



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.0.3.0)

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 procesamiento 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 procesamiento 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 antecedentes 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 fichero python para efectuar entrenamiento e inferencias por línea de comandos.

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 artificial	3
2.1.3 Red Neuronal Artificial	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 conteo, 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 Análisis de requisitos, diseño e implementación	20
3.1 Datos iniciales	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

<i>Índice general</i>	3
6 Pruebas	33
Referencias	40

Índice de cuadros

2.1	Pipeline que siguen los datos de inicio a fin. GT significa <i>groundtruth</i> 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 <i>groundtruth</i> 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

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 Convolutiva 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 $7 \times 7 \times 3$, volumen generado al aplicar padding de 1 px a una imagen de tamaño 5×5 con 3 canales. El resultado es un volumen de $3 \times 3 \times 2$	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 Stanford 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	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	Arquitectura U-Net media. Batch=1. Znorm. Hay sobreajuste.	33
6.2	Arquitectura U-Net media. Batch=1. Adam con mejores parámetros. Hay sobreajuste.	34
6.3	Arquitectura U-Net media. Con data augmentation. Data augmentation elimina el sobreajuste.	35
6.4	Arquitectura U-Net completa. Bordes. Data augmentation. Apex. El autoescalado del factor de pérdida de Apex aumenta la velocidad de entrenamiento.	36
6.5	Arquitectura U-Net completa. Bordes. Data augmentation. Apex.	36
6.6	Arquitectura U-Net completa. Bordes. Data augmentation. Apex.	37
6.7	U-Net completa. Espaciado. Data augmentation. Apex	37
6.8	U-Net completa. Espaciado. Data augmentation. Apex	38
6.9	U-Net completa. Sin espaciado. Data augmentation. Apex	38
6.10	U-Net completa. Sin espaciado. Data augmentation. Apex	39
6.11	U-Net completa. Sin espaciado. Data augmentation. Apex	39

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 desarrolladas en las células. Sin embargo, esta información es omitida en la mayoría de estudios a causa del esfuerzo que conlleva.

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 n° 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.

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 popularización del algoritmo de propagación hacia atrás, **backpropagation** (Rumelhart, Hinton, y Williams, 1986) para entrenar redes neuronales artificiales 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 artificial

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

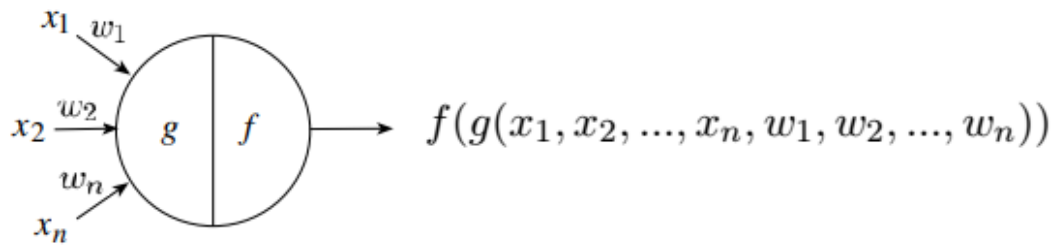


Figura 2.1: Neurona artificial genérica

Siendo:

- (x_1, x_2, \dots, x_n) el vector de entrada.
- (w_1, w_2, \dots, 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.

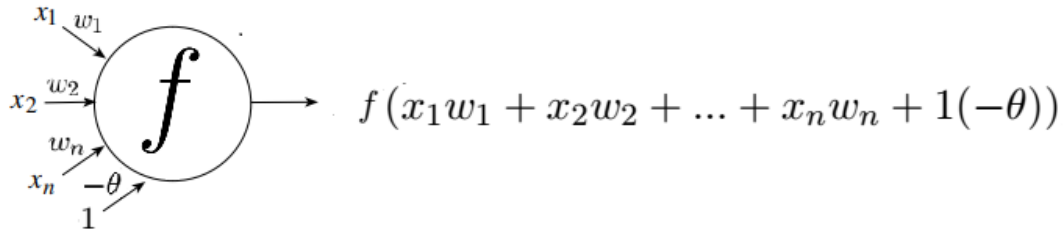


Figura 2.2: Representación de una Neurona Artificial con $\sum x_i w_i$ como función de integración

- 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 neural 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):

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

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).

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

2.1.3. Red Neuronal Artificial

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 artificiales se consigue concatenando las salidas de unas con la entradas de otras y obteniendo así una red neuronal artificial (ANN).

En la figura 2.3 se ve una Red Neuronal Artificial 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.

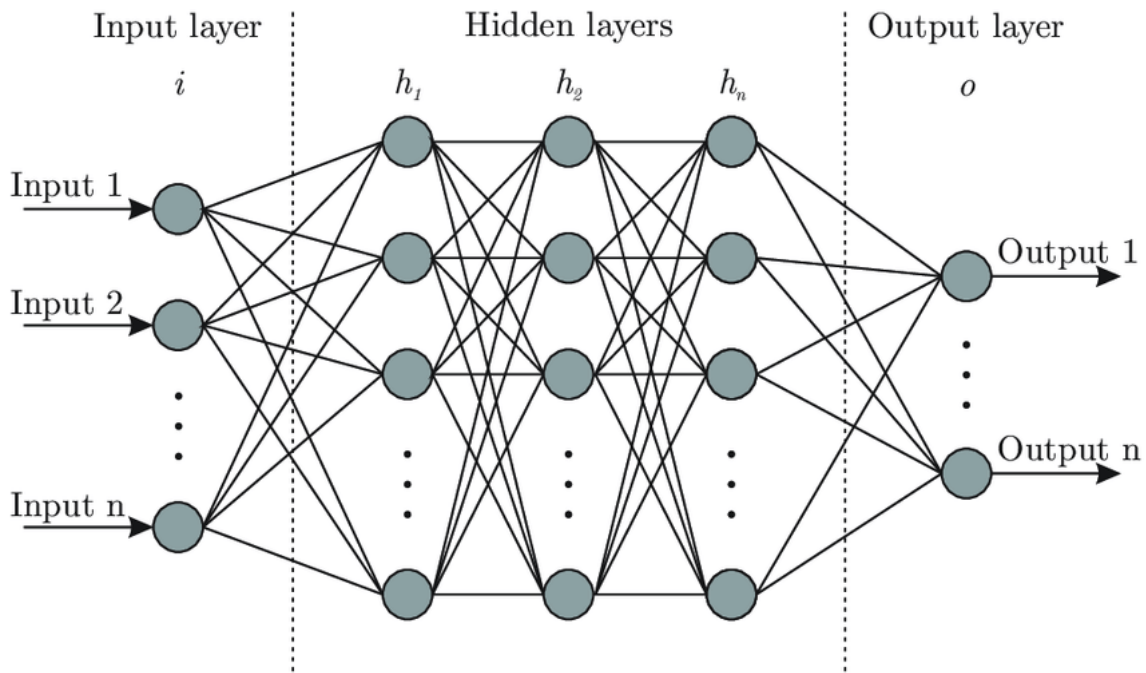


Figura 2.3: Red Neuronal Artificial completamente conectada.

- Capas ocultas (h_1, h_2, \dots, h_m) (*hidden layers*). Cada capa oculta podrá tener un n° 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_{ij} 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.

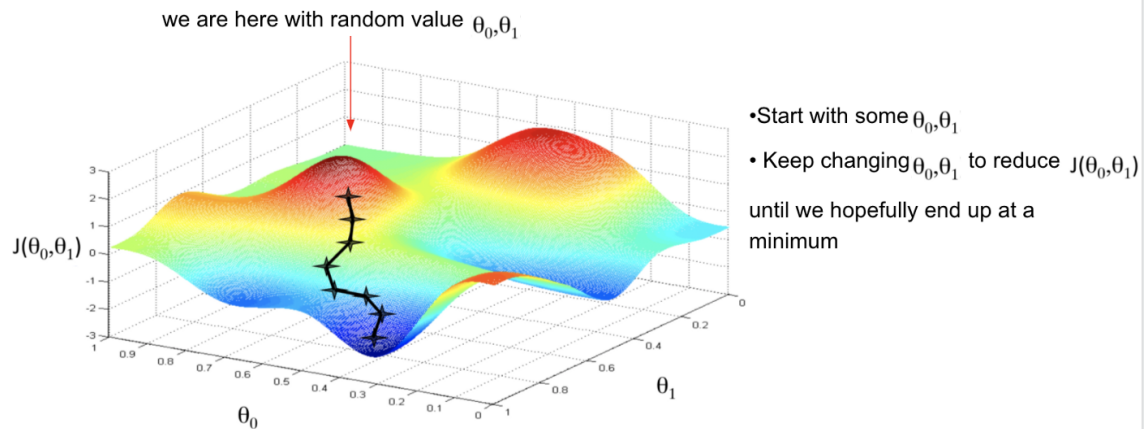


Figura 2.4: Ilustración del algoritmo de descenso por gradiente. θ_0 y θ_1 son los pesos de la ANN y J la función de pérdida. Se puede observar cómo se produce un "descenso" hacia un mínimo de la función.

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 $4 \times 4 \text{ px}$ y 4 neuronas en la única capa oculta, se tendrían $16 \times 4 = 64$ pesos. Si aplicáramos este sistema a una imagen 3D en escala de grises con $124 \times 124 \times 70 = 1076320 \text{ vox}$, 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 n° de pesos necesarios en la red neuronal y aprovecharán técnicas del procesamiento de imágenes para encontrar patrones.

