ESCUELA TÉCNICA SUPERIOR DE INGENIERÍA INFORMÁTICA

Proyecto fin de Grado

Grado en Ingeniería Informática: Tecnologías Informáticas

Segmentación de células en imágenes 3D con técnicas de machine learning

Realizado por
(ponente): Adrián López Carrillo

Dirigido por
María José Jiménez Rodríguez

Departamento
Matemática Aplicada I

Profesor Colaborador
Luis María Escudero Cuadrado
Departamento de Biología Celular, Universidad de Sevilla

Sevilla, 09 de Septiembre de 2020 (v.1.0)

Resumen

En este proyecto se han utilizado técnicas modernas para la segmentación de células en imágenes 3D. Actualmente los mejores resultados para tratamiento de imágenes en deep learning se obtienen con las redes neuronales convolucionales (CNN), obteniendose muy buenos resultados en segmentación celular 3D con la arquitectura UNet.

En este proyecto se han estudiado varios algoritmos y métodos para hacer segmentación celular y se han implementado aquellos que han obtenido mejores resultados en los últimos años. El desarrollo se ha basa fuertemente en (Wolny y cols., 2020) y (Falk y cols., 2019), con una estrategia de segmentación de bordes + DT Watershed como un método y con segmentación de células añadiendo espaciado entre ellas como otro método.

Se ha desarrollado un programa sencillo que puede ser usado y modificado fácilmente. Se puede entrenar nuevos modelos y hacer inferencia con modelos ya entrenados.

Índice general

Índice general **2**

Índice de cuadros **4**

Índice de figuras **5**

1 Presentación del problema **1**

2 Análisis de antecedentes **2**
2.1 Red Neuronal Artificial . . . 2
2.1.1 Introducción a Redes Neuronales Artificiales . . . 2
2.1.2 Neurona artifical . . . 3
2.1.3 Red Neuronal Artifical . . . 4
2.1.4 Descenso por Gradiente . . . 5
2.2 Red Neuronal Convolucional . . . 6
2.2.1 Tipos de capas . . . 7
2.2.2 Funciones de Pérdida . . . 10
2.3 Arquitecturas para segmentación semántica . . . 11
2.3.1 Fully Convolutional Network . . . 12
2.3.2 Deconvnet . . . 12
2.3.3 U-Net . . . 14
2.4 Aplicaciones recientes . . . 14
2.4.1 U-Net para cuenteo, detección y morfometría, detección y morfometría de células. (2019) . . . 15
2.4.2 Precisa y versátil segmentación 3D de tejido vegetal a resolución celular. (2020) 17

3 Datos de entrada **20**

4 Diseño **22**
4.1 Esquema general . . . 22
4.2 Preprocesado local . . . 24
4.3 Entrenamiento . . . 26
4.4 Inferencia . . . 27
4.5 Arquitectura de las Redes Neuronales Convolucionales . . . 28

5 Implementación **30**
5.1 Lenguaje y framework . . . 30
5.2 Preprocesado local . . . 31
5.2.1 Consideraciones . . . 31
5.3 Carga de datos . . . 32
5.3.1 Consideraciones . . . 33

5.4 U-Net 34
5.5 Postprocesado 36

6 Resultados **37**

7 Análisis temporal **45**

8 Conclusiones **46**

9 Manual **47**

Referencias **48**

Índice de cuadros

2.1 Pipeline que siguen los datos de inicio a fin. GT significa *groundtruth* y se refiere a la imagen ya segmentada. IM es la imagen de entrada. 15
2.2 Pipeline que siguen los datos de inicio a fin. GT significa *groundtruth* y se refiere a la imagen ya segmentada. IM es la imagen de entrada. 17
2.3 Pruebas relevantes realizadas. 18

3.1 Valores mínimo y máximo de los píxeles de las imágenes de entrada. 20

7.1 Tabla de tiempos 45

Índice de figuras

2.1 Neurona artificial genérica 3
2.2 Representación de una Neurona Artificial con $\sum x_i w_i$ como función de integración 4
2.3 Red Neuronal Artificial completamente conectada. 5
2.4 Ilustración del algoritmo de descenso por gradiente. θ_0 y θ_1 son los pesos de la ANN y J la función de pérdida. Se puede observar cómo se produce un "descenso"hacia un mínimo de la función. 6
2.5 Imagen de 4x4 px aplanada y usada como entrada en una FCNN con 4 neuronas en su única capa oculta. No se muestra su capa de salida. Imagen del curso Intro to Deep Learning with PyTorch de Udacity. 6
2.6 Imagen de 4x4 px usada como entrada en una CNN. Ambas figuras muestran la misma arquitectura, estando en la figura de la izquierda la imagen aplanada y en la figura de la derecha se muestra la matriz 4x4 como entrada. Imágenes del curso Intro to Deep Learning with PyTorch de Udacity. 7
2.7 Red Neuronal Convolucional simple en la que la imagen de un barco es clasificada. 7
2.8 Capa de convolución de profundidad 2 (se usan dos filtros) con kernel de tamaño $3 \times 3 \times 3$ aplicado a un volumen de entrada de tamaño $7x7x3$, volumen generado al aplicar padding de 1 px a una imagen de tamaño $5x5$ con 3 canales. El resultado es un volumen de $3x3x2$ 9
2.9 Imagen 4×4 a la que se ha aplicado un pooling de $F = 2, S = 2$. Cada color de la imagen de la izquierda indica las entradas del para la operación MAX, cada color de la imagen de la derecha indica la salida de dicha operación. Figura tomada del curso CS231n de Standford University. 10
2.10 Arquitectura de FCN32, FCN16 y FCN8. 12
2.11 Arquitectura Deconvnet. 13
2.12 Figura que ilustra las operaciones unpooling y deconvolución. 13
2.13 Arquitectura original U-Net propuesta por Ronneberger et al en U-Net: Convolutional Networks for Biomedical Image Segmentation. 14
2.14 Segmentación volumétrica. (a) Segmentación de una imagen con un modelo que se ha entrenado con imágenes del mismo dataset. (b) Segmentación de una imagen con un modelo entrenado con un dataset distinto. 16
2.15 Segmentación de tejido vegetal usando PlantSeg. En el primer paso se predicen los bordes de las células usando una red U-Net 3D. En el segundo paso se aplica un algoritmo de particionamiento de grafo para segmentar cada célula. 18
2.16 Interfaz gráfica del programa PlantSet. Se pueden ver las distintas opciones para cada paso del procesado. 19

3.1 Histograma de todos los valores 21
3.2 Histograma de todos los valores con el eje y limitado a 4000 21

4.1 Diseño general 23

4.2 Preprocesado llevado a cabo en la máquina local. 24
4.3 Pasos llevados a cabo en el entrenamiento 26
4.4 Se muestran dos métodos distintos para obtener la imagen correctamente etiquetada. 27
4.5 Arquitectura U-Net completa. 28
4.6 Arquitectura U-Net completa. 29

6.1 Modelo miniunet2.1. 100 epochs. Se usa MiniUnet3D. Imagen objetivo con espaciado entre células. Sin data augmentation. 38
6.2 Modelo unet4.1. 200 epochs. Imagen objetivo con espaciado entre células. Sin data augmentation. 38
6.3 Modelo unet4.13. 200 epochs. Imagen objetivo con espaciado entre células. Sin data augmentation. 39
6.4 Modelo unet4.8. 200 epochs. Imagen objetivo con espaciado entre células. Con data augmentation. 39
6.5 Modelo unet4.8. 200 epochs. Imagen objetivo con espaciado entre células. Con data augmentation. Sin Apex 40
6.6 Modelo unet5.2. 200 epochs. Imagen objetivo con espaciado entre células. Con data augmentation. Con Apex. 40
6.7 Modelo unet6t. 100 epochs. Imagen objetivo con espaciado entre células. Con data augmentation. Pérdida Dice. Sin postprocesado. Primera fila imagen de entrada. Segunda fila segmentación objetivo. Tercera fila predicción. Z=20,25,45,50 en las columnas. IoU 0.72 41
6.8 Modelo unet6t. 100 epochs. Imagen objetivo con espaciado entre células. Con data augmentation. Sin postprocesado. 41
6.9 Modelo unet6b. 100 epochs. Imagen objetivo bordes de células. Con data augmentation. Pérdida Dice. Sin postprocesado. Primera fila imagen de entrada. Segunda fila segmentación objetivo. Tercera fila predicción. Z=20,25,45,50 en las columnas. IoU 0.84 42
6.10 Modelo unet6b. 100 epochs. Imagen objetivo bordes de células. Con data augmentation. Pérdida Dice. Sin postprocesado. 42
6.11 DT Watershed aplicado a la salida del modelo unet6b. Primera fila bordes. Segunda fila transformación de la distancia. Filtro gaussiano. Cuarta fila watershed con mínimos locales como semilla. Z=20,25,45,50 en las columnas. 43
6.12 Modelo unet6b. 100 epochs. Imagen objetivo bordes de células. Con data augmentation. Pérdida Dice. Componentes conexas con etiquetas distintas. IoU 0.72 . . . 44
6.13 DT Watershed aplicado a la salida del modelo unet6b. IoU 0.70 44

CAPÍTULO 1

Presentación del problema

Gracias a los avances en microscopía se provee a los investigadores una gran cantidad de datos de imágenes. Antes de que estas imágenes puedan ser analizadas por investigadores se requiere un cierto tratamiento de los datos en bruto. A veces este tratamiento requiere etiquetar manualmente miles de células o dibujar sus contornos. Este es un trabajo tedioso no muy gratificante de realizar. Por ejemplo en estudios neurocientíficos se suele requerir cuantificar el número de neuronas que expresan opsinas (un tipo de proteínas) o la localización de nuevas opsinas desarolladas en las células. Sin embargo, esta información es omitida en la mayoría de estudios a causa del esfuerzo que conlleva.

El Departamento de Biología Celular de la Facultad de Biología de la Universidad de Sevilla tenía un problema similar. Necesitaban un método para segmentar con precisión y velocidad imágenes tomadas por un microscopio confocal con gran detalle. Tras probar varias técnicas decidieron intentar hacerlo utilizando herramientas de machine learning. El procesado digital de estas imágenes actualmente se hace de forma manual teniendo una duración de una a dos semanas, por lo que la automatización de este proceso conllevará un gran ahorro de tiempo.

CAPÍTULO 2

Análisis de antecedentes

En este capítulo se hará una breve introducción a las redes neuronales, se justificará el uso del tipo de red neuronal CNN para tratamiento de imágenes así como el de la arquitectura CNN U-Net para la segmentación de células.

2.1– Red Neuronal Artificial

En esta sección se hará una breve introducción a las redes neuronales artificiales, compuesta por neuronas artificiales. Después se describirá qué es una neurona artificial y se definirán 3 tipos: perceptrón, sigmoide y unidad lineal rectificada (ReLU). Por último se hablará sobre el entrenamiento.

2.1.1. Introducción a Redes Neuronales Artificiales

Desde la antigüedad la humanidad ha sentido interés en la posibilidad de emular la inteligencia de forma artificial. Con los avances en neurociencia hemos sido capaces de entender cómo funcionan las neuronas, la unidad básica en el funcionamiento de los cerebros. Siendo el cerebro un ejemplo funcional de un sistema inteligente, es natural que haya interés en replicar su funcionamiento.

Un hito importante se produjo gracias al desarrollo de la teoría del aprendizaje biológico, introducida por Warren McCulloch y Walter Pitts en 1943 (McCulloch y Pitts, 1943), popularizando lo que fue llamado como *cibernética* (Goodfellow, Bengio, y Courville, 2016, p13). Gracias a esto surgió el ADALINE (elemento lineal adaptativo) (Widrow y Hoff, 2015), que es un caso concreto del algoritmo descenso por gradiente estocástico (**SGD**), algoritmo que con pequeñas modificaciones es usado en la actualidad en el proceso de aprendizaje(Goodfellow y cols., 2016, p14). Fue también gracias al estudio de McCulloch y Pitts que en 1958 Frank Rosenblatt introdujo por primera vez el **perceptrón**, un modelo general de neurona artificial (Rosenblatt, 1958), que fue perfeccionado por Minsky Y Papert en 1969 (Minsky y Papert, 1969).

El perceptrón es un modelo lineal que, dado un conjunto de elementos con dos categorías distintas como entrada, puede clasificar cada elemento en una de esas dos categorías. Un ejemplo sencillo son las puertas lógicas, siendo la entrada un conjunto de dos elementos con las categorías 0 o 1 y la salida sería un 0 o un 1.

Minsky y Papert encontraron un problema en los modelos lineales y lo demostraron con la función XOR, siendo imposible para un modelo lineal compuesto de una neurona artificial aprender esta función. Esto causó un declive en el interés sobre este campo.

En la década de 1980 resurgió el interés gracias en parte al conexionismo, cuya idea central es que un gran número de unidades de computación simples pueden tener un comportamiento inteligente al estar conectadas entre sí (Goodfellow y cols., 2016, p16). Durante esta etapa se hicieron importantes contribuciones como la representación distribuida (G. E. Hinton, 1986), donde se habla sobre representar las entradas de un sistema en base a sus características, reconocidas por patrones de actividad en redes neuronales. Otra gran contribución fue la populariación del algoritmo de propagación hacia atrás, **backpropagation** (Rumelhart, Hinton, y Williams, 1986) para entrenar redes neuronales artificales y actualizar sus pesos, siendo este algoritmo el más usado en la actualidad.

En este punto de la historia los algoritmos más importantes involucrados en las redes neuronales artificiales usadas en la actualidad habían sido descubiertos, pero no se estaban obteniendo resultados tan buenos como los esperados. Desde un principio lo que se había estado buscando era replicar de forma artificial el funcionamiento del cerebro, siendo el cerebro un sistema de computación genérico, capaz de aprender todo tipo de conocimiento distinto sin la necesidad de cambiar su arquitectura o su método para aprender. Era imposible conocer el algoritmo de aprendizaje usado en el cerebro ya que para ello haría falta monitorizar una gran cantidad de neuronas con gran precisión, lo cual es imposible incluso en la actualidad.

En 2006 se produjo un hito importante que comenzó la etapa del **aprendizaje profundo**, cuando Geoffrey Hinton demostró que era posible entrenar de forma eficiente una red neuronal profunda con un gran número de capas ocultas (G. Hinton, Osindero, y Teh, 2006).

2.1.2. Neurona artifical

En la figura 2.1 se puede ver una neurona artifical genérica con la que pueden ser descritos los distintos tipos que se verán en esta sección.

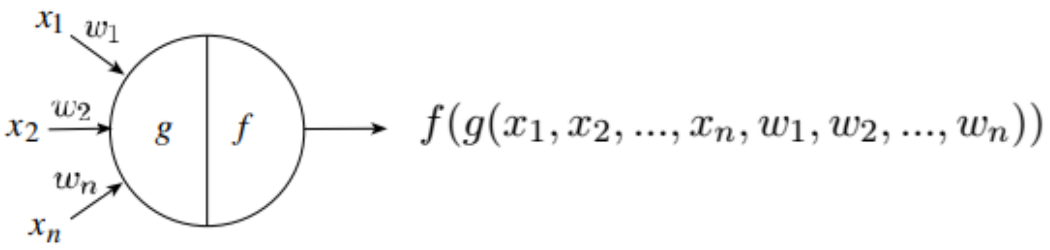

Figura 2.1: Neurona artificial genérica

Siendo:

- $(x_1, x_2, ..., x_n)$ el vector de entrada.
- $(w_1, w_2, ..., w_n)$ el vector de pesos.
- g la función de integración, encargada de reducir el vector de entrada a un único valor.
- f la función de activación, encargada de producir la salida de este elemento.

Se puede simplificar la representación al asumir que siempre se usará $\sum x_i w_i$ como función de integración. Además toda neurona artificial tendrá una entrada y un peso por defecto, independientemente del vector de entrada, esto hará referencia al *bias*. Será común ver una representación como 2.3 en la que f indicará la función de activación.

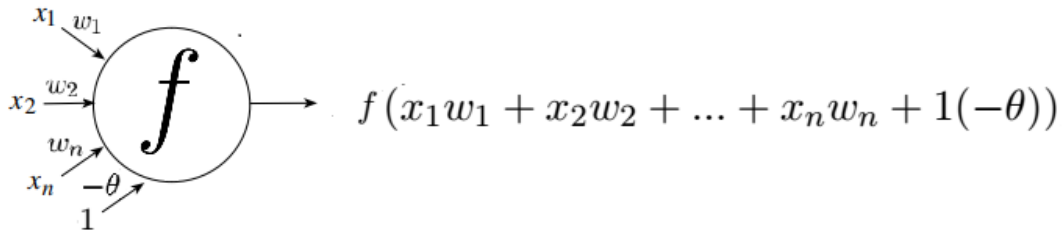

Figura 2.2: Representación de una Neurona Artificial con $\sum x_i w_i$ como función de integración

- Al vector de entradas se le añade un elemento de valor constante 1, siendo ahora de tamaño $n + 1$.
- Al vector de pesos se le añade un elemento de valor inicial $-\theta$, siendo ahora de tamaño $n + 1$. A este valor se le llamará *bias*.
- f indicará la función de activación de la neurona artificial.

Sigmoide

La función sigmoide como función de activación es una función no lineal usada principalmente en redes neuronales prealimentadas (feedforward neurals networks), que son las que usaremos en este proyecto. Es una función real, acotada y diferenciable (a diferencia de la usada en el perceptrón). Su definición es la siguiente relación (Nwankpa, Ijomah, Gachagan, y Marshall, 2018):

$$f(x) = \sigma(x) = \frac{1}{1 + e^{-x}} \tag{2.1}$$

Unidad Lineal Rectificada (ReLU)

La unidad lineal rectificada (ReLU) fue propuesta como función de activación en 2010 por Nair y Hinton (Nair y Hinton, 2010) y desde entonces ha sido la más usada en aplicaciones de aprendizaje profundo (deep learning, DL). Si se compara con la función de activación Sigmoide, ofrece un mejor rendimiento y es más generalista (Nwankpa y cols., 2018).

$$f(x) = max(0, x) = \begin{cases} x_i & \text{si } x_i \geq 0 \\ 0 & \text{si } x_i < 0 \end{cases} \tag{2.2}$$

2.1.3. Red Neuronal Artifical

Considerando la neurona artificial como una unidad de computación básica, según el conexionismo (que más tarde evolucionó en lo que hoy conocemos como *deep learning*, se podría emular un comportamiento inteligente al conectar neuronas artificiales entre sí. La conexión entre las neuronas artificales se consigue concatenando las salidas de unas con la entradas de otras y obteniendo así una red neuronal artifical (ANN).

En la figura 2.3 se ve una Red Neuronal Artifical completamente conectada (*fully connected neural network* o FCNN) en la que todos los nodos de una capa están conectados con todos los nodos de la capa siguiente. Contaría con los siguientes elementos:

- Capa de entrada i (*Input layer*) con n nodos. Cada nodo representa un valor del vector de entrada x. En esta capa no se altera el valor de x, está para representar los pesos de cada elemento del vector de entrada con los nodos de la primera capa oculta.

