ESCUELA TÉCNICA SUPERIOR DE INGENIERÍA INFORMÁTICA

Proyecto fin de Carrera Grado

GITI

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

Sevilla, 12 de Agosto de 2020 (v.0.1.0)

Resumen

En la actualidad existen microscopios que emplean técnicas ópticas para reconstruir imágenes 3D con una alta precisión. Gracias a esto los investigadores tienen a su alcance imágenes microscópicas de alta calidad y pueden sacar conclusiones de ellas. Estas imágenes suelen requerir un procesado digital con el objetivo de discernir los elementos importantes del resto.

En este proyecto se abordará el problema de la segmentación de células en imágenes 3D. Para ello se usarán imágenes cedidas por el Departamento de Biología Celular de la Facultad de Biología de la Universidad de Sevilla. En cada imagen hay decenas de células, todas en contacto con otras células sin espacio entre ellas.

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.

Respecto a las técnicas usadas, este proyecto se centrará en el uso de redes neuronales para la segmentación de células. Se validará la efectividad del uso de redes neuronales y se estudiará qué tipo de red neuronal y arquitectura será mejor para esta tarea, teniendo en cuenta la exactitud de la segmentación y el coste computacional.

En el análisis de antecedenes se comprobará que la CNN (Convolutional Neural Network o Red Neuronal Convolucional) será con la que se obtienen mejores resultados en el reconocimiento de patrones en imágenes. También se verá que al diseñar una CNN con la arquitectura U-Net se obtienen buenos resultados.

Se probarán varios modelos de CNNs para la segmentación de células, esperándose el mejor resultado de la arquitectura U-Net.

Adicionalmente, se estudiará el uso de un preprocesado y postprocesado para aumentar la eficacia de la segmentación, así como un posterior mapeado a color de las células para ayudar a su visualización.

El código fuente de esta herramienta estará disponible, así como Jupyter Notebooks y un ficher python para efectuar entrenamiento e inferencias por línea de comandos.

Índice general

Índice de cuadros

Índice de figuras

CAPÍTULO 1

Definición de objetivos

Este proyecto tiene como objetivo segmentar correctamente las células mostradas en imágenes 3D mediante la aplicación de técnicas de deep learning. Posteriormente se colorearán las células de distintos colores para facilitar su visualización.

Este proceso podrá ser reproducido por un usuario ejecutando los archivos .ipynb con Jupyter Notebook o por línea de comandos. El usuario podrá entrenar el modelo con un nuevo conjunto de imágenes o inferir la segmentación de una o más imágenes.

Se intentará que esta herramienta sea accesible al mayor nº de usuarios posibles, para ello se optimizará todo lo posible el proceso de entrenamiento e inferencia y se estudiarán varios modelos con distinto grado de complejidad. Con esto se reducirá el hardware necesario.

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.

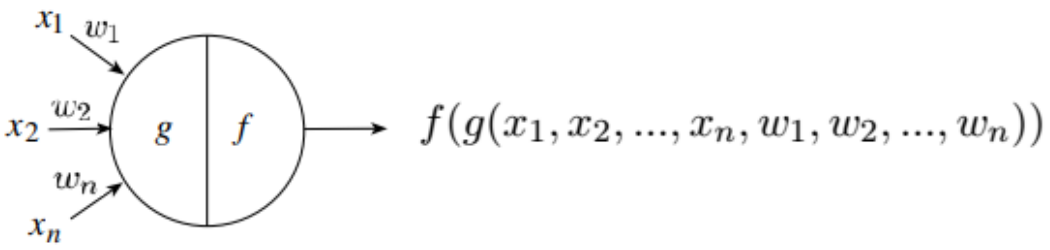

Figura 2.1: Neurona artificial genérica

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.

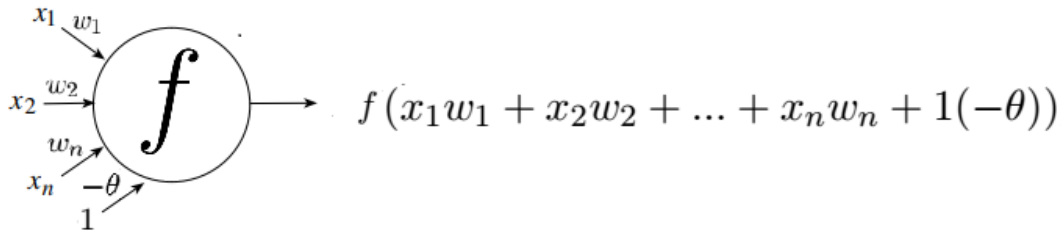

Figura 2.2: Representación de una Neurona Artificial con $\sum x_i w_i$ como función de integración

- 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.

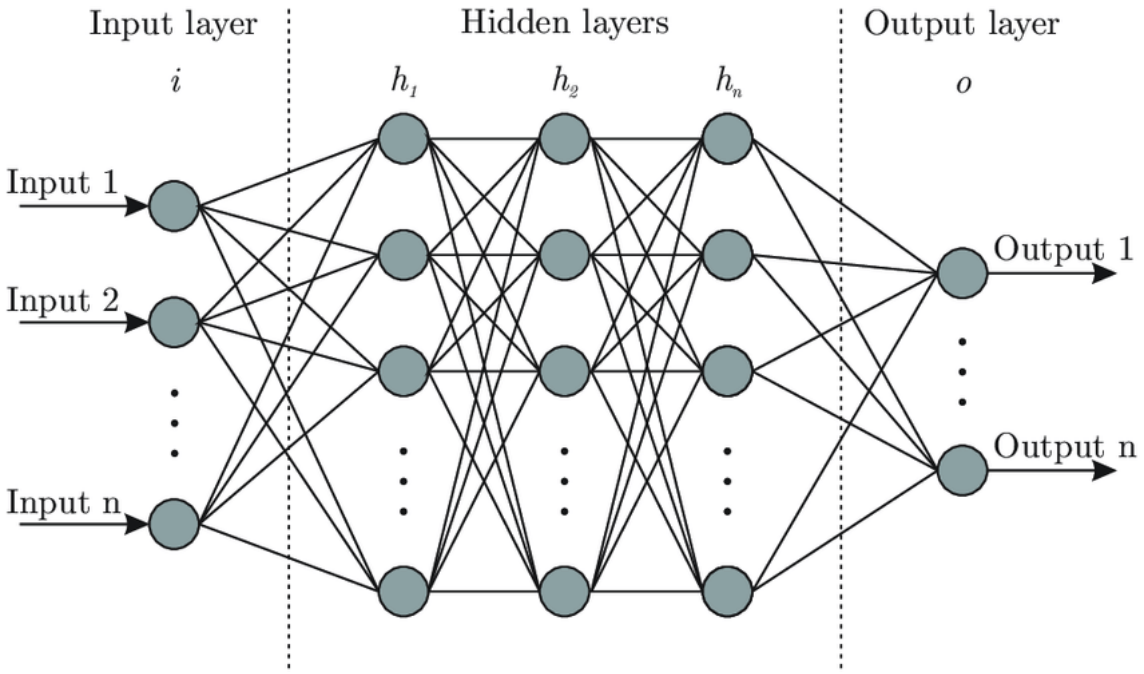

Figura 2.3: Red Neuronal Artificial completamente conectada.

- 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.

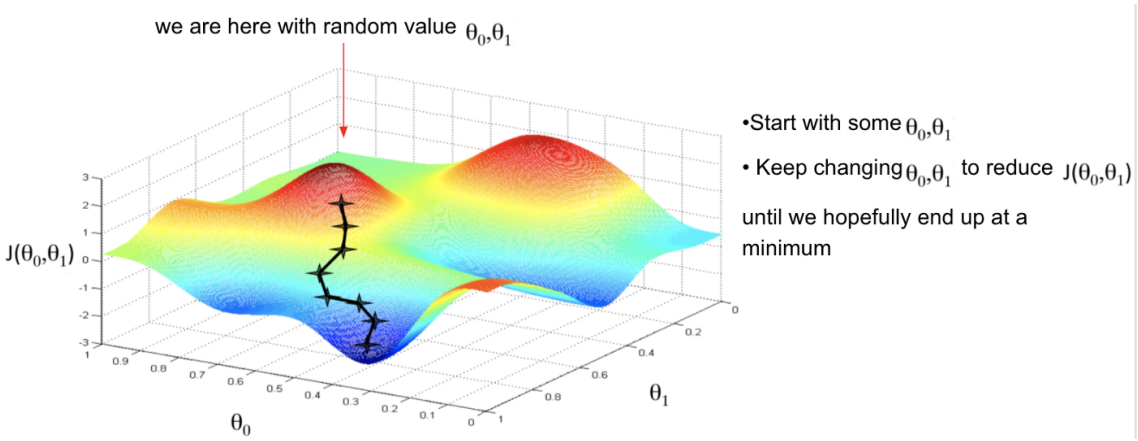

Figura 2.4: Ilustración del algoritmo de descenso por gradiente. θ_0 y θ_1 son los pesos de la ANN y J la función de pérdida. Se puede observar cómo se produce un ”descenso”hacia un mínimo de la función.