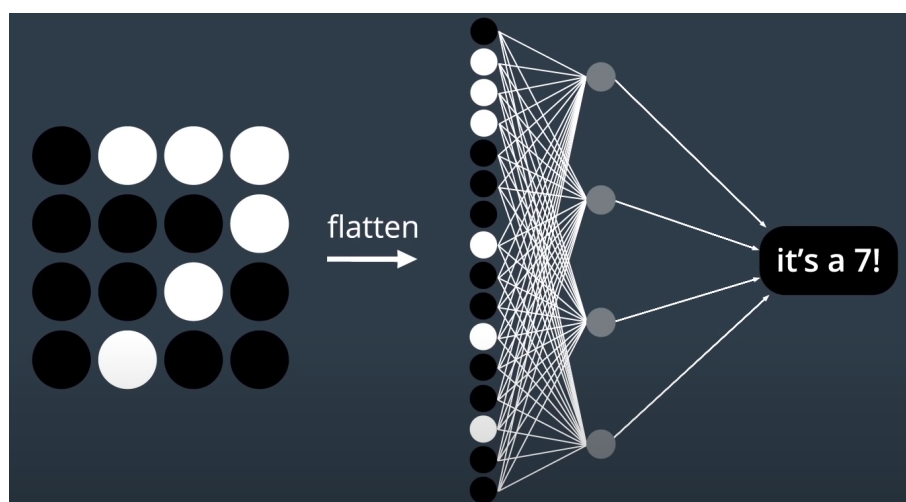


Figura 2.5: Imagen de $4 \times 4 \text{ px}$ aplanada y usada como entrada en una FCNN con 4 neuronas en su única capa oculta. No se muestra su capa de salida. Imagen del curso Intro to Deep Learning with PyTorch de Udacity.

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.

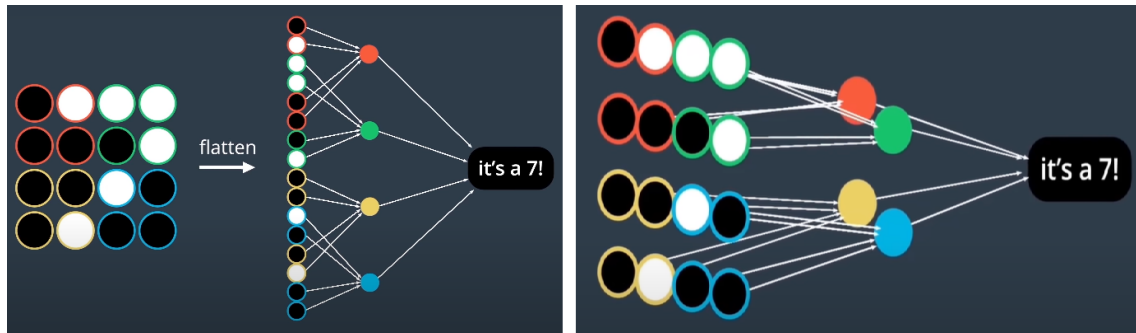


Figura 2.6: Imagen de 4x4 px usada como entrada en una CNN. Ambas figuras muestran la misma arquitectura, estando en la figura de la izquierda la imagen aplanada y en la figura de la derecha se muestra la matriz 4x4 como entrada. Imágenes del curso Intro to Deep Learning with PyTorch de Udacity.

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 Convolutiva, 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.

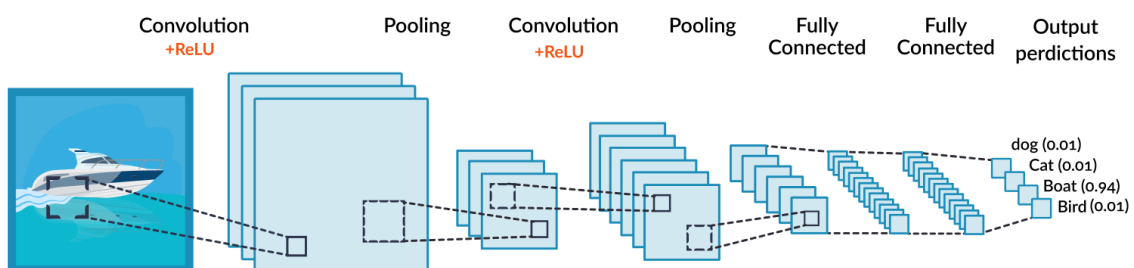


Figura 2.7: Red Neuronal Convolutiva simple en la que la imagen de un barco es clasificada.

Capa convolutiva

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 n° 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 inicializarán 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:

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

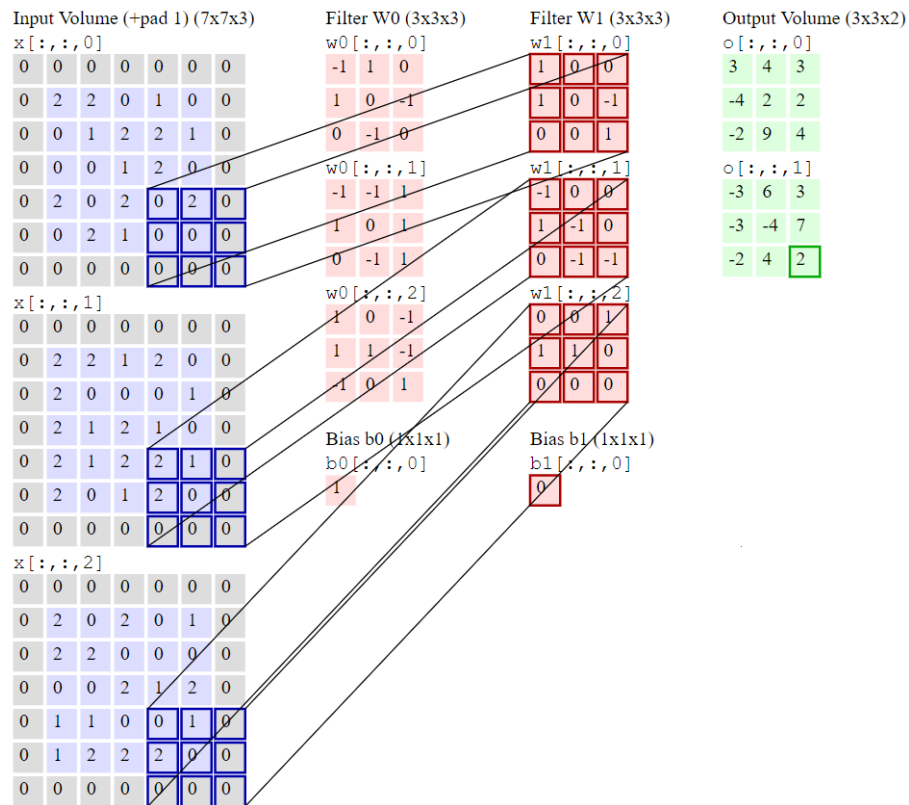


Figura 2.8: Capa de convolución de profundidad 2 (se usan dos filtros) con kernel de tamaño $3 \times 3 \times 3$ aplicado a un volumen de entrada de tamaño $7 \times 7 \times 3$, volumen generado al aplicar padding de 1 px a una imagen de tamaño 5×5 con 3 canales. El resultado es un volumen de $3 \times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

- $D_2 = D_1$

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.

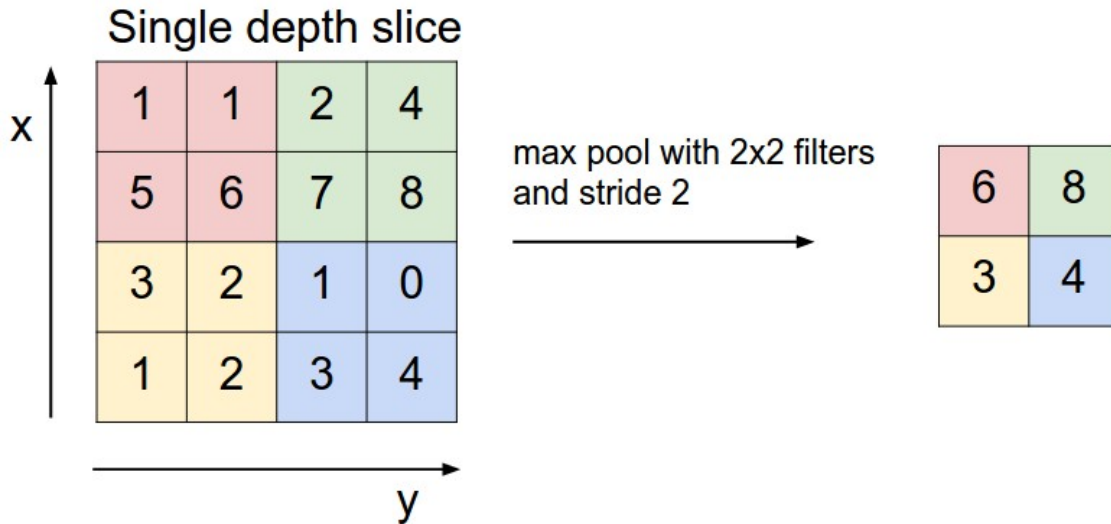


Figura 2.9: Imagen 4×4 a la que se ha aplicado un pooling de $F = 2, S = 2$. Cada color de la imagen de la izquierda indica las entradas del para la operación MAX, cada color de la imagen de la derecha indica la salida de dicha operación. Figura tomada del curso CS231n de Stanford University.

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 n° de clases. En problemas de segmentación esta capa tendrá tantos nodos como clases haya multiplicado por el n° 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:

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

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:

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

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:

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

Siendo $p \in [0, 1]$ la probabilidad de que un píxel/vóxel pertenezca a una clase, siendo la suma de 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 competición 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 n° 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 (? , ?).

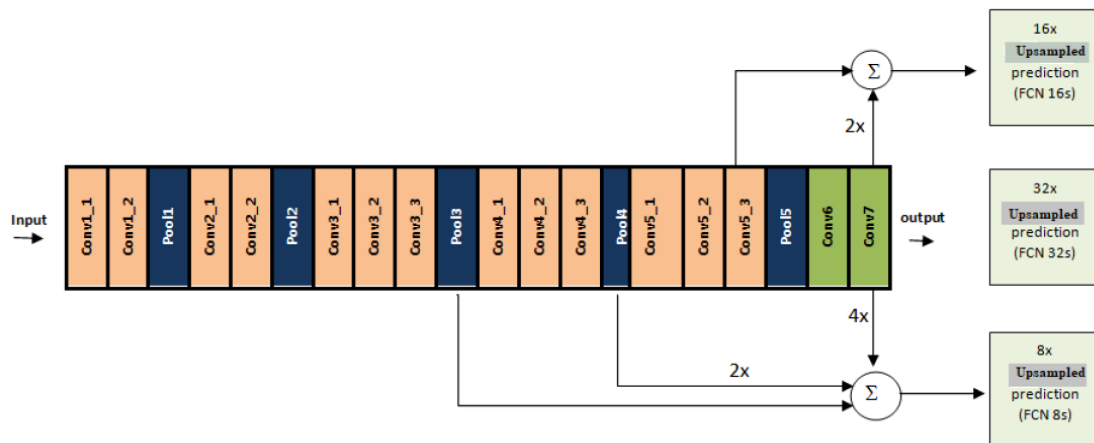


Figura 2.10: Arquitectura de FCN32, FCN16 y FCN8.

