



ESCUELA TÉCNICA SUPERIOR DE INGENIERÍA INFORMÁTICA

Proyecto fin de Grado

Grado en Ingeniería Informática: Tecnologías Informáticas

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

**Realizado por
(ponente): Adrián López Carrillo**

**Dirigido por
María José Jiménez Rodríguez**

**Departamento
Matemática Aplicada I**

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

Sevilla, 27 de Septiembre de 2020 (v.1.1)

Resumen

[REDACTED]

Índice general

Índice general	2
Índice de cuadros	5
Índice de figuras	6

I Introducción

1 Nacimiento del proyecto	3
2 Objetivos	6
3 Planteamiento inicial	7
3.1 Plan inicial	7
3.2 Nueva metodología	7
3.3 Plan seguido	8

II Estudio previo

4 Red Neuronal Artificial	11
4.1 Introducción a Redes Neuronales Artificiales	11
4.2 Neurona artificial	12
4.2.1 Sigmoide	13
4.2.2 Unidad Lineal Rectificada (ReLU)	13
4.3 Red Neuronal Artificial	14
4.4 Descenso por Gradiente	14
5 Red Neuronal Convolucional	16
5.1 Tipos de capas	19
5.1.1 Capa convolucional	19
5.1.2 Capa Pooling	21
5.1.3 Capa ReLU	21
5.1.4 Capa FC	22
5.2 Funciones de Pérdida	22
5.2.1 Binary Cross-Entropy	22
5.2.2 Weighted Binary Cross-Entropy	22
5.2.3 Dice Loss	22

6 Arquitecturas para segmentación semántica	24
6.1 Fully Convolutional Network	26
6.2 Deconvnet	26
6.2.1 Unpooling	27
6.2.2 Deconvolución	27
6.3 U-Net	27
7 Aplicaciones recientes	29
7.1 U-Net para conteo, detección y morfometría, detección y morfometría de células. (2019)	29
7.1.1 Proyecto	29
7.1.2 Pipeline	29
7.1.3 Resultados	30
7.1.4 Software	30
7.2 Segmentación 3D precisa y versátil de tejido vegetal a resolución celular. (2020)	31
7.2.1 Proyecto	31
7.2.2 Pipeline	31
7.2.3 Resultados	32
7.2.4 Software	33
 III Desarrollo	
8 Computación en la nube	36
9 Esquema general	39
10 Sistema inicial	43
10.1 Lenguaje y framework	43
10.2 Metas y métricas	44
10.3 Arquitectura CNN	45
10.4 Tamaño de la imagen en disco	45
10.5 Tamaño de la imagen en memoria	46
10.6 Segmentación semántica multiinstancia con U-Net	47
10.7 Carga de datos	48
10.8 Guardar modelo	49
10.9 Resultados	49
11 Mejora 1: Función de pérdida Dice	51
11.1 Resultados	51
12 Mejora 2: Data augmentation	53
12.1 Resultados	54
13 Mejora 3: Arquitectura U-Net completa	56
13.1 Resultados	56
14 Mejora 4: Normalización	59
14.1 Resultados	60

15 Mejora 5: Apex, precisión mixta	62
15.1 Resultados	63

IV Cierre

16 Análisis temporal y de costes	68
16.1 Análisis temporal	68
16.2 Resumen	69
16.2.1 Costes directos	69
16.2.2 Costes indirectos	69
Referencias	71

Índice de cuadros

7.1	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.	29
7.2	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.	31
7.3	Pruebas relevantes realizadas.	33
10.1	Cuadro de métricas y metas.	44
10.2	Valores mínimo y máximo de los píxeles de las imágenes de entrada.	46
10.3	Métricas sistema inicial. En la primera fila está cómo de bien se ha predicho el fondo. En la segunda fila está cómo de bien se han predicho las células.	49
11.1	Métricas pérdida dice.	51
12.1	Métricas data augmentation.	54
13.1	Métricas.	56
14.1	Valores mínimo y máximo de los píxeles de las imágenes de entrada.	59
14.2	Métricas normalización.	60
15.1	Métricas normalización.)	63
15.2	Métricas precisión mixta O1. Se han hecho 2 pruebas con la misma configuración.	63
15.3	Métricas precisión mixta O2. Se han hecho 2 pruebas con la misma configuración.	63
16.1	Tabla de resumen de la planificación.	68
16.2	Tabla de tiempos	68
16.3	Tabla de resumen de los costes.	69

Índice de figuras

1.1	Formas geométricas prisma, tronco y escutoide	4
1.2	Segmentación de dos capas de una glándula salivar de Drosophila	5
4.1	Neurona artificial genérica	12
4.2	Neurona artificial con $\sum x_i w_i$ como función de integración.	13
4.3	Red Neuronal Artificial completamente conectada.	14
4.4	Ilustración del algoritmo descenso por gradiente.	15
5.1	Imagen como entrada en una ANN.	16
5.2	Imagen como entrada en una CNN.	17
5.3	Resultado de aplicar un kernel a una imagen.	17
5.4	Resultado de aplicar un kernel a una imagen con padding.	18
5.5	Resultado de aplicar un kernel a una imagen con padding y pasos 2 y 3 para horizontal y vertical.	18
5.6	Resultado de aplicar un filtro de reconocimiento de bordes a una imagen.	18
5.7	Ejemplo de CNN en la que se clasifica una imagen con un barco.	19
5.8	Ejemplo de capa de convolución.	20
5.9	Ejemplo de pooling.	21
6.1	Ejemplo de segmentación semántica.	25
6.2	Arquitectura de FCN32, FCN16 y FCN8.	26
6.3	Arquitectura Deconvnet.	27
6.4	Figura que ilustra las operaciones unpooling y deconvolución.	27
6.5	Arquitectura original U-Net.	28
7.1	Segmentación volumétrica.	31
7.2	Segmentación de tejido vegetal usando PlantSeg.	32
7.3	Interfaz gráfica del programa PlantSet.	33
8.1	Métricas CNN de prueba en local.	37
8.2	Métricas CNN de prueba en la nube.	38
9.1	Diseño Sistema inicial.	40
9.2	Diseño Sistema Final.	42
10.1	Arquitectura U-Net reducida.	45
10.2	Pérdida de entrenamiento y validación de sistema inicial.	50
10.3	Segmentación de prueba de sistema inicial.	50
11.1	Pérdida sistema con función de pérdida DICE.	52
11.2	Ejemplo de segmentación sistema con función de pérdida DICE.	52

12.1 Pérdida de entrenamiento sistema con pérdida DICE en 200 iteraciones.	53
12.2 Pérdida de entrenamiento y validación de sistema con data augmentation.	54
12.3 Ejemplo de segmentación sistema de con data augmentation.	55
13.1 Arquitectura U-Net completa. .	57
13.2 Pérdida de entrenamiento y validación de sistema con U-Net completa.	57
13.3 Ejemplo de segmentación de sistema con U-Net completa.	58
14.1 Histogramas de aplicar la normalización tipificada y el cambio a escala	60
14.2 Pérdida de entrenamiento y validación de sistema con normalización.	60
14.3 Ejemplo de segmentación de sistema con normalización.	61
15.1 Pérdida de entrenamiento y validación de sistema con precisión mixta tipo O1. .	64
15.2 Pérdida de entrenamiento y validación de sistema con precisión mixta tipo O2. .	64

Parte I

Introducción

CAPÍTULO 1

Nacimiento del proyecto

Uno de los mayores intereses del autor de este proyecto en el mundo del software siempre ha sido el uso de técnicas de inteligencia artificial para resolver problemas actuales.

Aprovechando que cursaba la asignatura de Procesamiento de Imágenes Digitales con la profesora María José Jiménez Rodríguez para preguntarle si sabía de algún TFG en el que estuviesen involucradas imágenes y pudiese desarrollar una solución de inteligencia artificial. La principal motivación par esto era que nunca había trabajado con imágenes digitales en el ámbito de machine learning.

María José habló sobre un proyecto muy interesante dirigido por el investigador Luis María Escudero Cuadrado del Departamento de Biología Celular de la Universidad de Sevilla. Parte de este proyecto consistía en procesar imágenes 3D de glándulas salivales tomadas por un microscopio confocal para obtener la segmentación de las células. Un punto importante era que este procesado fuese automático ya que se podría ahorrar mucho tiempo, por lo que estaban valorando el uso de técnicas de machine learning.

Tras ver que este proyecto era interesante se planeó una primera visita al Departamento de Biología Celular, siendo recibidos por Luis María Escudero, Pablo Vicente-Manuera y Pedro Gómez-Gálvez. En esta reunión se habló en más detalle sobre la investigación que realizaban y el problema que se iba a abordar en el TFG.

Estaban estudiando la arquitectura a nivel celular del tejido epitelial. El tejido epitelial es uno de los 4 tejidos animales básicos, por lo que gran parte del organismo está compuesto por éste. Aunque las células de este tejido siempre se han representado en forma de prisma (cuando las células están en el mismo plano) o en forma de tronco (cuando se produce un pliegue), han demostrado que en condiciones de curvatura adoptan forma de escudoide para estabilizar energéticamente el tejido. La forma geométrica escudoide fue descubierta por este grupo de investigación (Gómez-Gálvez y cols., 2018). En la figura 1.1 se pueden ver las forma de prisma, tronco y escudoide.

Para obtener un modelo visual del tejido epitelial empiezan utilizando un microscopio confocal del que obtienen un stack de imágenes 2D, consiguiendo así una imagen 3D. Esta imagen es procesada computacionalmente utilizando herramientas de MATLAB y Fiji (ImageJ) para conseguir una imagen con todas las células etiquetadas correctamente, proceso conocido como segmentación semántica. El resultado final sería una imagen en la que cada voxel del fondo tiene valor 0 y los voxels pertenecientes a cada célula tienen valor 1, 2, 3...n, donde n es el nº de células en la imagen. Un ejemplo de dos capas de la imagen 3D original y la imagen etiquetada puede verse en la figura 1.2

Pasar de la primera fila de la figura 1.2 a la segunda fila es un proceso que puede llevar hasta una semana ya que se quiere conseguir un etiquetado perfecto. Esto provoca que se invierta mucho tiempo en una tarea repetitiva y poco interesante y se reste tiempo de las tareas de interés científico.

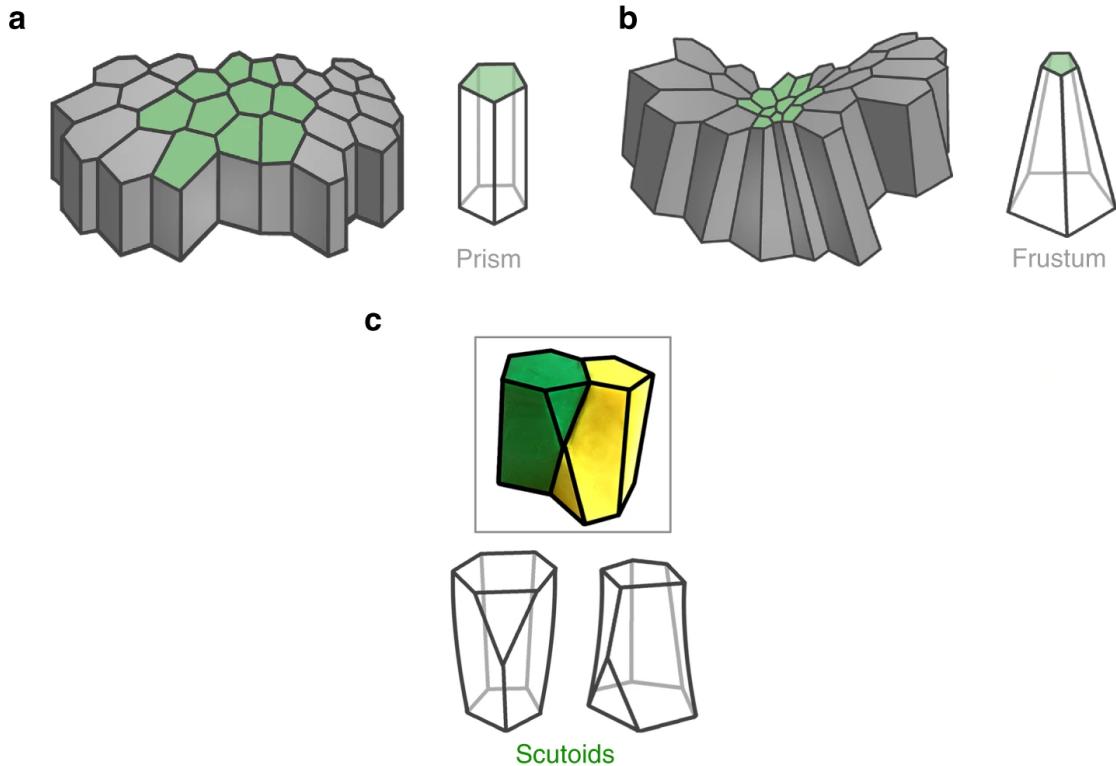


Figura 1.1: **a** Representación de las células del tejido epitelial cuando están en una estructura plana. Se suele representar en forma de prisma. **b** Representación de las células del tejido epitelial cuando se produce un pliegue. Se suele representar en forma de tronco. **c** Geometría propuesta caracterizada por tener al menos un vértice en un plano distinto al de sus dos bases. En la figura de arriba se ven dos elementos adyacentes con forma escutoide. En la figura de abajo se ven estos elementos por separado. Las células del tejido epitelial tendrían esta geometría. (Figura editada procedente de Gómez-Gálvez et. al 2018)