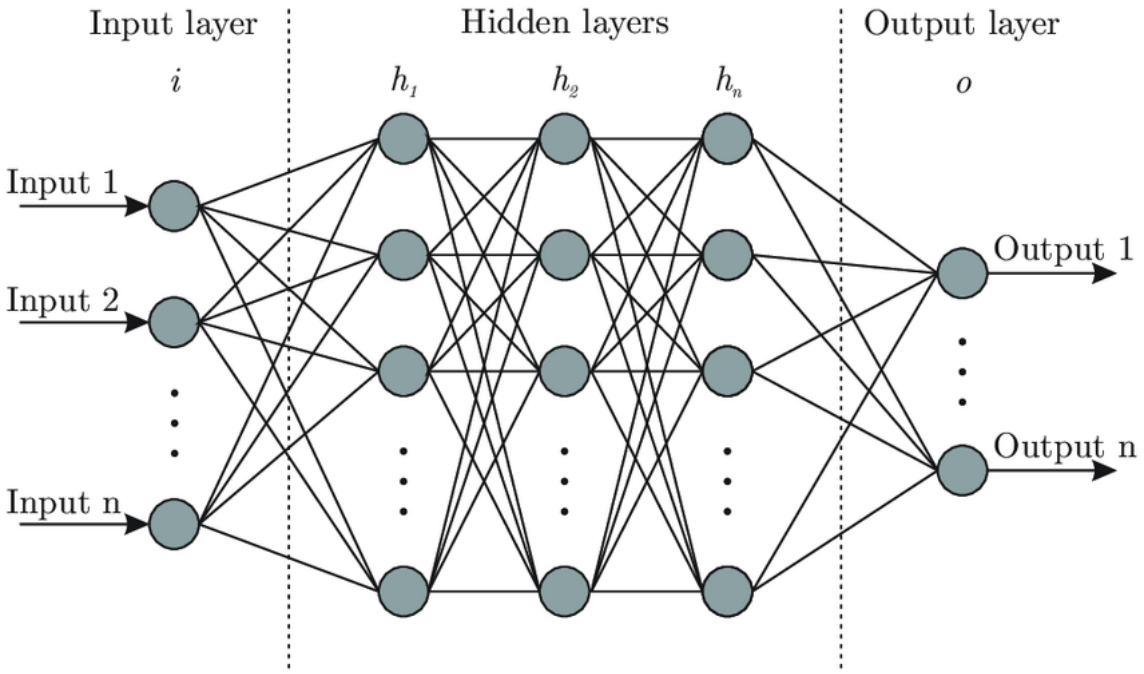

Figura 2.3: Red Neuronal Artificial completamente conectada.

- Capas ocultas $(h1, h_2, ..., h_m)$ (*hidden layers*). Cada capa oculta podrá tener un nº de nodos distintos. Es en estas capas donde se reconocen los patrones del conjunto de datos y tiene el mayor coste computacional. La última capa oculta está conectada con las entradas de la capa de salida.

- Capa de salida o (*output layer*) con c nodos. La última capa de la red neuronal, la salida de esta capa nos dará un vector de tamaño c como salida.

2.1.4. Descenso por Gradiente

En cálculo de varias variables calcular el gradiente de una función (∇f) dará como resultado un vector indicando la dirección en la que esa función tiene un mayor incremento, siendo el módulo el ritmo de variación. Para el descenso del gradiente será interesante usar $-\nabla f$ ya que nos dará el vector en el que la función decrece más. Como es habitual en problemas de optimización, si suponemos que tenemos una función que nos da el error cometido, nuestro objetivo será minimizar dicha función.

La función a optimizar se llamará **función de pérdida** (*loss function*) en la que se comparará la salida obtenida por la red con la salida deseada. Hay que tener que este es un algoritmo de aprendizaje y sólo tendrá sentido usarlo cuando se conozca el resultado correcto para una entrada determinada.

La salida obtenida por una red para una entrada determinada y, por lo tanto, el valor obtenido en la función de pérdida, dependerá únicamente de los pesos de dicha red. Esto significa que lo ideal será encontrar el mínimo global de la función de pérdida al cambiar el valor de los pesos de la red. Para actualizar los pesos se usará la fórmula $w_{ij} = w_{ij} - \eta \nabla J(W)$ donde $w_i j$ es el peso desde el nodo i al nodo j, η es el factor de aprendizaje o **learning rate** y $\nabla J(W)$ es la función de coste dados unos pesos determinados.

No es extraño en deep learning encontrar una red con millones de pesos pero, para facilitar la visualización, se ha usado como ejemplo una ANN con 2 pesos (θ_0 y θ_1). En la figura 2.4 se puede ver una ilustración de cómo en cada punto (marcado por una X negra) se calcula el gradiente de $J(\theta_0, \theta_1)$ y se actualizan los pesos, cambiando el valor de $J(\theta_0, \theta_1)$ hasta alcanzar un mínimo.

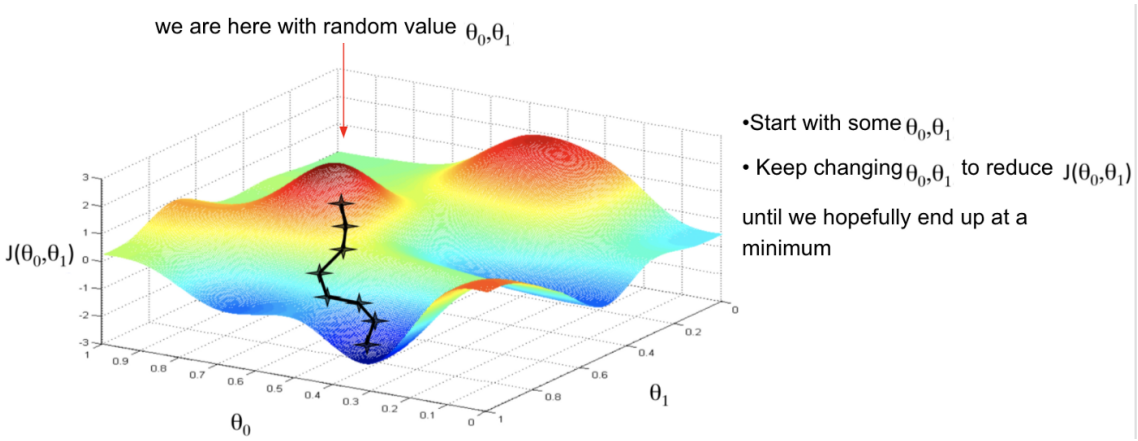

Figura 2.4: Ilustración del algoritmo de descenso por gradiente. θ_0 y θ_1 son los pesos de la ANN y J la función de pérdida. Se puede observar cómo se produce un ”descenso”hacia un mínimo de la función.

2.2– Red Neuronal Convolucional

Hasta ahora hemos supuesto que la entrada a una ANN es un vector, algo válido para un gran número de aplicaciones en deep learning. El problema está cuando la entrada de la red neuronal es una imagen. En este caso antes de utilizar la imagen como entrada hay que aplanarla para contener la imagen en un vector, de esta forma cada píxel (o vóxel) será un elemento del vector de entrada y estará conectado a cada neurona de la capa siguiente. Para el caso de imágenes pequeñas (como la de la figura 2.5) puede ser viable, pero teniendo tan sólo una imagen de $4\text{x}4px$ y 4 neuronas en la única capa oculta, se tendrían $16 * 4 = 64$ pesos. Si aplicáramos este sistema a una imagen 3D en escala de grises con $124 * 124 * 70 = 1076320vx$, se necesitarían más de un millón de pesos por cada neurona que haya en la primera capa oculta. Esto hace que sea completamente inviable usar este tipo de redes para imágenes a partir de cierto tamaño. En esta sección se presentan las redes neuronales convolucionales (CNN), que reducirán en gran medida el nº de pesos necesarios en la red neuronal y aprovecharán técnicas del procesamiento de imágenes para encontrar patrones.

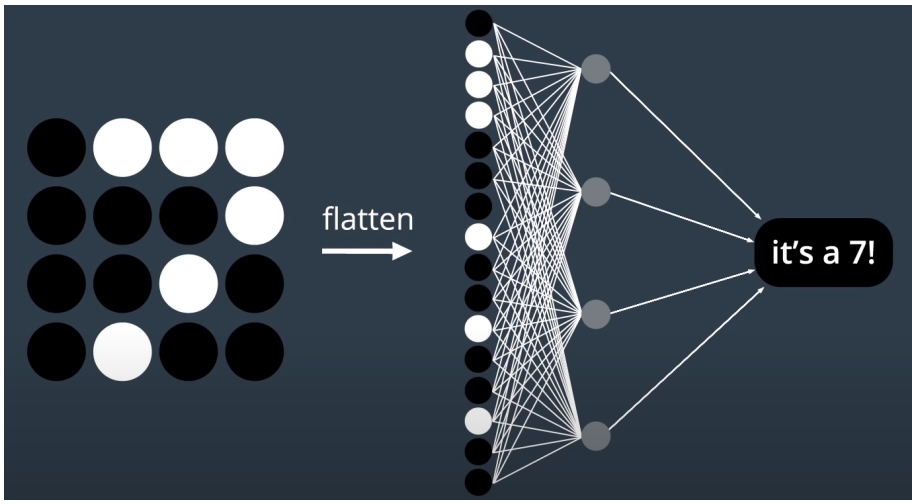

Figura 2.5: Imagen de 4x4 px aplanada y usada como entrada en una FCNN con 4 neuronas en su única capa oculta. No se muestra su capa de salida. Imagen del curso Intro to Deep Learning with PyTorch de Udacity.

Cambiando la arquitectura de la red vista en la figura 2.5 por una CNN, obtendríamos una arquitectura similar a la vista en la figura 2.6. En esta CNN se ha reducido el nº de conexiones de la capa de entrada a la capa oculta de 64 a 16, además, como veremos más adelante, los pesos de las 4 neuronas son compartidos, esto significará que sólo necesitamos 4 pesos distintos.

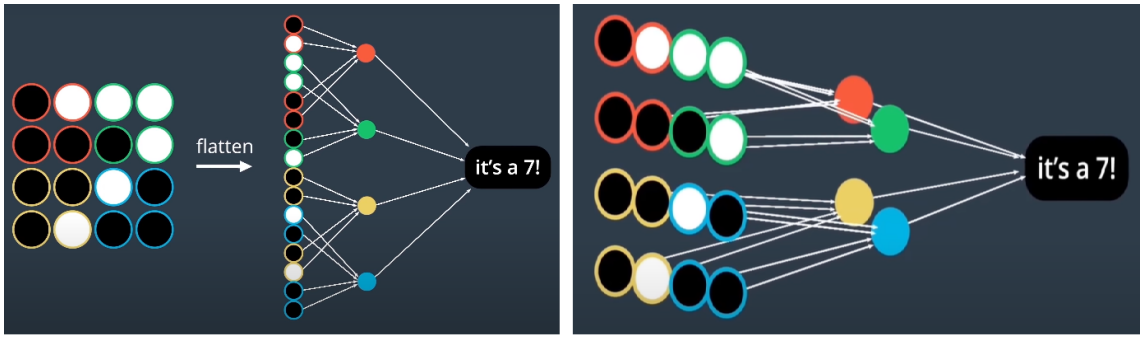

Figura 2.6: Imagen de 4x4 px usada como entrada en una CNN. Ambas figuras muestran la misma arquitectura, estando en la figura de la izquierda la imagen aplanada y en la figura de la derecha se muestra la matriz 4x4 como entrada. Imágenes del curso Intro to Deep Learning with PyTorch de Udacity.

2.2.1. Tipos de capas

Antes de describir cada tipo de capa con la que puede construirse una CNN es importante mencionar la **profundidad**. Cada capa tendrá una profundidad asociada que no hay que confundir con la profundidad de una ANN. En una CNN si una capa tiene profundidad k, querrá decir que en esa capa hay un stack de k imágenes en escala de grises. Lo normal es que cada imagen del stack represente características distintas de la imagen de entrada.

Las capas más comunes usadas en una CNN son: Capa Convolucional, Capa de Pooling, Capa ReLU y Capa Completamente Conectada (FC). En la figura 2.7 (missinglink, 2020) se puede ver un ejemplo en el que la imagen de un barco pasa por varias capas de convolución + ReLU, Pooling y por último FC, dando la predicción de la clase de la imagen.

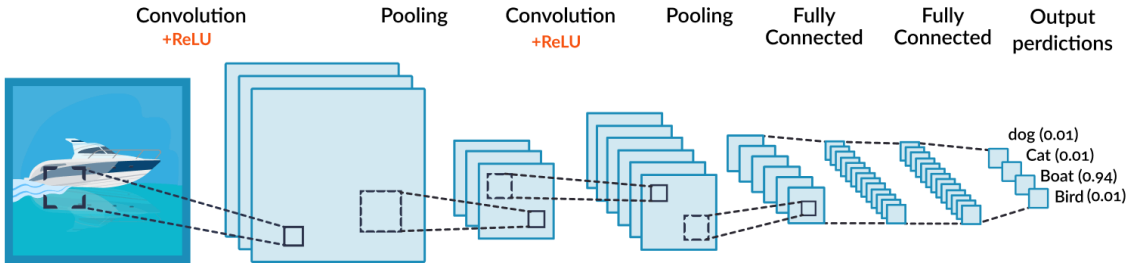

Figura 2.7: Red Neuronal Convolucional simple en la que la imagen de un barco es clasificada.

Capa convolucional

Es la capa principal de este tipo de redes. Es descrita por 4 hiperparámetros:

- Número de filtros, **K**.
- Tamaño del kernel, **F**.

- Paso, **S**.
- Padding, **P**.

Esta capa va a tener como entrada una imagen con una profundidad K_0, le aplicará un padding de **P** píxeles/vóxeles alrededor de la imagen y realiza la operación de convolución del stack de imágenes y de K filtros de tamaño F en todas sus dimensiones excepto en la dimensión de la profundidad, que será de tamaño K_0. El paso con el que desplazamos los filtros sobre las imágenes será S. Suponiendo que la imagen inicial tiene 2D, esta operación se hará frente a una entrada de tamaño $W_1 \times H_1 \times D_1$ y dará como resultado una imagen de tamaño:

- $W_2 = \frac{W_1 - F}{S} + 1$
- $H_2 = \frac{H_1 - F}{S} + 1$
- $D_2 = K$

De la misma forma que se han calculado W y H se puede calcular cualquier nº de dimensiones. También es importante notar que aunque se puede elegir cualquier valor para S, F y P, es habitual en las arquitectura modernas que las convoluciones se realicen con $S = 1$, $F = 3$ y $P = 1$, de esta forma quedaría: $W_2 = \frac{W_1 - F + 2P}{S} + 1 = \frac{W_1 - 3 + 2}{1} + 1 = W_1$. Haciendo esto no varía el tamaño de la imagen.

En la figura 2.8(Li, Krishna, y Xu, 2020) se muestra un ejemplo de una capa de convolución de profundidad 2 con imagen de entrada 5×5 con $1px$ de padding y profundidad 3, siendo los filtros de tamaño 3 y aplicándose con un paso de 3. Se obtendrá una imagen 3×3 con profundidad 2.

Para que salida tenga profundidad 2 será necesario usar 2 filtros, cada uno de estos filtros será de tamaño $3 \times 3 \times 3$, por lo que cada uno tendrá 27 valores, uno por cada píxel. Estos valores son los pesos de las neuronas de la red neuronal y no están definidos por el usuario como los hiperparámetros, en cambio se incializarán de forma aleatoria y se irán modificando acorde al algoritmo de optimización utilizado. Entrenar una CNN significa encontrar unos valores para los filtros que minimicen el error (dado por la función de pérdida). Adicionalmente, también habrá que entrenar la capa FC, que no es más que una red neuronal como ya se ha visto previamente.

Capa Pooling

El objetivo de esta capa es reducir el tamaño de las imágenes para reducir la memoria necesaria y el coste computacional.

De forma similar a la capa de convolución, en la capa de pooling o reducción se usa un filtro que se aplica a toda la imagen. Se diferencian en que esta capa usa un sólo filtro de profundidad 1 que es aplicado a todas las imágenes del stack de entrada, por lo que esta capa mantiene la misma profundidad de la capa anterior. Otra diferencia está en la operación a realizar, en la capa de convolución se aplica un kernel con determinados valores a toda la imagen, en la capa de pooling se aplica es la función MAX.

Los parámetros necesarios para definir una capa de pooling son:

- Tamaño del kernel, F.
- Paso, S.

Y si se tiene una entrada de tamaño $W_1 \times H_1 \times D_1$, la salida será:

- $W_2 = \frac{W_1 - F}{S} + 1$
- $H_2 = \frac{H_1 - F}{S} + 1$

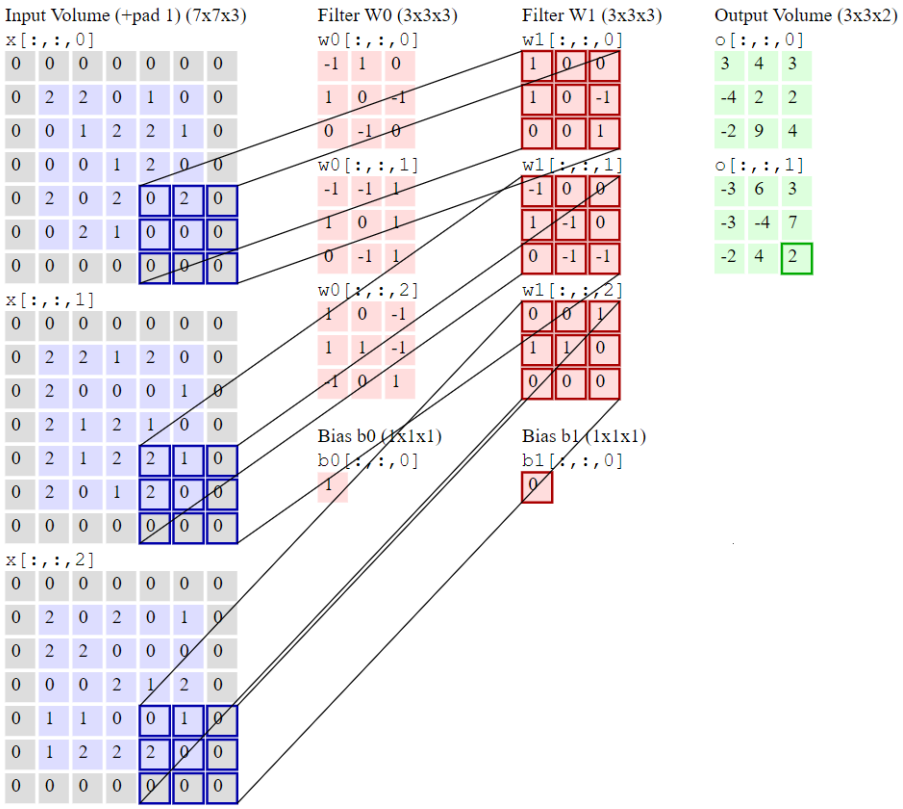

Figura 2.8: Capa de convolución de profundidad 2 (se usan dos filtros) con kernel de tamaño $3 \times 3 \times 3$ aplicado a un volumen de entrada de tamaño $7x7x3$, volumen generado al aplicar padding de 1 px a una imagen de tamaño $5x5$ con 3 canales. El resultado es un volumen de $3x3x2$

- $D_2 = D_1$

Los parámetros F y S determinan cómo se reducirá la imagen, siendo común usar $F = 2$ y $S = 2$ para reducir el tamaño a la mitad. Reducir demasiado la imagen al hacer pooling puede provocar un efecto muy destructivo.

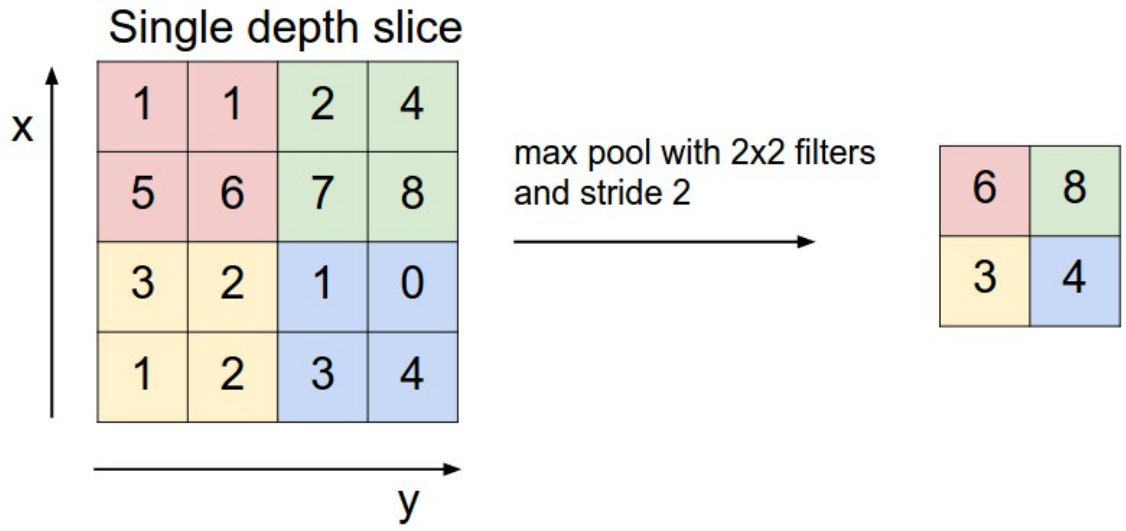

Figura 2.9: Imagen 4×4 a la que se ha aplicado un pooling de $F = 2, S = 2$. Cada color de la imagen de la izquierda indica las entradas del para la operación MAX, cada color de la imagen de la derecha indica la salida de dicha operación. Figura tomada del curso CS231n de Standford University.