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 deconvolucional, 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 ??.

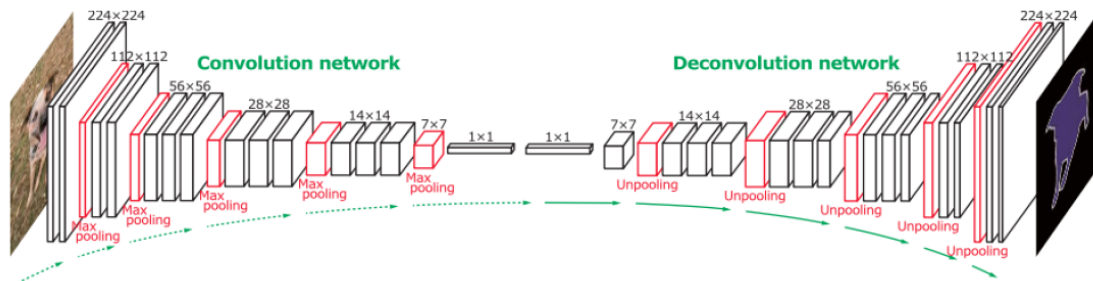


Figura 2.11: Arquitectura Deconvnet.

A continuación 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 manteniendo 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).

Deconvolución

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.

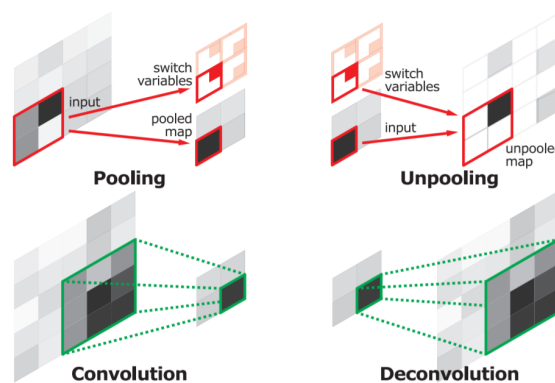


Figura 2.12: Figura que ilustra las operaciones unpooling y deconvolución.

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 "convolución hacia arriba", similar a la operación unpooling + deconvolución. La última capa es una convolución 1x1 para mapear cada píxel/vóxel con el n° correspondiente de clases. En la arquitectura original se tienen en total 23 capas convolucionales. En la figura 2.13 puede verse la arquitectura original.

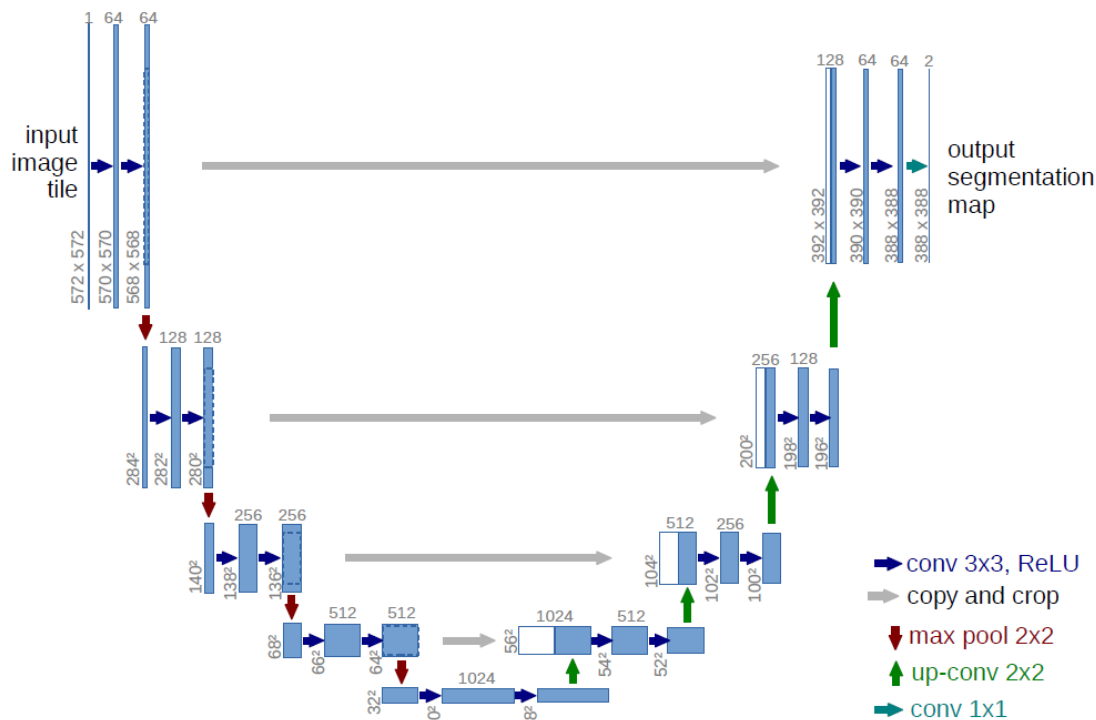


Figura 2.13: Arquitectura original U-Net propuesta por Ronneberger et al en U-Net: Convolutional Networks for Biomedical Image Segmentation.

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 mostrarán resultados obtenidos y por último mencionará el software resultante de estas investigaciones.

2.4.1. U-Net para conteo, 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} $\beta_1 : 0,9$, $\beta_2 : 0,999$ 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 imagen va a tener un peso propio. El peso de cada vóxel viene dado por la siguiente fórmula:

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

Donde $\lambda \in \mathbb{R}_{\geq 0}$ controla la importancia de la separación de instancia. w'_{bal} y w_{sep} son calculados de la siguiente forma:

$$w'_{bal}(x) := \begin{cases} 1 & y(x) > 0 \\ v_{bal} + (1 - v_{bal}) * \exp\left(-\frac{d_1^2(x)}{2\sigma_{bal}^2}\right) & y(x) = 0 \\ 0 & y(x) \text{ desconocido} \end{cases} \quad (2.7)$$

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

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

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

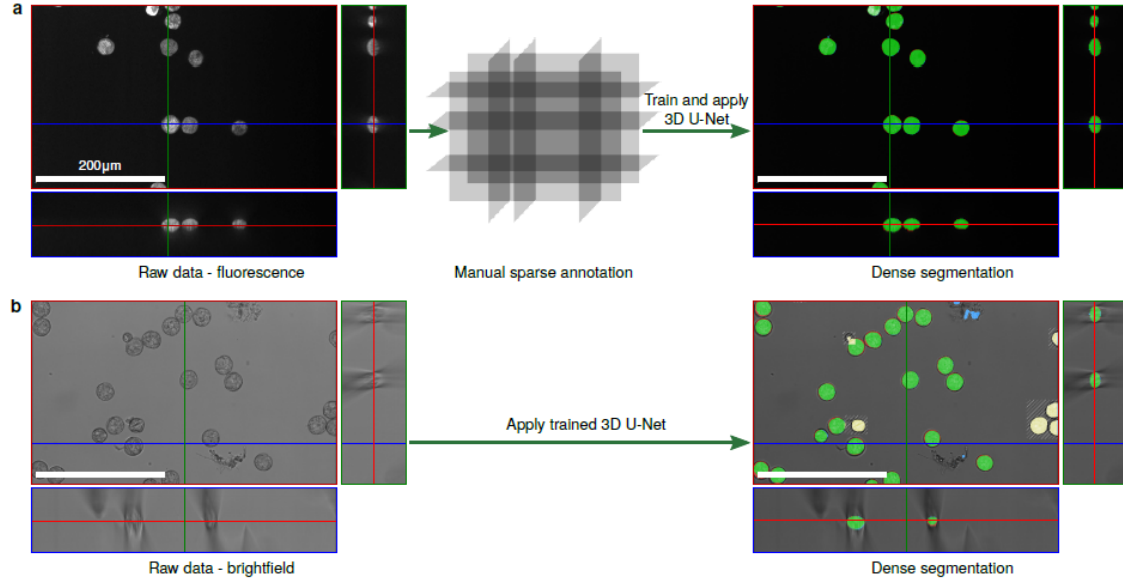


Figura 2.14: Segmentación volumétrica. (a) Segmentación de una imagen con un modelo que se ha entrenado con imágenes del mismo dataset. (b) Segmentación de una imagen con un modelo entrenado con un dataset distinto.

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

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

Donde A es el conjunto de vóxeles pertenecientes a la imagen perfectamente etiquetada y B el conjunto de vóxeles pertenecientes a la predicción. Un $M_{IoU} \in [0, 1]$ 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 GT:Desenfoque Gaussiano IM y GT:Data Augmentation - Volteo Horizontal y Vertical - Rotación en plano XY - Deformación elástica IM: Noise Augmentation IM,GT:Troceo de imagen a 170x170x80 vx	U-Net Funciones de pérdida probadas: -Entropía Cruzada Binaria -Pérdida Dice Optimizador ADAM: -Learning rate 10^{-5} - $\beta_1 : 0,9$, $\beta_2 : 0,999$ 100000 iteraciones	Transformada de la distancia Detectado de centroides Algoritmo watershed Grafo de adyacencia Particionamiento de grafo

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 (?, ?) (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

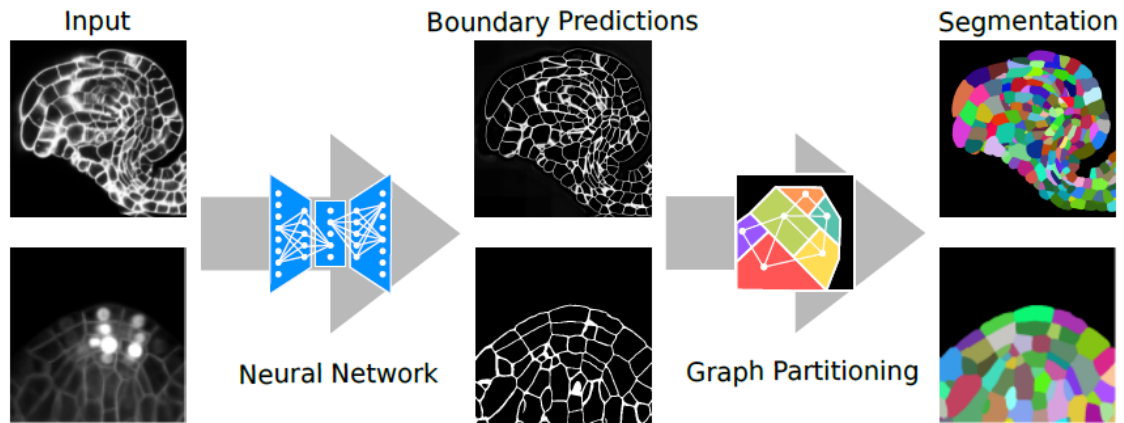


Figura 2.15: Segmentación de tejido vegetal usando PlantSeg. En el primer paso se predicen los bordes de las células usando una red U-Net 3D. En el segundo paso se aplica un algoritmo de particionamiento de grafo para segmentar cada célula.