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.

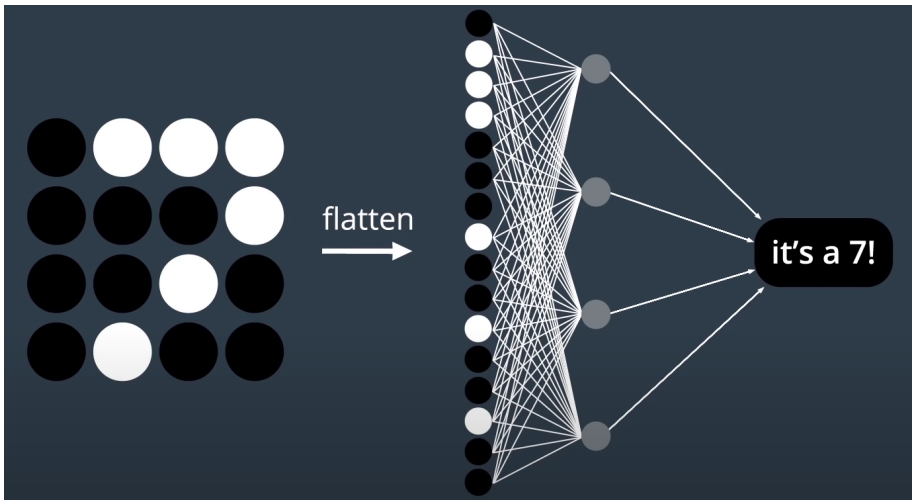

Figura 2.5: Imagen de 4x4 px aplanada y usada como entrada en una FCNN con 4 neuronas en su única capa oculta. No se muestra su capa de salida. Imagen del curso Intro to Deep Learning with PyTorch de Udacity.

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.

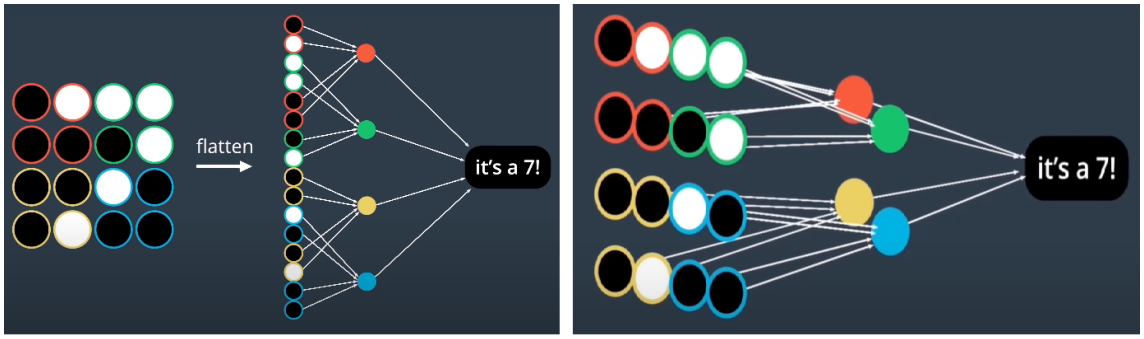

Figura 2.6: Imagen de 4x4 px usada como entrada en una CNN. Ambas figuras muestran la misma arquitectura, estando en la figura de la izquierda la imagen aplanada y en la figura de la derecha se muestra la matriz 4x4 como entrada. Imágenes del curso Intro to Deep Learning with PyTorch de Udacity.

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 (?, ?) 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.

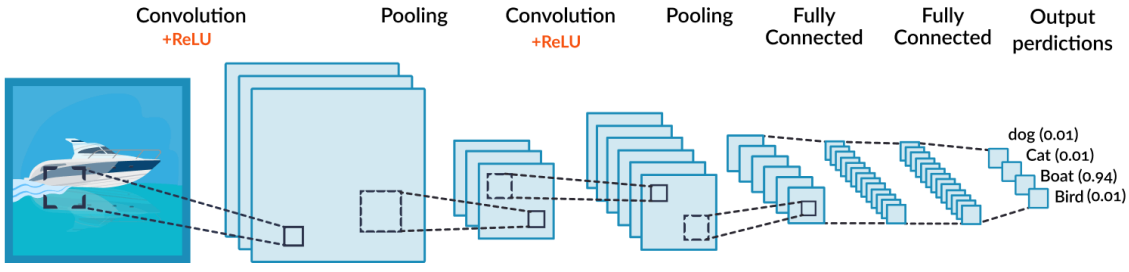

Figura 2.7: Red Neuronal Convolucional simple en la que la imagen de un barco es clasificada.

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(?, ?) se muestra un ejemplo de una capa de convolución de profundidad 2 con imagen de entrada 7×7 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:

- $W_2 = \frac{W_1 - F}{S} + 1$
- $H_2 = \frac{H_1 - F}{S} + 1$
- $D_2 = D_1$

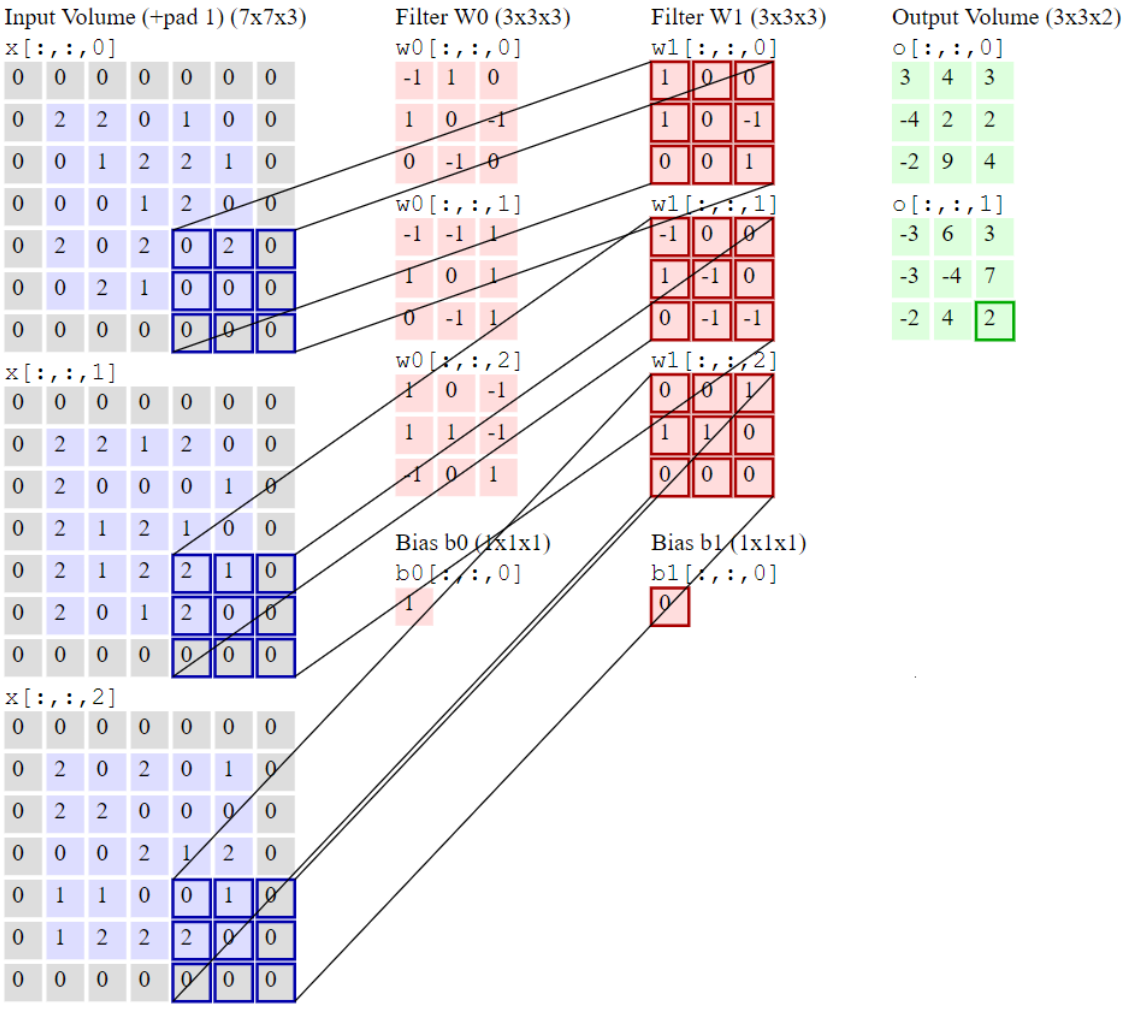

Figura 2.8: Imagen de 4x4 px usada como entrada en una CNN. Ambas figuras muestran la misma arquitectura, estando en la figura de la izquierda la imagen aplanada y en la figura de la derecha se muestra la matriz 4x4 como entrada. Imagen tomada del curso CS231n de Standford University.