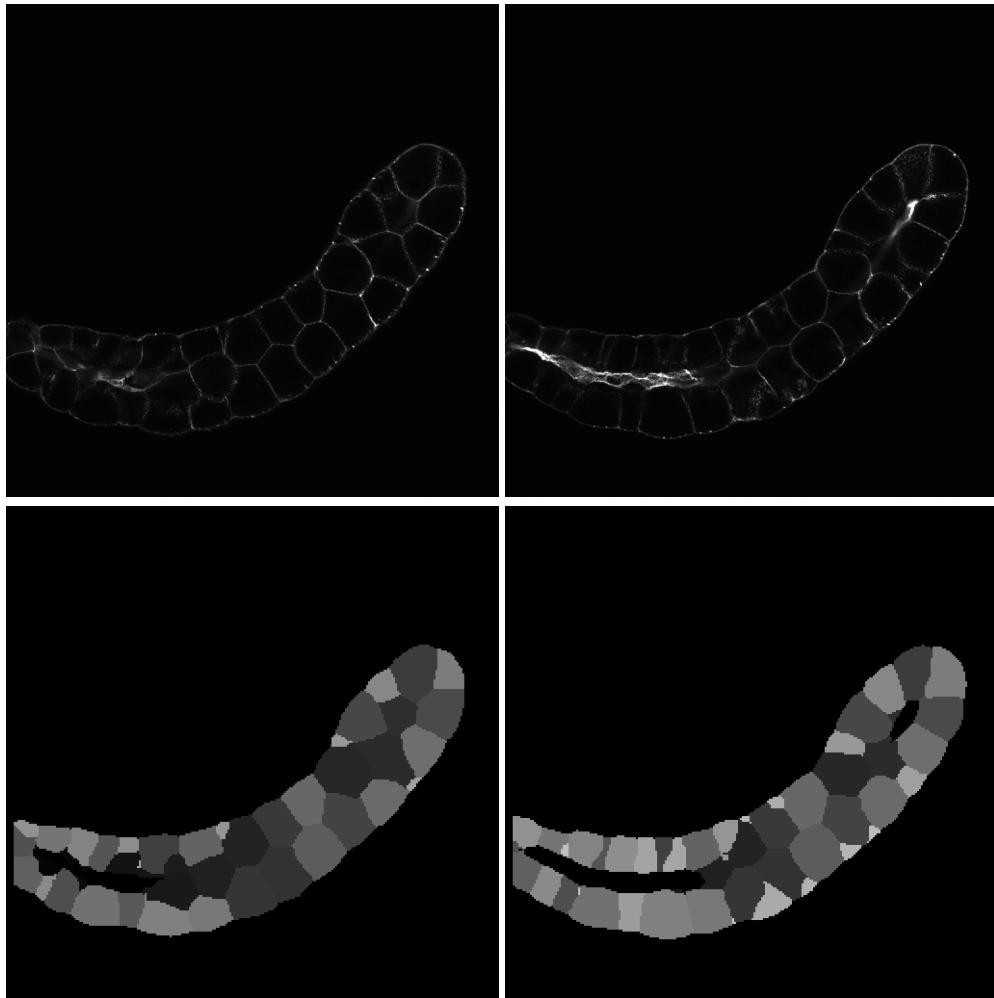


Figura 1.2: Esta figura muestra el resultado de aplicar el procesado a la imagen 3D de una glándula salivar de *Drosophila* (un género de moscas), obtenida por microscopía. La imagen tiene las dimensiones $1024 * 1024 * 234$, seleccionándose para su representación $Z=77$ y $Z=100$. En la primera fila se muestra la imagen obtenida por microscopía y la segunda fila se muestra la obtenida al aplicar segmentación a las células. En la primera columna se muestra la capa $Z=77$ y en la segunda la capa $Z=100$.

Querían reducir el tiempo necesario para la segmentación y para ello pensaron que sería buena idea recurrir a técnicas de machine learning. Sería el objetivo principal del TFG, por tanto, obtener una segmentación automática que redujera el tiempo de segmentación manual, siendo lo ideal obtener directamente la segmentación perfecta.

CAPÍTULO 2

Objetivos

Este proyecto comenzó con la idea de reducir el tiempo necesario para tener una segmentación correcta de las células, por lo que ese será el objetivo principal. Los objetivos se podrían enumerar como sigue:

1. Utilizar técnicas de machine learning para desarrollar un modelo que, a partir de una imagen 3D de tejido epitelial, calcule una segmentación de las células lo más cercano posible a una segmentación perfecta.
2. Debido a que especies con distintos fenotipos pueden presentar glándulas con distinta morfología a nivel celular, es útil la posibilidad de poder entrenar un modelo con distintos datasets.
3. Debido a la alta resolución de las imágenes, hacer que el proyecto funcione correctamente con una capacidad de computación limitada.
4. Debido a que el proyecto será utilizado por personal del ámbito científico y no tecnológico, hacer que el proyecto sea de uso e instalación sencilla.

CAPÍTULO 3

Planteamiento inicial

3.1– Plan inicial

El primer enfoque para abordar este proyecto fue utilizar herramientas actuales de machine learning para la segmentación semántica de las imágenes provistas y, a partir de los resultados obtenidos, investigar cómo mejorar la segmentación. Sin embargo esto no fue posible debido a la alta resolución de las imágenes provistas y al limitado acceso a hardware de alto rendimiento. Por ello se ideó un plan inicial en el que se construía una herramienta que tuviese menor requisito de hardware. La mayor limitación era por la VRAM necesaria, por lo que se pensó en utilizar Apache Spark (*ApacheSpark - Unified Analytics Engine for Big Data*, s.f.) para aprovechar HDD y RAM.

En este primer plan se dividieron las tareas en fases, siendo la fase 0 realmente un proceso de iniciación, necesaria para planificar el resto de las fases.

Fase 0 Estudio previo. Investigar las soluciones actuales al problema o problemas similares, las posibles técnicas de machine learning a usar en este proyecto y tecnologías para implementarlas.

Fase 1 Aprender a implementar una CNN en Python. Usar datos de prueba para familiarizarme con las tecnologías a usar en este proyecto. Se explica en más detalle por qué usar CNN en la sección 6. El uso de Python se explica en la sección 10.1.

Fase 2 Implementar, entrenar y probar una o más CNN.

Fase 3 Analizar resultados obtenidos.

Fase 4 Investigar sobre mejoras pre y post procesado.

Fase 5 Realizar estas operaciones usando Apache Spark.

Fase 6 Hacer que el entrenamiento y la inferencia sean usables por un usuario.

3.2– Nueva metodología

Gracias a la investigación inicial realizada en la **Fase 0** se pudo definir el resto de fases del plan inicial, aunque no se tuvieron en cuenta varios factores que se empezaron a ver en la **Fase 1** y **Fase 2**. Esto provocó que se tuviese que realizar un ajuste en el plan inicial. Lo que no se tuvo en cuenta fue:

- Las fases 2, 3 y 4 deben ser fases iterativas. Cada vez que se obtengan resultados hay que estudiarlos y hacer cambios para mejorar los resultados.
- No se tiene en cuenta el rendimiento de los algoritmos utilizados ni los cuellos de botella.
- Se quiere usar Spark con la premisa de aprovechar la RAM o el HDD cuando la VRAM se agote, pero esto no es viable ya en la VRAM se tiene que poder almacenar toda la información necesaria para ejecutar cada algoritmo.

Por todo ello, se buscó información sobre una metodología adecuada. El capítulo 11 del libro Deep Learning (Goodfellow, Bengio, y Courville, 2016) habla sobre una metodología práctica para la aplicación de técnicas en deep learning. Esta metodología se centra en utilizar feedback del sistema construido en cada iteración para modificar el sistema acorde a los objetivos. Los pasos a seguir propuestos por esta metodología son:

1. Determinar las metas. Qué métrica usar como error y un valor de la métrica aceptable como objetivo.
2. Establecer lo antes posible un sistema completo que pueda dar un valor para dicha métrica.
3. Utilizar herramientas adecuadas para determinar cuellos de botella en el rendimiento. Comprobar qué componentes están dando peores resultados en el sistema y comprobar si los resultados negativos son debidos a overfitting, underfitting, a un fallo con los datos o en el software.
4. Hacer cambios de forma a los datos, ajuste de hiperparámetros o cambiar algoritmos con el objetivo de mejorar la métrica usada. Volver al punto 3.

Estos pasos se resumirán como “desarrollo del sistema” constituirá la mayor parte del proyecto.

3.3– Plan seguido

Fase 0 Estudio previo. En esta fase se investigará sobre artículos de segmentación celular y se aprenderá sobre machine learning y las tecnologías necesarias para llevar a cabo resultados similares aplicados al problema actual.

Fase 1 Desarrollo del sistema. Aplicar la metodología de la sección 3.2.

1. Determinar metas con métricas.
2. Sistema inicial.
3. Usar herramientas para determinar cuellos de botella en el rendimiento.
4. Cambios en el sistema.
5. Si no se ha alcanzado el valor objetivo de la métrica, volver al punto 3.

Fase 2 Mejoras de usabilidad. Que el proyecto sea usable por un usuario.

Fase 3 Cierre. Terminar de escribir la memoria con los resultados obtenidos, las conclusiones, tabla de tiempos y manual de usuario.

Parte II

Estudio previo

CAPÍTULO 4

Red Neuronal Artificial

En este capítulo se hará un recorrido histórico haciendo énfasis en los avances del siglo XX en redes neuronales artificiales (ANNs). Después se definirá la neurona artificial y se distinguirán 3 tipos según su función de activación (visto en más detalle en la sección 4.2): perceptrón, sigmoide y unidad lineal rectificada (ReLU). Siendo la ReLU la función de activación más usada en redes neuronales artificiales (Ramachandran, Zoph, y Le, 2017) y en las soluciones de este proyecto. Por último se hablará sobre el entrenamiento de la ANN.

4.1– 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 y cols., 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 hi-

cieron 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** (o deep learning), 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).

Es en esta etapa de deep learning en la que se enmarca el desarrollo de este trabajo.

4.2– Neurona artificial

En la figura 4.1 se puede ver una neurona artificial genérica con la que pueden ser descritos los distintos tipos que se verán en esta sección.

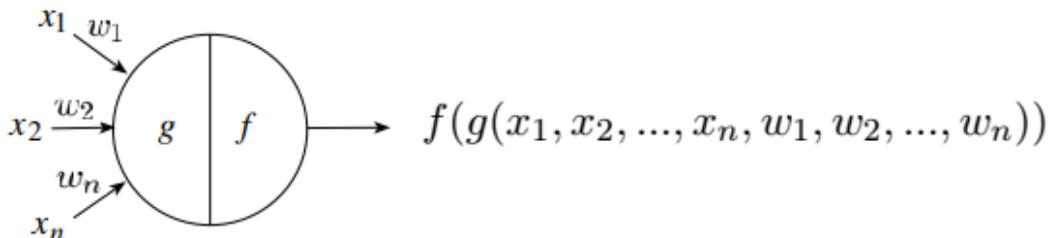


Figura 4.1: Neurona artificial genérica

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 4.1 en la que f indicará la función de activación.

- Al vector de entradas se le añade un elemento de valor constante 1, siendo ahora de tamaño $n + 1$.

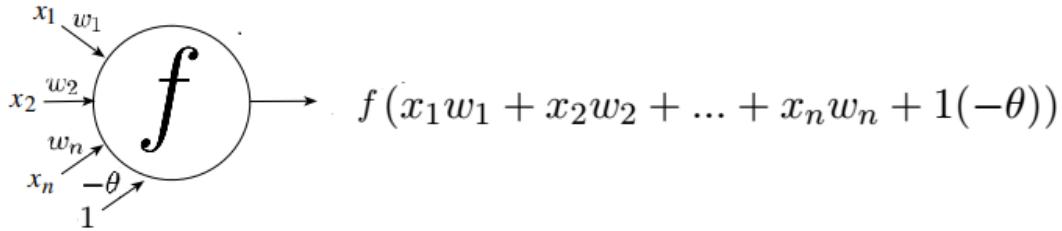


Figura 4.2: Representación de una Neurona Artificial con $\sum x_i w_i$ como función de integración

- 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.

4.2.1. Sigmoid

La función sigmoide como función de activación es una función no lineal usada principalmente en redes neuronales prealimentadas (feedforward neural networks), que son las redes neuronales donde las conexiones entre neuronas no forman ciclos (más información en la sección 4.3). 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):

$$f(x) = \sigma(x) = \frac{1}{1 + e^{-x}} \quad (4.1)$$

4.2.2. 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 Sigmoid, ofrece un mejor rendimiento y es más generalista (Nwankpa y cols., 2018).

$$f(x) = \max(0, x) = \begin{cases} x & \text{si } x \geq 0 \\ 0 & \text{si } x < 0 \end{cases} \quad (4.2)$$

4.3– Red Neuronal Artificial

Considerando la neurona artificial como una unidad de computación básica, según el conexionismo (que más tarde evolucionó en lo que hoy se conoce 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).