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

- Precisión: N° vóxeles correctamente etiquetados como borde en la predicción entre el n° de vóxeles etiquetados como borde en la predicción.
- Exhaustividad: N° de vóxeles etiquetados correctamente en la predicción entre el n° 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:

$$VOI = H(seg|GT) + H(GT|seg) \quad (2.10)$$

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 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 preentrenados 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>.

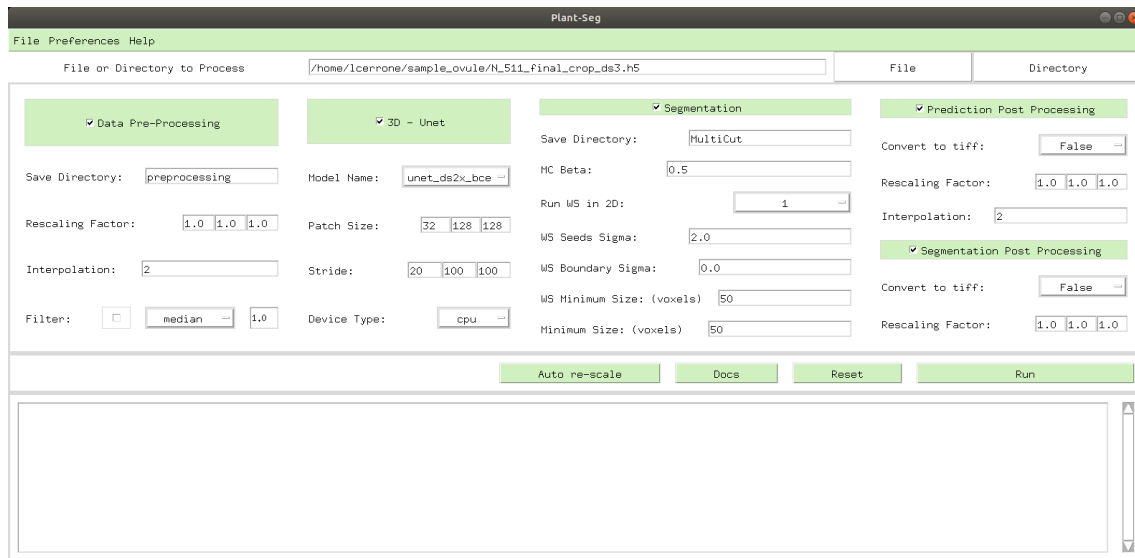


Figura 2.16: Interfaz gráfica del programa PlantSet. Se pueden ver las distintas opciones para cada paso del procesado.

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

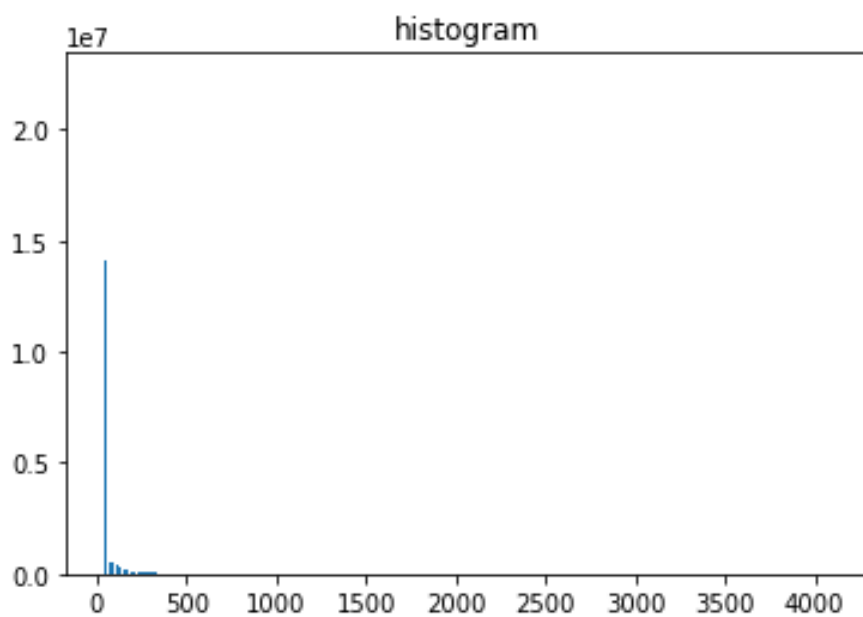


Figura 3.1: Histograma de todos los valores

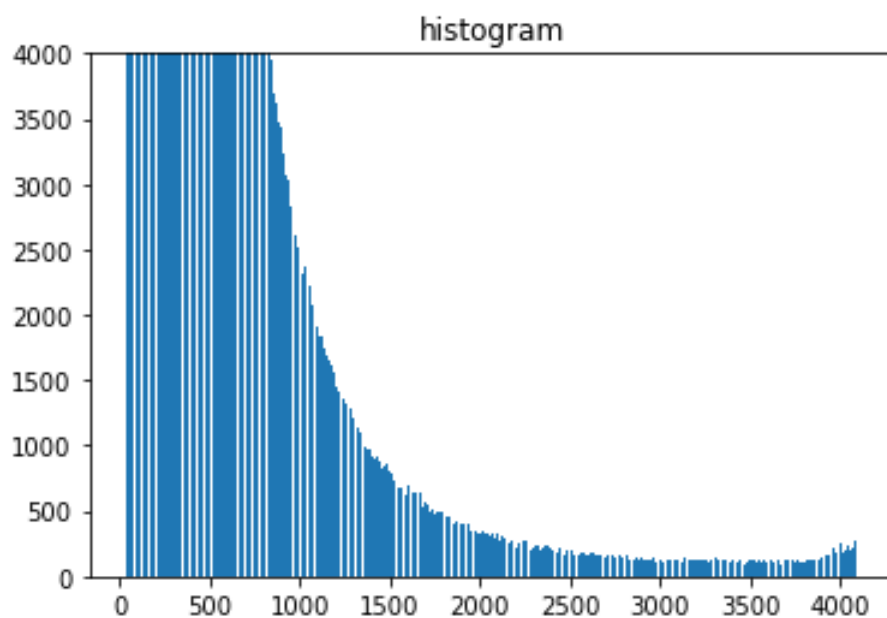


Figura 3.2: Histograma de todos los valores con el eje y limitado a 4000

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 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 $(200, 1024, 1024) \times 8$ 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 n° 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.

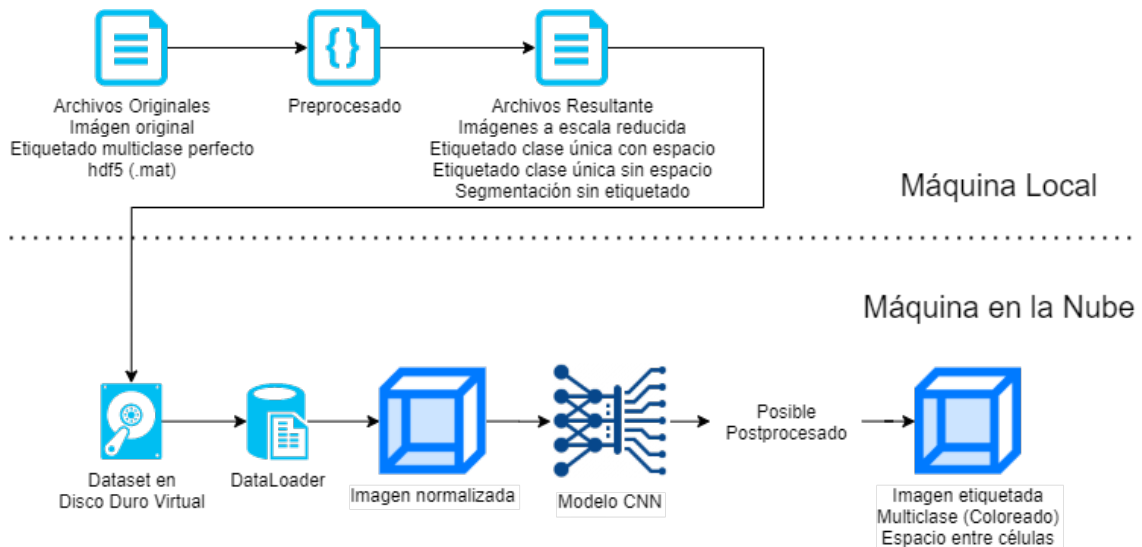


Figura 4.1: Diseño general

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 prepararán 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