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.

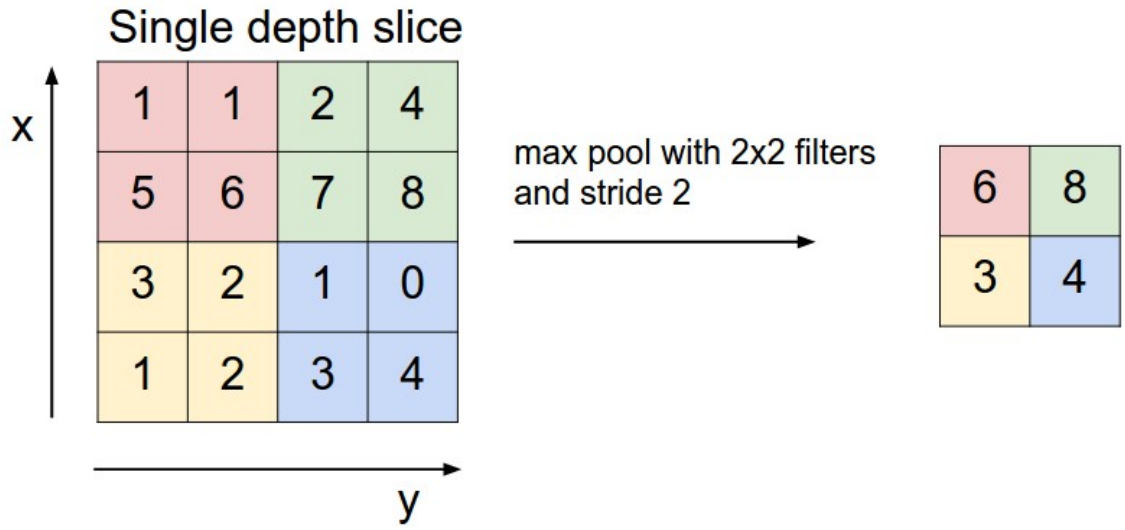

Figura 2.9: Imagen 4×4 a la que se ha aplicado un pooling de $F = 2, S = 2$. Cada color de la imagen de la izquierda indica las entradas del para la operación MAX, cada color de la imagen de la derecha indica la salida de dicha operación. Figura tomada del curso CS231n de Standford University.

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 $\beta < 1$ (?, ?).

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 (?, ?). 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 usadas en segmentación

2.4– Software desarrollado

2.5– Aportación Realizada

CAPÍTULO 3

Análisis de requisitos, diseño e implementación

3.1– Datos iniciales

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.

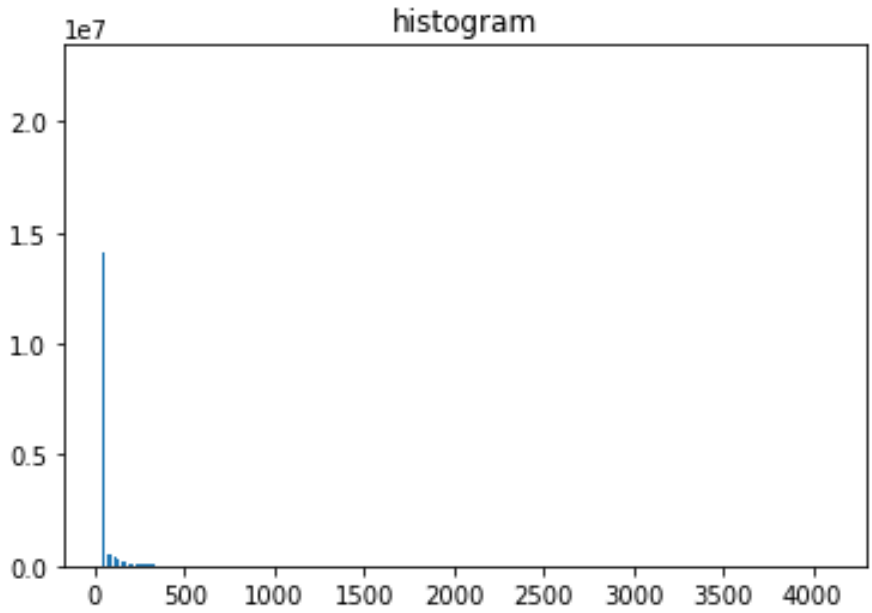

Figura 3.1: Histograma de todos los valores

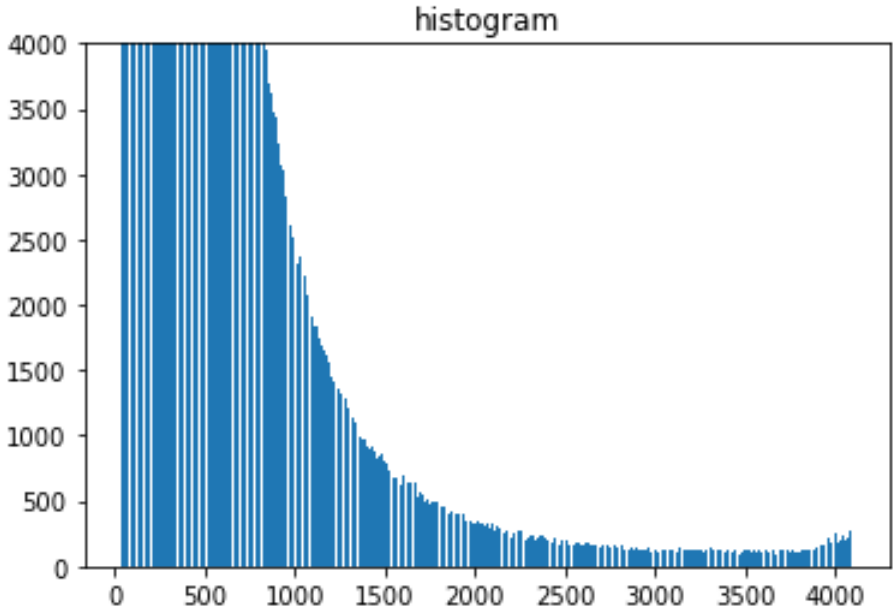

Figura 3.2: Histograma de todos los valores con el eje y limitado a 4000

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.