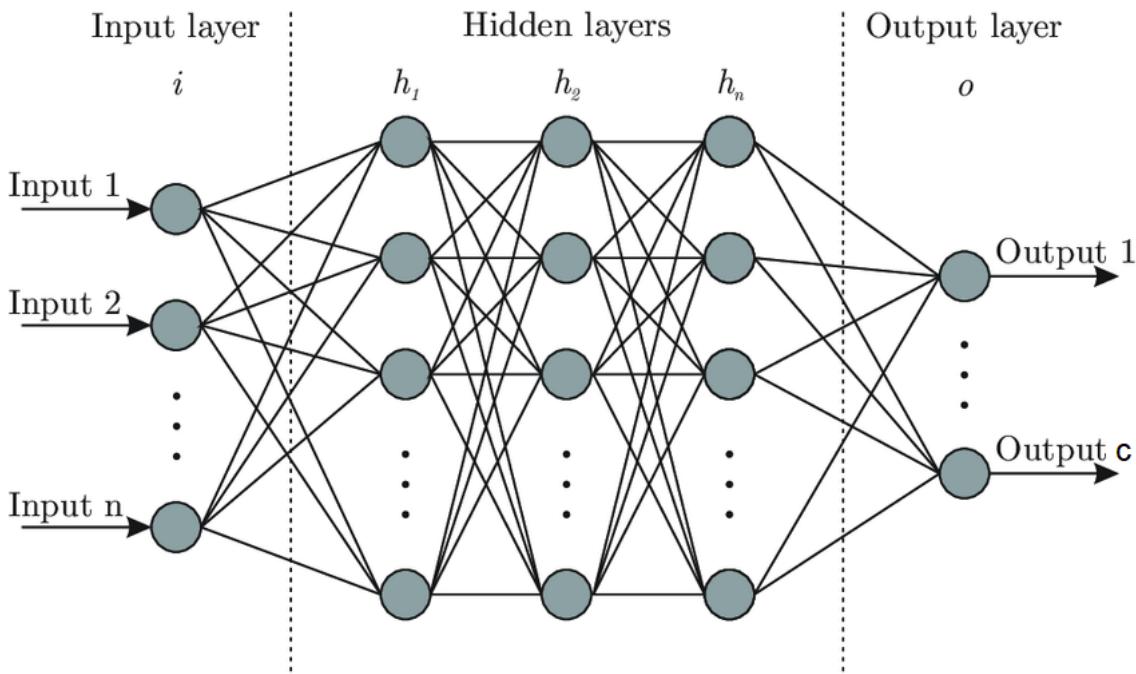


Figura 4.3: Red Neuronal Artificial completamente conectada.

En la figura 4.3 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.
- Capas ocultas (h_1, h_2, \dots, 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 dará un vector de tamaño c como salida.

4.4– Descenso por Gradiente

En cálculo de varias variables calcular el gradiente de una función en un punto ($\nabla f(x)$) dará como resultado un vector indicando la dirección en la que esa función tiene una mayor variación, siendo el módulo el ritmo de la variación. Con $\nabla f(x)$ (positivo) se obtiene un vector indicando la dirección y sentido donde $f(x)$ se acerca a un máximo local. De forma equivalente,

$-\nabla f(x)$ (negativo) se obtiene un vector indicando la dirección y sentido donde $f(x)$ se acerca a un mínimo local.

Si la función f calcula el error cometido en un procedimiento y es una función definida y diferenciable alrededor de un punto a , entonces $-\nabla f(a)$ calcula la dirección y sentido que, desde a , se debe tomar para conseguir el mayor decrecimiento. Calcular el mayor decrecimiento en un punto de una función que calcule un error cometido significará que el error disminuirá respecto al obtenido en ese punto, acercándose a un mínimo local. Este procedimiento está descrito por la ecuación 4.3, dónde η es un real positivo llamado factor de aprendizaje o **learning rate**.

$$a_{n+1} = a_n - \eta \nabla f(a_n) \quad (4.3)$$

En el contexto actual, esta función se llamará **función de pérdida** (*loss function*). La función de pérdida devuelve un valor real indicando cuánto se aleja el resultado obtenido del deseado, tomando como entrada la salida predicha de un modelo (segmentación predicha) y la salida ideal que podría haber obtenido (segmentación perfecta). Se verá con más detalle en 5.2.

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. Por lo tanto 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

$$w_{ij} = w_{ij} - \eta \nabla f(a) \quad (4.4)$$

Donde w_{ij} es el peso desde el nodo i al nodo j .

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 4.4 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.

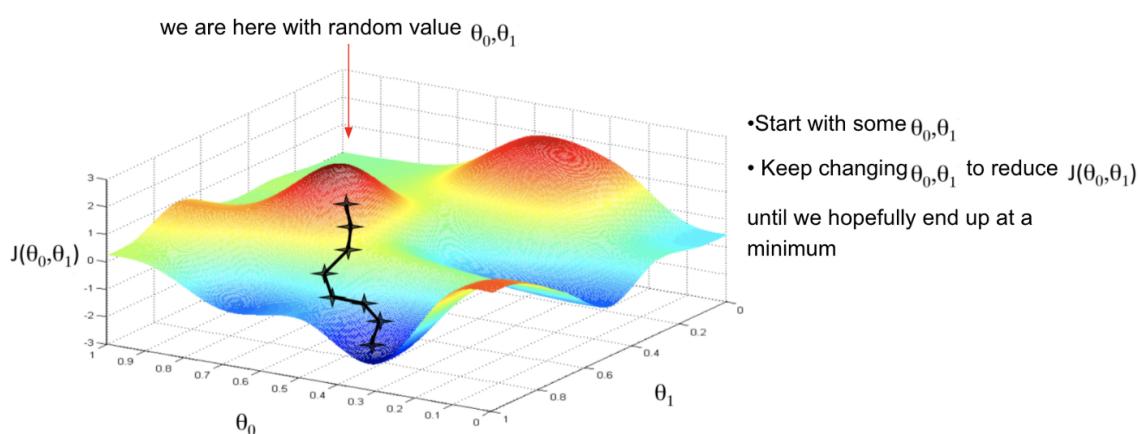


Figura 4.4: Ilustración del algoritmo de descenso por gradiente. θ_0 y θ_1 son los pesos de la ANN y J la función de pérdida. Se puede observar cómo se produce un "descenso" hacia un mínimo de la función.

CAPÍTULO 5

Red Neuronal Convolucional

En este capítulo se estudiarán los fundamentos de la red neuronal convolucional (CNN), un subtipo de ANN especializado en el tratamiento de imágenes y usado por aplicaciones actuales de segmentación semántica 7.

Hasta ahora se ha 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 voxel) 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 5.1) 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 $16 * 4 = 64$ pesos. Si aplicáramos este sistema a una imagen 3D en escala de grises con $124 * 124 * 70 = 1076320 \text{ vx}$, 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 este capítulo 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.

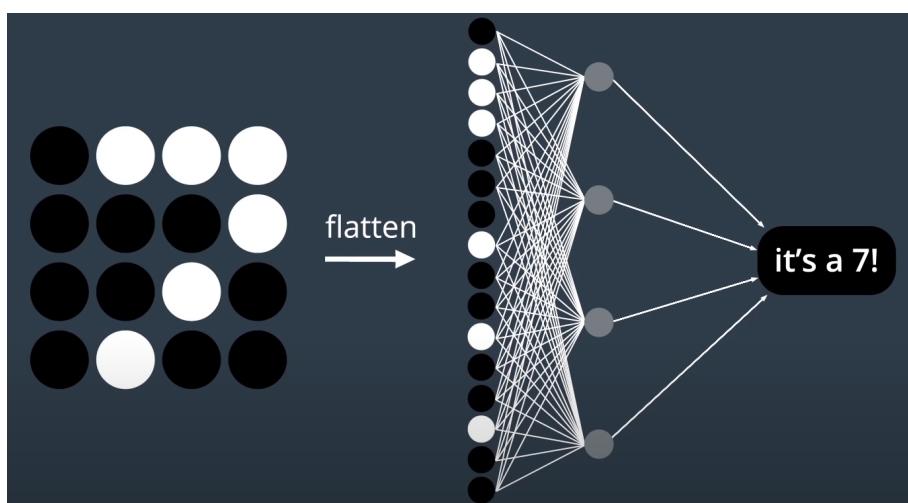


Figura 5.1: Imagen de $4 \times 4 \text{ px}$ aplanada y usada como entrada en una FCNN con 4 neuronas en su única capa oculta. No se muestra su capa de salida. Imagen del curso Intro to Deep Learning with PyTorch de Udacity.

Cambiando la arquitectura de la red vista en la figura 5.1 por una CNN, obtendríamos una arquitectura similar a la vista en la figura 5.2. 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 se necesitarán 4 pesos distintos.

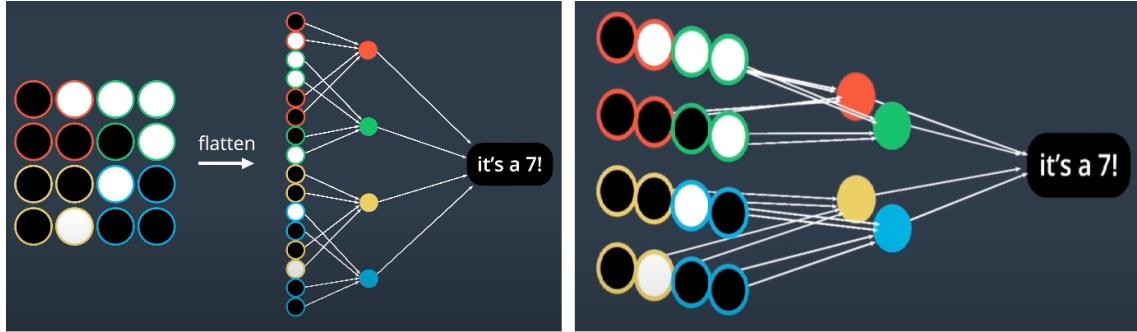


Figura 5.2: Imagen de 4x4 px usada como entrada en una CNN. Ambas figuras muestran la misma arquitectura, estando en la figura de la izquierda la imagen aplanada y en la figura de la derecha se muestra la matriz 4x4 como entrada. Imágenes del curso Intro to Deep Learning with PyTorch de Udacity.

Conceptos de procesamiento de imagen

Antes de describir los tipos de capas será necesario introducir varios conceptos propios del procesamiento de imagen que son usados en la capa convolucional (subsección 5.1.1) y la capa pooling (subsección 5.1.2)

Lo primero es saber que una convolución es una operación matemática entre dos funciones (f y g) que produce una tercera función ($f * g$) que expresa cómo la forma de f es cambiada por g . (*Convolution*, 2020)

En este caso f será la imagen original y g será una matriz con el mismo número de dimensiones que la imagen. Para facilitar esta explicación se asumirá que la imagen original tiene dos dimensiones, aunque en este proyecto se trabaje con imágenes de tres dimensiones.

Siguiendo este esquema, a g se le conocerá como **kernel** y definirá la naturaleza de la operación realizada a lo largo de f . Si se tiene un kernel X de tamaño (m, m) y una imagen Y , esta operación respecto al punto (y_1, y_2) de Y puede definirse cómo la ecuación 5.1, con un ejemplo visual en la figura 5.3:

$$\sum_{k=0}^{m-1} \sum_{l=0}^{m-1} X_{j,k} * Y_{y_1-i-[l/2], y_2-j-[l/2]} \quad (5.1)$$

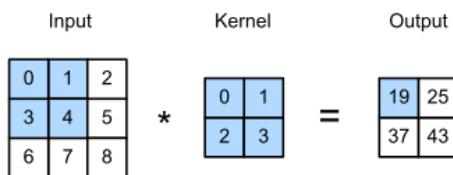


Figura 5.3: Kernel 2x2 aplicado al elemento (0, 0) de una matriz 3x3. Imagen tomada de d2l.ai.

En la figura 5.3 se puede observar un problema al intentar aplicar el kernel al punto (0, 2). El kernel “se sale” de la imagen, por lo que no se puede computar esta operación y se pierde información. Para solucionar esto se aplica lo que se conoce como **padding**.

El padding es una técnica que consiste en añadir píxeles de relleno alrededor de una imagen. En este contexto se añade 0 para evitar alterar los resultados. Un ejemplo de padding puede verse en la figura 5.4

Input	Kernel	Output
$\begin{bmatrix} 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & 2 & 0 \\ 0 & 3 & 4 & 5 & 0 \\ 0 & 6 & 7 & 8 & 0 \\ 0 & 0 & 0 & 0 & 0 \end{bmatrix}$	$\begin{bmatrix} 0 & 1 \\ 2 & 3 \end{bmatrix}$	$=$ $\begin{bmatrix} 0 & 3 & 8 & 4 \\ 9 & 19 & 25 & 10 \\ 21 & 37 & 43 & 16 \\ 6 & 7 & 8 & 0 \end{bmatrix}$

Figura 5.4: Kernel 2x2 aplicado al elemento (0,0) de una matriz 3x3 con padding 1 aplicado. Imagen tomada de d2l.ai.

Esta operación se debe realizar para toda la imagen de entrada. Si se quiere realizar esta operación para todos los puntos de la imagen, el kernel debe “deslizarse” sobre la imagen elemento a elemento, tanto en horizontal como en vertical. En este caso se dirá que se tiene un **paso** de 1 horizontal y vertical. En la imagen 5.5 se puede ver un ejemplo con un paso horizontal de 2 y vertical de 3. De aquí en adelante se supondrá que el paso horizontal y vertical tienen el mismo valor, por lo que el paso estará definido por un solo número.