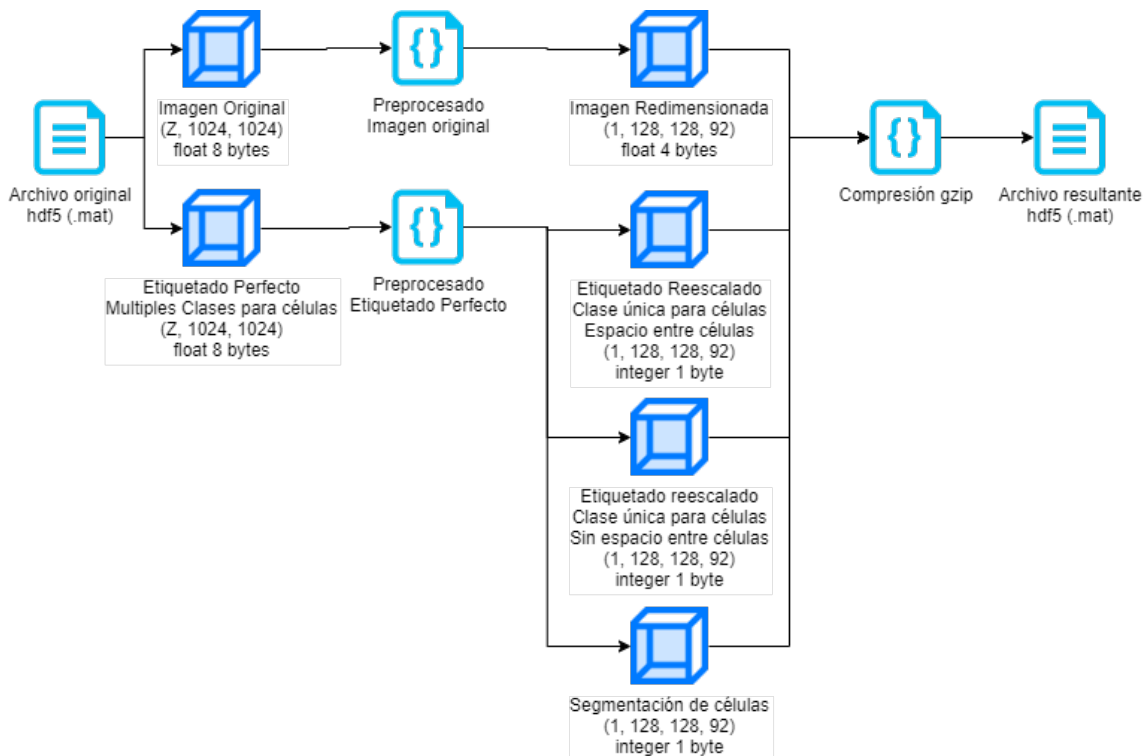


Figura 4.2: Preprocesado llevado a cabo en la máquina local.

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 $Z \in [216, 368]$. 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 “célula”), 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 n° 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 byte, 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

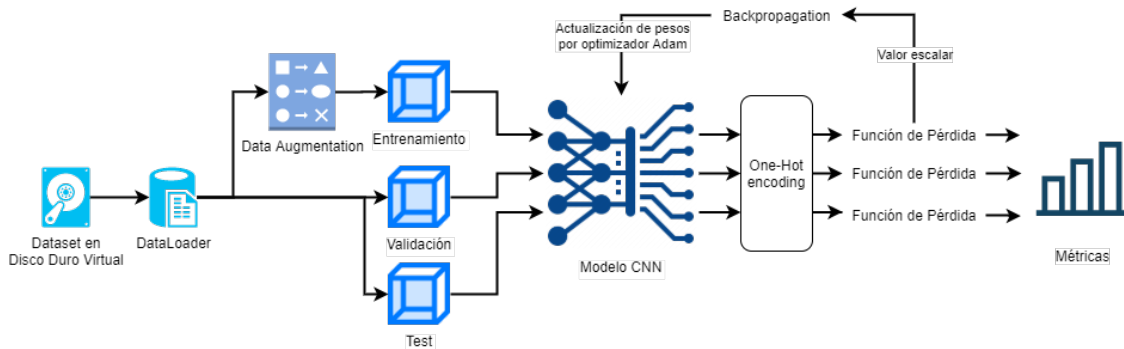


Figura 4.3

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 $N \in [1, 4]$. 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 n° 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 pérdida. 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

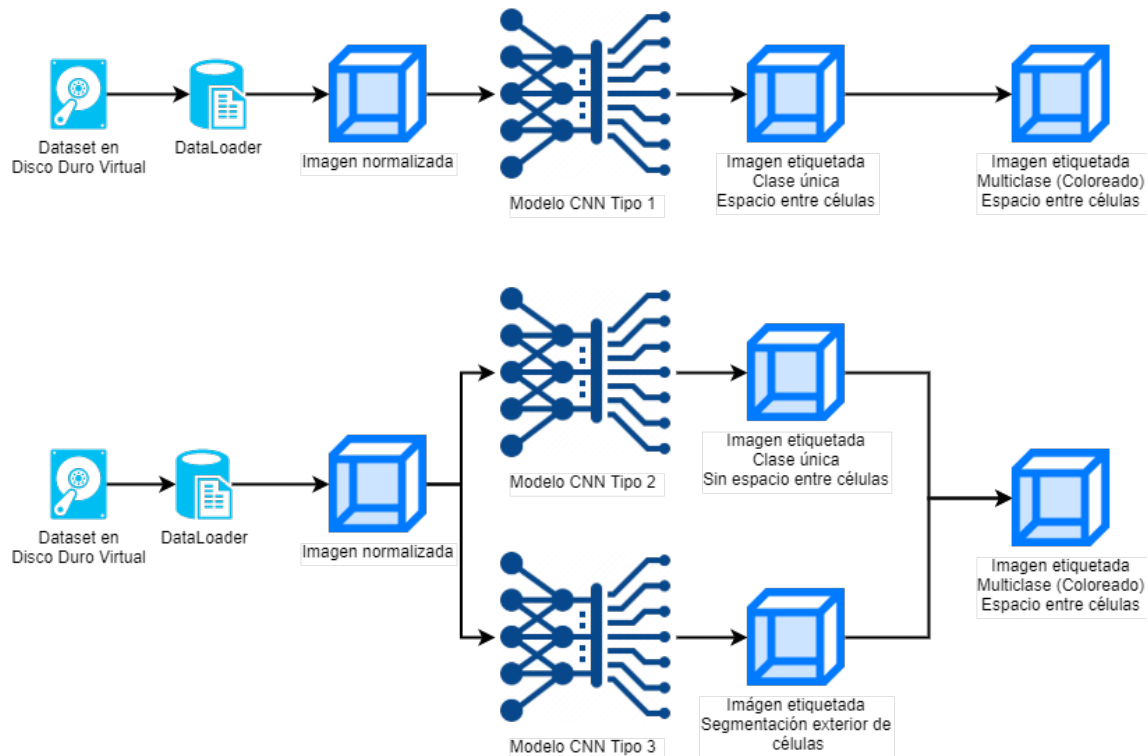


Figura 4.4: Se muestran dos métodos distintos para obtener la imagen correctamente etiquetada.

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 3** 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 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 n° de canales que tiene la imagen. Los n° mostrados en las capas determinan la profundidad de cada capa o, lo que es lo mismo, el n° 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.

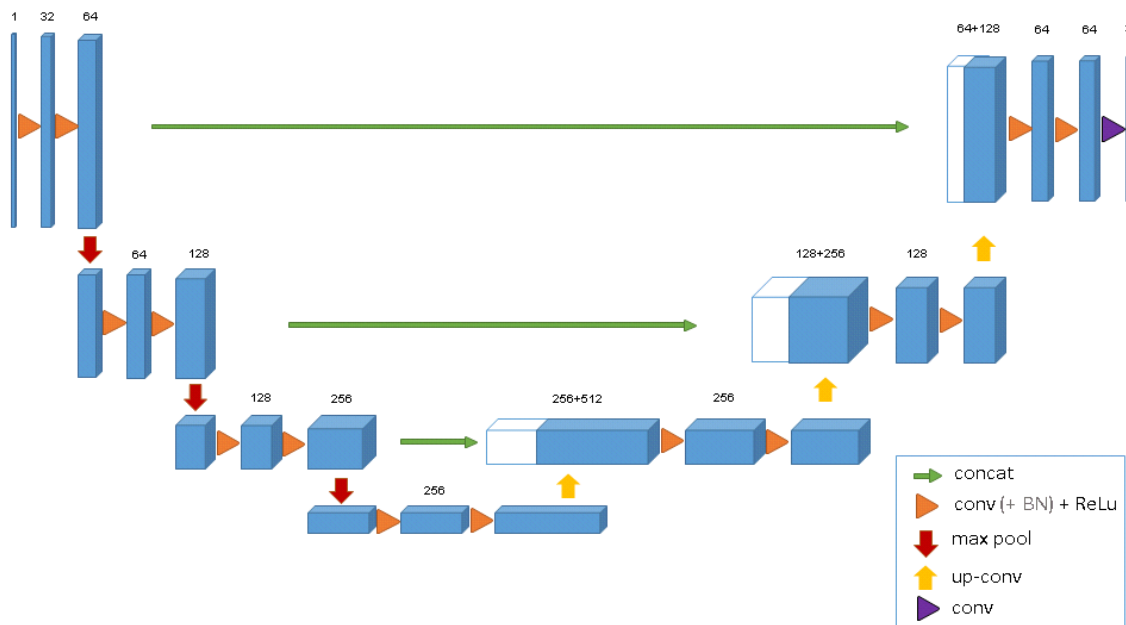


Figura 4.5: Arquitectura U-Net completa.

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.