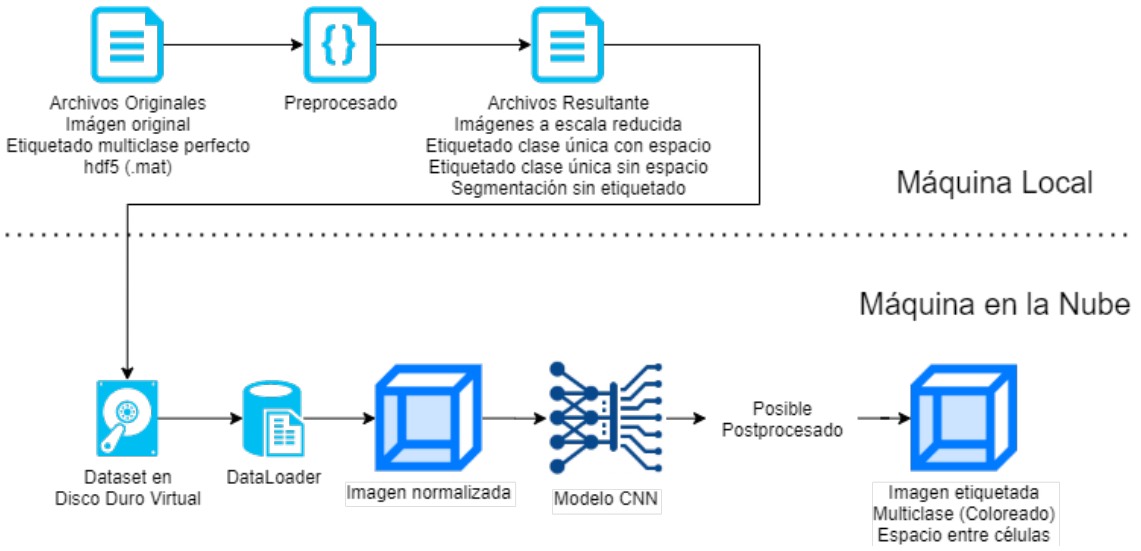

Figura 4.1: Diseño general

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

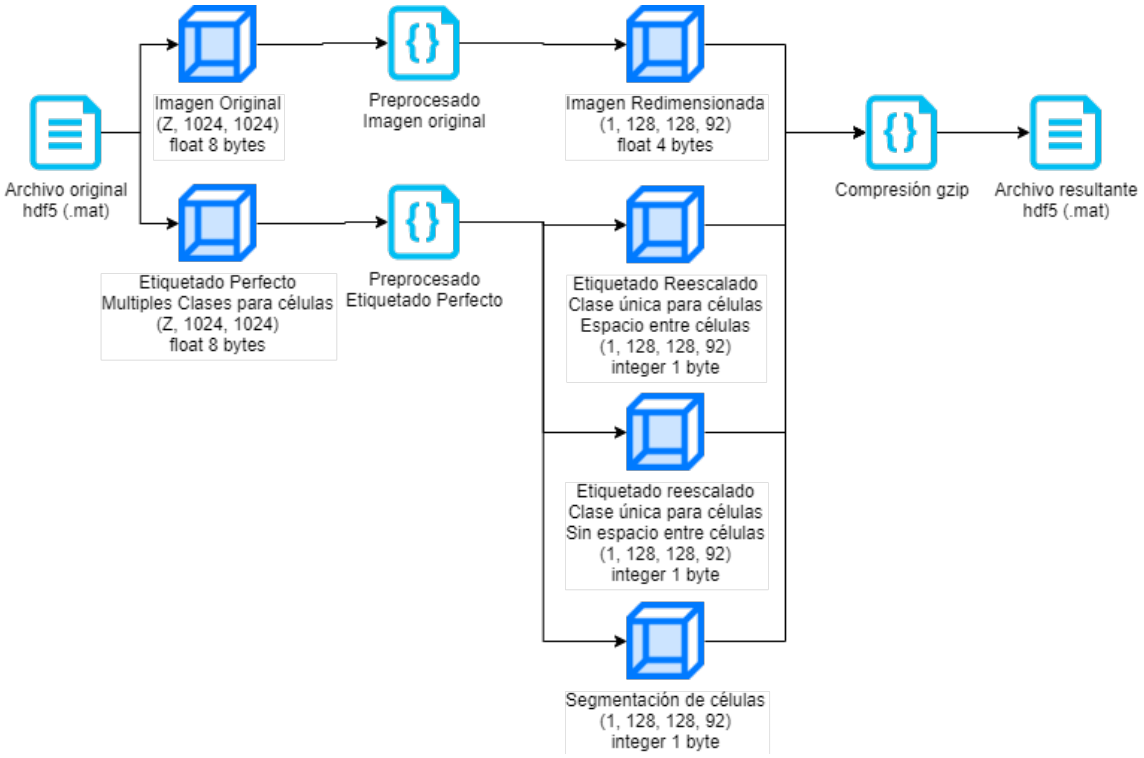

Figura 4.2: Preprocesado llevado a cabo en la máquina local.

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 çélula"), 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. Cambiar las etiquetas a 1 sin añadir espaciado y entrenar un modelo A para su predicción. Encontrar los bordes exteriores de las células y entrenar un modelo B para su predicción. La predicción final será la predicción del modelo A menos la predicción del modelo B.

Los 3 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 que usa valores enteros entre 0 y 100. Es importante tener en cuenta que al almacenar tensores en la memoria de la GPU, la imagen no tendrá ningún tipo de compresión, cada elemento ocupará espacio en memoria. Si hacemos que la imagen de entrada pase de 8 bytes a 4 bytes, estaremos reduciendo su tamaño a la mitad. De forma similar si hacemos que la imagen del etiquetado 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.

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

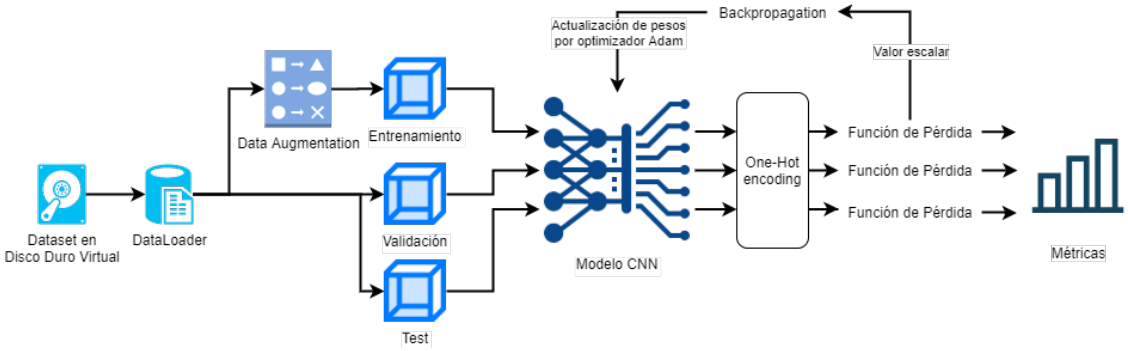

Figura 4.3

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, 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 intentará deformar 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

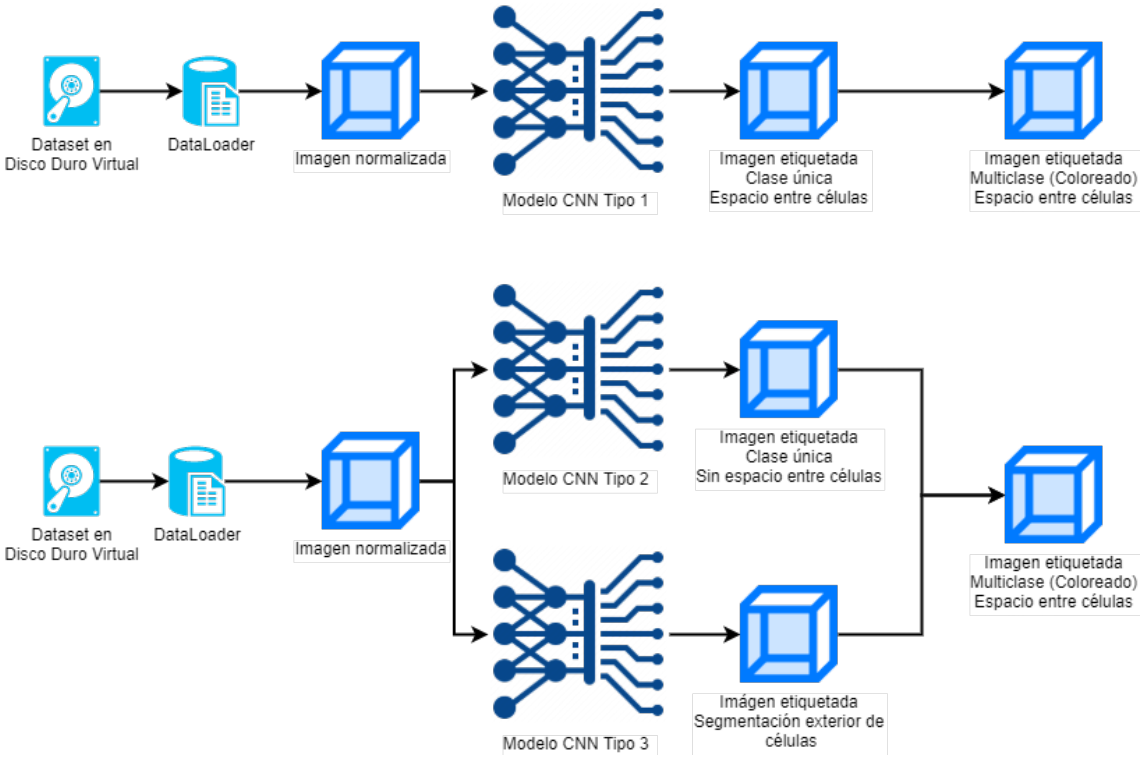

Figura 4.4: Se muestran dos métodos distintos para obtener la imagen correctamente etiquetada.

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: Usar dos modelos y operar con los resultados. El modelo de **Tipo 2** dará como predicción un etiquetado con las células con valor 1, pero sin espacio entre ellas. El modelo de **Tipo 2** dará como predicción un etiquetado con los bordes de las células de valor 1. La predicción final será el resultado de restar la predicción del modelo de Tipo 3 a la predicción del modelo de Tipo 2.

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 citeCicek2016, basada en la arquitectura U-Net propuesta por Ronneberger et al (Ronneberger, Fischer, y Brox, 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.