Input	Kernel	Output
$\begin{bmatrix} 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & 2 & 0 \\ 0 & 3 & 4 & 5 & 0 \\ 0 & 6 & 7 & 8 & 0 \\ 0 & 0 & 0 & 0 & 0 \end{bmatrix}$	$\begin{bmatrix} 0 & 1 \\ 2 & 3 \end{bmatrix}$	$=$ $\begin{bmatrix} 0 & 8 \\ 6 & 8 \end{bmatrix}$

Figura 5.5: Kernel 2x2 aplicado a una matriz 3x3 con padding 1, paso horizontal 2 y vertical 3. Imagen tomada de d2l.ai.

Al determinar el kernel, paso y padding se definirá un **filtro**. Los filtros son de gran interés en el procesamiento de imagen ya que al aplicarlos se pueden producir efectos como enfoque, realce o detección de bordes. En la figura 5.6 se muestra un ejemplo de aplicar un filtro a una imagen de entrada.

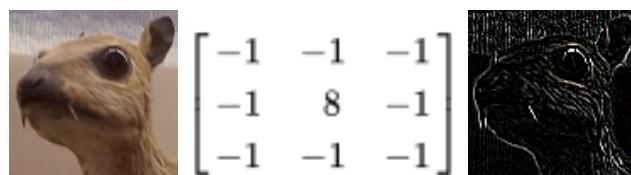


Figura 5.6: Efecto producido al aplicar un filtro de detección de bordes a una imagen de entrada. Imágenes tomadas de la página “Núcleo (procesamiento digital de imágenes)” de Wikipedia.

5.1– Tipos de capas

Antes de describir los tipos de capa usados para construir 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.

A continuación se describirán las capas más comunes usadas en una CNN: Capa Convolutacional, Capa de Pooling, Capa ReLU y Capa Completamente Conectada (FC). En la figura 5.7 (missinglink.ai, 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.

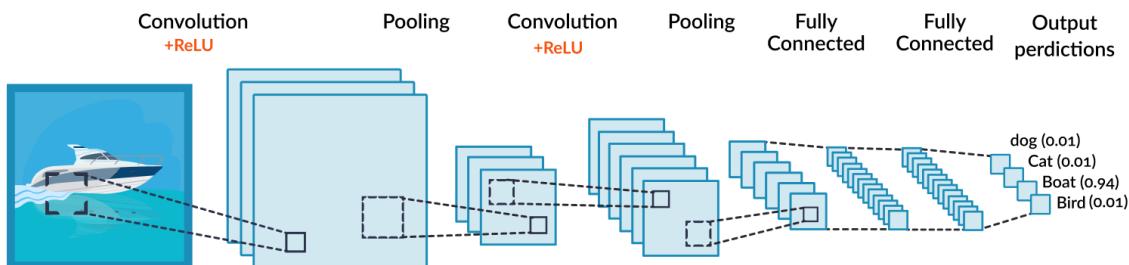


Figura 5.7: Red Neuronal Convolucional simple en la que la imagen de un barco es clasificada.

5.1.1. Capa convolucional

Es la capa principal de este tipo de redes, es donde se aplican los filtros a las imágenes usando convoluciones. 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 \mathbf{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 arquitecturas 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 5.8(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 la salida tenga profundidad 2 será necesario usar 2 filtros. Cada uno de estos filtros tendrá un kernel de tamaño $3 \times 3 \times n$, siendo $n = 3$ la profundidad de la imagen de entrada a esta capa. Cada kernel 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.

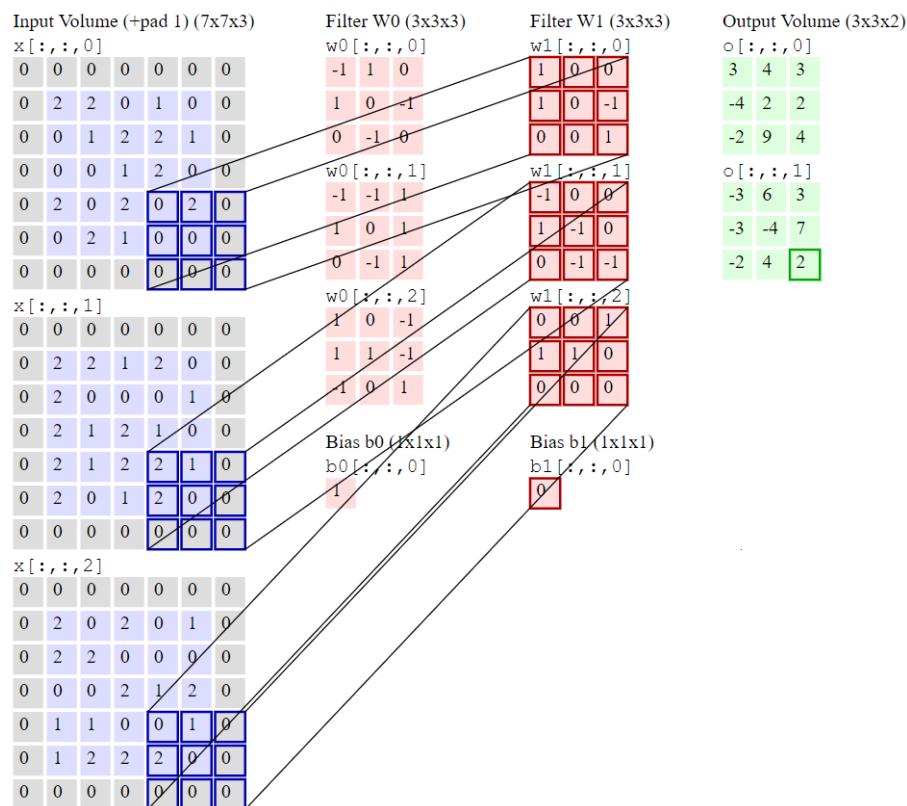


Figura 5.8: Capa de convolución de profundidad 2 (se usan dos filtros) con kernel de tamaño $3 \times 3 \times 3$ aplicado a una imagen de entrada de tamaño 5×5 con profundidad 3 y padding 1. El resultado es un volumen de $3x3x2$.

5.1.2. 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 solo 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 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:

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

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.

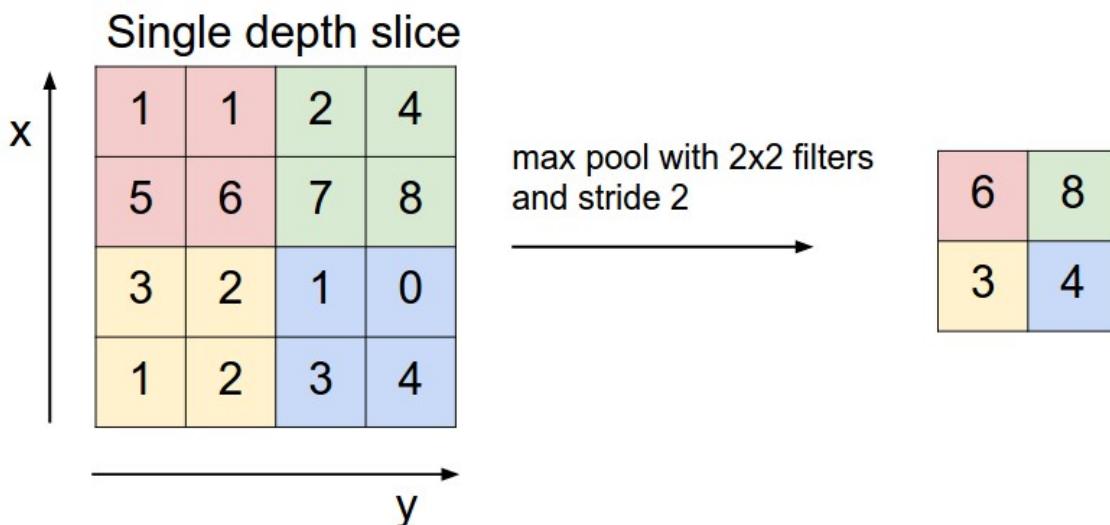


Figura 5.9: Imagen 4×4 a la que se ha aplicado un pooling de $F = 2, S = 2$. Cada color de la imagen de la izquierda indica las entradas del para la operación MAX, cada color de la imagen de la derecha indica la salida de dicha operación. Figura tomada del curso “CS231n”de Standford University.

5.1.3. Capa ReLU

Esta capa aplica la función de activación no lineal *ReLU* (vista en el apartado 4.2.2) a cada elemento (píxel o voxel) 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).

5.1.4. 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.

5.2– Funciones de Pérdida

En esta sección se describirán varias funciones de pérdida que se tuvieron en cuenta en el capítulo 11, seleccionando finalmente la entropía cruzada binaria.

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. En este proyecto el objetivo viene dado por la segmentación perfecta realizada de forma manual a la que se llamará y . Esta segmentación perfecta se comparará con la predicción obtenida (\hat{y}) por el modelo CNN al tomar como dato de entrada la imagen original sin procesar. Como se vio en la sección del descenso por gradiente (4.4), se usará una función que tome como entrada la segmentación perfecta y la predicción y devuelva un valor numérico indicando cuánto se aleja la predicción del objetivo. Esta función será la función de pérdida. A continuación se describirán varias funciones de pérdida en relación con el problema de segmentación semántica.

5.2.1. Binary Cross-Entropy

La entropía cruzada binaria es una medida utilizada para calcular la diferencia entre dos distribuciones de probabilidad. (Jadon, 2020). Resultará útil si comparamos el objetivo y con la predicción \hat{y} . La fórmula de la entropía cruzada binaria (BCE) es la siguiente:

$$L_{BCE}(y, \hat{y}) = -(y \log(\hat{y}) + (1 - y) \log(1 - \hat{y})) \quad (5.2)$$

5.2.2. Weighted Binary Cross-Entropy

La entropía cruzada 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:

$$L_{WBCE}(y, \hat{y}) = -(\beta y \log(\hat{y}) + (1 - y) \log(1 - \hat{y})) \quad (5.3)$$

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

5.2.3. 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:

$$DL(y, \hat{p}) = 1 - \frac{2y\hat{p} + 1}{y + \hat{p} + 1} \quad (5.4)$$

Siendo $\hat{p} \in [0, 1]$ la probabilidad de que un píxel/vóxel pertenezca a una clase, dada por la predicción del modelo, 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.

CAPÍTULO 6

Arquitecturas para segmentación semántica

En este capítulo se verán varias arquitecturas que popularizaron técnicas importantes en deep learning y redes convolucionales en concreto. El objetivo será llegar a la arquitectura U-Net, ya que será la utilizada durante este proyecto, que será implementada en el capítulo 13.

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. En la figura 6.1 (Jeong, Yoon, y Park, 2018) se muestra un ejemplo de segmentación semántica.

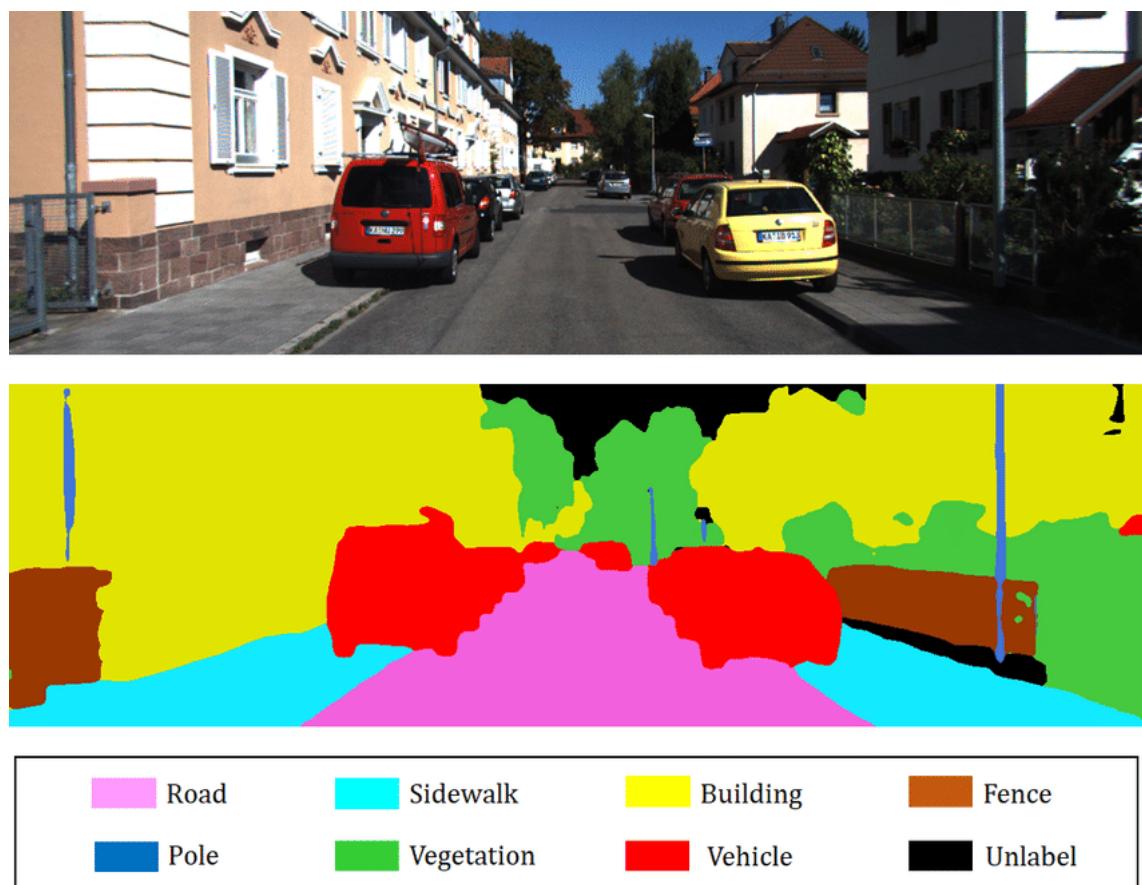


Figura 6.1: Ejemplo de segmentación semántica. Se clasifica cada píxel en función del tipo de objeto (clase) al que pertenezca.(Imagen tomada de Jeong, Yoon, y Park, 2018)

6.1– Fully Convolutional Network

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

Se compararon los modelos FCN-AlexNet, FCN-VGG16 y FCN-GoogLeNet, obteniendo los mejores resultados con FCN-VGG16 y convirtiéndose este en el modelo base. Con el objetivo de ganar más nivel de detalle en la salida, se aumentó la resolución de la salida en 32x usando interpolación bilineal. También se usaron *skip connections* para conectar varias capas con la capa final. En la figura 6.2 se puede ver un ejemplo de la arquitectura FCN (Sultana, Sufian, y Dutta, 2020).

