronal convolucional UNet que nos permite obtener buenos resultados para el conjunto de datos de validación y prueba.

Este sistema fue desarrollado tras realizar una investigación sobre el actual Estado del Arte en Deep Learning relacionado con la segmentación de imágenes médicas, que confirmó que el mejor enfoque en este campo es el uso de RNC, en concreto UNet, ya que logra un muy buen rendimiento en aplicaciones de segmentación biomédica muy diferentes.

Tras explorar las posibles plataformas y herramientas para trabajar con redes neuronales convolucionales, se decidió implementar en Keras, bajo TensorFlow, ya que permite un desarrollo rápido y sencillo.

Finalmente, para desarrollar y perfeccionar la red neuronal convolucional UNet inicialmente propuesta, se han realizado experimentos y mejoras sobre ella, evaluando su rendimiento a base de prueba y error.

Todo ello, me ha permitido familiarizarme con herramientas y frameworks de Inteligencia Artificial, y adquirir conocimientos en Machine Learning y Deep Learning que no he aprendido durante el transcurso de mi Grado. Por lo que este trabajo me ha servido para adquirir dichas competencias.

6.2 Líneas Futuras

Para continuar con la investigación abordada en este proyecto se pueden seguir varias vías que permitan seguir obtener resultados interesantes:

- Realizar experimentos adicionales sobre la arquitectura implementada. Se puede seguir evaluando y mejorando el modelo con otras configuraciones de hiperparámetros, con diferentes funciones de pérdida, o aplicando otras técnicas de entrenamiento.
- Aumentar el conjunto de imágenes de entrenamiento añadiendo las anotaciones proporcionadas por otros participantes del challenge, Silver-standard corpus (silver turh) (Sección 3.2).
- Buscar un aumento de datos que mejore notablemente los resultados. Los resultados de los experimentos realizados no fueron tan buenos como lo esperado al entrenar el modelo con aumento de datos. Trabajos futuros podrían centrarse en analizar las deformaciones adecuadas para conseguir mejores resultados, como por ejemplo aplicar deformaciones elásticas.
- Investigación y experimentos con otras arquitecturas de red.

Debido a la extensión en el tiempo del proyecto y la velocidad a la que avanza el campo de la visión artificial, pueden haber surgido otras arquitecturas o algoritmos que compitan con la arquitectura UNet, para las tareas de segmentaciones en imagen médica.

Bibliografía

- [1] Maška, M., Ulman, V., Svoboda, D., Matula, P., Matula, P., Ederra, C., ... & Karas, P. (2014). A benchmark for comparison of cell tracking algorithms. *Bioinformatics*, 30(11), 1609-1617.
- [2] Cell Tracking Challenge. In URL <http://celltrackingchallenge.net/>.
- [3] Smal, I., Loog, M., Niessen, W., & Meijering, E. (2009). Quantitative comparison of spot detection methods in fluorescence microscopy. *IEEE transactions on medical imaging*, 29(2), 282-301.
- [4] McCarthy, J., Minsky, M. L., Rochester, N., & Shannon, C. E. (2006). A proposal for the dartmouth summer research project on artificial intelligence, august 31, 1955. *AI magazine*, 27(4), 12-12.
- [5] Goodfellow, I., Bengio, Y., & Courville, A. (2016). Deep learning. Book in preparation for MIT Press. URL: <http://www.deeplearningbook.org>, 1.
- [6] Géron, A. (2019). Hands-on machine learning with Scikit-Learn, Keras, and TensorFlow: Concepts, tools, and techniques to build intelligent systems. O'Reilly Media.
- [7] L. Kinsey. A machine learning primer. (2016). In URL <https://medium.com/@libbykinsey/a-machine-learning-primer-6d7b5a96a3b0>.
- [8] Russell, S. J., & Norvig, P. (2002). Artificial Intelligence-A modern Approach, 2nd edn. Indian Reprint.
- [9] Bcnvision. Deep Learning, en el punto de mira de la visión artificial. (2019). In URL <https://www.bcnvision.es/blog-vision-artificial/deep-learning-vision-artificial/>.
- [10] Deng, L., & Yu, D. (2016). Deep learning: methods and applications.(2014).
- [11] Hinton, G. E., Osindero, S., & Teh, Y. W. (2006). A fast learning algorithm for deep belief nets. *Neural computation*, 18(7), 1527-1554.

- [12] NUSANTARA, B. (2017). Convolutional Neural Network. Dostupné také z: <http://socs.binus.ac.id/2017/02/27/convolutionalneural-network>.
- [13] H. Norman. AI Basics: Training vs Inference – What’s the Difference?. (2019) In URL <https://community.arm.com/developer/ip-products/processors/b/ml-ip-blog/posts/ai-basics-training-vs-inference-whats-the-difference>.
- [14] Long, J., Shelhamer, E., & Darrell, T. (2015). Fully convolutional networks for semantic segmentation. In Proceedings of the IEEE conference on computer vision and pattern recognition (pp. 3431-3440)..
- [15] Guo, Y., Liu, Y., Oerlemans, A., Lao, S., Wu, S., & Lew, M. S. (2016). Deep learning for visual understanding: A review. *Neurocomputing*, 187, 27-48.
- [16] Badrinarayanan, V., Kendall, A., & Cipolla, R. (2017). Segnet: A deep convolutional encoder-decoder architecture for image segmentation. *IEEE transactions on pattern analysis and machine intelligence*, 39(12), 2481-2495.
- [17] Krizhevsky, A., Sutskever, I., & Hinton, G. E. (2012). Imagenet classification with deep convolutional neural networks. In *Advances in neural information processing systems* (pp. 1097-1105).
- [18] Russakovsky, O., Deng, J., Su, H., Krause, J., Satheesh, S., Ma, S., ... & Berg, A. C. (2015). Imagenet large scale visual recognition challenge. *International journal of computer vision*, 115(3), 211-252.
- [19] Simonyan, K., & Zisserman, A. (2014). Very deep convolutional networks for large-scale image recognition. *arXiv preprint arXiv:1409.1556*.
- [20] A. Anwar. (2019). Difference between AlexNet, VGGNet, ResNet, and Inception. In URL <https://towardsdatascience.com/the-w3h-of-alexnet-vggnet-resnet-and-inception-7baaecccc96>.
- [21] Chen, L. C., Papandreou, G., Kokkinos, I., Murphy, K., & Yuille, A. L. (2014). Semantic image segmentation with deep convolutional nets and fully connected crfs. *arXiv preprint arXiv:1412.7062*..
- [22] E. P. Sánchez. (2020). Development and validation of data analysis automation methods using pattern recognition.
- [23] Abadi, M., Barham, P., Chen, J., Chen, Z., Davis, A., Dean, J., ... & Kudlur, M. (2016). Tensorflow: A system for large-scale machine learning. In 12th {USENIX} symposium on operating systems design and implementation ({OSDI} 16) (pp. 265-283).