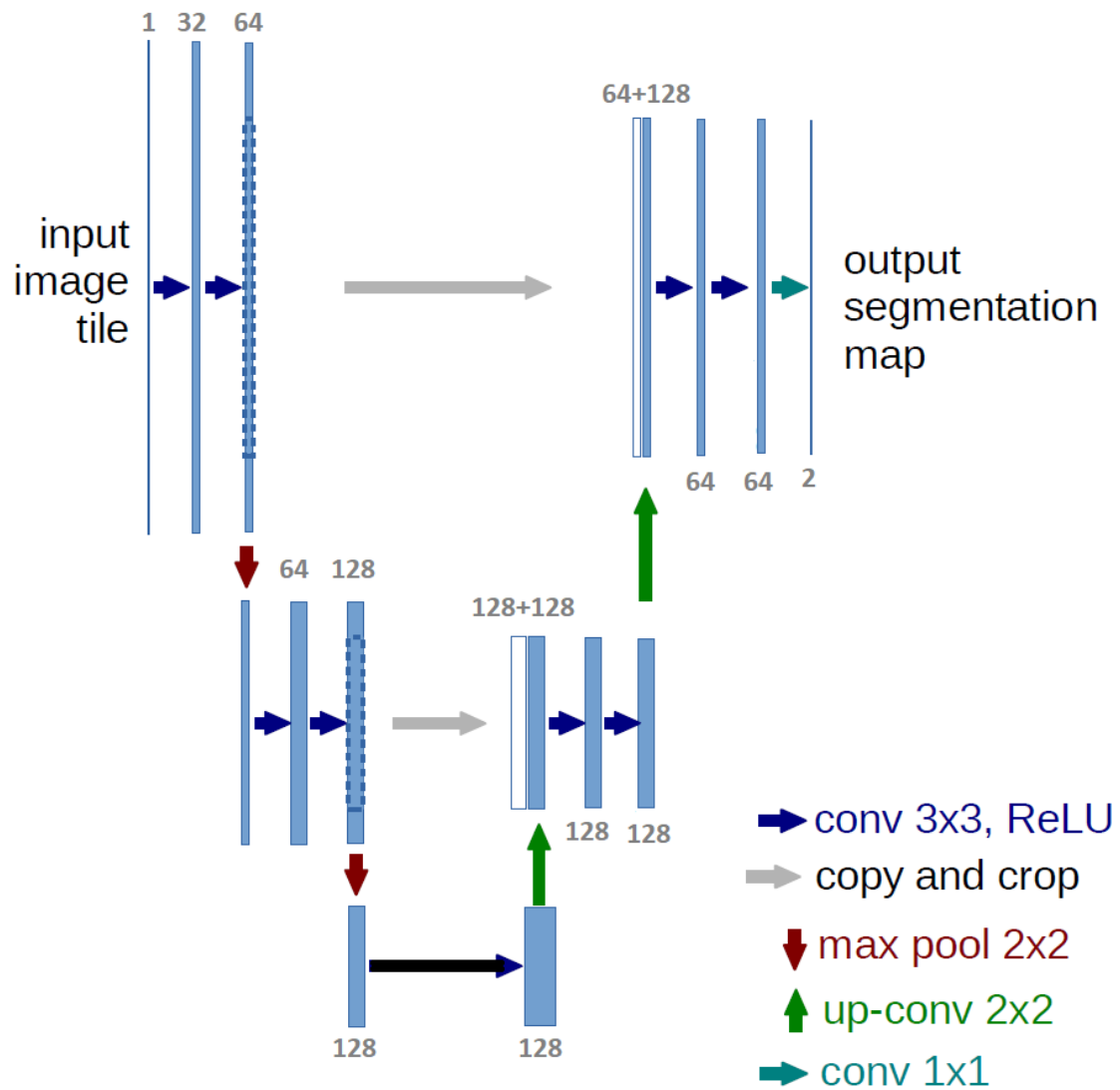


Figura 4.6: Arquitectura U-Net completa.

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 learning", "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 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.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, convirtiéndose 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** (?, ?). Aporta una interfaz para trabajar con el formato de datos HDF5 (?, ?), 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 seleccionando 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 n° 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.

Pruebas

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

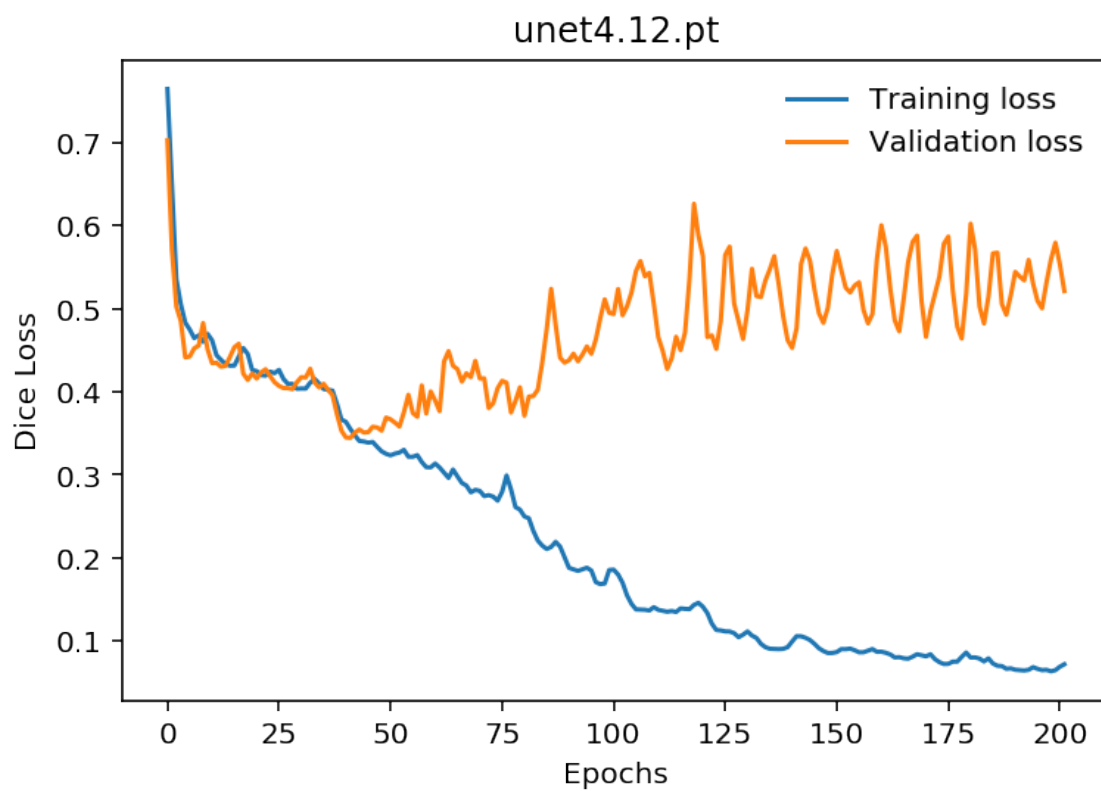


Figura 6.1: Arquitectura U-Net media. Batch=1. Znorm. Hay sobreajuste.

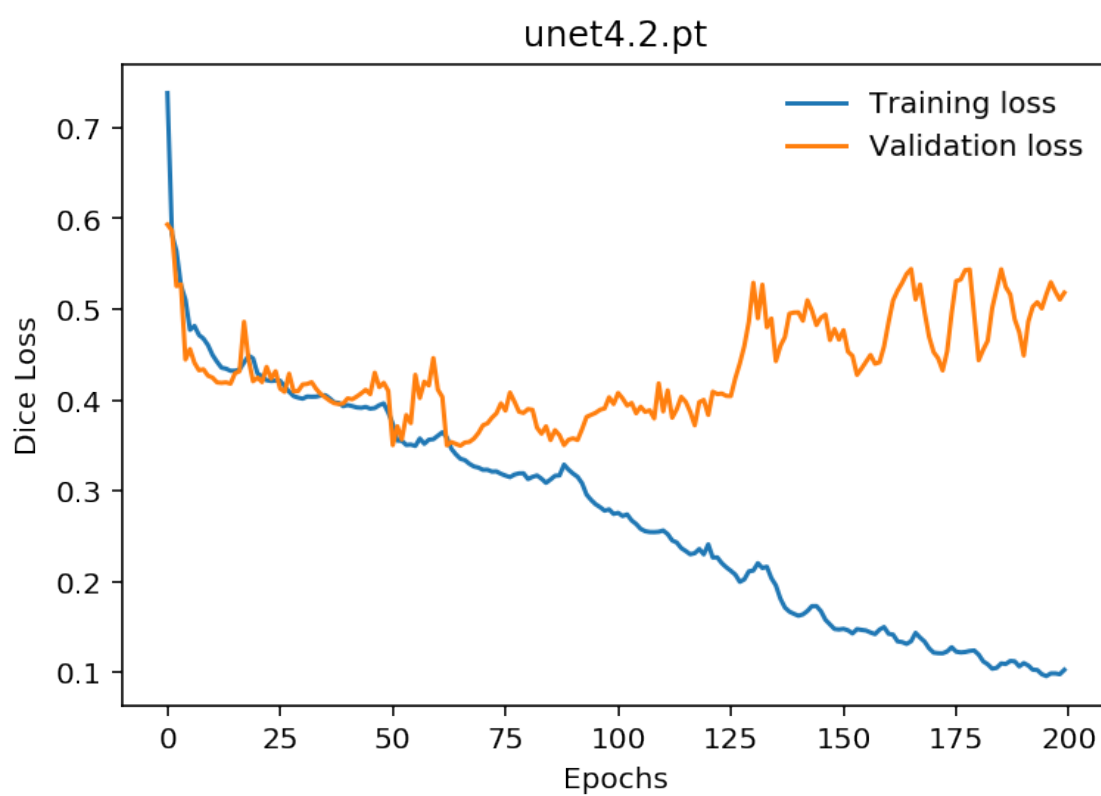


Figura 6.2: Arquitectura U-Net media. Batch=1. Adam con mejores parámetros. Hay sobreajuste.