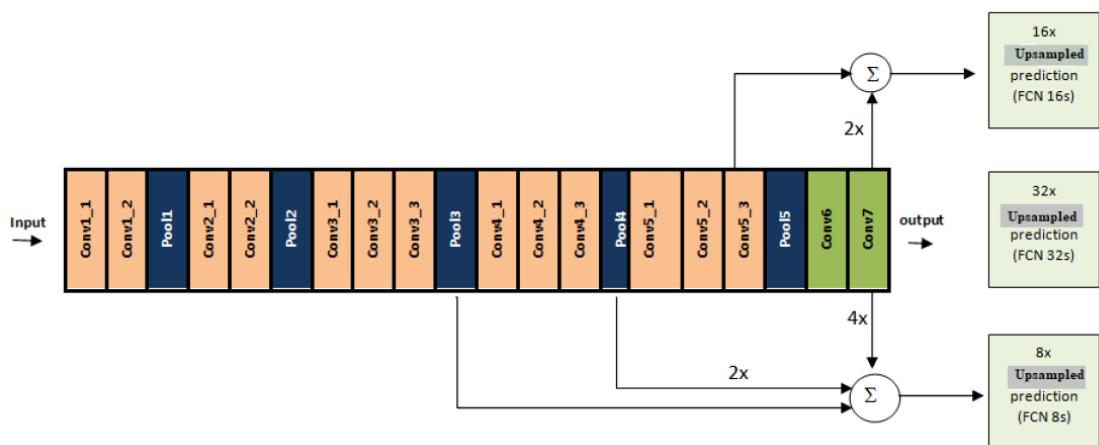


Figura 6.2: Arquitectura de FCN32, FCN16 y FCN8.

6.2– 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 6.3 (Noh y cols., 2015).

A continuación se describen brevemente las operaciones de unpooling y deconvolución (más correctamente llamada convolución transpuesta). En la figura 6.4 (Noh y cols., 2015) se puede ver un ejemplo gráfico.

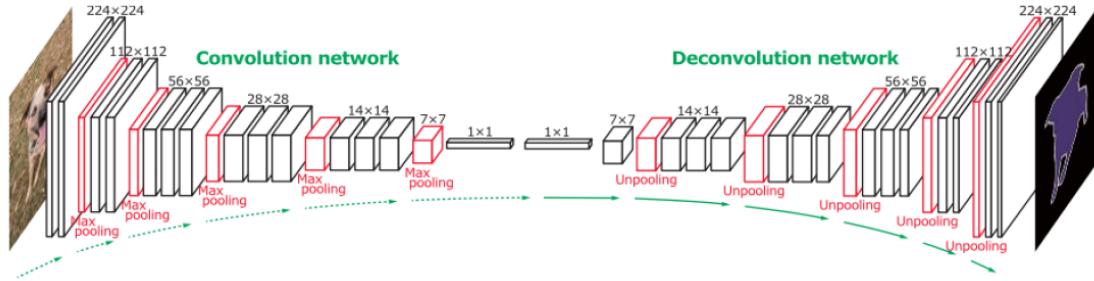


Figura 6.3: Arquitectura Deconvnet.

6.2.1. 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 6.4). 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).

6.2.2. 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 desconvolución (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.

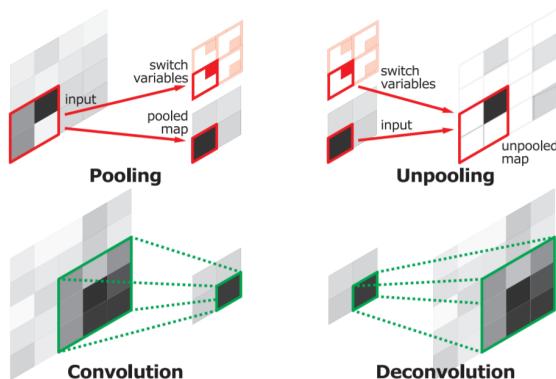


Figura 6.4: Figura que ilustra las operaciones unpooling y deconvolución.

6.3– 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 la arquitectura ganadora en 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, 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 sustituida 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 nº correspondiente de clases. En la arquitectura original se tienen en total 23 capas convolucionales. En la figura 6.5 puede verse la arquitectura original.

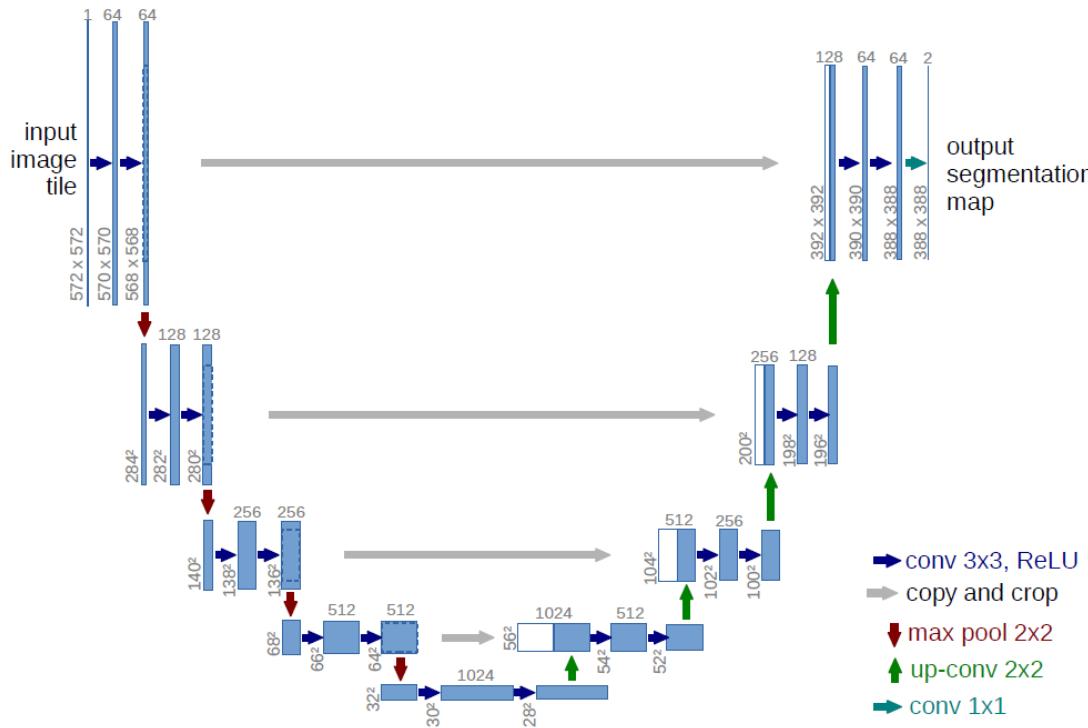


Figura 6.5: Arquitectura original U-Net propuesta por Ronneberger et al en U-Net: Convolutional Networks for Biomedical Image Segmentation.

CAPÍTULO 7

Aplicaciones recientes

En este capítulo se describirán dos aplicaciones recientes en las que se realiza segmentación celular usando la arquitectura U-Net. Estas aplicaciones fueron la principal fuente de información para el desarrollo del proyecto descrito en esta memoria.

Primero se describirá cada aplicación en general, luego se definirá brevemente el algoritmo de segmentación seguido dividiendo las operaciones en: preprocessado (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.

7.1– U-Net para conteo, detección y morfometría, detección y morfometría de células. (2019)

7.1.1. 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.

7.1.2. Pipeline

Preprocesado	Procesado	Postprocesado
GT:Espaciado de 1 voxel 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 espacio entre células Optimizador ADAM: -Learning rate 10^{-5} $\beta_1 : 0,9, \beta_2 : 0,999$ 150000 Iteraciones	Ninguno

Cuadro 7.1: 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:

$$w(x) := w'_{bal} + \lambda w_{sep} \quad (7.1)$$

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

$$w'_{bal}(x) := \begin{cases} 1 & y(x) > 0 \\ v_{bal} + (1 - v_{bal}) * \exp(-\frac{d_1^2(x)}{2\sigma_{bal}^2}) & y(x) = 0 \\ 0 & y(x) \text{desconocido} \end{cases} \quad (7.2)$$

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.

$$w_{sep}(x) := \exp(-\frac{(d_1(x) + d_2(x))^2}{2\sigma_{sep}^2}) \quad (7.3)$$

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

7.1.3. Resultados

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

$$M_{IoU}(A, B) := \frac{|A \cap B|}{|A \cup B|} \quad (7.4)$$

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.

7.1.4. 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/resourcesopensource/unet>

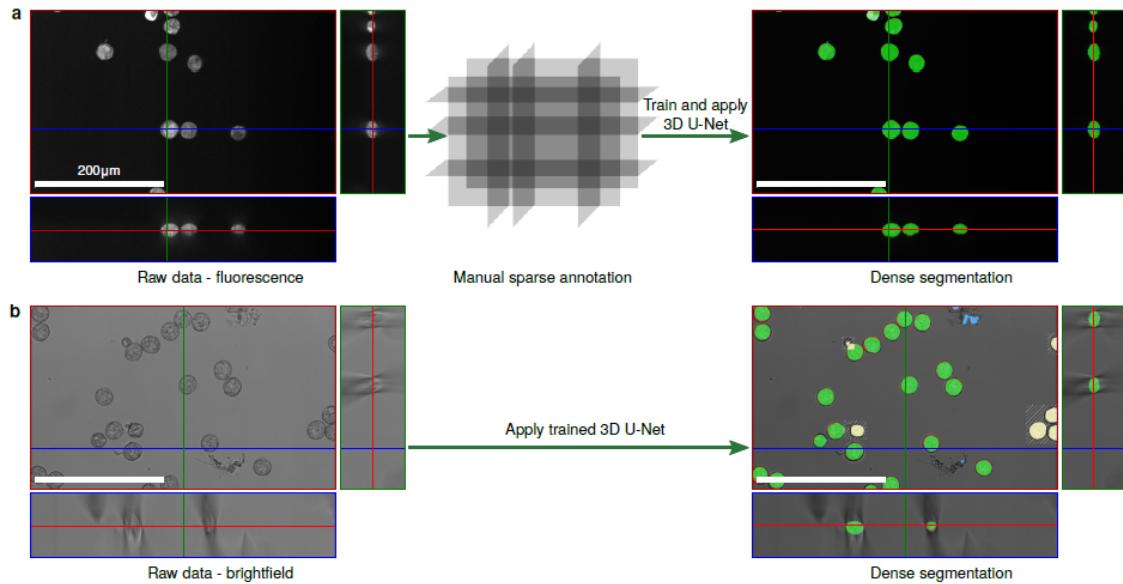


Figura 7.1: Segmentación volumétrica. (a) Segmentación de una imagen con un modelo que se ha entrenado con imágenes del mismo dataset. (b) Segmentación de una imagen con un modelo entrenado con un dataset distinto.

7.2– Segmentación 3D precisa y versátil de tejido vegetal a resolución celular. (2020)

7.2.1. 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.

7.2.2. Pipeline

Preprocesado	Procesado	Postprocesado
GT:Bordes con 2 véxeles 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 7.2: 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 preprocessado, se parte de una imagen con etiquetado perfecto y se le aplica la función *find_boundaries* de la librería scikit (Pedregosa y cols., 2011) (van der Walt y cols., 2014) obte-

niendo los bordes de las células con 2 véxeles 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 véxel 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 véxel de fondo un valor igual al véxel 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 véxeles 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 véxel etiquetado se insertan en una cola prioritaria teniendo más prioridad aquellos con un valor más bajo.
3. El véxel 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.

7.2.3. Resultados

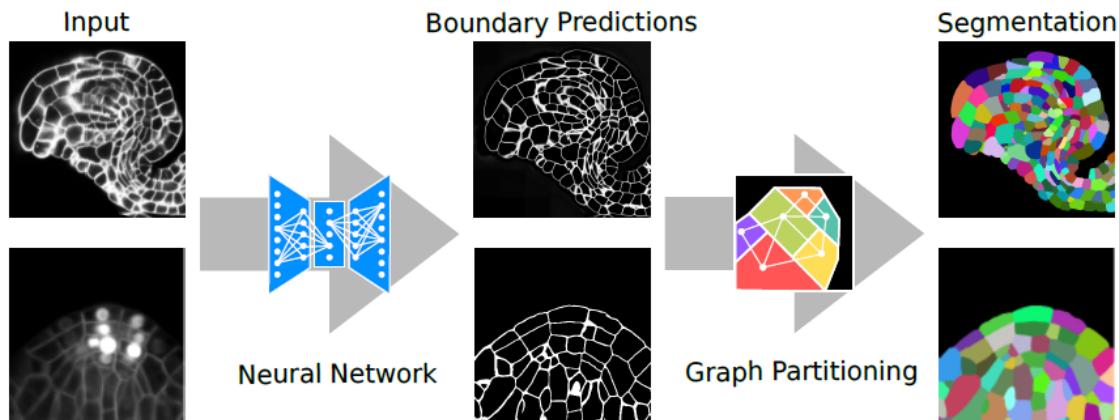


Figura 7.2: Segmentación de tejido vegetal usando PlantSeg. En el primer paso se predicen los bordes de las células usando una red U-Net 3D. En el segundo paso se aplica un algoritmo de particionamiento de grafo para segmentar cada célula.