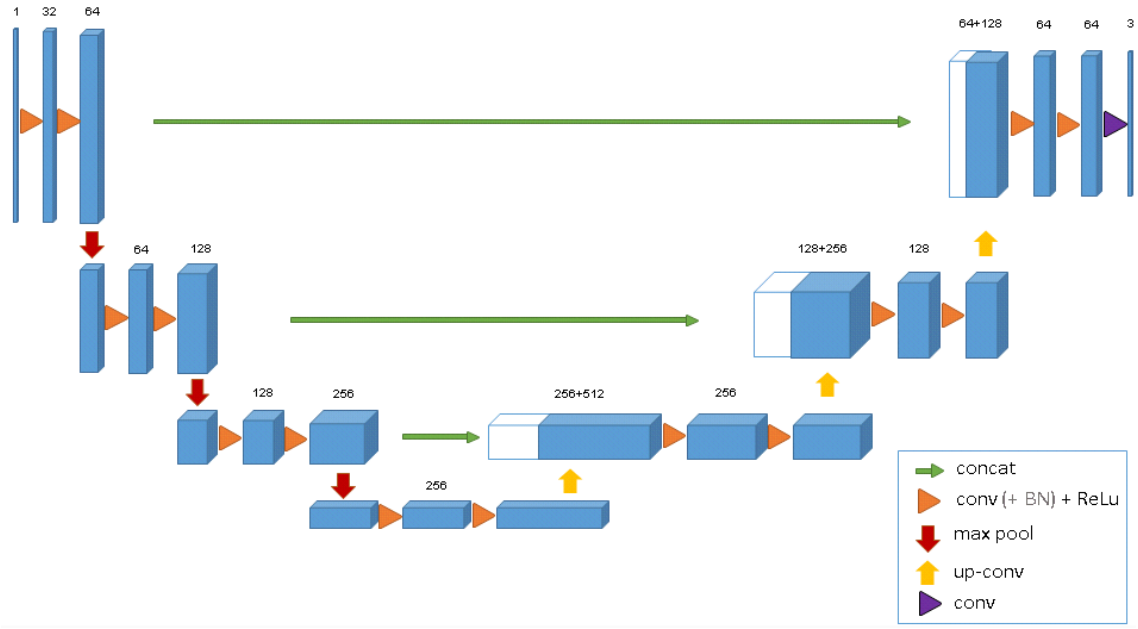

Figura 4.5: Arquitectura U-Net completa.

En la figura 6.11 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.

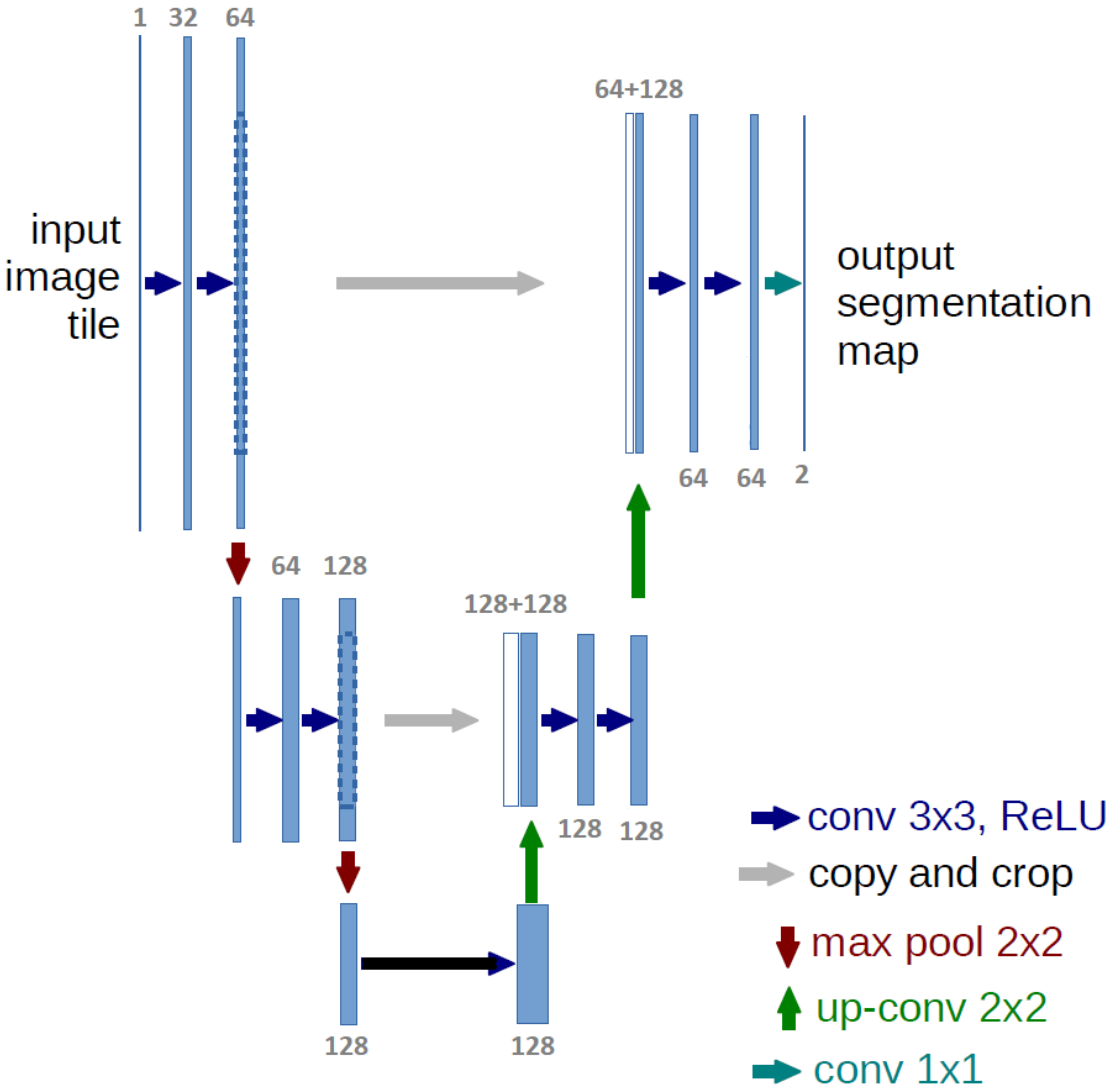

Figura 4.6: Arquitectura U-Net completa.

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 se su velocidad. 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.searchboundaries)* (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.

CAPÍTULO 6

Pruebas

Pongo sólo las imágenes por ahora. Todas las métricas están guardadas.

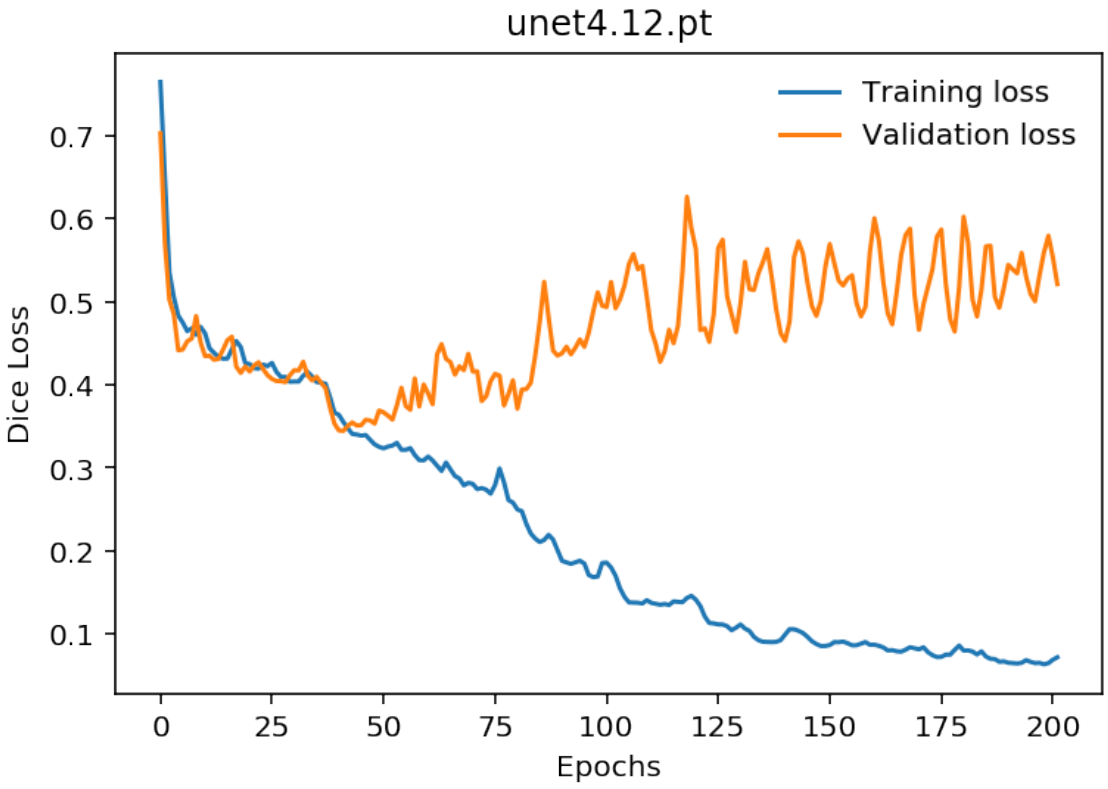

Figura 6.1: Arquitectura U-Net media. Batch=1. Znorm. Hay sobreajuste.