Capa ReLU

Esta capa aplica la función de activación no lineal *ReLU* ($max(0, x)$) a cada elemento (píxel o vóxel) de la entrada. Se introduce después de la capa de convolución y es a veces llamada la etapa de detección (Goodfellow y cols., 2016, 335).

Capa FC

Se usa para obtener la puntuación de clase de cada píxel/vóxel de la capa anterior, a la que está completamente conectada (cada nodo de la capa anterior está conectado a todos los de esta capa). Es la salida de la red ya que es la última capa. En problemas de clasificación esta capa tendrá C nodos, siendo C el nº de clases. En problemas de segmentación esta capa tendrá tantos nodos como clases haya multiplicado por el nº píxeles/vóxeles que haya en la capa anterior, entendiéndose la segmentación como etiquetar cada píxel/vóxel con la probabilidad que tiene de pertenecer a cada clase.

Esta capa funciona como una ANN normal, incluyendo los pesos y su actualización.

2.2.2. Funciones de Pérdida

En las CNNs, al igual que en la inmensa mayoría de algoritmos de Deep Learning, se usa el descenso por gradiente estocástico o alguna variación como método para optimizar y aprender hacia un objetivo (y). Para ello necesitamos una representación matemática de dicho objetivo, que será la función de pérdida. Esta función de pérdida deberá evaluar correctamente cómo de buena es la predicción ($\hat{y}$). A continuación se describirán varias funciones de pérdida en relación con el problema de segmentación.

Binary Cross-Entropy

La entropía cruzada es una medida utilizada para calcular la diferencia entre dos distribuciones de probabilidad. **??**. Resultará útil si comparamos el objetivo y con la predicción $\hat{y}$. La fórmula de la entropía cruzada binaria (BCE) es la siguiente:

$$L_{BCE}(y, \hat{y}) = -(ylog(\hat{y}) + (1 - y)log(1 - \hat{y})) \tag{2.3}$$

Weighted Binary Cross-Entropy

Cross entropía binaria con pesos (WCE) es una variante de BCE. En esta variante se aplica un coeficiente a cada ejemplo positivo. Es muy útil cuando los datos están sesgados, como por ejemplo una segmentación de un elemento muy pequeño en comparación con el fondo. La formula es la siguiente:

$$L_{WBCE}(y, \hat{y}) = -(\beta ylog(\hat{y}) + (1 - y)log(1 - \hat{y})) \tag{2.4}$$

Para reducir el número de falsos negativos usar $\beta > 1$, para reducir el número de falsos positivos usar $\beta < 1$ (Jadon, 2020).

Dice Loss

El coeficiente Dice se usa como métrica para calcular la similitud entre dos imágenes. En 2016 se adaptó para usarlo como función de pérdida (Cardoso y cols., 2017). La fórmula es la siguiente:

$$DL(y, \hat{p}) = 1 - \frac{2y\hat{p} + 1}{y + \hat{p} + 1} \tag{2.5}$$

Siendo $p\epsilon[0, 1]$ la probabilidad de que un píxel/vóxel pertenezca a una clase, siendo la suma de todas todas las probabilidades (para un determinado píxel/vóxel) igual a 1.

Se le añade 1 en el numerador y denominador para evitar que haya 0 en el numerador o denominador.

2.3– Arquitecturas para segmentación semántica

En los últimos años se han desarrollado diversas arquitecturas para CNN consolidándola como el mejor método actual para resolver ciertos problemas en tratamiento de imágenes, como la clasificación o la segmentación. Esto ha sido posible gracias en parte a los avances obtenidos en el reto ImageNet (Deng y cols., 2009). Imagenet es una base de datos con más de 10 millones de imágenes etiquetadas a mano, entre las que hay 1000 categorías distintas. Desde 2010 se ha organizado una compteción anual donde los participantes tienen que desarrollar y entrenar el mejor modelo para clasificar estas imágenes.

- El ganador del reto de 2012 fue un equipo de la Universidad de Toronto con la arquitectura AlexNet (Krizhevsky, Sutskever, y Hinton, 2012), usando filtros 11x11 y siendo el primer modelo en usar ReLU como función de activación, algo que se ha convertido en estándar.

- En 2014 VGGNet (VGG), del Grupo de Geometría Visual de la Universidad de Oxford, fue de los que obtuvo mejor resultado en la competición con 2 versiones distintas, VGG16 con 16 capas y VGG19 con 19 capas (Simonyan y Zisserman, 2014). Ambas versiones emparejan convolución y pooling, usan filtros para las capas de convolución, 2x2 para las capas de pooling, acabando la red en 3 capas completamente conectadas. VGG fue el primer modelo en usar filtros tan pequeños, mostrando que se obtenían mejores resultados al hacerlo.

- El ganador del reto de 2015 fue ResNet, desarrollada por Microsoft Research (He, Zhang, Ren, y Sun, 2015). ResNet utiliza una misma distribución de capas repetida durante toda la red. Posee 152 capas, siendo la primera vez que se alcanzaba un número tan alto de capas de forma práctica. Era difícil tener tantas capas a causa del problema del desvanecimiento de gradiente (Hochreiter, 1998) que se da al entrenar por backpropagation y resulta en que, al hacer la propagación hacia atrás, cada capa consecutiva reduce el error propagado llegando ”desvanecerse” resultando en un difícil entrenamiento. Esto es solucionado por ResNet al conectar capas lejanas para que el error pueda propagarse entre estas más rápido, llamando a estas conexiones *skip connections*.

Estas arquitecturas resultaron en hitos importantes para el desarrollo de CNN, a continuación se describirán varias arquitecturas para segmentación semántica.

2.3.1. Fully Convolutional Network

Fue propuesta en 2014 por investigadores de la Universidad de California en Berkeley (Long, Shelhamer, y Darrell, 2014) obteniendo los mejores resultados hasta la fecha en segmentación semántica mediante el uso de redes convolucionales. Un punto clave fue tomar una entrada de un tamaño arbitrario y producir una salida de un tamaño correspondiente. Usa como base los modelos AlexNet, VGGNet y GoogLeNet (Szegedy y cols., 2014), donde se sustituyeron las capas completamente conectadas con capas convolucionales con filtros 1x1 y añadieron una capa convolucional con filtro 1x1 y profundidad C+1 para predecir las puntuaciones de cada clase, C el nº de clases y añadiendo una más para el fondo.

Se compararon los modelos FCN-AlexNet, FCN-VGG16 y FCN-GoogLeNet, obteniendo los mejores resultados con FCN-VGG16 y convirtiéndose este en el modelo base. Con el objetivo de ganar más nivel de detalle en la salida, se aumentó la resolución de la salida en 32x usando interpolación bilineal. También se usaron *skip connections* para conectar varias capas con la capa final. En la figura 2.10 se puede ver un ejemplo de la arquitectura FCN (?, ?).

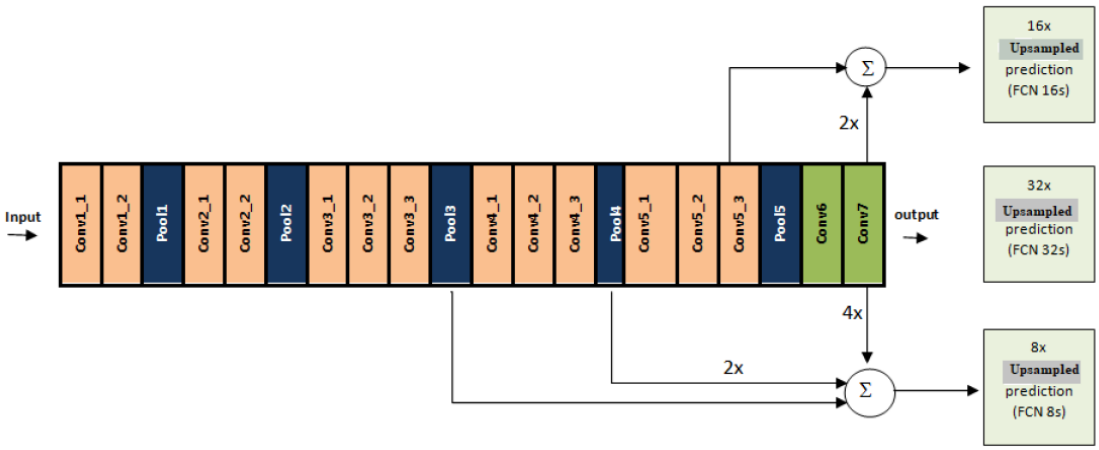

Figura 2.10: Arquitectura de FCN32, FCN16 y FCN8.

2.3.2. Deconvnet

La red Deconvnet (Noh, Hong, y Han, 2015) tiene una red convolucional y otra red deconvolucional. Para la red convolucional utiliza casi la misma topología que VGG16, con 13 capas de convolución 2 capas FC, cambiando sólo en la última capa que no es de clasificación ya que en Deconvnet está conectada a la red deconvolucional. La red deconvolucional tiene una topología

inversa a la red convolucional, resultando en una salida de igual tamaño a la entrada. Esta red tiene capas de deconvolución, de *un-pooling* y de ReLU. Todas las capas de una Deconvnet extraen características de la entrada excepto la última capa de la red deconvolocuional, que genera una probabilidad de pertenencia a cada clase para cada píxel/vóxel. Una arquitectura de esta red se puede ver en la figura 2.11 **??**.

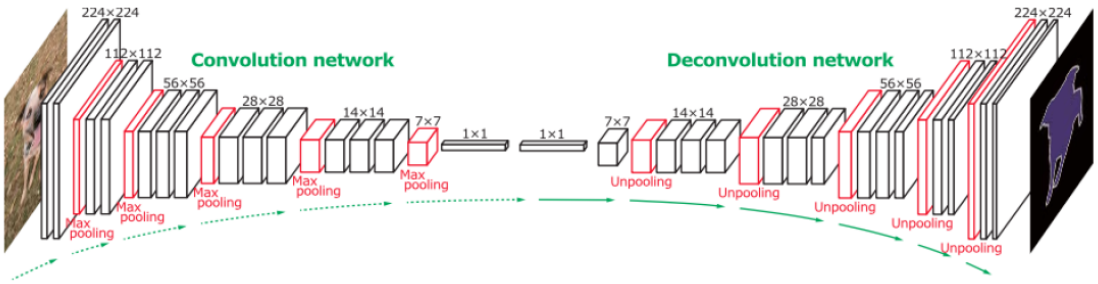

Figura 2.11: Arquitectura Deconvnet.

A continucación se describen brevemente las operaciones de unpooling y deconvolución (más correctamente llamada convolución transpuesta). En la figura 2.12 (Noh y cols., 2015) se puede ver un ejemplo gráfico.

Unpooling

La operación unpooling es una operación que reconstruye la zona de pooling mantiendo sólo el píxel correspondiente a aquel que fue seleccionado en la capa de pooling correspondiente (figura 2.12. Para implementar esto se usan variables *switch*, que almacena la posición en la que se encontraba el valor máximo al hacer pooling. Esta estrategia resuelve el problema de pérdida de información espacial que tiene el pooling (Zeiler, Taylor, y Fergus, 2011).

Deconvoloución

La salida de una capa de unpooling aporta información espacial pero muy dispersa. Para aumentar la densidad de información se usan las operaciones de deconvolución (Noh y cols., 2015). Las operación de convolución toma varias entradas y las transforma en un único valor al aplicar un filtro. La operación deconvolución toma un único valor y lo transforma en varias salidas al aplicar el mismo filtro usado en la convolución correspondiente.

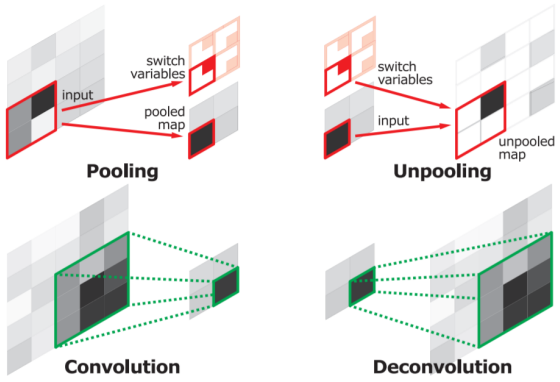

Figura 2.12: Figura que ilustra las operaciones unpooling y deconvolución.

2.3.3. U-Net

U-Net es una arquitectura en forma de U con una parte reductora (izquierda) y otra parte expansiva (derecha) simétrica. Esta arquitectura ha mostrado muy buenos resultados con pocos ejemplos de entrenamiento al beneficiarse fuertemente de la aumentación de datos (data-augmentation)(Ronneberger, Fischer, y Brox, 2015). Fue el ganador de 2015 del reto ISBI para segmentación neuronal con resultados muy superiores al segundo puesto.

La parte reductora tiene una arquitectura típica de CNN. Cada paso de esta parte tiene dos convoluciones consecutivas con filtros 3x3 (sin padding), cada convolución seguida de por una capa ReLU, con una capa de pooling 2x2 al final de ambas convoluciones. La salida de la última capa de cada paso es usada en la capa equivalente de la parte expansiva. Después de varios pasos como este comienza la parte expansiva con una topología inversa a la convolucional, en donde en cada paso expansivo es añadida la salida de la última capa de cada paso reductor. Cada capa de pooling de la parte izquierda es sustituido en la parte derecha por una çonvolución hacia arriba", similar a la operación unpooling + deconvolución. La última capa es una convolución 1x1 para mapear cada píxel/vóxel con el nº correspondiente de clases. En la arquitectura original se tienen en total 23 capas convolucionales. En la figura 2.13 puede verse la arquitectura original.

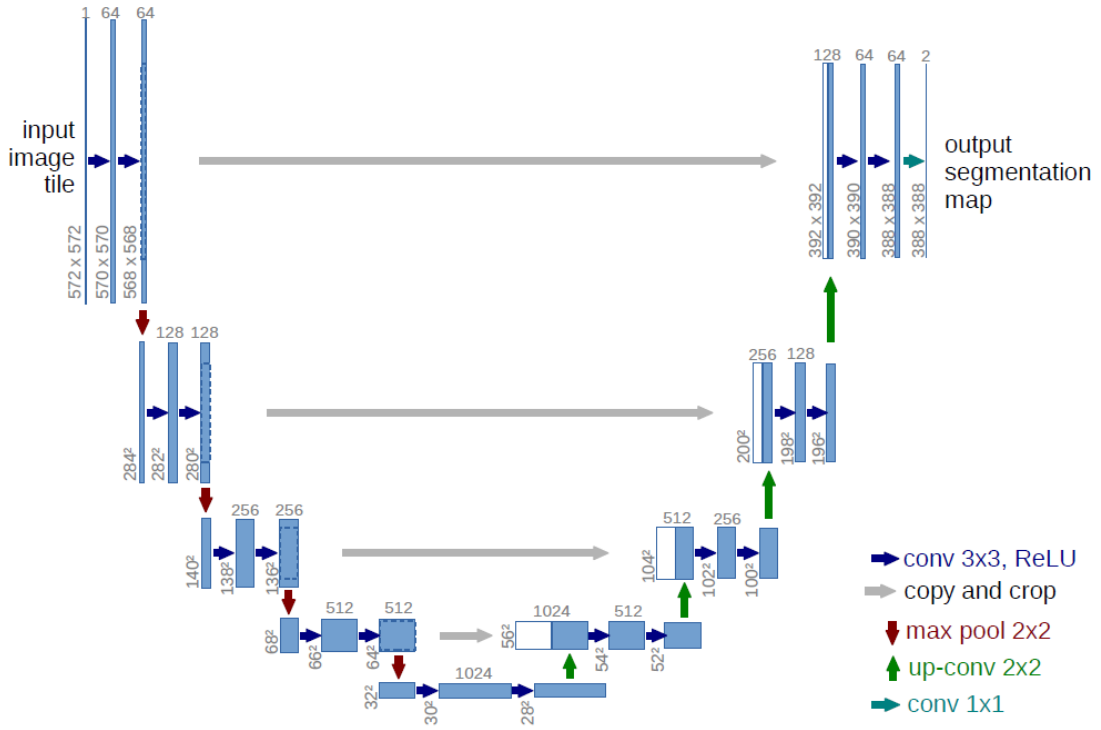

Figura 2.13: Arquitectura original U-Net propuesta por Ronneberger et al en U-Net: Convolutional Networks for Biomedical Image Segmentation.

2.4– Aplicaciones recientes

En esta sección se describirán dos aplicaciones recientes en las que se realiza segmentación celular usando la arquitectura U-Net. Primero se describirá el proyecto en general, luego se definirá brevemente el algoritmo de segmentación seguido dividiendo las operaciones en: preprocesado (preparar los datos para usarlos en el modelo U-Net, procesado (características del modelo) y postprocesado (operaciones posteriores al resultado dado por el modelo), después se mostrará resultados obtenidos y por último mencionará el software resultante de estas investigaciones.

2.4.1. U-Net para cuenteo, detección y morfometría, detección y morfometría de células. (2019)

Proyecto

Ha sido desarrollado por investigadores de la Universidad de Friburgo (Alemania) en colaboración con la Universidad de Berna (Suiza) y la Universidad de París V Descartes (Francia) (Falk y cols., 2019). El coautor designado es Olaf Ronneberger, uno de los principales contribuyentes en la creación de la arquitectura U-Net (Ronneberger y cols., 2015).

En este proyecto se aplica la arquitectura U-Net para solucionar el problema del tratamiento de imagen que hay que realizar al gran volumen de datos originados por microscopios antes de que puedan ser analizados por investigadores.

Pipeline

Preprocesado	Procesado	Postprocesado
GT:Espaciado de 1 vóxel entre células GT:Células etiqueta 1, fondo 0 IMyGT:Troceo de imagen a 236x236x100 vx IMyGT:Data Augmentation: -Rotación -Deformación suave -Incremento de intensidad	U-Net Función de pérdida: -Entropía Cruzada con Pesos -Peso alto en espaciado entre células Optimizador ADAM: -Learning rate 10^{-5} β_1 : 0,9, β_2 : 0,999 150000 Iteraciones	Ninguno

Cuadro 2.1: Pipeline que siguen los datos de inicio a fin. GT significa *groundtruth* y se refiere a la imagen ya segmentada. IM es la imagen de entrada.

La segmentación semántica etiqueta cada vóxel con la clase correspondiente, no distingue entre distintas instancias de un mismo objeto si estos están en contacto. Para conseguir esta distinción entre células se ha aplicado un espaciado de un vóxel alrededor de cada célula en la imagen segmentada a mano. De esta forma se podrá comprobar la forma de todas las células y se podrán contabilizar.

Para que la imagen ocupe menos memoria y el entrenamiento sea más rápido, se ha dividido la imagen en trozos de 236x236x100.

Se usa data augmentation (aumentación de datos) tanto en la imagen de entrada como en la imagen objetivo para mejorar el aprendizaje. Gracias a esto en este proyecto se sostiene que no son necesarias más de 10 imágenes anotadas para el correcto entrenamiento de un modelo.

Respecto al modelo entrenado, se ha usado como función de pérdida la entropía cruzada con pesos a nivel de vóxel. Esto quiere decir que cada vóxel en cada imágen va a tener un peso propio. El peso de cada vóxel viene dado por la siguiente fórmula:

$$w(x) := w_{bal}^{'} + \lambda w_{sep} \tag{2.6}$$

Donde $\lambda \epsilon R_{\geq} 0$ controla la importancia de la separación de instancia. $w_{bal}^{'}$ y w_{sep} son calculados de la siguiente forma:

$$w_{bal}^{'}(x) := \begin{cases} 1 & y(x) > 0 \\ v_{bal} + (1 - v_{bal}) * exp(-\frac{d_1^2(x)}{2\sigma_{bal}^2}) & y(x) = 0 \\ 0 & y(x) desconocido \end{cases} \tag{2.7}$$

Donde d_1 es la distancia hacia la instancia de célula más cercana, $v_{bal} \epsilon [0, 1]$ es un factor se usa para reducir la importancia de los vóxeles de fondo y σ_{bal} es la desviación estándar deseada.

$$w_{sep}(x) := exp(-\frac{(d_1(x) + d_2(x))^2}{2\sigma_{sep}^2}) \tag{2.8}$$

Donde d_2 es la distancia a la segunda instancia de célula más cercana y σ_{sep} es la desviación estándar deseada.