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

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

$$VOI = H(seg|GT) + H(GT|seg) \quad (7.5)$$

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 7.3 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 7.3: Pruebas relevantes realizadas.

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

7.2.4. 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>.

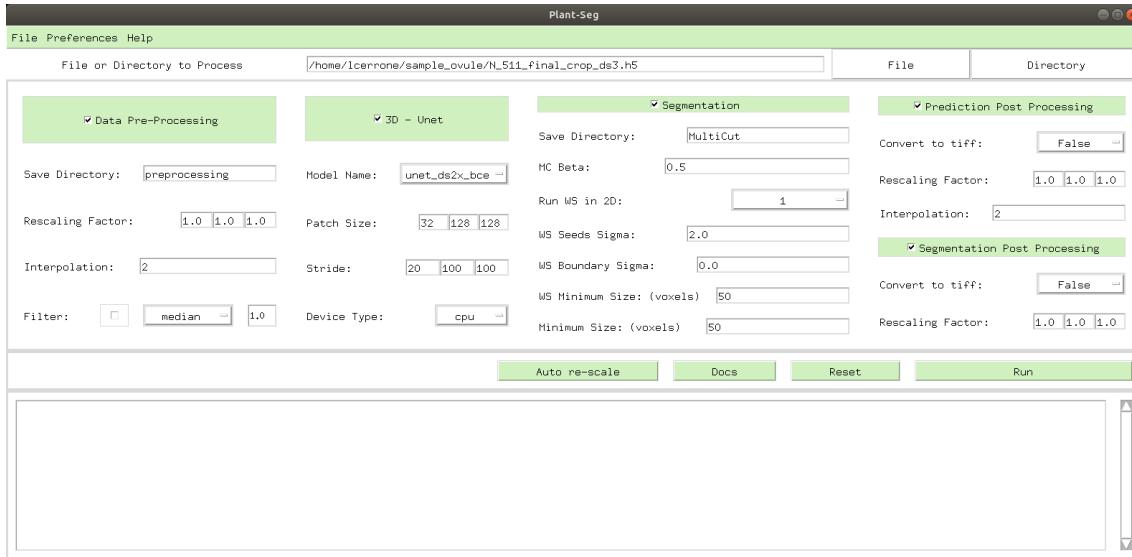


Figura 7.3: Interfaz gráfica del programa PlantSet. Se pueden ver las distintas opciones para cada paso del procesado.

Parte III

Desarrollo

CAPÍTULO 8

Computación en la nube

En este capítulo se mostrará el problema encontrado al intentar desarrollar en local con un hardware limitado. Se pasará de probar en local un ejemplo de fastai a probarlo en un ordenador en la nube y se compararán los resultados.

Después de investigar sobre las soluciones actuales (Capítulo 7) e intentar utilizarlas se llega a la conclusión de que dichas soluciones están pensadas para un hardware superior al que hay disponible para este proyecto. Por lo tanto se van a implementar redes neuronales convolucionales de forma manual. Lo más probable es que no se obtengan resultados tan buenos que los que obtendría por las soluciones vistas en el estudio previo, pero se tendrá libertad a la hora de decidir la complejidad del modelo y otros factores, lo que reducirá el coste de entrenarlo y usarlo.

Se empezará por seguir los pasos del “Getting started” del framework fastai (Howard y cols., 2018) de Python. Este framework es una capa de abstracción por encima de PyTorch (Paszke y cols., 2019). Abstiene muchos elementos del programador, haciendo que probar distintas arquitecturas sea rápido, además tiene implementado mejoras de optimización a bajo nivel. También es importante tener en cuenta que este framework tiene implementado la arquitectura u-unet (y otras de procesamiento de imágenes), arquitectura vista en el estudio previo como el estado del arte en segmentación semántica.

Computación local

Se ha seguido un tutorial en el que se usa una u-net para segmentar imágenes 2D. El tutorial se puede encontrar en el siguiente notebook:

<https://www.kaggle.com/dipam7/image-segmentation-using-fastai>

Se han tenido que modificar parámetros para poder ejecutar el notebook en la máquina local, que tiene una GPU con 2 GB de VRAM. Los parámetros cambiados son:

- *image_size* = $(720, 960)/8 = (90, 120)$ En el tutorial ya se reduce el tamaño de cada dimensión a la mitad, pero ha sido necesario reducirlo a una octava parte para que el entrenamiento quepa en la memoria GPU.
- *dataset_size* = 0,1 Esto quiere decir que se usa un 10 % del dataset.
- *batch_size* = 8 - > 2 El nº de imágenes que se usan a la vez en cada ciclo del entrenamiento.
- *#learn.recorder.plot()* Esta operación muestra una gráfica de “Learning rate” vs “Loss”, en el que se podría comprobar qué ratio de aprendizaje tiene menor pérdida. No se ha podido ejecutar porque consume mucha memoria.

epoch	train_loss	valid_loss	acc_camvid	time
0	2.539388	1.924914	0.542044	00:05
1	1.946895	1.438631	0.589393	00:04
2	1.632087	1.342975	0.657553	00:04
3	1.400821	1.216354	0.642960	00:04
4	1.246845	1.040034	0.760023	00:05
5	1.210604	1.110132	0.708900	00:04
6	1.092206	0.946427	0.779377	00:04
7	1.022060	0.886610	0.782427	00:04
8	0.942024	1.057799	0.700746	00:04
9	0.913496	0.883105	0.764337	00:04

Figura 8.1: Métricas obtenidas al entrenar durante 10 epochs. acc_camvid es el nº de píxeles clasificados correctamente entre el nº de píxeles totales. El tiempo es en segundos.

En la figura 8.1 se pueden ver las métricas obtenidas durante el entrenamiento. Esta tarea es de segmentación, por lo que a cada píxel de la imagen se le da un valor dependiendo de a la clase a la que pertenezca. Carece de interés analizar en detalle este caso ya que se ha hecho como primer ejemplo de uso de esta librería para segmentación.

Es importante tener en cuenta que ha sido necesario disminuir la imagen original de $(720, 960)px$ a $(90, 120)px$, que cuyo tamaño de imagen es $\frac{1}{8*8} = \frac{1}{64}$ del original.

Computación en la nube

Se ha intentando trabajar con la GPU local (2GB vRAM) en este ejemplo 2D, realizando optimizaciones y reduciendo los datos de entrada. Se han podido obtener resultados pese a reducir mucho el tamaño de imagen, por lo que entrenar con la GPU local podría ser válido para datos 2D. Aún así, en este proyecto se va a trabajar con datos 3D, por lo tanto no es realista usar la GPU local para el entrenamiento e inferencia.

Por ello se ha investigado sobre el uso de GPUs en Cloud Computing. Se ejecutará el código anterior en un notebook Jupyter gratuito con una GPU P5000, que tiene 16GB vRAM (el coste de adquirir un equipo similar al ofrecido superaría los 2000€). La empresa que ofrece este servicio es Paperspace (Paperspace, 2020), que tiene una tier gratuita en la que ofrecen un notebook con la GPU P5000 con pocas limitaciones.

Gracias a esto se ha conseguido ejecutar el notebook con los siguientes parámetros:

- $image_size = (720, 960)/2 = (360, 480)px$
- $dataset_size = 1$
- $batch_size = 16$

En la figura 8.2 se pueden ver las métricas obtenidas en las que se muestra cómo se ha conseguido un acc_camvid superior, que es la métrica usada para medir cómo de buena es la predicción, donde 1 sería una predicción perfecta. Las pérdidas de entrenamiento y validación son menores, por lo que el modelo considera que se ha acercado más a su objetivo.

En cada epoch de entrenamiento se han usado 16 imágenes de $(720, 960)/2px$. Cada imagen tiene $\frac{1}{4}$ del tamaño original y las 16 imágenes entran en memoria a la vez, por lo que se podría decir que en cada epoch se usa un dato de entrada con tamaño $\frac{1}{4} * 16 = 4$ veces la imagen original. En el caso anterior se han usado imágenes de $(720, 960)/8px$ con un batch de 2, por lo que se

epoch	train_loss	valid_loss	acc_camvid	time
0	2.101800	1.494480	0.684659	01:43
1	1.443275	0.909511	0.803556	01:43
2	1.067760	0.746240	0.822380	01:43
3	0.878190	0.676551	0.830904	01:43
4	0.738405	0.561852	0.841273	01:43
5	0.655896	0.562380	0.852621	01:43
6	0.629800	0.553934	0.844204	01:43
7	0.574289	0.421214	0.880067	01:43
8	0.523570	0.462477	0.870609	01:44
9	0.480465	0.337403	0.899350	01:43

Figura 8.2: Métricas obtenidas al entrenar durante 10 epochs. acc_camvid es el nº de píxeles clasificados correctamente entre el nº de píxeles totales. El tiempo es en segundos.

podría decir que en cada epoch se ha usado un dato de entrada con tamaño $\frac{1}{64} * 2 = \frac{1}{32}$ veces la imagen original.

Aunque solo se ha aumentado de 2GB a 16GB, la mejora ha sido de poder ser capaz de usar un dato de entrada $\frac{4}{1/32} = 128$ veces mayor.

CAPÍTULO 9

Esquema general

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.

Sistema inicial

En la figura 9.1 se muestra un esquema general sobre el sistema inicial, explicado en más detalle en el capítulo 10.

Los dos primeros recuadros (azules) muestran los datos de entrada y el preprocesado necesario para que puedan ser utilizados en el entrenamiento.

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.

Tras esto, se generarán nuevos archivos hdf5 y se subirán a un disco duro virtual, al cual se accederá por un Notebook. Las imágenes de entrada podrán ser usadas como entrada en el modelo seleccionado, cuya salida será una segmentación de células con espaciado.

En el recuadro siguiente (naranja), se muestra el proceso que se lleva a cabo durante el entrenamiento de un modelo. Como ya se mencionó en el capítulo 8 este proceso deberá realizarse con una GPU de alto rendimiento, por lo que en este proyecto se utilizará una máquina en la nube. Antes de comenzar este proceso los datos ya habrán sido cargados en un disco duro virtual. Se dividirá el dataset en 70 % entrenamiento, 15 % validación y 15 % test.

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 el modelo ha mejorado de forma programática. Si el modelo no mejora al disminuir la pérdida significará que habrá que cambiar la función de pérdida.

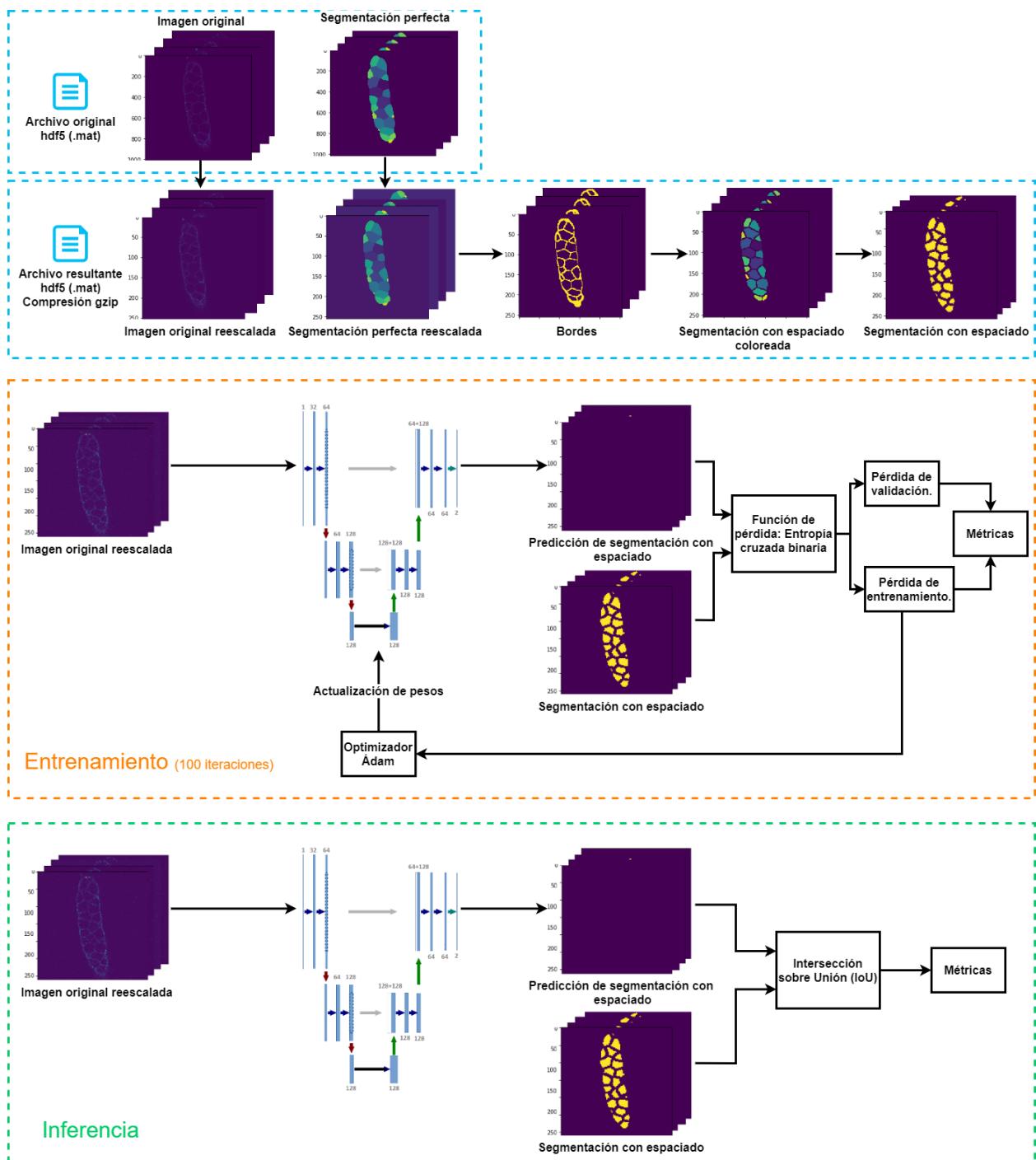


Figura 9.1: Diseño del sistema inicial.