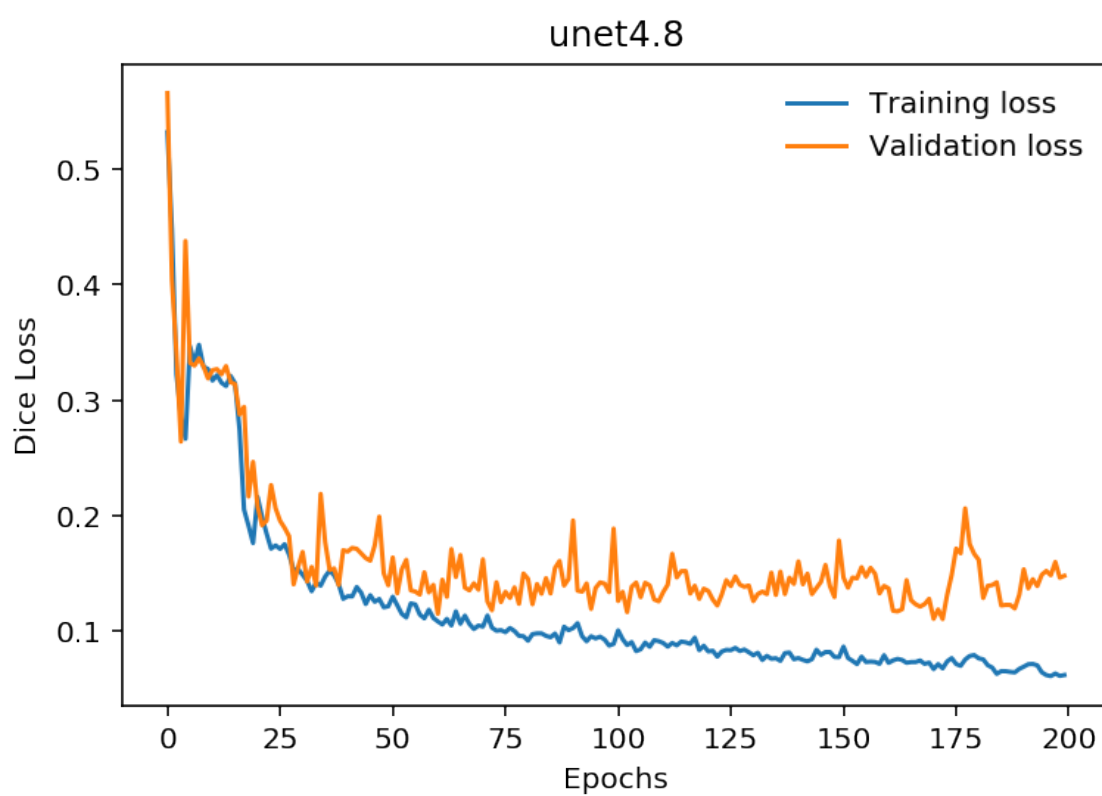


Figura 6.3: Arquitectura U-Net media. Con data augmentation. Data augmentation elimina el sobreajuste.

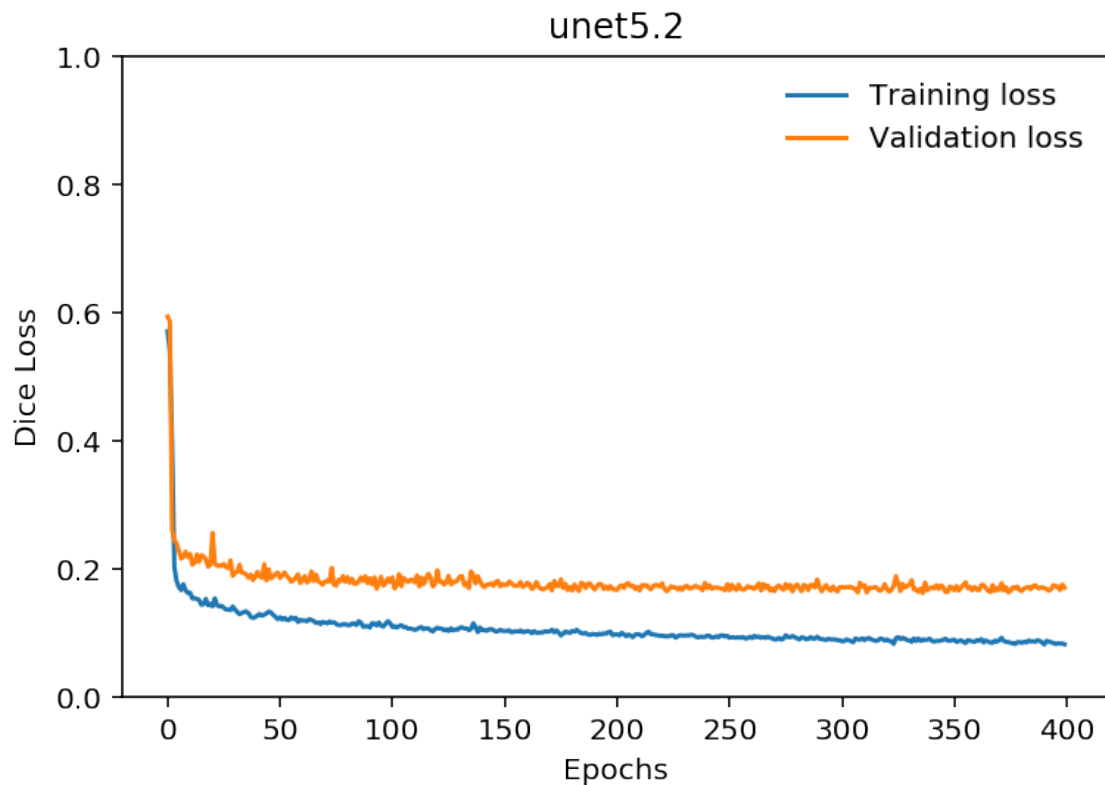


Figura 6.4: Arquitectura U-Net completa. Bordes. Data augmentation. Apex. El autoescalado del factor de pérdida de Apex aumenta la velocidad de entrenamiento.

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.

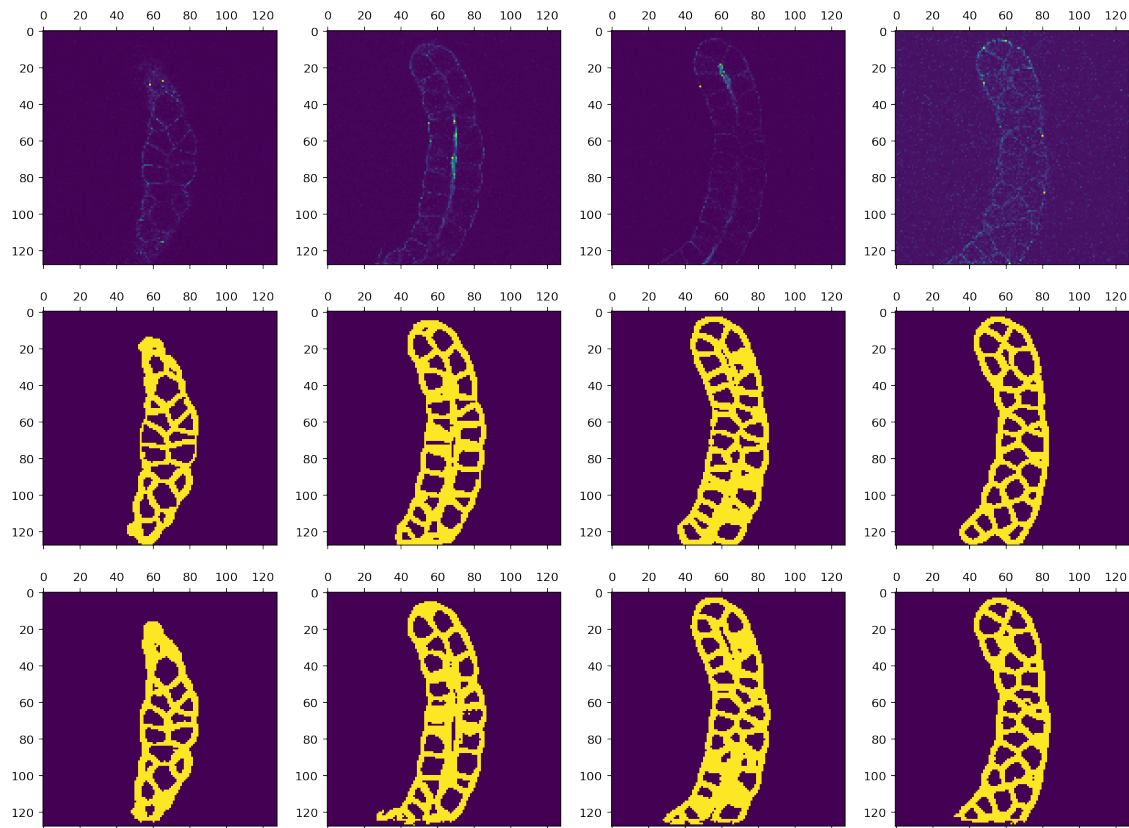


Figura 6.6: Arquitectura U-Net completa. Bordes. Data augmentation. Apex.