En el experimento realizado con datos volumétricos se han

Resultados

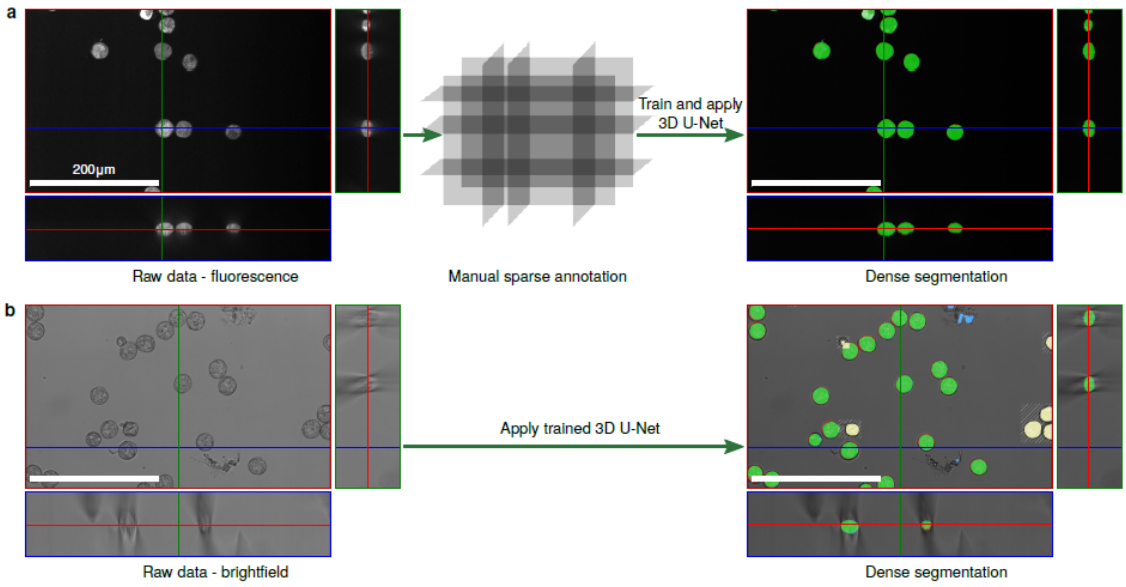

Figura 2.14: Segmentación volumétrica. (a) Segmentación de una imagen con un modelo que se ha entrenado con imágenes del mismo dataset. (b) Segmentación de una imagen con un modelo entrenado con un dataset distinto.

La métrica utilizada para cuantificar la calidad del resultado es intersección sobre unión (IoU).

$$M_{IoU}(A, B) := \frac{|A \cap B}{A \cup B} \tag{2.9}$$

Donde A es el conjunto de vóxeles pertenecientes a la imagen perfectamente etiquetada y B el conjunto de vóxeles pertencientes a la predicción. Un $M_{IoU} \epsilon [0, 1]$ igual a 1 equivale a una predicción perfecta mientras que igual a 0 equivale a una predicción en la que ningún vóxel coincide. Acorde a los experimentos realizados, se tomará un valor $\sim 0{,}7$ como una buena segmentación, siendo $\sim 0{,}9$ una segmentación equivalente a la humana.

Se hicieron varios experimentos, alcanzando siempre un $M_{IoU} > 0{,}8$ para datos volumétricos.

Software

Este algoritmo ha sido implementado como un plugin de FIJI (Schindelin y cols., 2012), una herramienta opensource para procesamiento de imágenes. Este plugin viene con modelos preentrenados para la segmentación de células y está disponible como repositorio oficial de imageJ *http://sites.imagej.net/Falk/plugins/*, con código fuente incluido.

El framework usado como backend es **caffe** (Jia y cols., 2014), que será necesario instalar previamente. Los binarios de caffe así como modelos entrenados para segmentación 2D y 3D están disponibles en *https://lmb.informatik.uni-freiburg.de/resources/opensource/unet*

2.4.2. Precisa y versátil segmentación 3D de tejido vegetal a resolución celular. (2020)

Proyecto

Ha sido realizado por investigadores de la universidad de Heidelberg (Alemania), colaborando con la Universidad Técnica de Munich (Alemania) y la Universidad de Warwick (UK). Los coautores son Adrian Wolny y Lorenzo Cerrone.(Wolny y cols., 2020)

Este proyecto está hecho con la idea de utilizar las últimas y mejores técnicas para segmentación precisa de datos volumétricos a nivel celular y hacerlo accesible a personas no expertas en la materia de visión por ordenador. El resultado de este proyecto es el software PlantSeg.

Pipeline

Preprocesado	Procesado	Postprocesado
GT:Bordes con 2 vóxeles de anchura	U-Net	Transformada de la distancia
GT:Desenfoque Gaussiano	Funciónes de pérdida probadas:	Detectado de centroides
IM y GT:Data Augmentation	-Entropía Cruzada Binaria	Algoritmo watershed
- Volteo Horizontal y Vertical	-Pérdida Dice	Grafo de adyacencia
- Rotación en plano XY	Optimizador ADAM:	Particionamiento de grafo
- Deformación elástica	-Learning rate 10^{-5}	
IM: Noise Augmentation	-β_1 : 0,9, β_2 : 0,999	
IM,GT:Troceo de imagen a 170x170x80 vx	100000 iteraciones	

Cuadro 2.2: Pipeline que siguen los datos de inicio a fin. GT significa *groundtruth* y se refiere a la imagen ya segmentada. IM es la imagen de entrada.

En el preprocesado, se parte de una imagen con etiquetado perfecto y se le aplica la función *find_ boundaries* de la librería scikit (Pedregosa y cols., 2011) (van der Walt y cols., 2014) obteniendo los bordes de las células con 2 vóxeles de anchura. A la imagen con los bordes se le aplica un desenfoque Gaussiano para reducir los componentes de alta frecuencia, lo que ayuda a prevenir el sobreajuste. Luego se aplica varias técnicas de *data augmentation* en el espacio a la imagen de entrada y a la imagen de bordes. Tras esto se aplica *noise augmentation* a la imagen de entrada. Por último se divide la imagen en trozos de 170x170x80 vx.

En el procesado, se usa una arquitectura U-Net y se prueba con Entropía Cruzada Binaria y con Pérdida Dice, usando un optimizador ADAM en ambos casos.

En el postprocesado, se parte de la imagen con la probabilidad de que cada vóxel pertenezca al borde de la imagen, se aplica un umbral para binarizar la imagen, donde cualquier probabilidad $> 0{,}4$ se considera borde. A la imagen binarizada se le aplica la transformada de la distancia, dando a cada vóxel de fondo un valor igual al vóxel de borde más cercano. A esta imagen se le aplica un suavizado gaussiano con $\sigma = 2{,}0$ y se calculan los mínimos locales para seleccionar las semillas. Estas semillas son usadas en el algoritmo watershed para obtener la segmentación final.

El algoritmo watershed trata el valor de los vóxeles como si describiese una topología.

1. Se colocan unas semillas iniciales, desde aquí se empezará la inundación. Cada semilla tendrá una etiqueta distinta.
2. Los vecinos de cada vóxel etiquetado se insertan en una cola prioritaria teniendo más prioridad aquellos con un valor más bajo.
3. El vóxel con más prioridad sale de la cola, si todos sus vecinos etiquetados tienen la misma etiqueta, se le pone esa etiqueta. Todos los nuevos vecinos sin marcar son puestos en la cola prioritaria.
4. Se vuelve al paso 3 hasta que se vacía la cola.

Resultados

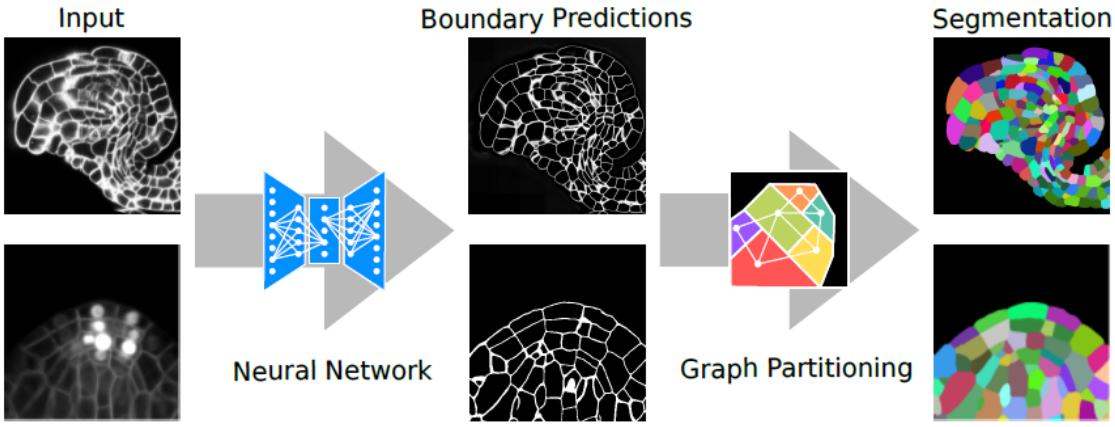

Figura 2.15: Segmentación de tejido vegetal usando PlantSeg. En el primer paso se predicen los bordes de las células usando una red U-Net 3D. En el segundo paso se aplica un algoritmo de particionamiento de grafo para segmentar cada célula.

Para la etapa de la predicción de bordes se han usado tres métricas:

- Precisión: Nº vóxeles correctamente etiquetados como borde en la predicción entre el nº de vóxeles etiquetados como borde en la predicción.
- Exhaustividad: Nº de vóxeles etiquetados correctamente en la predicción entre el nº de vóxeles etiquetados como bordes en la imagen con perfecto etiquetado.
- Puntuación F1: $F1 = 2 * \frac{Precision * Exhaustividad}{Precision + Exhaustividad}$

Para estas 3 métricas mientras más alto sea el valor mejor, siendo 1 la medida perfecta y 0 la peor.

Para la segmentación final se ha usado la variación de información (VOI), definida como:

$$VOI = H(seg|GT) + H(GT|seg) \tag{2.10}$$

Donde H es la entropía condicional, seg es la predicción de la segmentación y GT es la segmentación perfecta. Para esta métrica mientras más bajo sea el valor mejor, siendo 0 el valor perfecto.

Se han han realizado muchas pruebas en este proyecto, en la tabla 2.3 se puede ver dos pruebas en las que se compara la entropía cruzada binaria y la pérdida Dice.

Función de pérdida	Precisión	Exhaustividad	Puntuación F1
Entropía Cruzada Binaria	0.806 ± 0.071	0.799 ± 0.028	0.800 ± 0.036
Pérdida Dice	0.744 ± 0.096	0.933 ± 0.017	0.824 ± 0.062

Cuadro 2.3: Pruebas relevantes realizadas.

Como se ven en los resultados parece que la pérdida Dice es ligeramente superior aunque no por mucho.

Software

PlantSeg es un programa que puede ser ejecutado por una interfaz gráfica o por línea de comandos. Dispone de todos los modelos preentenados en este proyecto, así como de todas las opciones

para entrenar nuevos modelos o hacer inferencia. El software con las instrucciones para su uso están disponibles en *https://github.com/hci-unihd/plant-seg*.

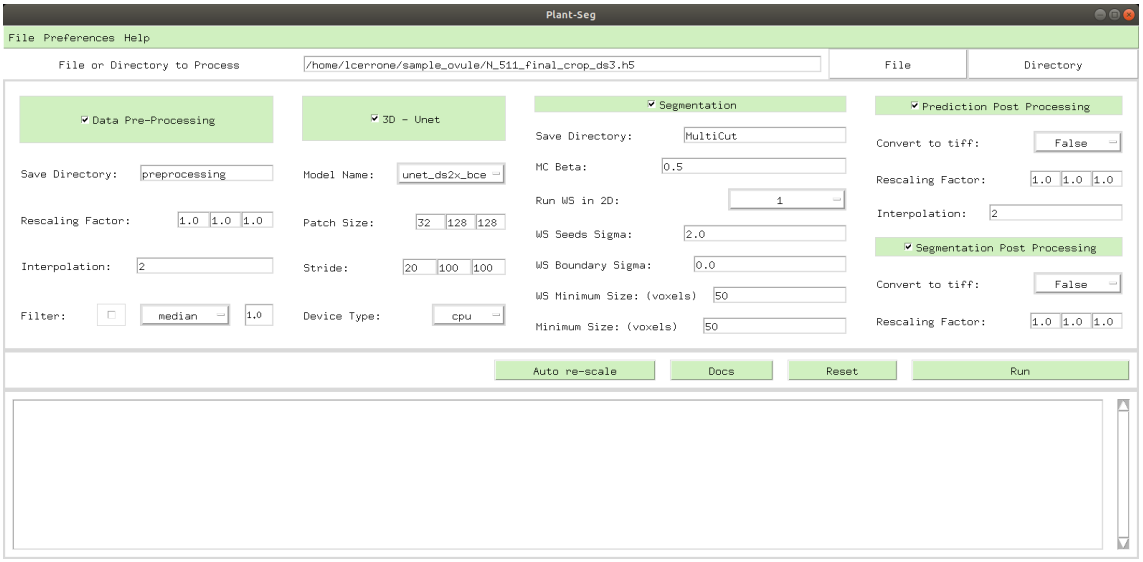

Figura 2.16: Interfaz gráfica del programa PlantSet. Se pueden ver las distintas opciones para cada paso del procesado.

CAPÍTULO 3

Datos de entrada

Los datos iniciales son 20 ficheros con formato hdf5, cada uno contiene la siguiente:

- **imageSequence**: Imagen inicial con la forma $(Z1, 1024, 1024)$
- **imageSequenceInterpolated**: Imagen incial interpolada, ahora con la forma $(Z2, 1024, 1024)$. Siendo $Z2 > Z1$.
- **labelledImage3D**: Resultado de etiquetar correctamente la imagen inicial interpolada $(Z2, 1024, 1024)$.
- **centroidOfRois**: Coordenadas del centroide de cada célula.
- **usedZScale**: Valor usado en la interpolación.

A continuación se analizarán las imágenes **imageSequenceInterpolated** y **labelledImage3D**.

imageSequenceInterpolated

El tipo usado para almacenar el valor de cada vóxel es *f8* que, según la documentación de numpy es el equivalente a un número de 64-bit float.

Se han comprobado los valores máximo y mínimo de las 20 imágenes y estos son los resultados:

35.0 4095.0	32.0 4095.0	33.0 4095.0	0.0 65535.0	31.0 4069.0	24.0 4095.0	19.0 4095.0	0.0 65535.0	0.0 65535.0	24.0 4095.0
22.0 4095.0	22.0 4095.0	23.0 4095.0	30.0 4095.0	31.0 4095.0	30.0 4095.0	25.0 4095.0	30.0 4095.0	23.0 4095.0	19.0 4095.0

Cuadro 3.1: Valores mínimo y máximo de los píxeles de las imágenes de entrada.

Se ha analizado el histograma de una imagen para entender mejor la distribución de valores. Podría ser que se pudiera simplificar el nº de valores distintos que tiene la imagen para aumentar el rendimiento. La imagen analizada tiene 32 de valor mínimo y 4095 de valor máximo.

En la figura 3.1 se puede ver un histograma en el que en el eje x representa el valor de un píxel y el eje y representa el nº de píxeles con ese valor. En esta figura se puede comprobar que los valores están concentrados en el rango $[0, 500]$, pero da la impresión de que no hay ningún elemento en el resto de valores. Para obtener una mejor visualización del resto en la figura 3.2 se puede ver otro histograma con el eje y limitado a 4000, donde se aprecia los distintos valores que un píxel puede tomar.

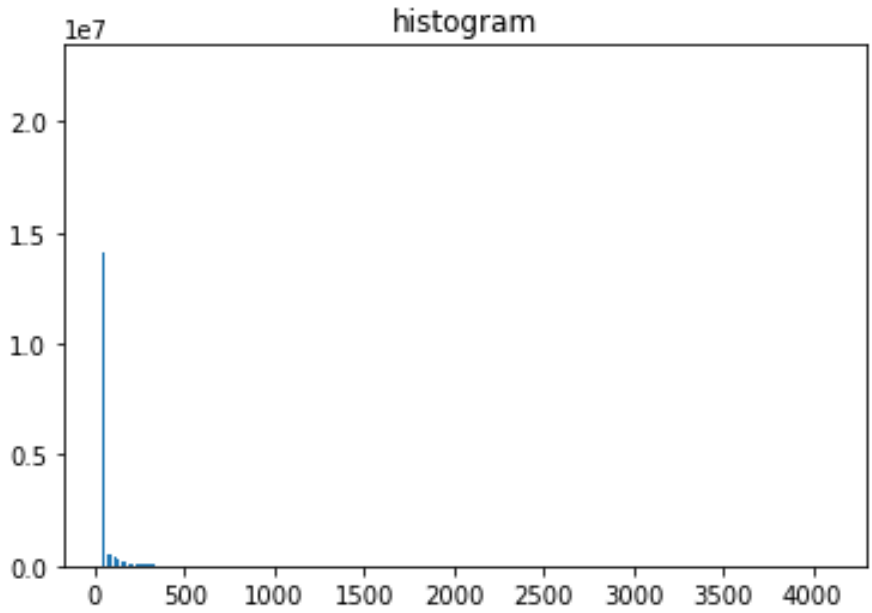

Figura 3.1: Histograma de todos los valores

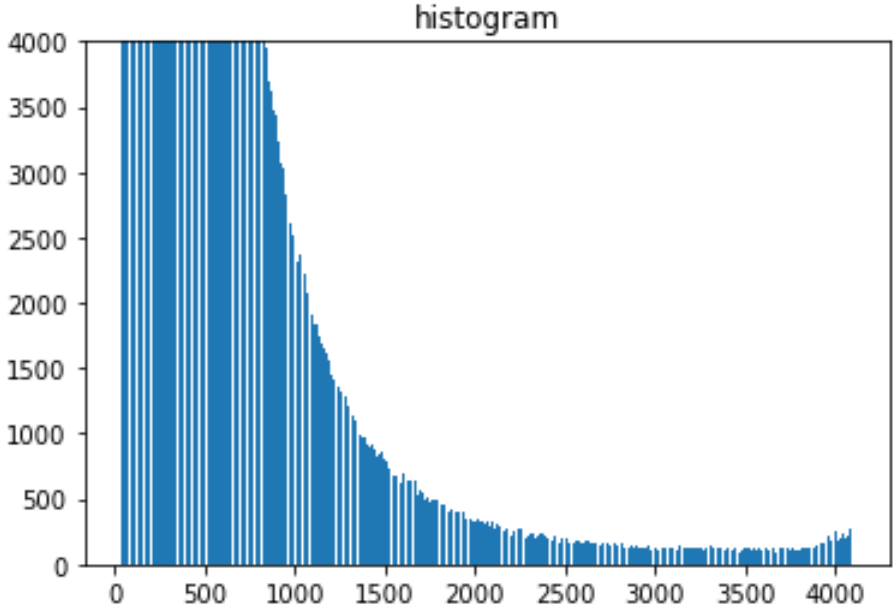

Figura 3.2: Histograma de todos los valores con el eje y limitado a 4000

CAPÍTULO 4

Diseño

En este capítulo se describirá el flujo por el que pasan los datos suministrados hasta dar lugar a la segmentación objetivo, sin entrar en detalles sobre la las herramientas usadas en la implementación.

4.1– Esquema general

El principal cuello de botella al usar técnicas de deep learning es el hardware requerido para ello.

Sería ideal ser capaz de entrenar una CNN usando las imágenes provistas directamente, pero a causa de su gran resolución no es viable. Una imagen de resolución $(200, 1024, 1024)vx$ con una precisión de 8 bytes ocupa en memoria (siempre hablaremos de memoria de GPU) $size(imagen) = \frac{1024*1024*200*8}{1024*1024} = 1600MB$. Si bien podríamos almacenar esta imagen en memoria, las CNN se caracterizan por aplicar un gran número de filtros distintos en paralelo a una imagen de entrada, utilizando los resultados de la aplicación de estos filtros en el siguiente paso de la CNN. Tal y cómo se verá en más detalle en la sección sobre las arquitecturas seleccionadas, es completamente inviable para nosotros usar la resolución original. Es por ello que como parte del preprocesado se han simplificado los datos de entrada.