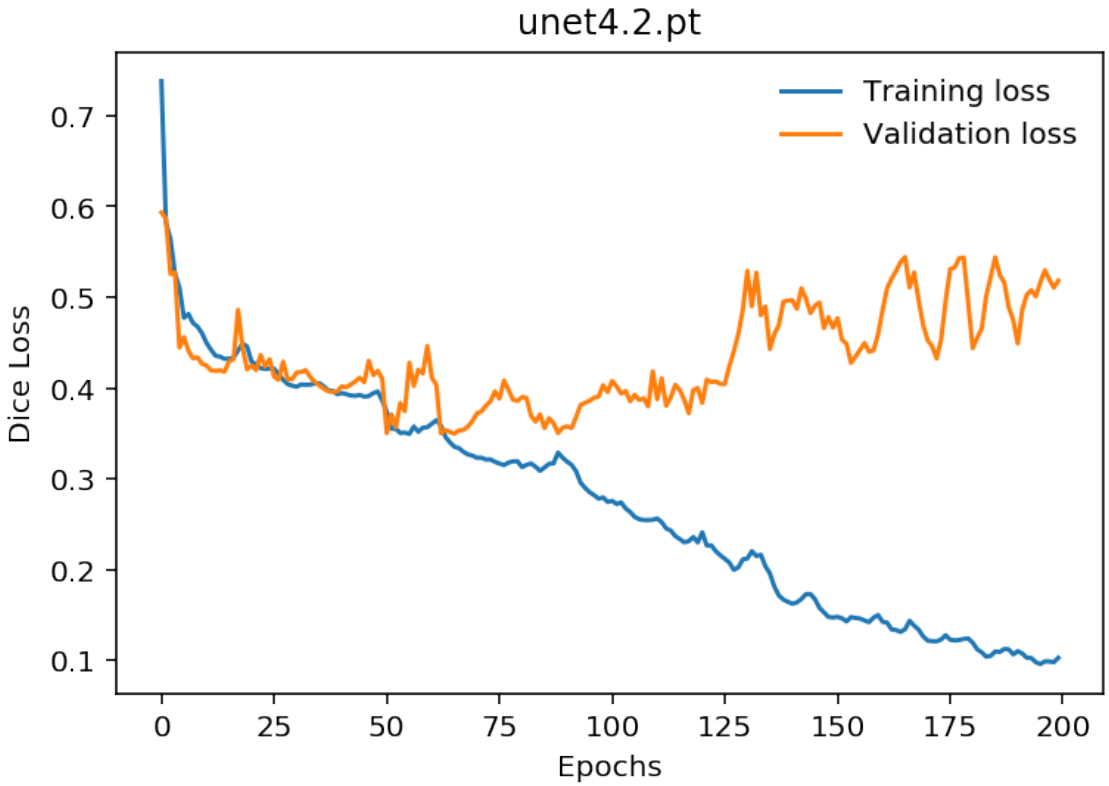

Figura 6.2: Arquitectura U-Net media. Batch=1. Adam con mejores parámetros. Hay sobreajuste.

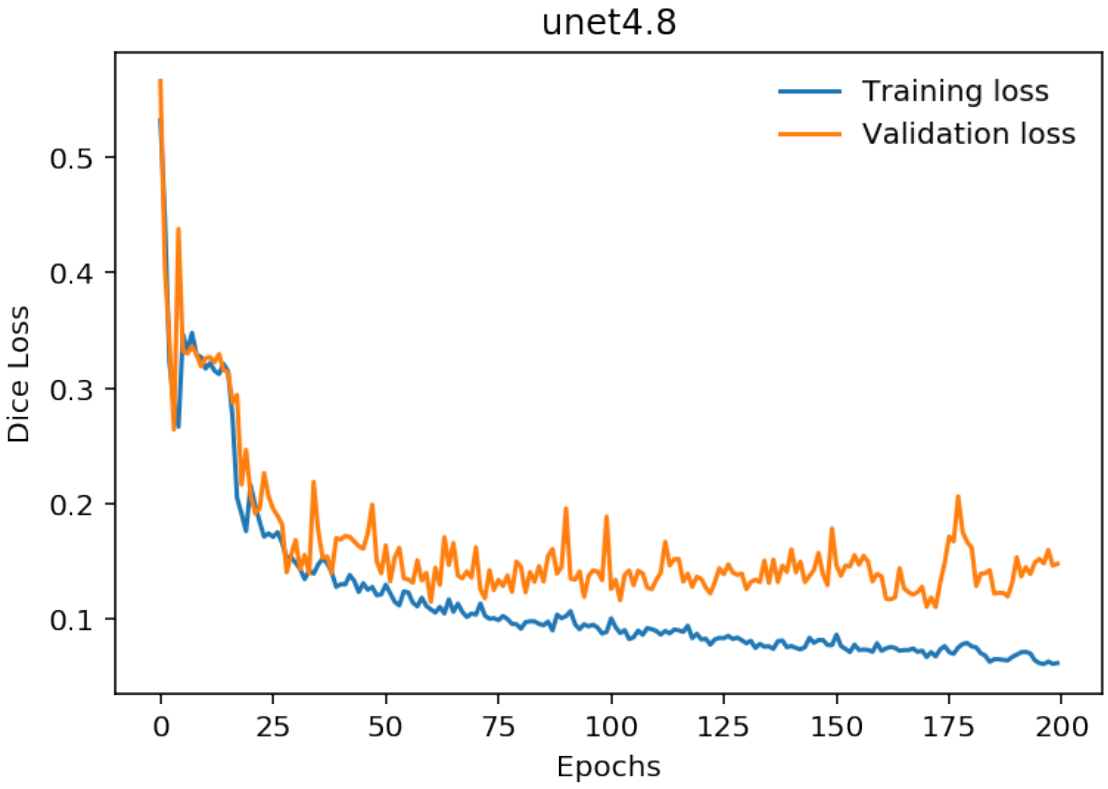

Figura 6.3: Arquitectura U-Net media. Con data augmentation. Data augmentation elimia el sobreajuste.