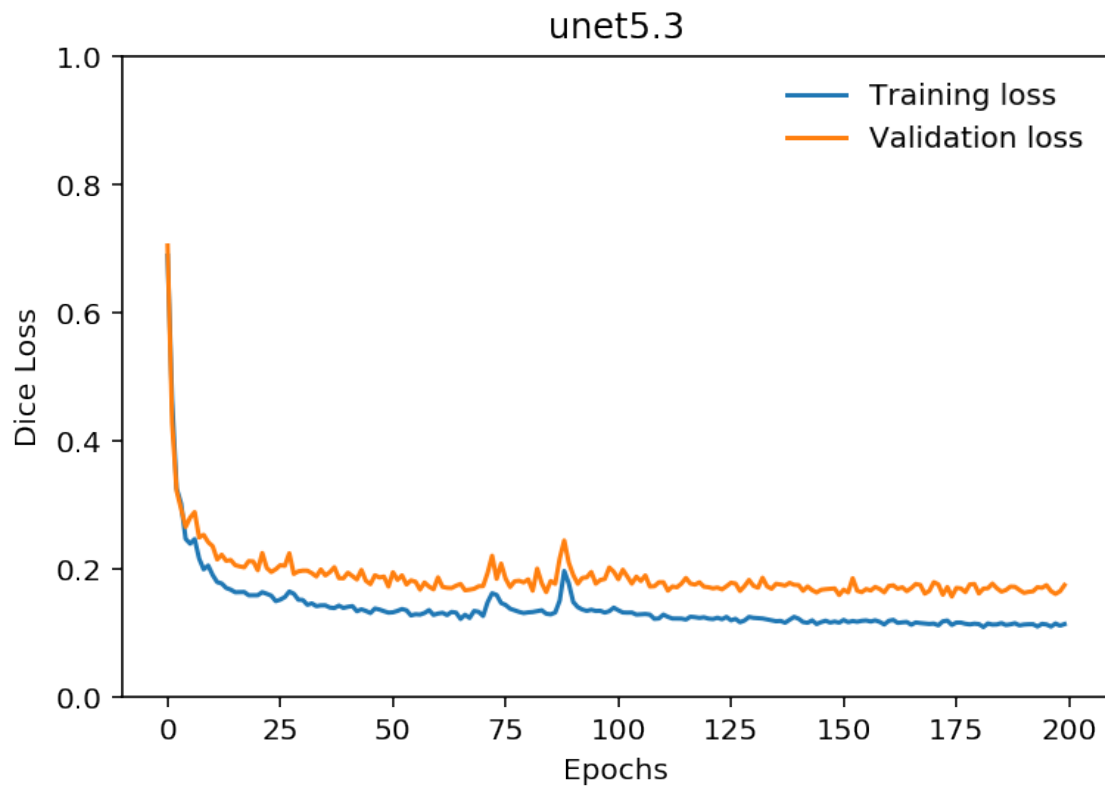


Figura 6.7: U-Net completa. Espaciado. Data augmentation. Apex

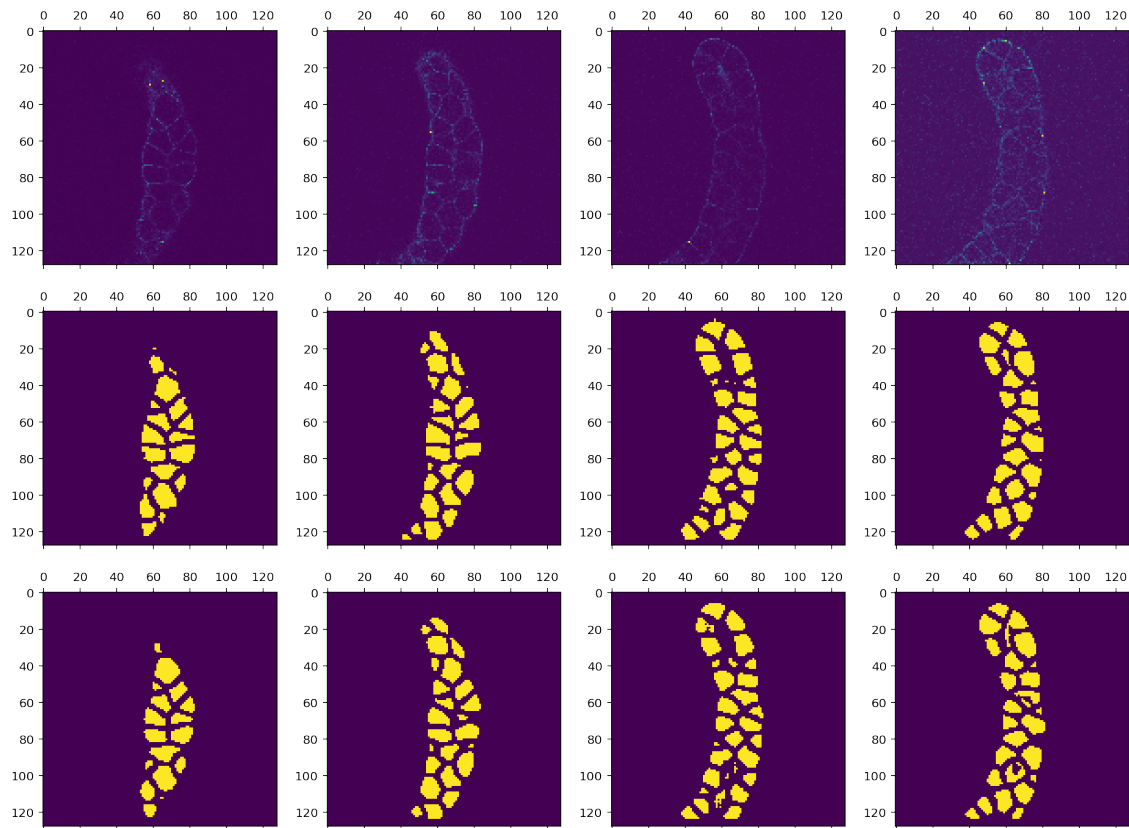


Figura 6.8: U-Net completa. Espaciado. Data augmentation. Apex

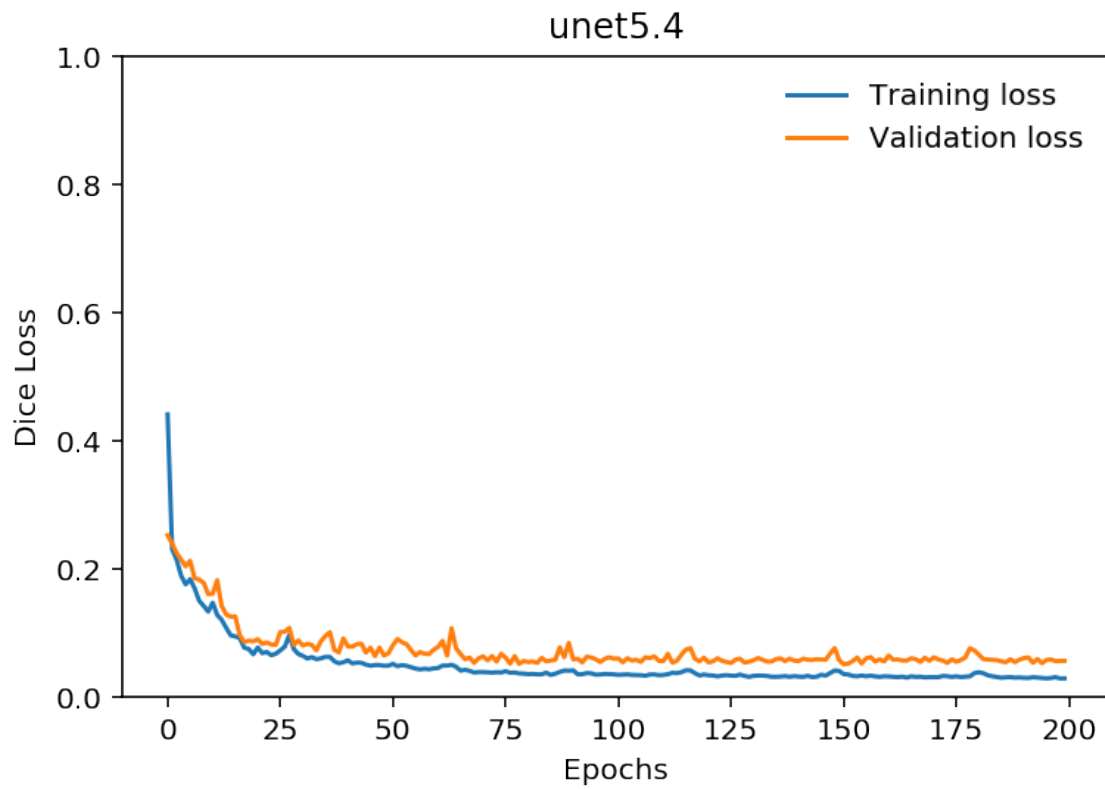


Figura 6.9: U-Net completa. Sin espaciado. Data augmentation. Apex

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

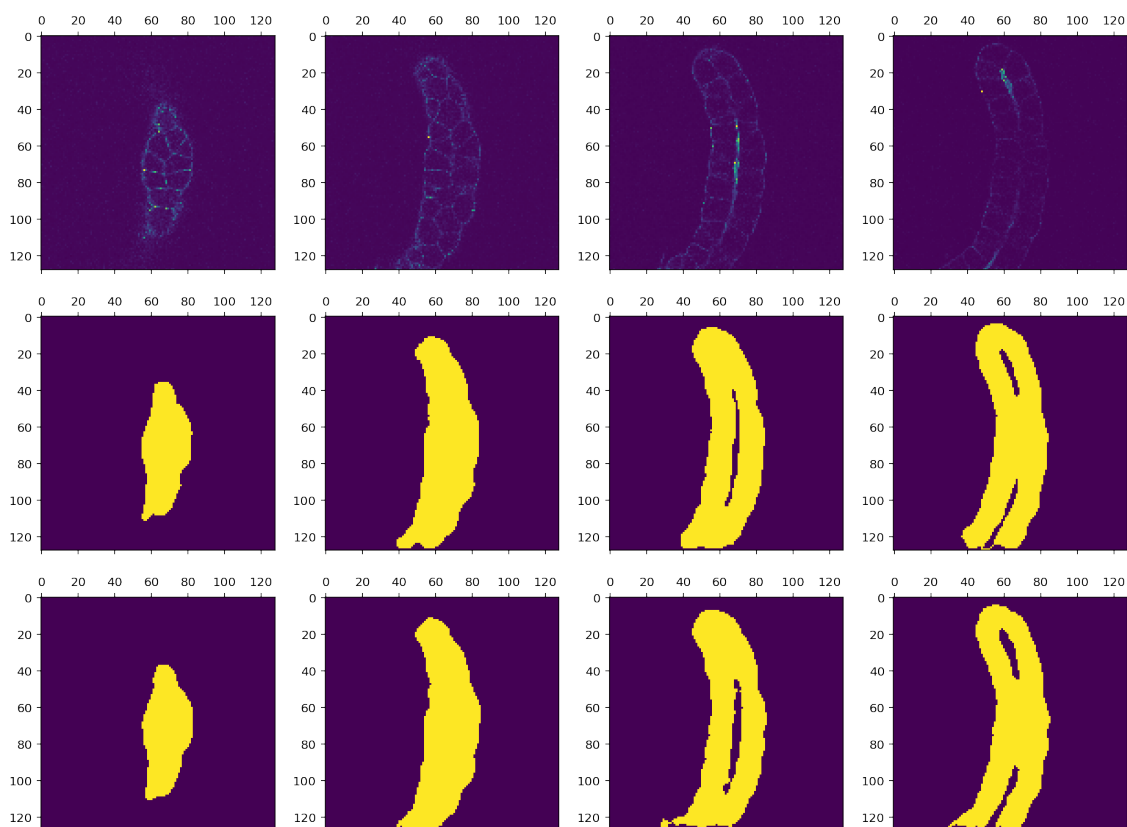


Figura 6.11: U-Net completa. Sin espaciado. Data augmentation. Apex

Referencias

- 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
- 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 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>
- 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
- 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*.
- 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).