Por otro lado, también sería ideal contar con la última tecnología en GPU o TPU. En un artículo sobre segmentación de tejido celular 3D usando U-Net se explica cómo han usado 8 NVIDIA GeForce RTX 2080 Ti GPUs para realizar 100K iteraciones (Wolny y cols., 2020). El acceso a un hardware superior permite el uso de arquitecturas más complejas, datos más precisos o mayor nº de iteraciones, lo que va a influir en el resultado obtenido.

Para tener acceso a mejor hardware se usarán tecnologías cloud, donde puedes alquilar el uso de GPUs por hora siendo común pruebas gratuitas especialmente para estudiantes.

Para la parte del entrenamiento y la inferencia se usará Jupyter Notebook, esto hará que sea fácil trasladar el código entre distintas máquinas, locales o en internet.

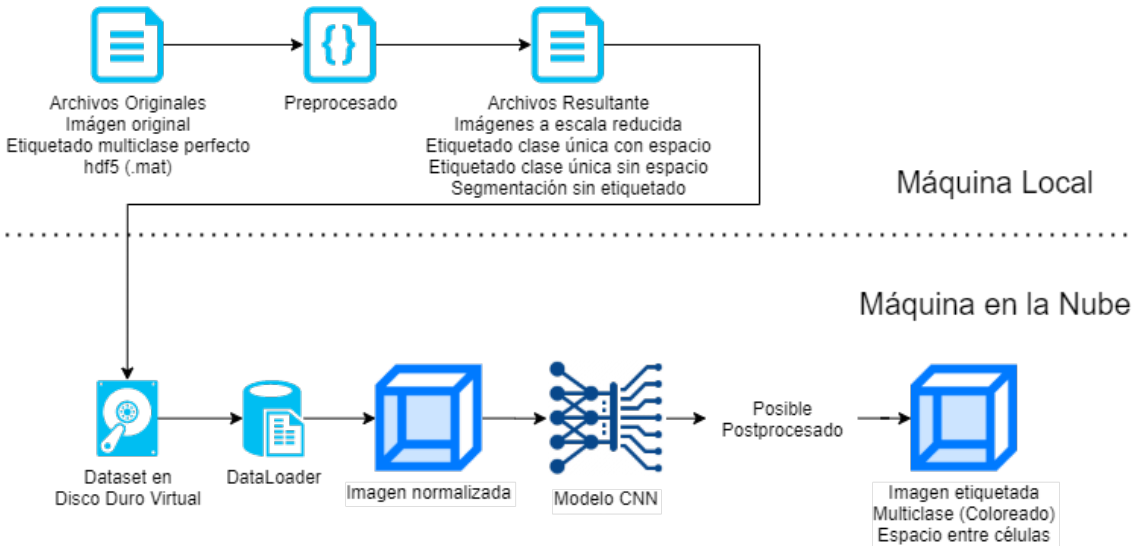

Figura 4.1: Diseño general

En la figura 4.1 se muestra el diseño general sobre el proceso en el que los datos pasan de los archivos hdf5 inicialmente provistos hasta completar la segmentación requerida.

La etapa de preprocesado no requerirá un uso intensivo de GPU, por lo que podrá realizarse en cualquier máquina. En este caso se usará la máquina local ya que la máquina en la nube es más costosa de utilizar.

En esta etapa se preparán los datos con 4 objetivos:

1. Reducir la resolución de la imagen de entrada, reduciendo así la memoria necesaria para almacenarla en GPU.

2. Generar las imágenes etiquetadas correctamente para tenerlas como objetivo.

3. Reducir el tamaño de los archivos resultantes, ya que estos serán usados en servicios cloud y tendrán que ser subidos y descargados con frecuencia.

4. Modificar el orden de las dimensiones y añadir una *singleton dimension* para el canal. Este formato es necesario para su uso en la CNN.

Tras esto, se generarán nuevos archivos hdf5 y se subirán a un disco duro virtual, al cual se accederá por un Notebook. El dataset será leído por un DataLoader, el cual realizará todo el preprocesado que faltase a las imágenes de entrada, como puede ser la normalización. Las imágenes de entrada podrán entonces ser usadas como entrada en el modelo seleccionado, cuya salida, dependiendo del modelo, podrá requerir un postprocesado o no. El resultado final será un etiquetado multiclase, que visualmente se traducirá a un coloreado de células.

4.2– Preprocesado local

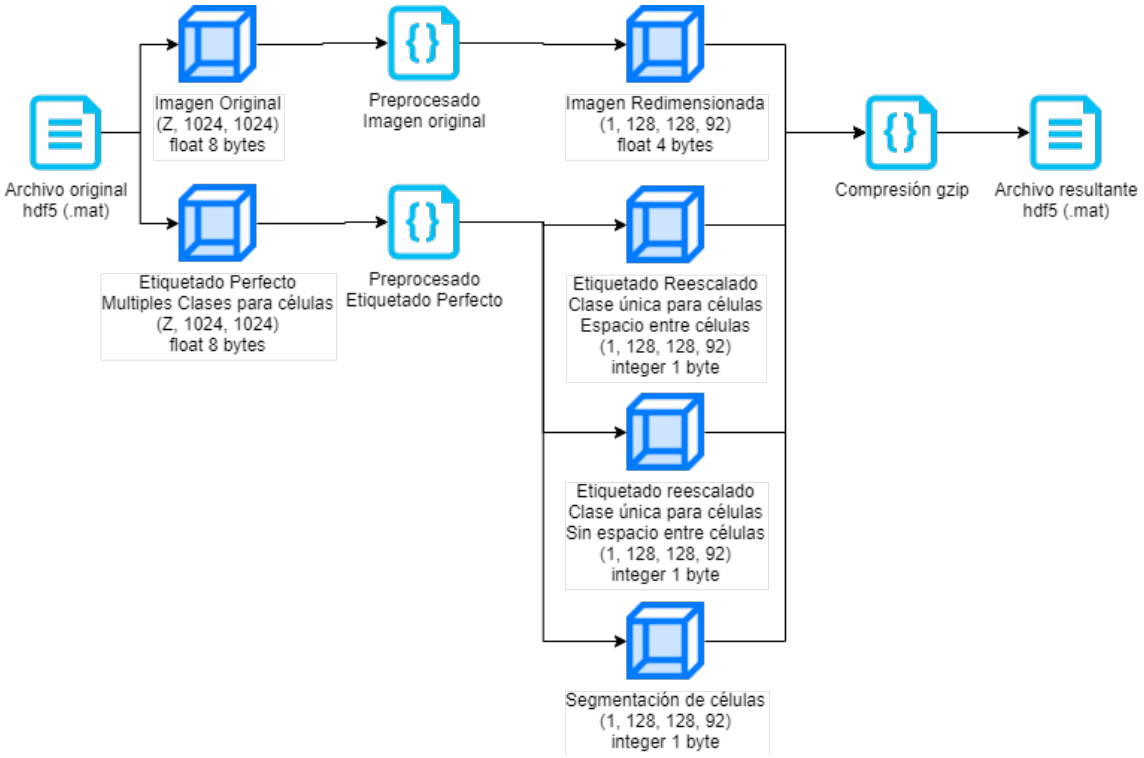

Figura 4.2: Preprocesado llevado a cabo en la máquina local.

En la figura 4.2 se puede ver con más detalle el resultado que se obtendría en esta etapa.

Reducir la resolución de la imagen de entrada

El formato de las imágenes inicialmente es (Z, X, Y), siendo para todas las imágenes $X = Y = 1024$ y $Z\epsilon[216, 368]$. Además de reducir la resolución de las imágenes, es importante que todas tengan la misma, de lo contrario no podrán ser usadas en operaciones batch. También es esencial que la imagen de entrada y la imagen etiquetada no se deformen demasiado.

Generar las imágenes etiquetadas correctamente

El problema principal que nos encontramos en las imágenes con el etiquetado perfecto es que las células, siendo instancias de una misma clase (la clase c̈élula"), tienen etiquetas distintas. En una segmentación semántica cada vóxel es etiquetado con la clase a la que se cree que pertenece. Si usáramos el etiquetado actual necesitaríamos una clase por cada célula, pero eso no tiene mucho sentido ya que el nº de células en una imagen puede variar, además todas las células tienen características similares. Probaremos dos soluciones a este problema:

1. Añadir espacio entre células para que ninguna esté en contacto y hacer que todas tengan 1 como etiqueta.

2. Etiquetar los bordes de las células con 1 para luego aplicarle DT Watershed.

Los 2 etiquetados distintos son preprocesados en este paso.

Reducir el tamaño de los archivos resultantes

Al estar trabajando con herramientas en la nube será recomendable reducir el tamaño de los archivos lo mayor posible.

Todas las imágenes tienen una precisión de 8 bytes, cuando en un estudio previo se concluyó que no era necesaria tanta precisión para los valores de esas imágenes, especialmente para el etiquetado, en el que usa valores enteros entre 0 y 100. Si hacemos que los valores de la imagen de entrada pasen de 8 bytes a 4 bytes, estaremos reduciendo su tamaño a la mitad. De forma similar si hacemos que la los valores de la imagen etiquetada pase de 8 bytes a 1 bytes, estaremos reduciendo su tamaño a una octava parte. Con 1 byte de precisión podremos almacenar hasta 256 valores distintos, suficiente ya que sólo tendremos 2 valores distintos: 0 para el fondo y 1 para la célula o los bordes.

Además se comprimirán las imágenes, aunque esto sólo reducirá el tamaño del archivo, el tamaño de los tensores almacenados en memoria será el mismo. Esta compresión será muy efectiva en las imágenes etiquetadas, ya que los elementos sólo tendrán 2 valores distintos.

Después de hacer estos cambios se habrá reducido el tamaño del fichero pero no de los tensores cargados en memoria al leer esas imágenes. Esto es así por dos motivos:

1. Los tensores al cargarlos en memoria estarán sin ningún tipo de compresión. Cada valor del tensor ocupará el espacio correspondiente a su tipo de dato.

2. Las operaciones implementadas en los frameworks de deep learning limitan el tipo de dato que pueden tener los tensores. Están implementados a muy bajo nivel para optimizar las operaciones, por lo que será necesario convertir los tipos a unos que acepten, si no lo son ya.

Modificar el orden de las dimensiones

El cambio de las dimensiones se debe principalmente a las herramientas usadas en etapas posteriores, que requieren los ejes ordenados como (x, y, z).

Añadir una *singleton dimension* es necesario para tener en cuenta el canal de la imagen, ya que esto es usado en los componentes de la CNN con arquitectura U-Net.

4.3– Entrenamiento

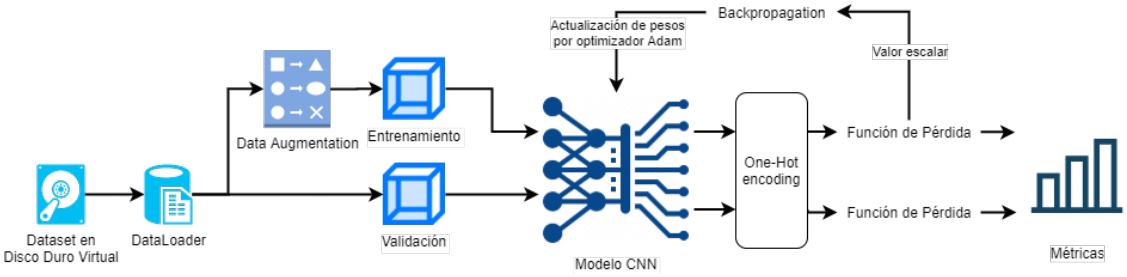

Figura 4.3: Pasos llevados a cabo en el entrenamiento

En la figura 4.3 se muestra el proceso que se lleva a cabo durante el entrenamiento de un modelo. Como ya se mencionó antes este proceso deberá realizarse con una GPU de alto rendimiento, por lo que en este proyecto se realizará en una máquina en la nube. Antes de comenzar este proceso los datos ya habrán sido cargados en un disco duro virtual.

Se utilizará un DataLoader, encargado de leer el dataset y realizar el preprocesado conveniente. Será importante normalizar los datos de entrada para que tengan un valor en rango $[0, 1]$, esto se hará siempre. Adicionalmente, se probarán distintas técnicas como estandarizar el histograma, o *data augmentation*. Al etiquetado objetivo no se le aplicará normalización ni estandarización, sólo se le aplicará *data augmentation*. Al ser importante que la segmentación de la imagen etiquetada siga siendo correcta, al aplicar *data augmentation* no se deformará la imagen.

Se dividirá el dataset en $70\,\%$ entrenamiento, $15\,\%$ validación y $15\,\%$ test. Se han escogido estos porcentajes ya que se cuenta con pocos datos de entrada y son similares entre sí.

En el entrenamiento se usará un *batch* de datos, se harán pruebas con valores en el rango $N\epsilon[1, 4]$. Durante cada *epoch*, tanto la predicción obtenida como la imagen con el etiquetado perfecto se transformarán con el *one-hot encoding*, lo que les dará el formato (N, C), donde N es el tamaño del *batch* y C el nº de clases, que siempre será $C = 2$. Esta disposición de datos transforma cada imagen en 2 vectores, uno por cada clase, lo cual hará que sea muy eficiente calcular diferencias entre la predicción y la imagen con etiquetado perfecto. Para calcular estas diferencias está la función de pérdida, que tendrá un valor bajo si hay poca diferencia y alto si hay mucha diferencia. El objetivo de cualquier algoritmo de optimización será disminuir esta función de perdida. Para la función de pérdida se probarán *Dice Loss* y *Cross Entropy Loss* con pesos.

El valor dado por la función de pérdida con los datos de entrenamiento se usará en la *back-propagation* para calcular los gradientes los cuales se usarán para optimizar los pesos con el optimizador Adam. El valor dado por la función de pérdida con los datos de validación será el que determine si se está mejorando el modelo de forma programática. Tras realizar el entrenamiento completo se usarán los datos de test para obtener predicciones y se analizarán con la matriz de confusión y el índice de Jaccard (*intersection over union* o *IoU*, apoyándonos con representación visual del resultado.

4.4– Inferencia

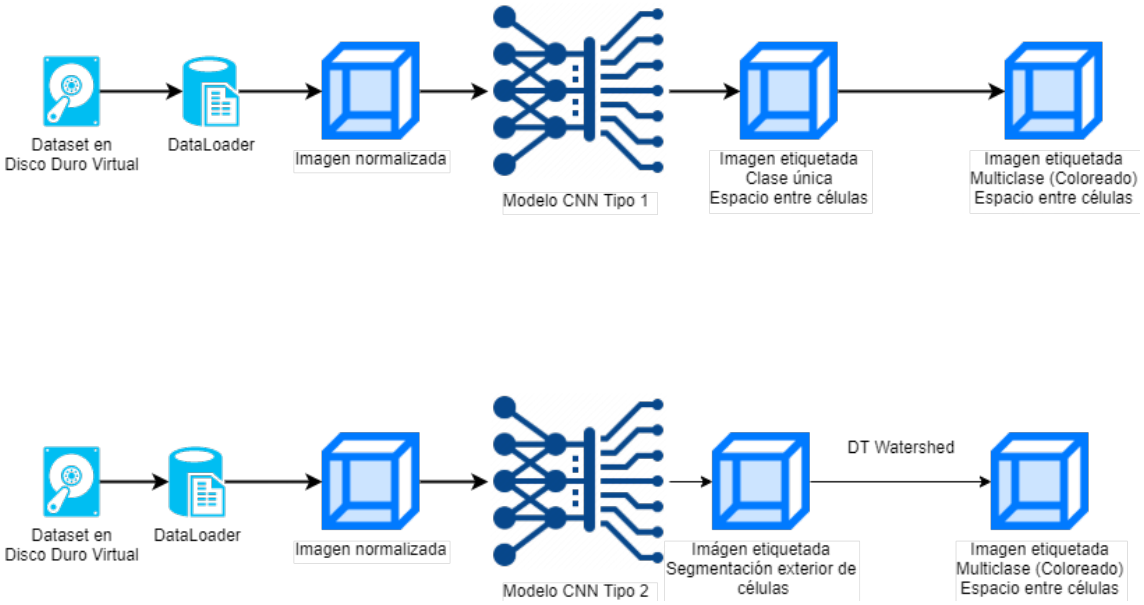

Figura 4.4: Se muestran dos métodos distintos para obtener la imagen correctamente etiquetada.

Una vez el entrenamiento ha finalizado correctamente querrá decir que las capas encargadas de la convolución tienen unos pesos adecuados para que la red pueda predecir con cierta confianza el etiquetado correspondiente, siempre y cuando tenga de entrada imágenes similares a las usadas en el entrenamiento.

Para conseguir una imagen con el etiquetado correcto se han seguido dos enfoques:

1. Enfoque 1: Usar un sólo modelo que llamaremos de **Tipo 1** que ha sido entrenado usando como objetivo imágenes en la que todas las células están etiquetadas con el valor 1 y hay un espaciado (valor 0) entre ellas. Una vez hecho esto se asignará una etiqueta distinta para cada célula.

2. Enfoque 2: El modelo de **Tipo 2** dará como predicción un etiquetado con los bordes de las células de valor 1. A la predicción se le aplica del algoritmo DT Watershed.

4.5– Arquitectura de las Redes Neuronales Convolucionales

Se probará principalmente una arquitectura de tipo U-Net. Se usará un modelo con una arquitectura reducida para facilitar las pruebas que decidirán el uso de técnicas como *data augmentation*, qué reescalado se hará a las imágenes, el tamaño del batch, la función de pérdida o el optimizador a utilizar. Una vez la mayor parte de estos aspectos haya sido decidido, se pasará a probar la arquitectura completa.

En la figura 4.5 se puede ver la arquitectura completa de la red U-Net utilizada, arquitectura usada con éxito para segmentación volumétrica densa (Özgün Çiçek, Abdulkadir, Lienkamp, Brox, y Ronneberger, 2016), basada en la arquitectura U-Net propuesta por Ronneberger et al (Ronneberger y cols., 2015). Se ha omitido el tamaño de las imágenes ya que la red acepta imágenes de cualquier tamaño, Lo único a tener en cuenta es el nº de canales que tiene la imagen. Los nº mostrados en las capas determinan la profundidad de cada capa o, lo que es lo mismo, el nº de filtros distintos usados en cada capa. En la capa inicial, que simboliza la imagen de entrada, el número 1 indica que la imagen debe tener tan sólo 1 canal de entrada (escala de grises). Este tipo de arquitectura también es válida para imágenes 2D o 3D, la diferencia estaría en que las convoluciones y el pooling sean sobre 2 dimensiones o sobre 3 dimensiones.

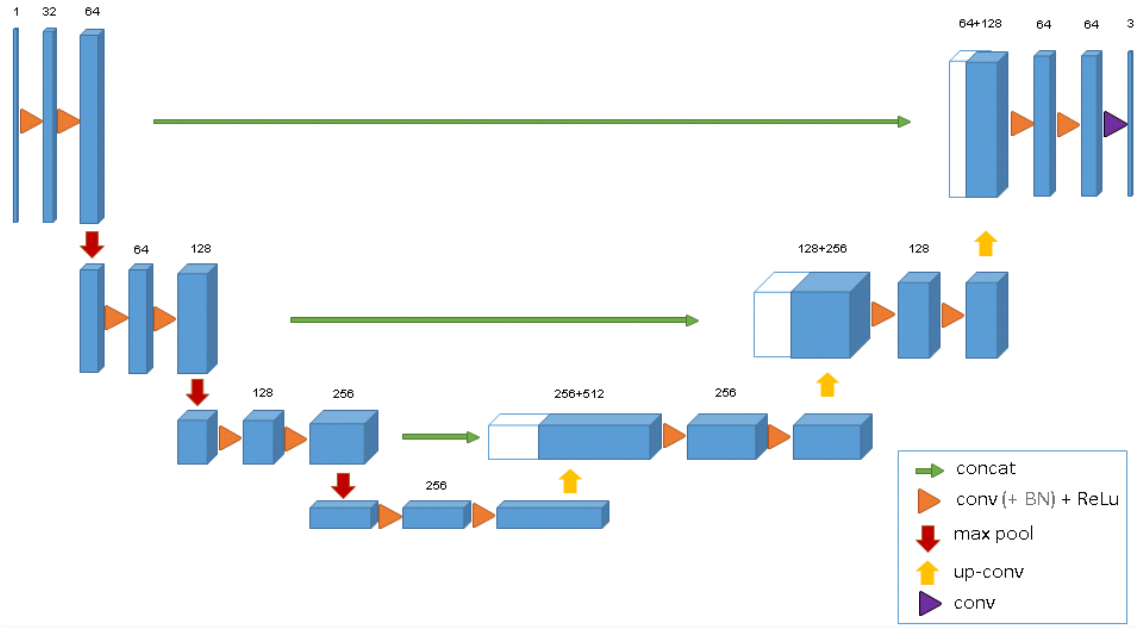

Figura 4.5: Arquitectura U-Net completa.