El último recuadro de la figura 9.1 (en verde) muestra el proceso de inferencia. Se utilizará como mejor modelo aquél que haya obtenido una menor pérdida de validación durante el entrenamiento. Por ejemplo, si se entrena durante 100 iteraciones y en la iteración 89 se obtuvo la menor pérdida de validación, se utilizará el modelo entrenado durante esta iteración y se ignorarán las iteraciones posteriores. Tras realizar el entrenamiento completo se usarán los datos de test para obtener predicciones y se analizarán con la intersección sobre unión (*intersection over union* o *IoU*, con apoyo de la representación visual del resultado).

Sistema final

En la figura 9.2 se muestra el diseño final del sistema tras aplicar todas las mejoras descritas en los capítulos siguientes.

Antes de utilizar la imagen original reescalada como entrada de la CNN, se han añadido dos pasos:

1. Normalización por unidad tipificada. Descrita en el capítulo 14, este proceso es necesario reducir las diferencias de intensidad de los véxeles y así facilitar el reconocimiento de patrones estructurales por la CNN.
2. Data augmentation. Descrito en el capítulo 12, este proceso es necesario para aumentar el nº de ejemplos de entrenamiento y reducir el sobreajuste.

Además, se ha cambiado la arquitectura U-Net por una completa, como se verá en el capítulo 13. La arquitectura del sistema inicial era temporal, pensada sólo para hacer iteraciones rápidas.

También se ha cambiado la función de pérdida por la pérdida DICE, como se verá en el capítulo 11. Utilizar una función de pérdida que represente correctamente el error del problema a tratar es algo esencial, de no ser así la pérdida obtenida durante el entrenamiento calculará un error carente de interés.

Por último, se ha implementado la precisión mixta (capítulo 15) para reducir la memoria y el tiempo necesarios durante el entrenamiento e inferencia. Esto hará que en algunas operaciones se utilice un tipo de dato de 16 bits en vez de uno de 32 bits.

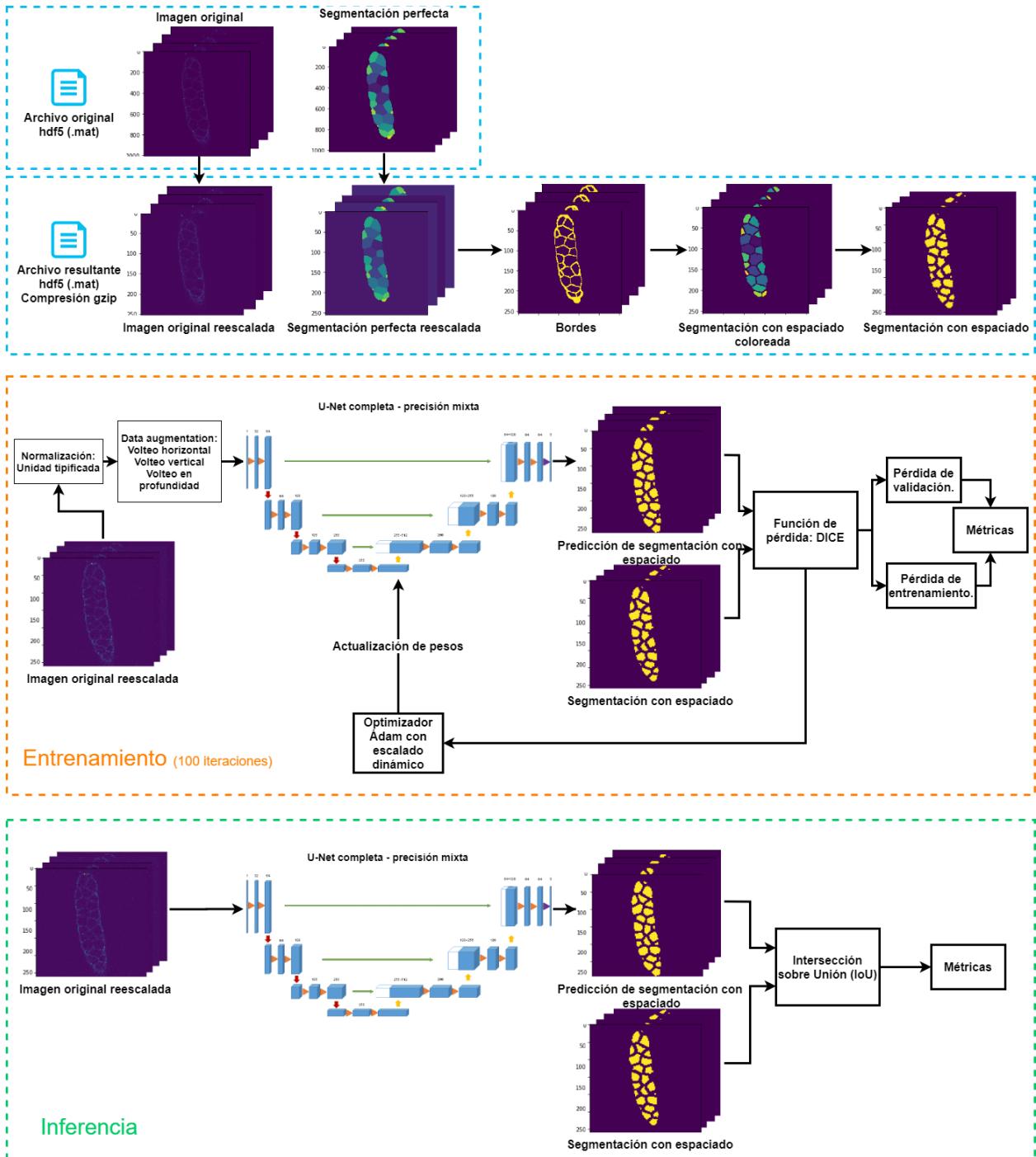


Figura 9.2: Diseño del sistema inicial.

CAPÍTULO 10

Sistema inicial

10.1– Lenguaje y framework

Lenguaje

El sistema se implementará completamente con el lenguaje de programación Python 3.7.

La elección del lenguaje Python es sencilla: los frameworks más utilizados de 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 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. Gracias a esto la semántica del lenguaje puede omitir los tipos al declarar y asignar variables, pero el rendimiento es menor.

Estas desventajas se pueden reducir gracias a que Python tiene la capacidad de llamar a subrutinas compiladas en 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.

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 multitud de operaciones de forma eficiente.

Framework

El framework utilizado será PyTorch 1.6 (Paszke y cols., 2019).

En capítulos anteriores se mencionó el uso de fastai. Desde un principio se pretendía usar fastai para la parte de deep learning, pero en fastai no hay modelos que tomen como dato de entrada imágenes 3D. El framework permite implementar arquitecturas personalizadas para los modelos, pero para ello hay que utilizar PyTorch, ya que fastai está construido por encima de PyTorch. A causa de esto se decide aprender a usar PyTorch con un curso de Udacity (Serrano, 2020, Deep Learning with PyTorch).

Una vez completado el curso, se decide seguir usando PyTorch en vez de fastai ya que al usar un software de más bajo nivel se tiene más control sobre las operaciones que se realizan.

Aún así, 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 uso fácil y buen rendimiento.

10.2– Metas y métricas

Métrica	Buen resultado	Resultado ideal
IoU	$\geq 0,7$	$\geq 0,9$

Cuadro 10.1: Cuadro de métricas y metas.

Una buena métrica para la segmentación semántica es la utilizada en el artículo descrito en la sección 7.1 (Falk y cols., 2019), donde se usa la métrica intersección sobre unión (IoU) para comprobar los resultados.

$$M_{IoU}(A, B) := \frac{|A \cap B|}{|A \cup B|} \quad (10.1)$$

Donde A es el conjunto de véxeles pertenecientes a la imagen perfectamente etiquetada y B el conjunto de véxeles pertenecientes a la predicción.

En ese mismo artículo se observa que si la tarea de etiquetado es realizada por un humano se obtiene un IoU $\sim 0,9$ y se define como buen resultado un IoU $\sim 0,7$.

10.3– Arquitectura CNN

Se ha implementado una arquitectura U-Net basada en la implementación de Ronneberger (Ronneberger y cols., 2015). Se ha usado una implementación en PyTorch (shiba24, 2017) con la arquitectura mostrada en la figura 10.1. Se ha desarrollado una versión de la arquitectura UNet3D que se ha llamado MiniUNet3D, con menos capas. Esta versión se ha usado para comprobar que el primer sistema se ha implementado correctamente sin provocar errores en tiempo de ejecución.

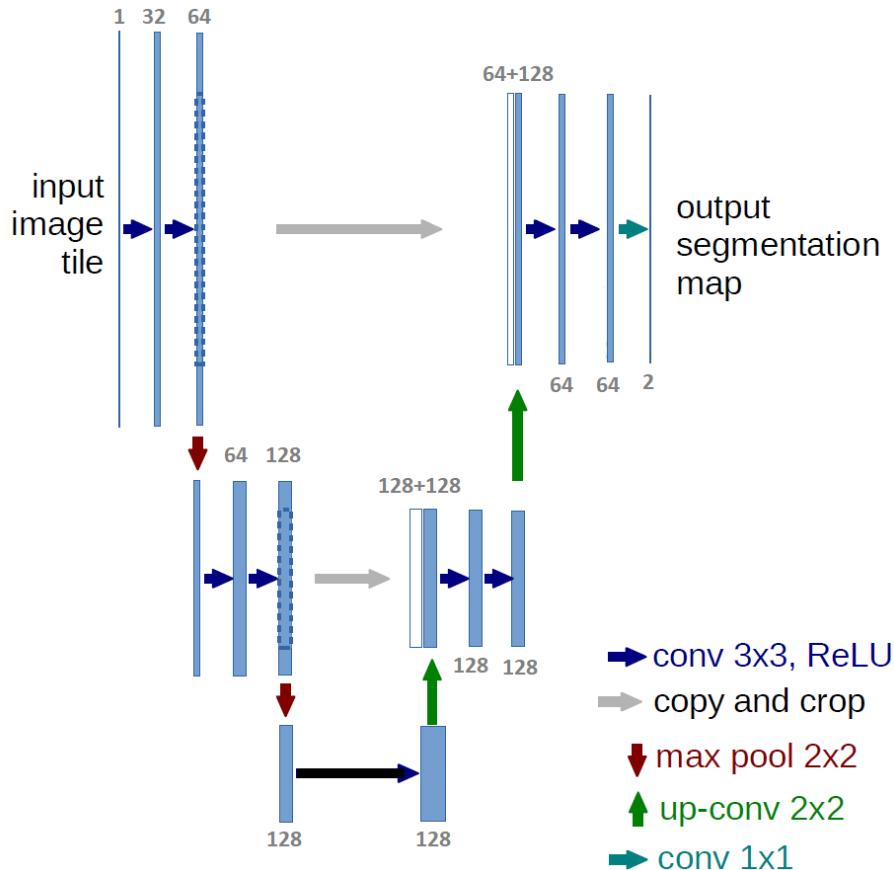


Figura 10.1: Arquitectura U-Net reducida. Arquitectura completa (vista en 6.3 eliminando varias capas de convolución y una de pooling). Los modelos generados por esta arquitectura serán menos fiables, pero el entrenamiento será más rápido, por lo que es útil para implementar un primer sistema.

10.4– Tamaño de la imagen en disco

Al estar trabajando con herramientas en la nube será recomendable reducir el tamaño de los archivos lo mayor posible. Los archivos originales ocupan en total 9,17GB, siendo el almacenamiento disponible de la máquina en la nube con la que se trabaja de 5GB.

Al principio no se tuvo en cuenta la solución al tamaño de la imagen en memoria (sección 10.5) en la que se reduce la resolución de las imágenes. Esto habría resuelto también el problema del tamaño en disco. Aún así, en esta sección se aplican técnicas para reducir aún más el tamaño en disco sin perder información.

Todas las imágenes utilizan float de 8 bytes (*f8*) para almacenar los valores de cada voxel. En el cuadro 10.2 se puede ver que la valor máximo de los voxels es 65535,0, por lo que no es necesaria una precisión de 8 bytes ($[2^{-1023}, 2^{1023}]$) para los valores de esas imágenes, especialmente para el etiquetado, en el que se usa valores enteros entre 0 y 100.