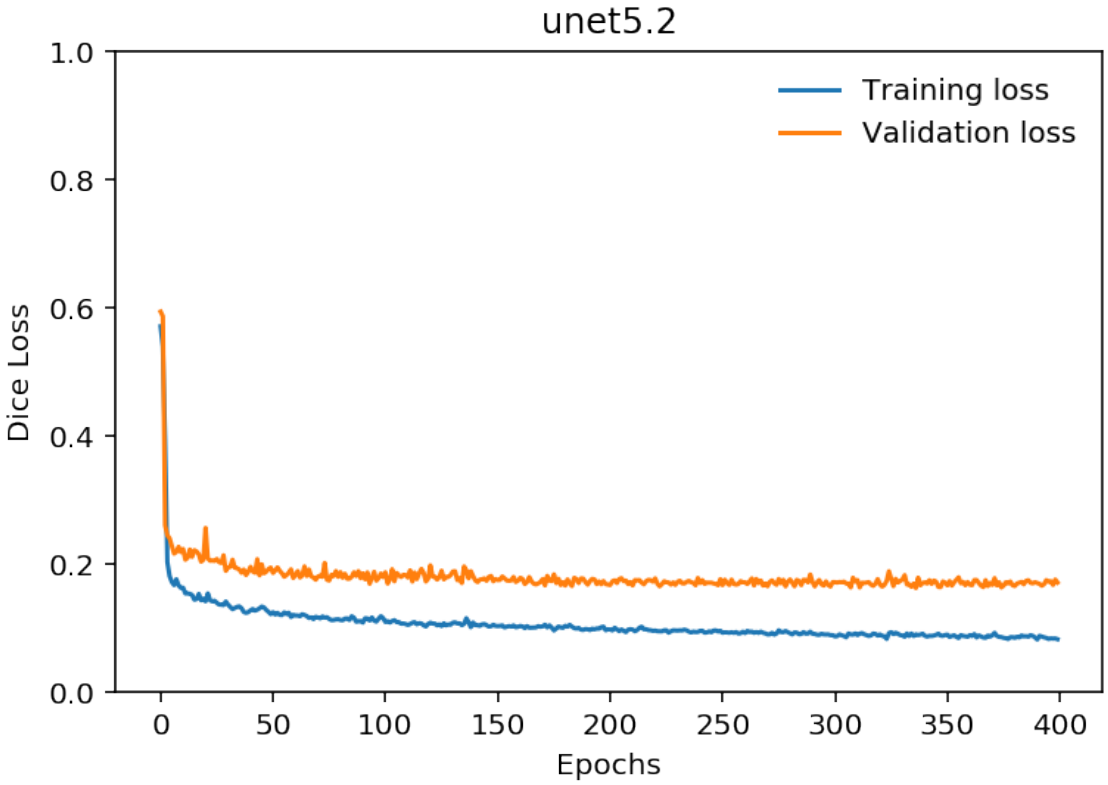

Figura 6.4: Arquitectura U-Net completa. Bordes. Data augmentation. Apex. El autoescalado del factor de pérdida de Apex aumenta la velocidad de entrenamiento.

```
Matriz de confusión por clases:
 tensor([[0.9896, 0.1620],
         [0.0104, 0.8380]])
miou:
 tensor([0.9709, 0.7708])
```

Figura 6.5: Arquitectura U-Net completa. Bordes. Data augmentation. Apex.

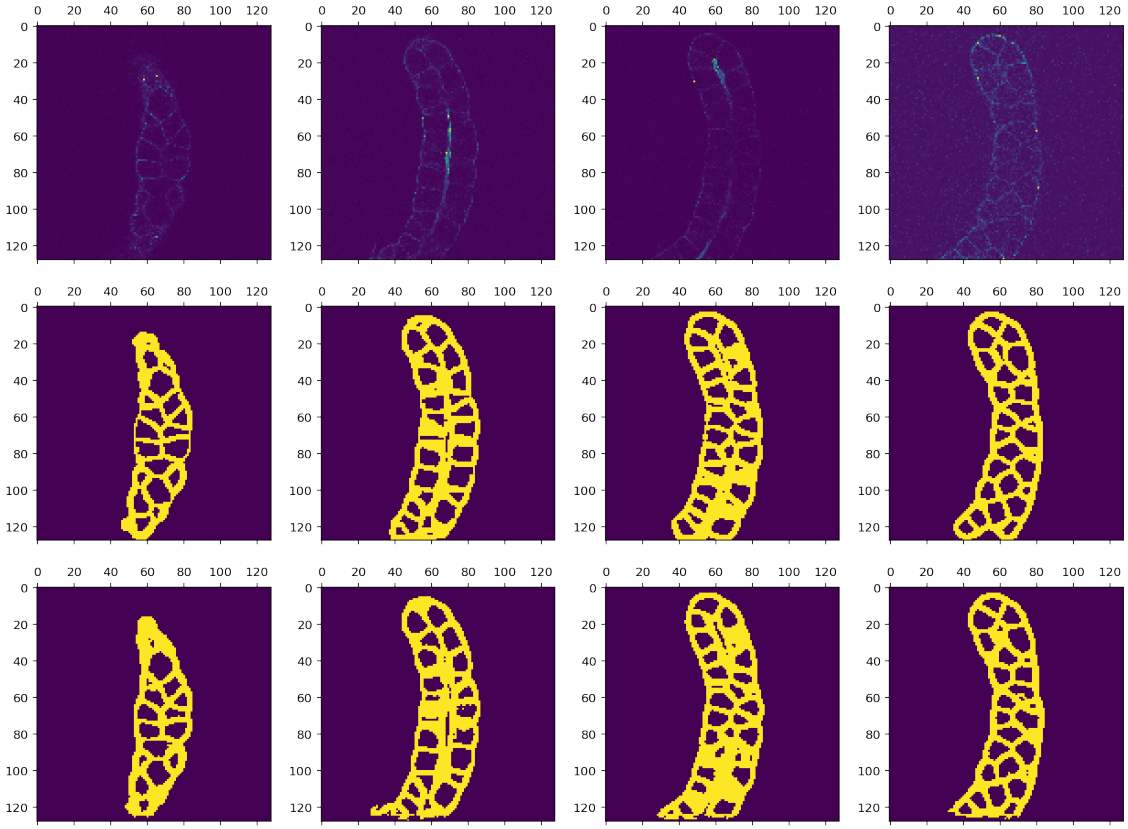

Figura 6.6: Arquitectura U-Net completa. Bordes. Data augmentation. Apex.