En la figura 4.6 se muestra la arquitectura anterior eliminando varias capas de convolución y una de pooling. Los modelos generados por esta arquitectura serán menos fiables, pero el entrenamiento será más rápido, por lo que es útil para realizar comparaciones al cambiar técnicas de preprocesado o de entrenamiento.

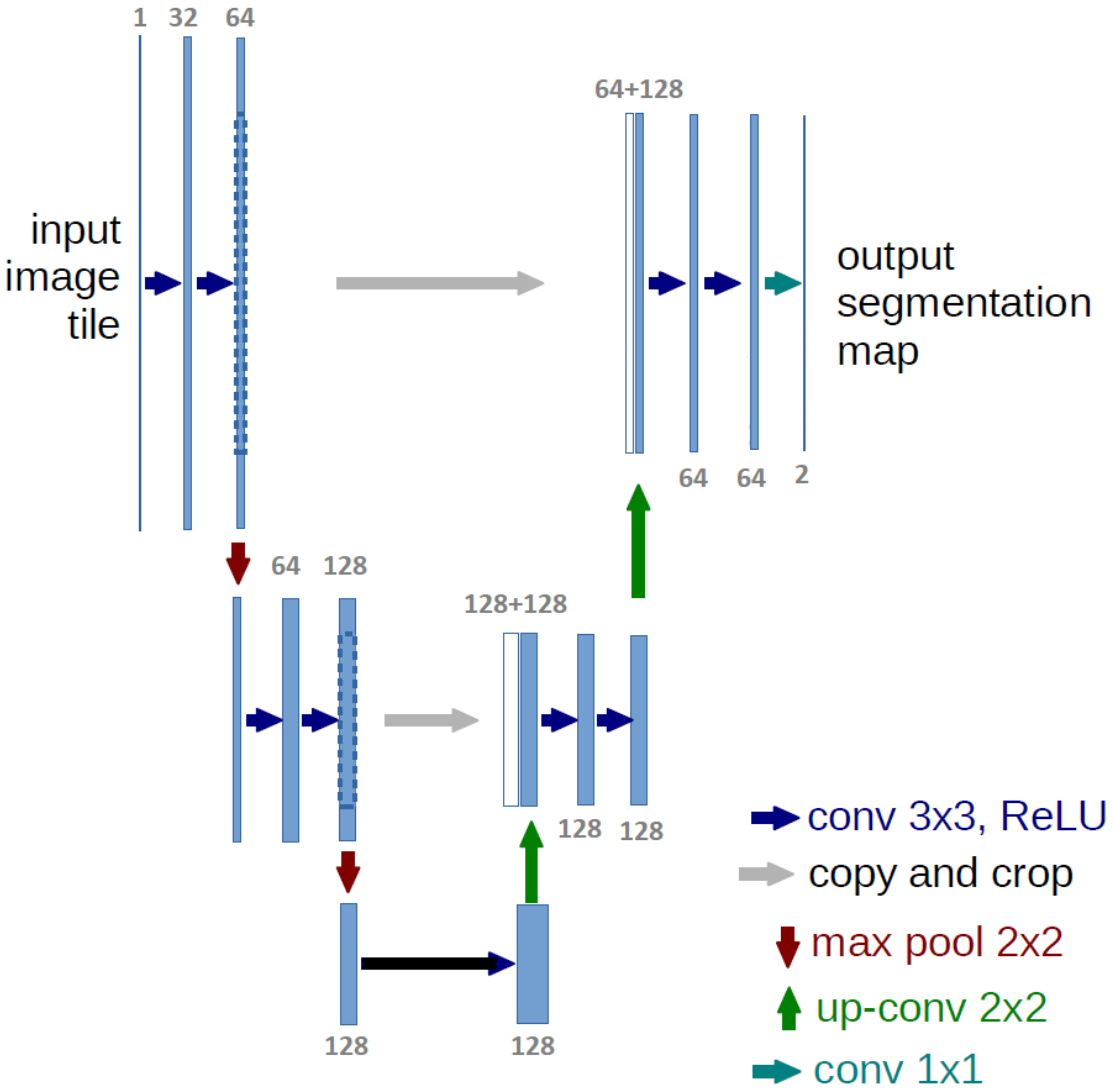

Figura 4.6: Arquitectura U-Net completa.

CAPÍTULO 5

Implementación

En este capítulo se mostrarán las tecnologías utilizadas para llevar a cabo el diseño deseado.

5.1– Lenguaje y framework

El proyecto se ha desarrollado completamente en lenguaje de programación Python 3.7 y el framework PyTorch 1.6 (Paszke y cols., 2019).

La elección del lenguaje Python es sencilla: los frameworks más utilizados en deep learning pueden ser usados en Python. Con una búsqueda rápida en GitHub bajo los términos ”deep learning”, ”machine learningz ”neural network”se puede ver que los frameworks más populares pueden ser usados en Python.

- TensorFlow con 148k estrellas.
- Keras con 49.4k estrellas.
- PyTorch con 41.5k estrellas.
- Caffe con 30.8k estrellas

Además, tal y como se muestra en el índice TIOBE (Tiobe, 2020), Python es el tercer lenguaje de programación con mejor puntuación teniendo en cuenta su presencia en los buscadores web.

Aún así Python tiene es lento cuando se compara con lenguajes como C y esto se debe principalmente a 2 motivos:

1. Es interpretado, no compilado. Al compilar un programa escrito en un lenguaje compilado el compilador optimiza el programa con una multitud de técnicas, como el desenrollado de bucles, que elimina o reduce las instrucciones de control de un bucle. En Python el intérprete sólo accede a la instrucción a realizar, no realiza ninguna tarea de optimización.

2. Es de tipado dinámico. Esto quiere decir que el intérprete no sabe de qué tipo es una variable hasta que intenta acceder a ella, por lo que antes de acceder a su valor debe comprobar el tipo de variable. Esto hace que las variables en Python sean fáciles de declarar y asignar, pero reduce el rendimiento.

Estas desventajas se pueden reducir gracias a que Python tiene la capacidad de llamar a subrutinas en C o C++. Prácticamente todos los frameworks en los que se realizan operaciones con un alto coste computacional tienen la parte crítica de su código escrita en C o C++.

En el ecosistema de Python, la base matemática para cualquier framework que haga operaciones matriciales es NumPy (Van Der Walt, Colbert, y Varoquaux, 2011), ya que añade soporte para arrays de n dimensiones y una gran multitud de operaciones de forma eficiente.

Todos los frameworks populares de Python son una buena elección, ya que todos tienen *bindings* a lenguajes de bajo nivel compilados, pero con la facilidad de uso de Python. Aunque Tensorflow puede ser difícil de utilizar, cuenta con Keras, que es un framework desarrollado por encima que abstrae muchas operaciones. Pytorch, basado en Torch, también ofrece un fácil uso y buen rendimiento.

5.2– Preprocesado local

El primer paso a realizar es preparar los datos, para ello se han usado las siguientes librerías:

- **scikit-image** (van der Walt y cols., 2014). Es una librería de procesamiento de imágenes. Se ha usado para encontrar los bordes de la imagen con etiquetado multiclase sin espaciado entre células. Esta imagen no podía ser usada directamente en el entrenamiento de la CNN, por lo que se ha seguido la estrategia de encontrar los bordes de cada célula con la función *(segmentation.search_boundaries())* (Wolny y cols., 2020), después de encontrar los contornos se substraen de la imagen original y se cambian todas las etiquetas a 1, consiguiendo así un espaciado entre células, tal y como se indica en (Falk y cols., 2019). También se ha usado esta librería (*measure_label()*) para asegurarnos de que el nº de células no varía al realizar las operaciones de preprocesado (por ejemplo, podría ser que dos células con etiqueta 1 eliminaran el espacio entre sí, convirtiendose en la misma instancia de célula).

- **SciPy** (Virtanen y cols., 2020). Contiene diversos módulos con algoritmos útiles en el ámbito científico. En el submódulo *ndimage* se pueden encontrar los algoritmos relativos al procesamiento de imagen. Se ha utilizado la función *ndimage.zoom()* para reescalar las imágenes de entrada, haciendo que ocupen menos memoria y sea viable usarlas para el entrenamiento.

- **h5py** (Collette, 2013). Aporta una interfaz para trabajar con el formato de datos HDF5 (The HDF Group, 2000-2010), útil para usar en python ficheros generados en programas como MATLAB. El tratamiento de datos con esta librería es muy intuitivo ya que imita la sintaxis de NumPy. Ha sido usado para archivos .mat generados en MATLAB con los datos iniciales. Tras el preprocesado, se ha usado esta librería para generar un nuevo archivo con todas las imágenes en un nuevo archivo .mat.

5.2.1. Consideraciones

Reescalado

Para la operación de reescalado se ha usado la función *ndimage.zoom()*. Esta función hace interpolación con splines seleccionado un factor por el que hacer zoom por cada dimensión, el orden de la interpolación spline. Se han tomado las siguientes decisiones:

- **Factor para escalar hacia abajo.** Las imágenes originales ocupan $\sim 2GB$ en memoria, intentar usarlas en una U-Net sobrepasa en gran medida los 16GB de memoria disponibles en este proyecto. Se ha decidido reducir las dimensiones en $(1/8, 1/8, 1/4)$ de sus dimensiones (X, Y, Z) originales, consiguiendo una reducción de $8 * 8 * 4 = 256$ de su tamaño original, se pasa de $\sim 2GB$ a $\sim 8MB$. Con un tamaño de $8MB$ es posible realizar todo el proceso de entrenamiento e inferencia aunque se pierda calidad en la imagen. Gracias a la característica de U-Net de aceptar imágenes de cualquier dimensión, si no se hace entrenamiento por batch no es necesario asegurarse de que todas las imágenes tengan las mismas dimensiones. Por experimentación, he comprobado que los factores de las dimensiones no

deben de ser muy diferentes, ya que cambiaría el tamaño en μm^3 de cada vóxel, además de provocar una deformación elástica en los imágenes.

- **Escalado hacia arriba.** Después de escalar hacia abajo, si se quiere hacer entrenamiento por batch con la imagen completa es necesario que todas las imágenes tengan las mismas dimensiones. Para esto se comprueba las mayores dimensiones de las imágenes después de escalar hacia abajo y se elige como objetivo $(X_{obj}, Y_{obj}, Z_{obj})$. Para cada imagen (X, Y, Z) el procedimiento es crear una nueva matriz de ceros de tamaño $(X_{obj}, Y_{obj}, Z_{obj})$ y asignar a la imagen a los valores $([0 : X - 1], [0 : Y - 1], [0 : Z - 1])$. De esta forma el volumen de las células mantienen el mismo ratio entre todas las imágenes.

- **Orden.** Se ha utilizado orden 3, ya que es el que está por defecto y se han obtenido buenos resultados con él.

Encontrar bordes

Para encontrar los bordes se ha usado la función *segmentation.find_boundaries()*. Esta función realiza operaciones morfológicas para encontrar los bordes de los objetos de una imagen con valores enteros o binarios. Se han tomado las siguientes decisiones:

- **Conectividad.** Hay N tipos de conectividad siendo N el nº de dimensiones de la imagen de entrada, en nuestro caso N=3. Con conectividad 1 los vóxeles compartiendo al menos una cara son considerados vecinos. Con conectividad 3 se amplía el rango de vecinos al hacer que cualquier vóxel compartiendo una esquina también son considerados vecinos. Si dos vóxeles vecinos tienen distinta etiqueta, son bordes. Se ha elegido conectividad 3 para priorizar el que no haya células en contacto en ninguna de las 3 dimensiones.

- **Modo.** Este modo define qué vóxeles son marcados como bordes. El modo escogido es *outer*, que selecciona como bordes los vóxeles de fondo alrededor de los elementos, si hay 2 elementos en contacto, se marca su frontera como borde. Se ha escogido este modo ya que, al escoger el fondo como borde retiene la mayor cantidad de volumen celular original.

Crear nuevo fichero

La creación del nuevo fichero se ha hecho con la librería h5py. Los datos originales tienen dos puntos en los que se puede mejorar su almacenamiento para que ocupe menos espacio y sea más fácil de transferir.

- **Tipo de dato.** Tanto la imagen de entrada como la imagen etiquetada tienen el tipo de datos $f8$. Es posible pasar la imagen de entrada al tipo $f4$ y la imagen etiquetada al tipo $i1$ sin perder información. Esto va a reducir el tamaño en disco en más de la mitad. Por lo escribir las imágenes en el fichero se harán con los tipos $f4$ y $i1$.

- **Compresión.** Se ha optado por usar la compresión *gzip*, que ofrece una buena compresión a una velocidad no muy alta. Tiene 10 niveles de compresión $[0, 9]$. Se ha probado a comprimir con el nivel 9 consiguiendo una buena compresión pero es excesivamente lento, al final se ha optado por usar una compresión de nivel 4 que, aunque tiene $\sim 10\,\%$ más tamaño que con el nivel 9, el tiempo de la compresión se reduce a más de la mitad y de todas formas el resultado ya es muy bueno.

5.3– Carga de datos

Una vez todos los datos han sido preprocesados de forma local ya se puede proceder a la carga de datos. En la carga de datos también se hace un tratamiento de imagen y será necesario que éste sea rápido, ya que se realizará cada vez que se acceda a una imagen.

Para la carga de datos se han usado las siguientes librerías:

- **h5py.** La misma librería usada para crear los ficheros en la etapa anterior es usada para cargarlos.
- **PyTorch.** PyTorch es el framework de deep learning elegido para este trabajo, por lo que se sacará el máximo provecho de lo que ofrezca de base. Para procesamiento de imagen es especialmente útil el paquete *torchvision*.
    - Clase base para Dataset: Para indicar dónde encontrar los datos y qué hacer con ellos, se usará la clase *torchvision.Dataset* como base. Tan sólo hay que sobreescribir 3 métodos para que la clase sea funcional, estos son los métodos de inicialización (*__init__*), para obtener la longitud del dataset (*__len__*) y para obtener el siguiente ejemplo (*__getitem__*), en el que se realizará un preprocesado y se devolverá la imagen de entrada y la imagen etiquetada.
    - DataLoader: Es usado para administrar el dataset. Se encarga de dar un nº de ejemplos aleatorios igual al batch seleccionado.
- **TorchIO.** TorchIO (Pérez-García, Sparks, y Ourselin, 2020) es una librería que contiene herramientas para trabajar con datos 3D especializadas para imágenes médicas. Se ha utilizado la función *transforms.ZNormalization()* en la imagen de entrada, que resta sus valores por la media y los divide por la desviación estándar.
- **Kornia.** Kornia (Riba, Mishkin, Ponsa, Rublee, y Bradski, 2020) es una librería especializada en visión por ordenador con PyTorch como backend. Aquí es usada para data augmentation, volteando la imagen en cada una de las 3 dimensiones de forma aleatoria. Gracias a esto se pasa de 15 ejemplos de entrenamiento a 120. Esta diferencia en cantidad de ejemplos de entrenamiento hace que el modelo no se sobreajuste tan rápido.

Además, después de la ZNormalization, se han normalizado los valore al rango $[0{,}1, 0{,}9]$.

5.3.1. Consideraciones

Tamaño del batch

Se han hecho pruebas con varios tamaños de batch. El tamaño del batch determina cuántos ejemplos van a usarse en el entrenamiento del modelo. La entrada del modelo será de dimensión $(B, 1, X, Y, Z)$, donde B es el tamaño del batch. La ventaja de tener una entrada con este formato es se realizarán el mismo nº de operaciones matriciales sin importar el tamaño del batch. En el modelo MiniUnet3D al usar un batch de 1 los epochs tardan 11 segundos y al usar un batch de 4 tardan 7 segundos. Por cada batch utilizado se necesitan $\sim 3GB$ de memoria, por lo que se tomó como una buena opción para mejorar la velocidad de entrenamiento, aunque los resultados de este modelo eran malos ya que es una arquitectura reducida Unet3D. Al entrenar con el modelo Unet3D, se comprobó que el nº de batch máximo posible era 2 debido a que se requiere más memoria al haber más capas. Con un batch de 1 tarda 39 segundos por epoch, con un batch de 2 tarde 34 segundos. Sin embargo, los resultados para un batch de 1 eran mejores, por lo que al se optó por usar un batch de 1, provocando que el tamaño de las imágenes usadas en el entrenamiento no tenga por qué ser el mismo.

Normalización

Para normalizar los datos se han probado dos técnicas distintas, el cambio a escala $[0, 1]$ y la puntuación tipificada (ZNormalization).

El cambio a escala $[0, 1]$ es algo habitual en machine learning para evitar que se realicen operaciones con números muy altos, facilitando el cálculo de los gradientes. Cada vóxel pasará a tener el siguiente valor:

$$v_i^{'} = \frac{v_i - v_{min}}{v_{max} - v_{min}} \tag{5.1}$$

Donde $v_i^{'}$ el nuevo valor del vóxel, v_i el valor actual, v_{min} el valor mínimo en toda la imagen y v_{max} el valor máximo en toda la imagen.

La puntuación tipificada hace que los valores estén más cerca entre ellos, útil para cuando hay datos muy dispersos, su fórmula es:

$$v_i^{'} = \frac{v_i - \mu}{\rho} \tag{5.2}$$

Donde $v_i^{'}$ es el nuevo valor del vóxel, v_i es el valor actual, μ es la media de los valores de la imagen y ρ es la desviación estándar.

Tras varias pruebas, como mejor resultado se ha obtenido ha sido haciendo primero el cambia a escala $[0, 1]$ y después la ZNormalization, pese a que lo habitual en deep learning es preparar los datos de entrada con un valor $[0{,}1, 0{,}9]$.

Data Augmentation

Los primeros modelos se entrenaron sin utilizar ninguna técnica de data augmentation, lo que provocó un sobreajuste muy rápido, como se verá en el capítulo siguiente. PyTorch ofrece varias técnicas de data augmentation pero no están adaptadas para datos volumétricos, por lo que es necesario requerir a librerías externas, como Kornia.

Se ha optado por hacer transformaciones que no provoquen ninguna deformación elástica, ya que habría que aplicar una deformación elástica en la imagen etiquetada y, debido al reescalado hacia abajo hecho en la fase anterior en el que se ha perdido calidad, esto podría provocar células partidas o células en contacto entre sí.

Se han utilizado las funciones *RandomDepthicalFlip3D*, *RandomHorizontalFlip3D* y *RandomVerticalFlip3D* de Kornia para realizar estas 3 transformaciones en secuencia, todas con una probabilidad de 0.5.

Se ha probado *RandomRotation3D* pero no se han obtenido buenos resultados, ya que la parte etiquetada en la mayoría de los casos quedaba fuera de la imagen.

5.4– U-Net

Se ha implementado arquitectura una arquitectura U-Net basada basada en la implementación de Ronneberger (Ronneberger y cols., 2015). Se ha usado una implementación en PyTorch (shiba24, 2017) con la arquitectura mostrada en el capítulo anterior. Se ha desarrollado una versión de esta arquitectura llamada MiniUNet3D con menos capas. Esta versión se ha usado para iterar rápidamente y tomar decisiones como el uso da *data ugmentation* o el tamaño del batch, así como comprobar que las funciones de pérdida se han implementado correctamente.

Funciones de pérdida

Se han probado 3 funciones de pérdida: la entropía cruzada, la entropía cruzada con pesos y la función de pérdida DICE.