- [24] A. Bakshi. TensorFlow Tutorial Deep Learning Using TensorFlow. In URL <https://www.edureka.co/blog/tensorflow-tutorial/>..
- [25] Get started with TensorBoard. In URL https://www.tensorflow.org/tensorboard/get_started.
- [26] Developer guides, Keras. In URL <https://keras.io/guides/>.
- [27] Kingma, D. P., & Ba, J. (2014). Adam: A method for stochastic optimization. arXiv preprint arXiv:1412.6980.
- [28] Dumoulin, V., & Visin, F. (2016). A guide to convolution arithmetic for deep learning. arXiv preprint arXiv:1603.07285.
- [29] Bressert, E. (2012). SciPy and NumPy: an overview for developers. " O'Reilly Media, Inc."
- [30] Van der Walt, S., Schönberger, J. L., Nunez-Iglesias, J., Boulogne, F., Warner, J. D., Yager, N., ... & Yu, T. (2014). scikit-image: image processing in Python. PeerJ, 2, e453.
- [31] Pedregosa, F., Varoquaux, G., Gramfort, A., Michel, V., Thirion, B., Grisel, O., ... & Vanderplas, J. (2011). Scikit-learn: Machine learning in Python. the Journal of machine Learning research, 12, 2825-2830..
- [32] SciPy.org. In URL <https://www.scipy.org/>.
- [33] Jaitley, U. (2018). Why Data Normalization is necessary for Machine Learning models.
- [34] Falk, T., Mai, D., Bensch, R., Çiçek, Ö., Abdulkadir, A., Marrakchi, Y., ... & Dovzhenko, A. (2019). U-Net: deep learning for cell counting, detection, and morphometry. Nature methods, 16(1), 67-70.
- [35] Ronneberger, O., Fischer, P., & Brox, T. (2015, October). U-net: Convolutional networks for biomedical image segmentation. In International Conference on Medical image computing and computer-assisted intervention (pp. 234-241). Springer, Cham.
- [36] S. Team. (2018). The Ultimate Guide to Convolutional Neural Networks (CNN). In URL <https://www.superdatascience.com/blogs/the-ultimate-guide-to-convolutional-neural-networks-cnn>.
- [37] Neural Networks, Machine Learning for Artists. In URL https://ml4a.github.io/ml4a/neural_networks/.

- [38] P. b. Sharp. Neural Networks The Elements of Statistical Learning, Chapter 12 Presented by Nick Rizzolo. In URL: <https://slideplayer.com/slide/10091359/>.
- [39] Scherer, D., Müller, A., & Behnke, S. (2010, September). Evaluation of pooling operations in convolutional architectures for object recognition. In International conference on artificial neural networks (pp. 92-101). Springer, Berlin, Heidelberg.
- [40] U-Net, in DeepLearning documentation. In URL <http://www.deeplearning.net/tutorial/unet.html>.
- [41] Machine Learning Glossary. (2017). In URL https://ml-cheatsheet.readthedocs.io/en/latest/loss_functions.html.
- [42] Reddi, S. J., Kale, S., & Kumar, S. (2018). On the convergence of adam and beyond.(2018). In URL <https://openreview.net/forum>.
- [43] Torres, J. (2018). DEEP LEARNING Introducción pr´ctica con Keras. Lulu. com.
- [44] Wong, S. C., Gatt, A., Stamatescu, V., & McDonnell, M. D. (2016, November). Understanding data augmentation for classification: when to warp?. In 2016 international conference on digital image computing: techniques and applications (DICTA) (pp. 1-6). IEEE.
- [45] Hussain, Z., Gimenez, F., Yi, D., & Rubin, D. (2017). Differential data augmentation techniques for medical imaging classification tasks. In AMIA Annual Symposium Proceedings (Vol. 2017, p. 979). American Medical Informatics Association.
- [46] Amazon Machine Learning Guía para desarrolladores. Validación cruzada (p. 106). In URL https://docs.aws.amazon.com/es_es/machine-learning/latest/dg/cross-validation.html.
- [47] Long, J., Shelhamer, E., & Darrell, T. (2015). Fully convolutional networks for semantic segmentation. In Proceedings of the IEEE conference on computer vision and pattern recognition (pp. 3431-3440).
- [48] Jordan, J. (2018). An overview of semantic image segmentation.
- [49] Ortega, D. R., & Iznaga, A. M. (2008). Técnicas de Segmentación de Imágenes Médicas. 14 Convención científica de ingeniería y arquitectura, 1-7.