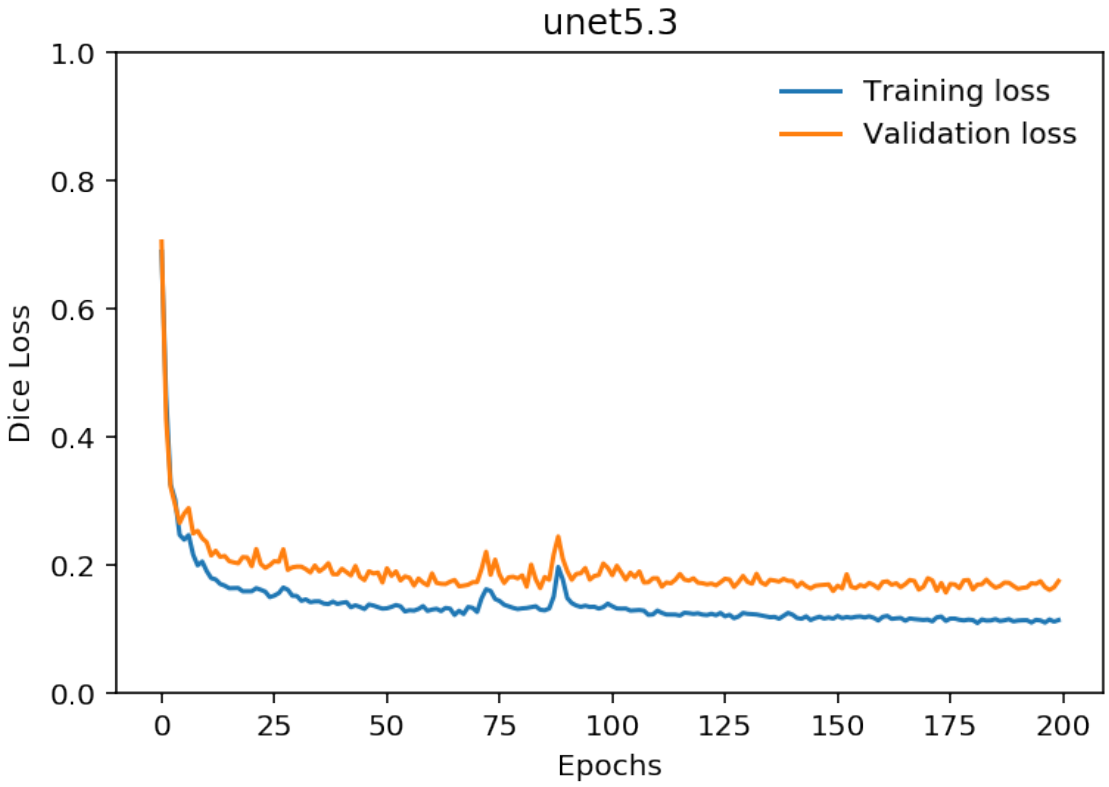

Figura 6.7: U-Net completa. Espaciado. Data augmentation. Apex

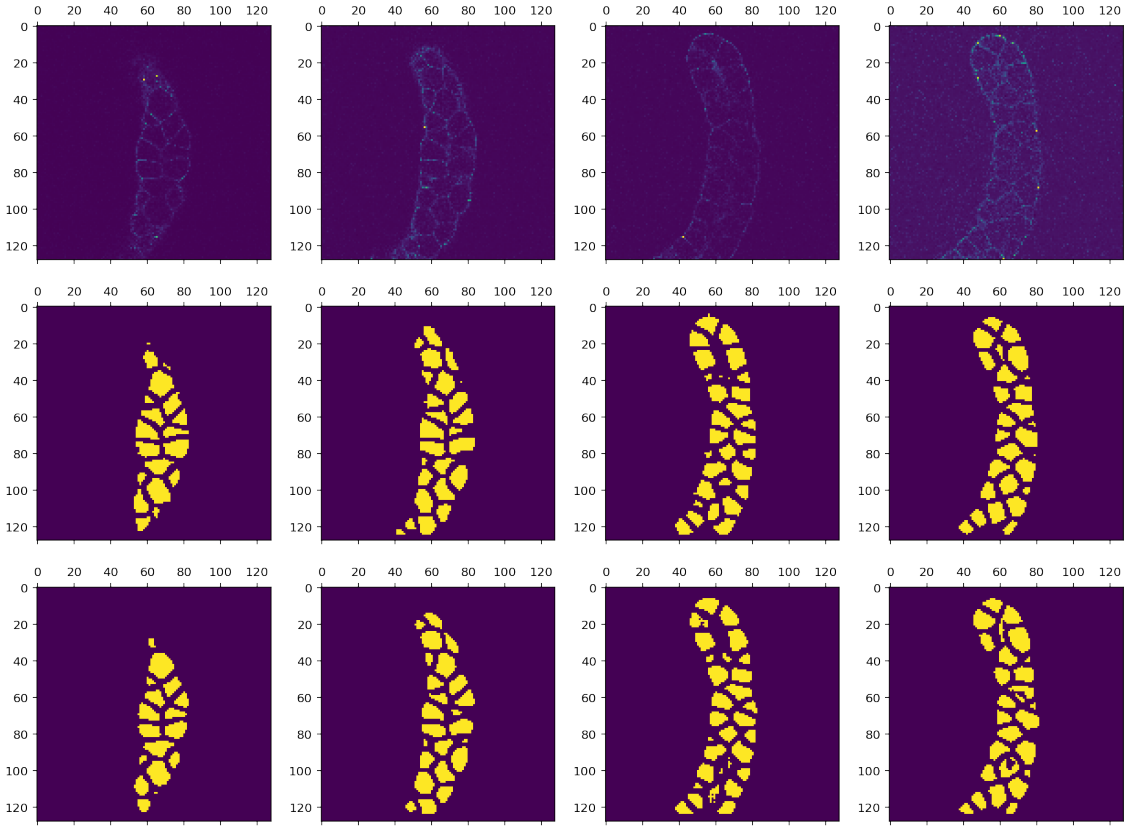

Figura 6.8: U-Net completa. Espaciado. Data augmentation. Apex

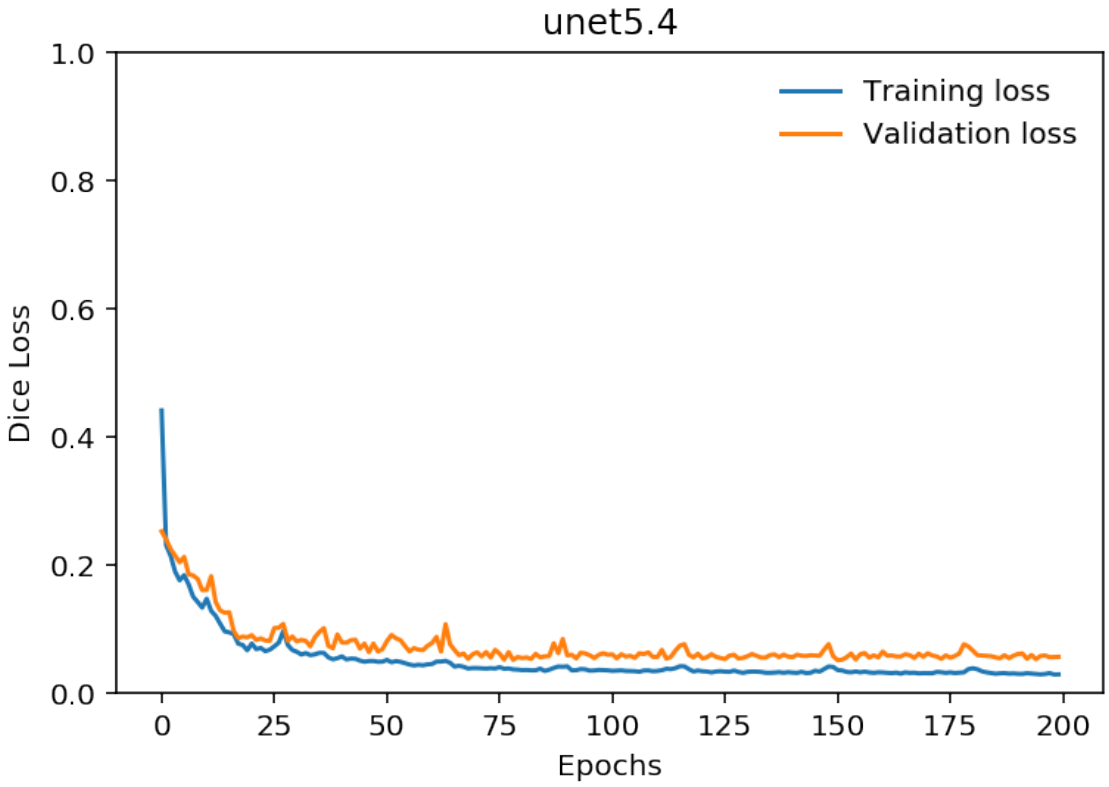

Figura 6.9: U-Net completa. Sin espaciado. Data augmentation. Apex

```
Matriz de confusión por clases de esta imagen:
tensor([[[0.9940, 0.0496],
         [0.0060, 0.9504]]])
IoU de esta imagen:
tensor([[0.9869, 0.9127]])
```

Figura 6.10: U-Net completa. Sin espaciado. Data augmentation. Apex

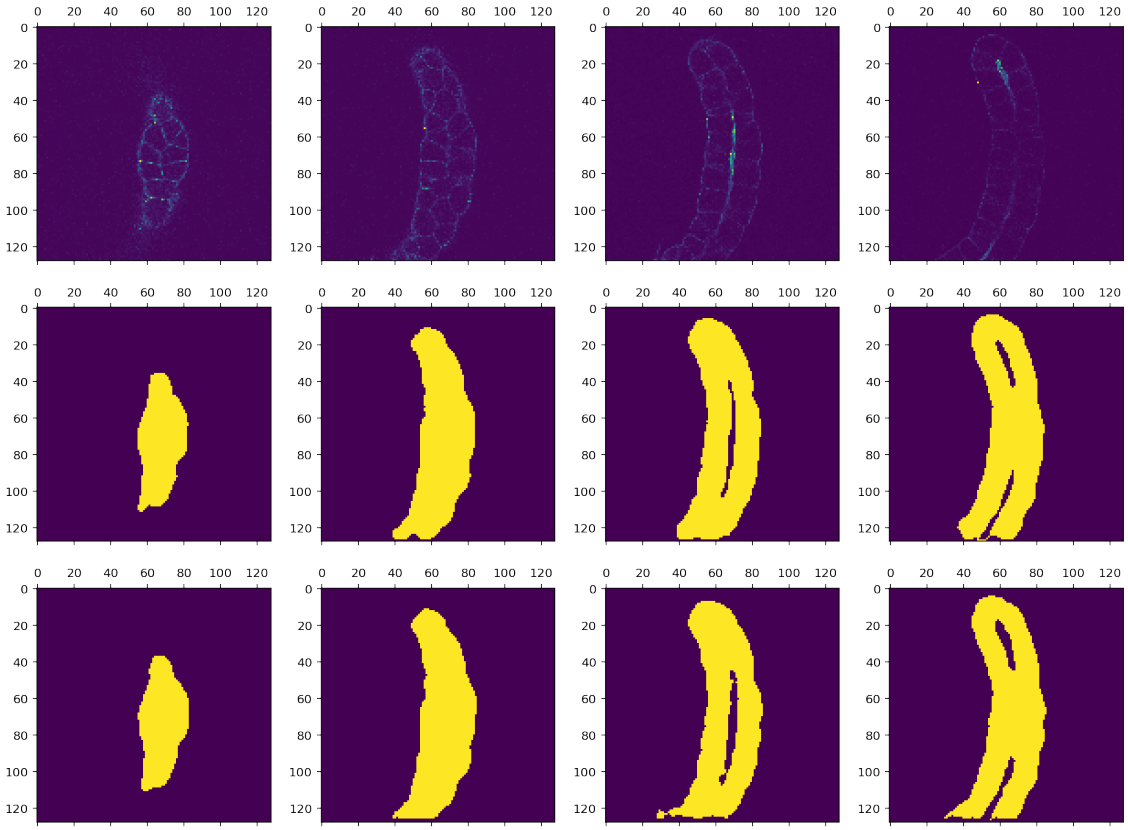

Figura 6.11: U-Net completa. Sin espaciado. Data augmentation. Apex

Referencias

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)

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.*

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.

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.

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

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

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