La entropía cruzada de descartó rápidamente ya que, al ser las dos clases muy desequilibradas daba demasiada prioridad al fondo sobre la segmentación y se tendía a perder mucha segmentación. Tras esto se probó la pérdida DICE. Para ello se utilizó la función *Softmax* en la salida del modelo (logits) para que la suma de las probabildades de que un vóxel pertenezca al fondo y a la segmentación sume 1. El resultado y el etiquetado perfecto se pasaron al formato *one hot* y se compararon con la función de pérdida DICE. Esto dio muy buenos resultados, consiguiendo por

primera vez un $IoU > 0{,}7$ para la segmentación. Por último se probó la entropía cruzada con pesos, dando como peso a la clase de segmentación 1 y se probó darle a la clase borde 0, $0{,}1$ y $0{,}2$, sin obtener buenos resultados en ningún caso. En el artículo estudiado en los antecedentes (Falk y cols., 2019) se usó entropía cruzada con pesos dando dando pesos distintos a cada vóxel. No se llegó a probar esta implementación, que podría haber sido buena para la segmentación con espaciado entre células.

Precisión Mixta

Al trabajar con datos numéricos de poca precisión, como puede ser float de 16 bits, se puede producir un desbordamiento. PyTorch no va a poder reconocer que ha habido desbordamiento, lo que provocará que en las operaciones como en la actualización de pesos intervengan valores incorrectos. Para solucionar esto existe *Apex*, una librería desarrollada por NVidia que gestiona estas operaciones al modificar directamente PyTorch. Hay varios niveles de optimización disponibles, se ha probado a utilizarlo con una optimización de nivel 1 y de nivel 2.

- En optimización de nivel 1 se cambia la entrada de algunas funciones para que usen FP16 (float de 16 bits) y se dejan otras que puedan beneficiarse de la precisión en FP32 (float de 32 bits).
- En optimización de nivel 2 se cambian los pesos del modelo a FP16, se cambian los métodos del modelo para que acepten FP16 y se mantienen unos pesos maestros en FP32 que son usados por el optimizador. En este caso no se cambia las entradas de las funciones como en el nivel 1.

En ambos casos se usa escalado dinámico de pérdida, que se usa como coeficiente para la pérdida. Comienza con un valor muy alto y, si hay desbordamiento, se divide en 2, si no hay desbordamiento durante un número de epochs (1000 por defecto), se multiplica por 2.

Se han probado ambos niveles de optimización y en ambos casos no se ha visto una reducción en la cantidad de memoria ni en el tiempo de entrenamiento. Sí se ha visto un beneficio de usar el escalado dinámico de pérdida, ya que se ha alcanzado más rápidamente una pérdida pequeña. Aún así, a lo largo de 100 o 200 epochs, esto no ha provocado que se alcance mejor error de validación. En los resultados se muestra esa diferencia.

Ficheros generados

En cada iteración se ha guardado en un fichero información sobre el estado actual del entrenamiento y del modelo, con esto se puede utilizar para inferencia, para continuar el entrenamiento o para comprobar métricas, la información guardada es:

- **epochs:** Nº de epochs entrenados.
- **best_model_state_dict:** Pesos del modelo con menor error de validación.
- **model_state_dict:** Pesos del modelo en la última iteración.
- **optimizer_state_dict:** Pesos del optimizador en la última iteración. Al usar el optimizador Adam se guarda el *learning rate* de cada parámetro de forma individual y este se va modificando a lo largo del aprendizaje, por lo que es beneficioso usar el último estado de estos pesos si se desea seguir con el entrenamiento.
- **train_losses:** Lista con todos los valores de pérdida en el entrenamiento.
- **valid_losses:** Lista con todos los valores de pérdida en la validación. train_losses y valid_losses son usados para dibujar las curvas de aprendizaje.

- **test_conf_matrix:** Matriz de confusión por clases obtenida al evaluar el conjunto de test.
- **test_miou:** Media del IoU para el conjunto de test.
- **amp_state_dict:** Estado del optimizador usado con precisión mixta.

5.5– Postprocesado

Se ha probado una técnica con postprocesado basada en la usada en el programa PlantSeg (Wolny y cols., 2020) donde se aplica el algoritmo Watershed. Para utilizar esta técnica se ha usado la librería scikit-image (skimage) y scipy, usando las siguientes funciones en este orden:

1. *scipy.ndimage.distance_transform_edt()* para calcular la transformada de la distancia a la predicción del modelo.
2. *skimage.filters.gaussian()* para aplicar un filtro gaussiano con $\sigma = 2$ al resultado anterior.
3. *skimage.feature.peak_local_max()* al $1 - G$ siendo G el resultado anterior. Esto hace que encuentre los mínimos locales.
4. *scipy.ndimage.label()* al resultado anterior, para generar marcadores al etiquetar cada mínimo local con una etiqueta distinta.
5. *skimage.segmentation.watershed()* utilizando los marcadores generados y la transformada de la distancia, esto termina la segmentación.

Se han hecho varias pruebas siguiendo este preprocesado al predecir los bordes frente a la predicción de bordes con espaciado, los resultados pueden verse en el siguiente capítulo.

CAPÍTULO 6

Resultados

En este capítulo se mostrarán los resultados obtenidos. Se hará uso de:

1. Las pérdidas de entrenamiento y validación en cada epochs.
2. La métrica IoU y el porcentaje de verdaderos positivos.
3. Representación visual de las células antes y después del procesado.

MiniUnet3D vs Unet3D

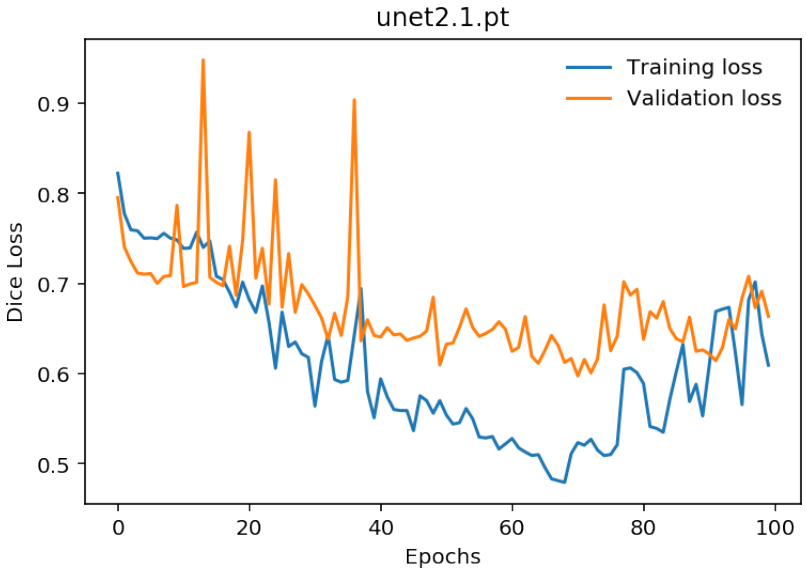

Figura 6.1: Modelo miniunet2.1. 100 epochs. Se usa MiniUnet3D. Imagen objetivo con espaciado entre células. Sin data augmentation.

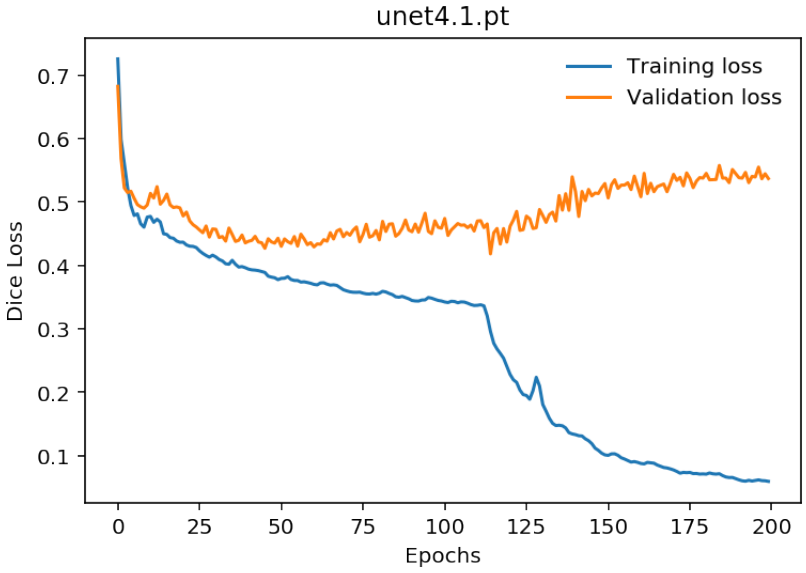

Figura 6.2: Modelo unet4.1. 200 epochs. Imagen objetivo con espaciado entre células. Sin data augmentation.

Sin data augmentation vs Con data augmentation

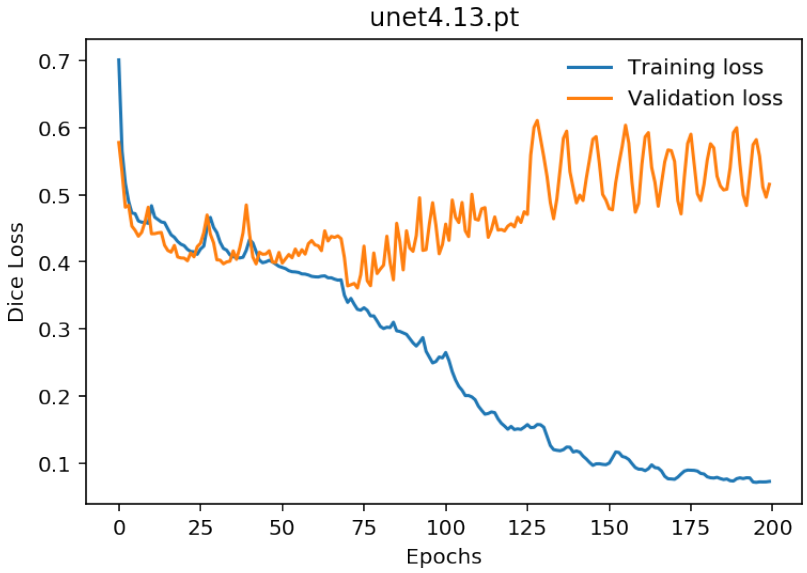

Figura 6.3: Modelo unet4.13. 200 epochs. Imagen objetivo con espaciado entre células. Sin data augmentation.

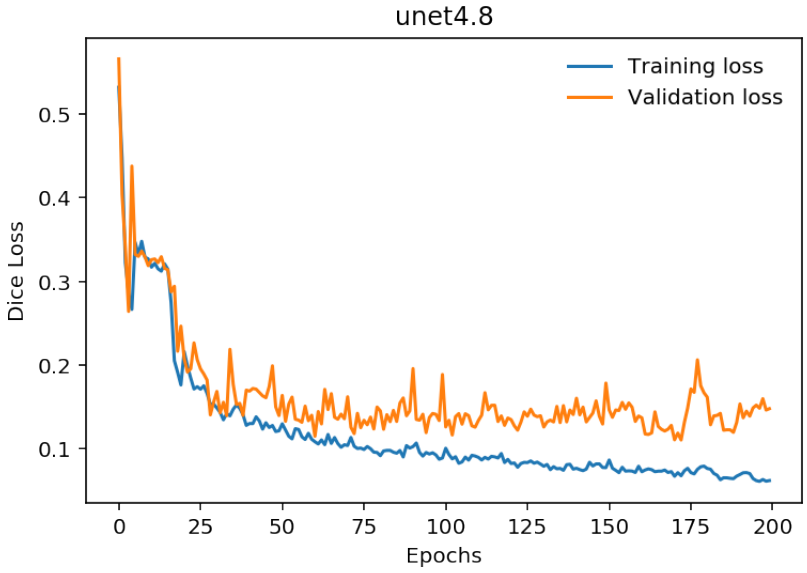

Figura 6.4: Modelo unet4.8. 200 epochs. Imagen objetivo con espaciado entre células. Con data augmentation.

Sin Apex vs Con Apex

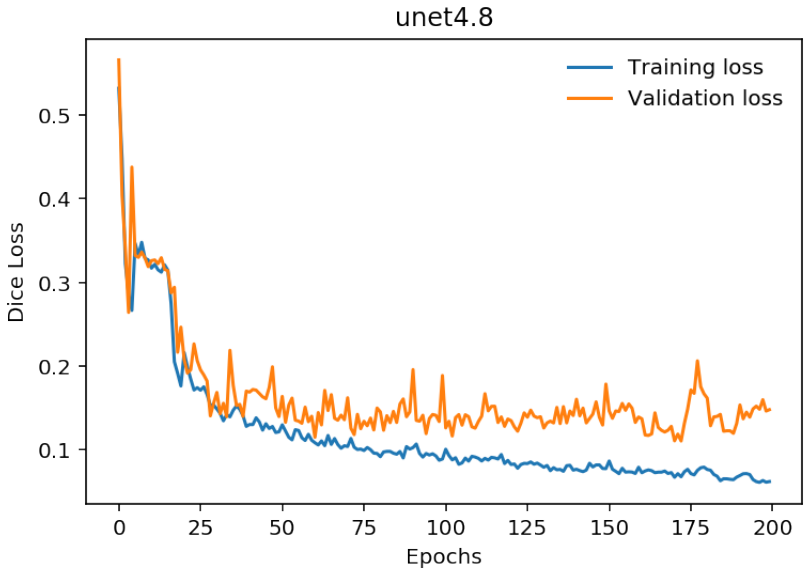

Figura 6.5: Modelo unet4.8. 200 epochs. Imagen objetivo con espaciado entre células. Con data augmentation. Sin Apex

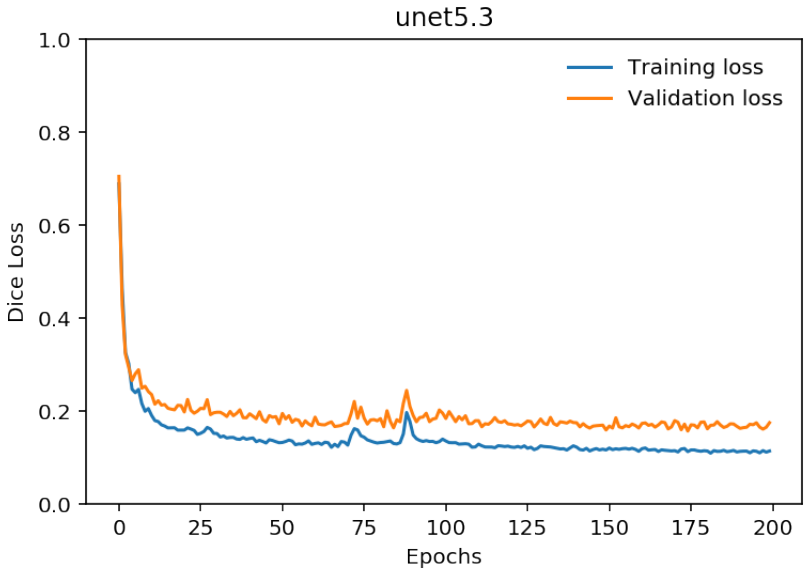

Figura 6.6: Modelo unet5.2. 200 epochs. Imagen objetivo con espaciado entre células. Con data augmentation. Con Apex.

Prueba de espaciado entre células.

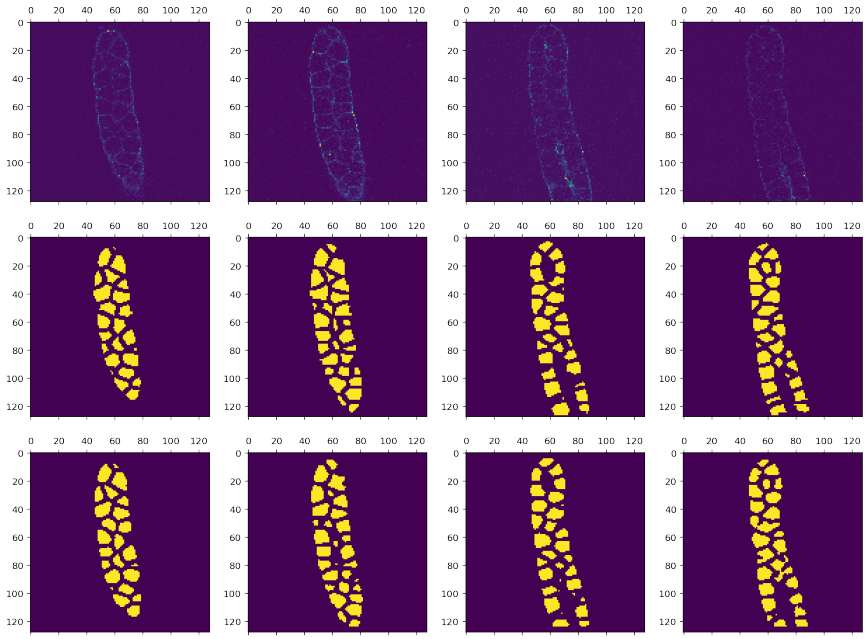

Figura 6.7: Modelo unet6t. 100 epochs. Imagen objetivo con espaciado entre células. Con data augmentation. Pérdida Dice. Sin postprocesado. Primera fila imagen de entrada. Segunda fila segmentación objetivo. Tercera fila predicción. Z=20,25,45,50 en las columnas. IoU 0.72

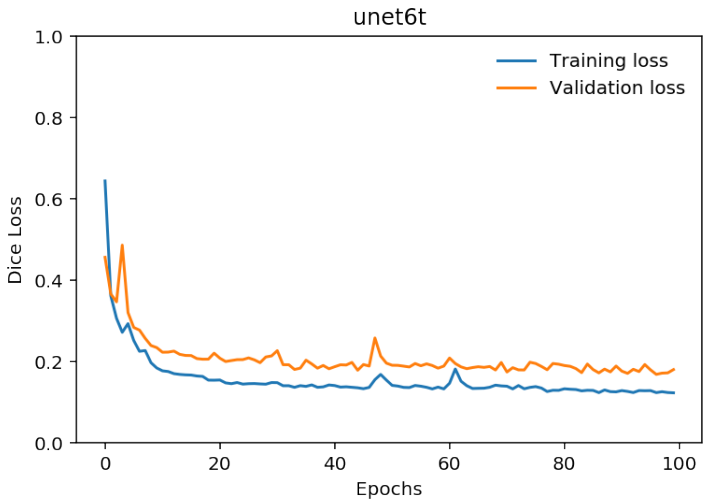

Figura 6.8: Modelo unet6t. 100 epochs. Imagen objetivo con espaciado entre células. Con data augmentation. Sin postprocesado.

Prueba de bordes.

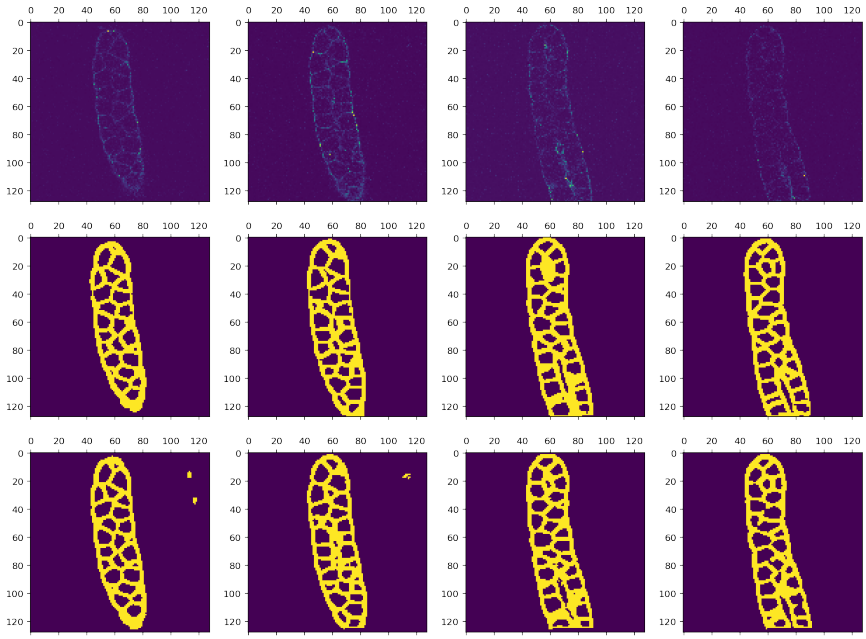

Figura 6.9: Modelo unet6b. 100 epochs. Imagen objetivo bordes de células. Con data augmentation. Pérdida Dice. Sin postprocesado. Primera fila imagen de entrada. Segunda fila segmentación objetivo. Tercera fila predicción. Z=20,25,45,50 en las columnas. IoU 0.84

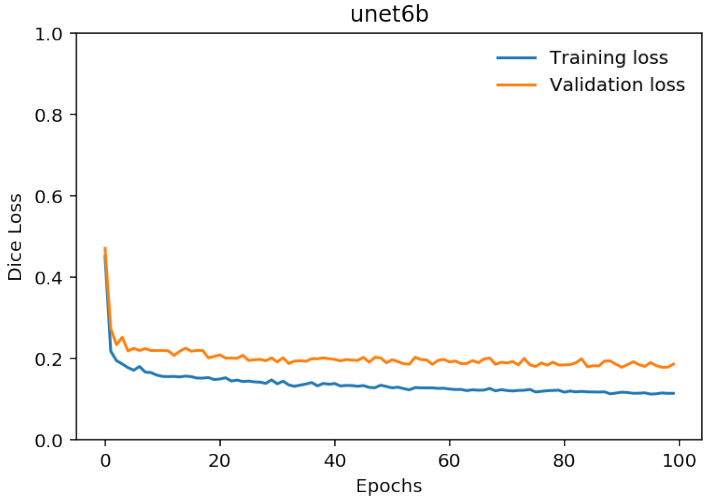

Figura 6.10: Modelo unet6b. 100 epochs. Imagen objetivo bordes de células. Con data augmentation. Pérdida Dice. Sin postprocesado.

DT Watershed a bordes.

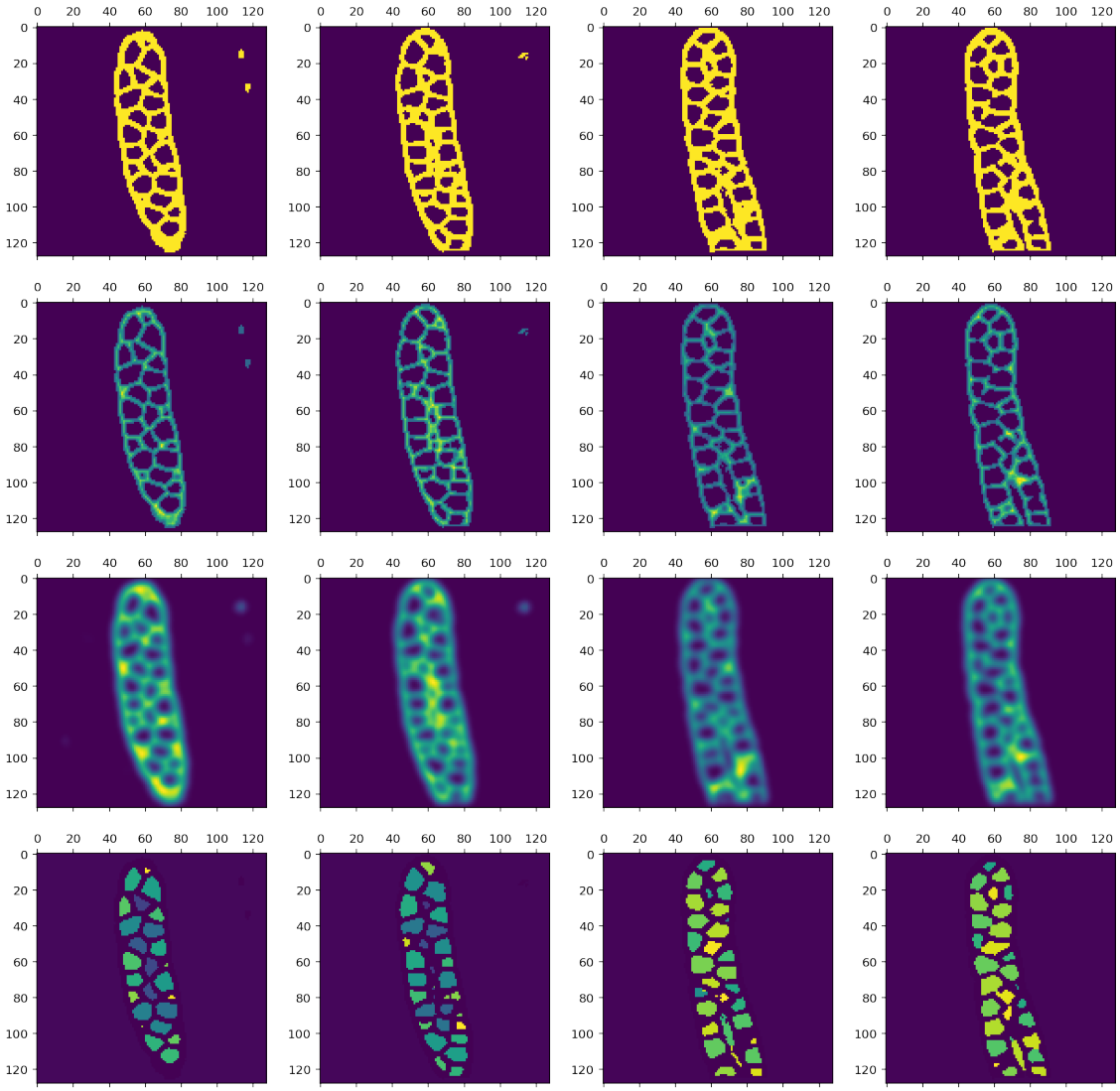

Figura 6.11: DT Watershed aplicado a la salida del modelo unet6b. Primera fila bordes. Segunda fila transformación de la distancia. Filtro gaussiano. Cuarta fila watershed con mínimos locales como semilla. Z=20,25,45,50 en las columnas.

Comparación DT Watershed con espaciado entre células.

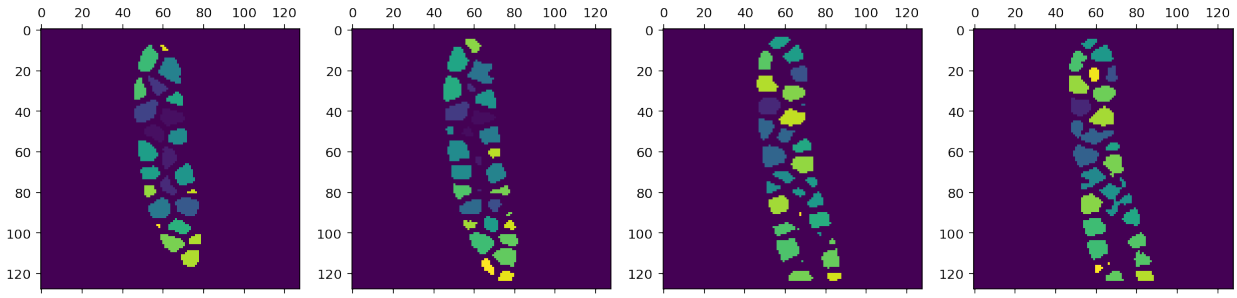

Figura 6.12: Modelo unet6b. 100 epochs. Imagen objetivo bordes de células. Con data augmentation. Pérdida Dice. Componentes conexas con etiquetas distintas. IoU 0.72

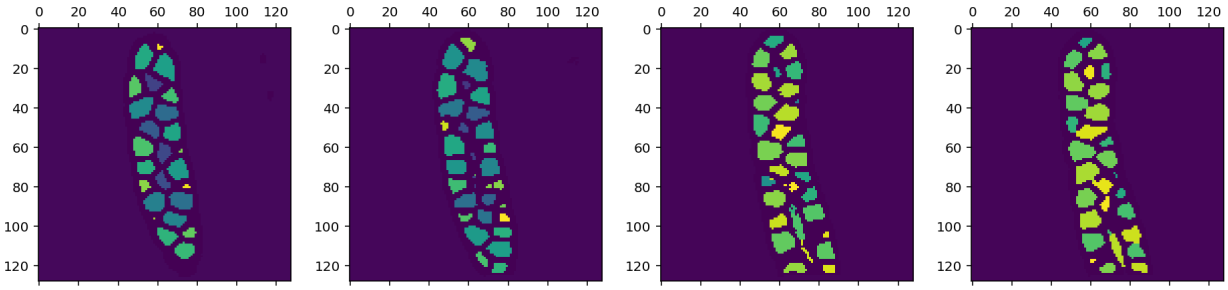

Figura 6.13: DT Watershed aplicado a la salida del modelo unet6b. IoU 0.70

CAPÍTULO 7

Análisis temporal

Tarea	Subtarea	Tiempo
Configurar servicios Cloud		16h 55m
Diseñar primera CNN para segm. semántica.	Primer modelo U-Net	40m
	Preparar Función de Pérdida y Optimizador	1h 21m
	Simplificar red o inputs para disminuir memoria	9h 40m
	Resultados de primer entrenamiento de red	4h 22m
	Elegir función de pérdida correcta	2h 50m
Usar métricas implementadas en kornia		2h 55m
Investigar sobre CNN		32h 18m
Mejoras en la CNN	General	11h 34m
	Añadir métricas de validación y test	3h
	Implementar Apex	2h 5m
	Watershed	2h 38m
Mejoras en los datos	Prepar datos para batch	2h 37m
	Data Augmentation	2h 16m
Memoria		71h 15m
Preparar Código		2h 20m
Preparar Dataset en PyTorch	Crear nuevo Dataset (clase en PyTorch)	29m
	Estudiar imágenes de entrada	19h 10m
	Añadir espacio entre células	10h 56m
	Otros	1h 47m
Total		201 h 7m

Cuadro 7.1: Tabla de tiempos

CAPÍTULO 8

Conclusiones

Durante el desarrollo de este trabajo han sido dos las principales dificultades que me he encontrado: **conocimiento en la materia** y **capacidad de computación**.

Podría parecer intuitivo pensar que la inteligencia artificial es la que soluciona un problema al aplicar un algoritmo. Sin embargo he experimentado que esto no es así. Es necesario un amplio conocimiento en varios dominios para afrontar un problema real. Empecé el proyecto intentando solucionar el problema de segmentación celular utilizando todas las herramientas disponibles para ello y rápidamente descubrí que los algoritmos de deep learning están a la vanguardia, en concreto la arquitectura U-Net. Intenté recurrir a soluciones de segmentación celular sin éxito, ya que estas soluciones no estaban especializadas en resolver exactamente el mismo problema o requerían de un hardware muy superior al disponible. Quedando como única opción una implementación más a bajo nivel.

Los resultados obtenidos al realizar 100 o 200 iteraciones entrenando el modelo UNet con imágenes de baja calidad son consideradas buenos ($IoU > 0{,}7$). Se puede usar el programa desarrollado sin hacer ningún cambio con imágenes de mayor calidad o iterando un mayor nº de veces, lo que mejorará mucho los resultados. En los artículos seguidos se han hecho 100k y 150k iteraciones sobre imágenes de mayor calidad, obteniendo a veces unos resultados casi perfectos. Con 100k iteraciones en el algoritmo desarrollado probablemente se consigan resultados comparables con los obtenidos en esos artículos.

CAPÍTULO 9

Manual

El proyecto está disponible en el siguiente enlace, donde hay instrucciones sobre su instalación `https://github.com/Orzzet/segmentacion_celular_unet3d`.

Prerequisitos:

- CUDA `https://docs.nvidia.com/cuda/cuda-installation-guide-linux/index.html`
- CUDNN `https://docs.nvidia.com/deeplearning/cudnn/install-guide/index.html`
- Python 3.5+ `https://www.python.org/downloads/`
- PyTorch `https://github.com/pytorch/pytorch`
- Jupyter Notebook `https://jupyter.readthedocs.io/en/latest/install.html` o un Software que pueda ejectura Notebooks

Cómo usarlo:

1. Descargar el proyecto del repositorio y descomprimir en el lugar deseado.
2. El notebook 0_preprocesado trata las imágenes y las transforma en el formato correcto. En caso de tener imágenes en un formato distinto, hay que modificar este fichero.
3. El notebook 1_procesado es para hacer el entrenamiento.
4. El notebook 2_inferencia es para utilizar el modelo, bien para estadísticas o para comprobar el resultado.

Referencias

Cardoso, M. J., Arbel, T., Carneiro, G., Syeda-Mahmood, T., Tavares, J., Moradi, M., ... Lu, Z. (2017). *Deep learning in medical image analysis and multimodal learning for clinical decision support: Third international workshop, dlmia 2017, and 7th international workshop, ml-cds 2017, held in conjunction with miccai 2017, québec city, qc, canada, september 14, proceedings*. doi: 10.1007/978-3-319-67558-9

Collette, A. (2013). *Python and hdf5*. O'Reilly.

Deng, J., Dong, W., Socher, R., Li, L.-J., Li, K., y Fei-Fei, L. (2009). ImageNet: A Large-Scale Hierarchical Image Database. En *Cvpr09*.

Falk, T., Mai, D., Bensch, R., Çiçek, Ö., Abdulkadir, A., Marrakchi, Y., ... Ronneberger, O. (2019, 01 de enero). U-net: deep learning for cell counting, detection, and morphometry. *Nature Methods*, *16*(1), 67-70. Descargado de https://doi.org/10.1038/s41592-018-0261-2 (https://github.com/lmb-freiburg/Unet-Segmentation) doi: 10.1038/s41592-018-0261-2

Goodfellow, I., Bengio, Y., y Courville, A. (2016). *Deep learning*. MIT Press. Descargado de https://www.deeplearningbook.org/ (http://www.deeplearningbook.org)

He, K., Zhang, X., Ren, S., y Sun, J. (2015). *Deep residual learning for image recognition.*

Hinton, G., Osindero, S., y Teh, Y.-W. (2006, 08). A fast learning algorithm for deep belief nets. *Neural computation*, *18*, 1527-54. doi: 10.1162/neco.2006.18.7.1527

Hinton, G. E. (1986). Learning distributed representations of concepts. En *Proceedings of the eighth annual conference of the cognitive science society.*

Hochreiter, S. (1998, 04). The vanishing gradient problem during learning recurrent neural nets and problem solutions. *International Journal of Uncertainty, Fuzziness and Knowledge-Based Systems*, *6*, 107-116. doi: 10.1142/S0218488598000094

Jadon, S. (2020). *A survey of loss functions for semantic segmentation.*

Jia, Y., Shelhamer, E., Donahue, J., Karayev, S., Long, J., Girshick, R., ... Darrell, T. (2014). Caffe: Convolutional architecture for fast feature embedding. *arXiv preprint arXiv:1408.5093*.

Krizhevsky, A., Sutskever, I., y Hinton, G. (2012, 01). Imagenet classification with deep convolutional neural networks. *Neural Information Processing Systems*, *25*. doi: 10.1145/3065386

Li, F.-F., Krishna, R., y Xu, D. (2020). *CS231n convolutional neural networks for visual recognition.* Descargado 2020-09-03, de https://cs231n.github.io/convolutional-networks/

Long, J., Shelhamer, E., y Darrell, T. (2014). *Fully convolutional networks for semantic segmentation.*

McCulloch, W. S., y Pitts, W. (1943, 01 de Dec). A logical calculus of the ideas immanent in nervous activity. *The bulletin of mathematical biophysics*, *5*(4), 115-133. Descargado de https://doi.org/10.1007/BF02478259 doi: 10.1007/BF02478259

Minsky, M., y Papert, S. (1969). *Perceptrons: An introduction to computational geometry*. Cambridge, MA, USA: MIT Press.

missinglink. (2020). *missinglink.ai.* Descargado 2020-09-03, de https://missinglink.ai/guides/convolutional-neural-networks/

convolutional-neural-network-tutorial-basic-advanced/

Nair, V., y Hinton, G. E. (2010). Rectified linear units improve restricted boltzmann machines. En *Proceedings of the 27th international conference on international conference on machine learning* (p. 807–814). Madison, WI, USA: Omnipress.

Noh, H., Hong, S., y Han, B. (2015). *Learning deconvolution network for semantic segmentation.*

Nwankpa, C., Ijomah, W., Gachagan, A., y Marshall, S. (2018). *Activation functions: Comparison of trends in practice and research for deep learning.*

Paszke, A., Gross, S., Massa, F., Lerer, A., Bradbury, J., Chanan, G., ... Chintala, S. (2019). Pytorch: An imperative style, high-performance deep learning library. En *Advances in neural information processing systems 32* (pp. 8024–8035). Curran Associates, Inc. Descargado de http://papers.neurips.cc/paper/9015-pytorch-an-imperative-style-high-performance-deep-learning-library.pdf

Pedregosa, F., Varoquaux, G., Gramfort, A., Michel, V., Thirion, B., Grisel, O., ... Duchesnay, E. (2011). Scikit-learn: Machine learning in Python. *Journal of Machine Learning Research*, *12*, 2825–2830.

Pérez-García, F., Sparks, R., y Ourselin, S. (2020, marzo). TorchIO: a Python library for efficient loading, preprocessing, augmentation and patch-based sampling of medical images in deep learning. *arXiv:2003.04696 [cs, eess, stat]*. Descargado 2020-03-11, de http://arxiv.org/abs/2003.04696 (arXiv: 2003.04696)

Riba, E., Mishkin, D., Ponsa, D., Rublee, E., y Bradski, G. (2020). Kornia: an open source differentiable computer vision library for pytorch. En *Winter conference on applications of computer vision.* Descargado de https://arxiv.org/pdf/1910.02190.pdf

Ronneberger, O., Fischer, P., y Brox, T. (2015). *U-net: Convolutional networks for biomedical image segmentation.*

Rosenblatt, F. (1958). The perceptron: A probabilistic model for information storage and organization in the brain. *Psychological Review*, 65–386.

Rumelhart, D. E., Hinton, G. E., y Williams, R. J. (1986, 01 de Oct). Learning representations by back-propagating errors. *Nature*, *323*(6088), 533-536. Descargado de https://doi.org/10.1038/323533a0 doi: 10.1038/323533a0

Schindelin, J., Arganda-Carreras, I., Frise, E., Kaynig, V., Longair, M., Pietzsch, T., ... Cardona, A. (2012, 01 de Jul). Fiji: an open-source platform for biological-image analysis. *Nature Methods*, *9*(7), 676-682. Descargado de https://doi.org/10.1038/nmeth.2019 doi: 10.1038/nmeth.2019

shiba24. (2017). *3d unet implementation.* https://github.com/shiba24/3d-unet. GitHub.

Simonyan, K., y Zisserman, A. (2014). *Very deep convolutional networks for large-scale image recognition.*

Szegedy, C., Liu, W., Jia, Y., Sermanet, P., Reed, S., Anguelov, D., ... Rabinovich, A. (2014). *Going deeper with convolutions.*

The HDF Group. (2000-2010). *Hierarchical data format version 5.* Descargado de http://www.hdfgroup.org/HDF5

Tiobe. (2020). *Tiobe index.* Descargado de https://www.tiobe.com/tiobe-index/

Van Der Walt, S., Colbert, S. C., y Varoquaux, G. (2011). The numpy array: a structure for efficient numerical computation. *Computing in Science and Engineering*, *13*(2), 22.

van der Walt, S., Schönberger, J. L., Nunez-Iglesias, J., Boulogne, F., Warner, J. D., Yager, N., ... the scikit-image contributors (2014, 6). scikit-image: image processing in Python. *PeerJ*, *2*, e453. Descargado de https://doi.org/10.7717/peerj.453 doi: 10.7717/peerj.453

Virtanen, P., Gommers, R., Oliphant, T. E., Haberland, M., Reddy, T., Cournapeau, D., ... Contributors, S. . . (2020). SciPy 1.0: Fundamental Algorithms for Scientific Computing in Python. *Nature Methods*, *17*, 261–272. doi: https://doi.org/10.1038/s41592-019-0686-2

Widrow, B., y Hoff, T. (2015). Adaptive linear neuron..

Wolny, A., Cerrone, L., Vijayan, A., Tofanelli, R., Barro, A. V., Louveaux, M., ... Kreshuk, A. (2020). Accurate and versatile 3d segmentation of plant tissues at cellular resolution. *bioRxiv*. Descargado de `https://www.biorxiv.org/content/early/2020/01/18/2020.01.17.910562` doi: 10.1101/2020.01.17.910562

Zeiler, M. D., Taylor, G. W., y Fergus, R. (2011). Adaptive deconvolutional networks for mid and high level feature learning. En *2011 international conference on computer vision* (p. 2018-2025).

Özgün Çiçek, Abdulkadir, A., Lienkamp, S. S., Brox, T., y Ronneberger, O. (2016). *3d u-net: Learning dense volumetric segmentation from sparse annotation.*