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

Si se hace que los valores de la imagen de entrada pasen de 8 bytes a 4 bytes, estaremos reduciendo su tamaño a la mitad. De forma similar si hacemos que los valores de la imagen etiquetada pase de 8 bytes a 1 bytes, estaremos reduciendo su tamaño a una octava parte. Con 1 byte de precisión podremos almacenar hasta 256 valores distintos, suficiente ya que sólo tendremos 2 valores distintos: 0 para el fondo y 1 para la célula o los bordes.

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

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

1. Los tensores al cargarlos en memoria estarán sin ningún tipo de compresión. Cada valor del tensor ocupará el espacio correspondiente a su tipo de dato.
2. Las operaciones implementadas en los frameworks de deep learning limitan el tipo de dato que pueden tener los tensores. Están implementados a muy bajo nivel para optimizar las operaciones, por lo que será necesario convertir los tipos a unos que acepten, si no lo son ya.

Para leer los ficheros originales y guardarlos con el mismo formato se ha usado la librería **h5py** (Collette, 2013). Esta aporta una interfaz para trabajar con el formato de datos HDF5 (The HDF Group, 2000-2010), útil para usar en python ficheros generados en programas como MATLAB.

El tratamiento de datos con esta librería es muy intuitivo ya que imita la sintaxis de NumPy. Ha sido usado para archivos .mat generados en MATLAB con los datos iniciales. Tras el preprocesado, se ha usado esta librería para generar un nuevo archivo con todas las imágenes en un nuevo archivo .mat.

- **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.

10.5– Tamaño de la imagen en memoria

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. Además, en el caso de U-Net, se guardan los resultados de algunas etapas para usarla en etapas futuras. Es completamente inviable usar la resolución original en este proyecto directamente como dato de entrada.

Escalado 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, se ha comprobado que los factores de las dimensiones no deben de ser muy diferentes, ya que cambiaría el tamaño en μm^3 de cada voxel, 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 la mayor dimensión 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 los valores $([0 : X - 1], [0 : Y - 1], [0 : Z - 1])$. De esta forma no se deforman las imágenes al hacer el escalado hacia arriba.

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.

Implementación

Para implementar esta solución se ha utilizado la librería **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.

Esta función hace interpolación con splines, polinomios de orden n definidos a trozos. Se ha utilizado orden 3 porque es el orden por defecto y ha dado buenos resultados.

10.6– Segmentación semántica multiinstancia con U-Net

El dataset disponible para este proyecto está compuesto por dos tipos distintos de imágenes. Uno es la imagen original y otro es una segmentación manual con el etiquetado perfecto. En el etiquetado perfecto cada célula está representada por un entero distinto sin espacio entre ellas, el fondo está representado por 0.

Si se usara el etiquetado actual sería necesario una clasificación con una clase por cada célula, pero eso no tiene mucho sentido ya que el nº de células en una imagen puede variar y las células

no tienen unas posiciones preestablecidas ni nada que las diferencien unas de otras (a priori, al menos). Que se sepa, todas las células tienen las mismas características por lo que deben tener la misma clase en una segmentación semántica. Tampoco se puede cambiar la etiqueta de todas las células a 1 (siendo 1 la clase célula) ya que las células están en contacto entre sí (no hay espacio entre células), lo que provocaría que los bordes originales de las células se perdieran.

La solución que se probará primero será añadir un espaciado entre células, tal y como se hace en el pipeline del artículo descrito en la sección 7.1 (Falk y cols., 2019). Al existir un espaciado entre células, estas no estarían tocándose entre sí, por lo que todas podrían tener la misma etiqueta y contar como instancias distintas.

Para añadir este espaciado entre células primero se encuentran los bordes de cada célula y después se substraen de la imagen original. Por último se cambian todas las etiquetas a 1.

Para hacer estas operaciones se ha utilizado la librería **scikit-image** (van der Walt y cols., 2014), una librería de procesamiento de imágenes.

Encontrar bordes

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

- Conectividad. Hay N tipos de conectividad siendo N el nº de dimensiones de la imagen de entrada, en nuestro caso N=3. Con conectividad 1 los vóxeles compartiendo al menos una cara son considerados vecinos. Con conectividad 3 se amplía el rango de vecinos al hacer que cualquier vóxel compartiendo una esquina también son considerados vecinos. Si dos vóxeles vecinos tienen distinta etiqueta, son bordes. Se ha elegido conectividad 3 para priorizar el que no haya células en contacto en ninguna de las 3 dimensiones.
- Modo. Este modo define qué vóxeles son marcados como bordes. El modo escogido es *outer*, que selecciona como bordes los vóxeles de fondo alrededor de los elementos. Si hay 2 elementos en contacto se marca su frontera como borde. Se ha escogido este modo ya que es el que retiene la mayor cantidad de volumen celular original.

Comprobar cantidad de células

También se ha usado la función (*measure_label()*) para comprobar que el nº de células no varía al hacer las operaciones de preprocessado (por ejemplo, podría ser que dos células con etiqueta 1 eliminaran el espacio entre sí, convirtiéndose en la misma instancia de célula).

10.7– Carga de datos

Una vez todos los datos han sido preprocessados y almacenados en nuevos ficheros ya se pueden utilizar como fuente de datos para cargar las imágenes en tiempo de ejecución. En la carga de datos también se hace un tratamiento de imagen y será necesario que éste sea rápido, ya que se realizará cada vez que se acceda a una imagen durante el entrenamientos.

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

- **h5py.** La misma librería usada para crear los ficheros en la etapa anterior es usada para cargarlos.
- **PyTorch.** PyTorch es el framework de deep learning elegido para este trabajo, por lo que se sacará el máximo provecho de lo que ofrezca de base. Para procesamiento de imagen es especialmente útil el paquete *torchvision*.

- Clase base para Dataset: Para indicar dónde encontrar los datos y qué hacer con ellos, se usará la clase `torchvision.Dataset` como base. Tan sólo hay que sobreescribir 3 métodos para que la clase sea funcional, estos son los métodos de inicialización (`__init__`), para obtener la longitud del dataset (`__len__`) y para obtener el siguiente ejemplo (`__getitem__`), en el que se realizará un preprocesado y se devolverá la imagen de entrada y la imagen etiquetada.
- DataLoader: Es usado para administrar el dataset. Se encarga de dar un nº de ejemplos aleatorios igual al batch seleccionado.

10.8– Guardar modelo

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

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

10.9– Resultados

Para comprobar los resultados se han utilizado 3 herramientas: la gráfica de la función de pérdida durante el entrenamiento, la métrica intersección sobre unión (IoU) y una representación visual de la segmentación.

IoU fondo	IoU células
0.9437	0.0018

Cuadro 10.3: Métricas sistema inicial. En la primera fila está cómo de bien se ha predicho el fondo. En la segunda fila está cómo de bien se han predicho las células.

En las métricas de la tabla 10.3 se muestra que el fondo está muy bien etiquetado, sin embargo las células están muy mal etiquetadas. Esto se puede ver visualmente en la figura 10.3.

Sin embargo la gráfica de la función de pérdida (figura 10.2 muestra que el modelo está acercándose a su objetivo correctamente. Este error se verá corregido en el capítulo siguiente, efectuando la primera mejora según la metodología vista en la sección 3.2

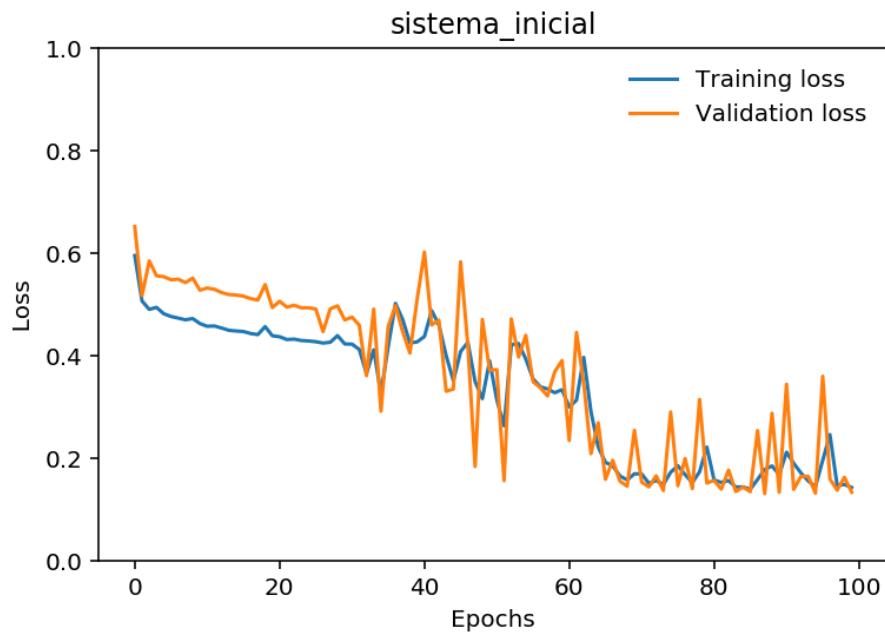


Figura 10.2: Pérdida de entrenamiento en cada iteración. 100 iteraciones. Se mide la pérdida media del conjunto de entrenamiento y el conjunto de validación.

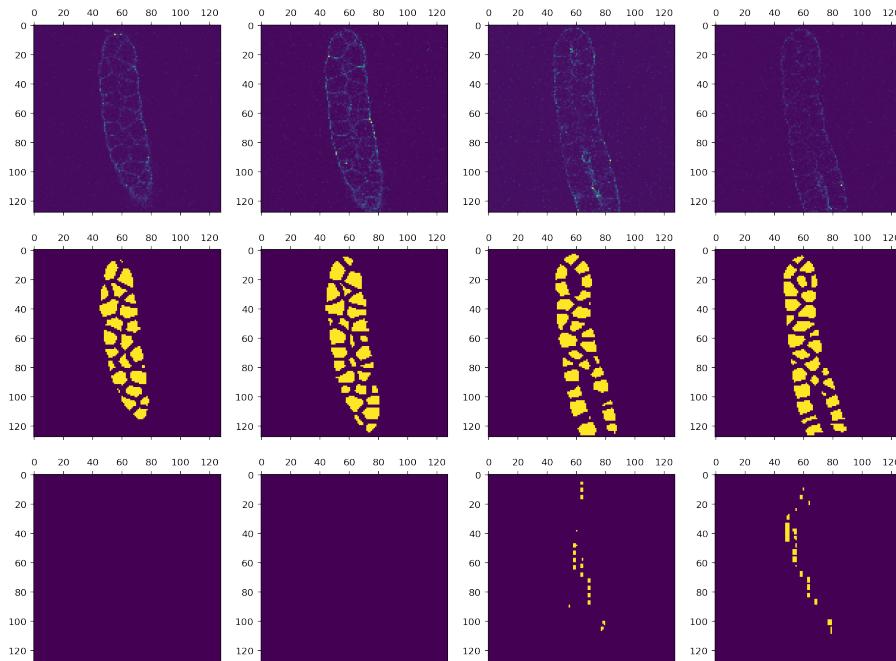


Figura 10.3: Ejemplo de segmentación del conjunto de test. Primera fila la imagen original, segunda fila segmentación objetiva, tercera fila segmentación predicha. Se muestra en cada columna $Z=20$, $Z=25$, $Z=45$, $Z=50$.

CAPÍTULO 11

Mejora 1: Función de pérdida Dice

En los resultados del capítulo anterior se mostró en cada iteración el algoritmo de aprendizaje hace el modelo se acerque más a su objetivo. Esto es lo que debe de ocurrir al entrenar un modelo, por lo que a priori todo está funcionando correctamente.

Sin embargo, los resultados son muy malos, obteniéndose un $IoU = 0,0018$, muy lejos del 0,7 propuesto como meta.

El motivo de que esta discrepancia ocurra es porque no se ha elegido una función de pérdida que represente correctamente el acercamiento a un objetivo útil para el problema actual.

La pérdida actual se calcula con la entropía cruzada (sin pesos):

$$L_{BCE}(y, \hat{y}) = -(y \log(\hat{y}) + (1 - y) \log(1 - \hat{y})) \quad (11.1)$$

Donde y es el objetivo y \hat{y} la predicción.

El problema principal se tiene en que las dos clases (fondo-0 y célula-1) están desbalanceadas y la entropía cruzada da la misma importancia a las dos clases. El modelo va a tender a etiquetar toda la imagen como fondo ya que así va a obtener una puntuación muy alta con esta función de pérdida.

La solución será cambiar la función de pérdida por otra que tenga en cuenta este desbalanceo entre clases. En el artículo estudiado en la sección 7.1 se utiliza la entropía cruzada binaria, mientras que en el artículo estudiado en la sección 7.2 se utiliza la pérdida Dice.

Se ha optado por probar primero la pérdida Dice ya que el cálculo de los pesos de la entropía cruzada binaria por pesos puede ser muy complejo. En un futuro se valorará el uso de esta pérdida.

11.1– Resultados

IoU fondo	IoU células
0.9381	0.2582

Cuadro 11.1: Métricas pérdida dice.

Esta vez la función de pérdida sí que representa correctamente el objetivo del problema actual. Se ha obtenido un $IoU = 0,2582$ y visualmente se puede ver cómo la segmentación empieza a formarse. Se ignorará el IoU del fondo ya que debería ser siempre mayor que 0,9.

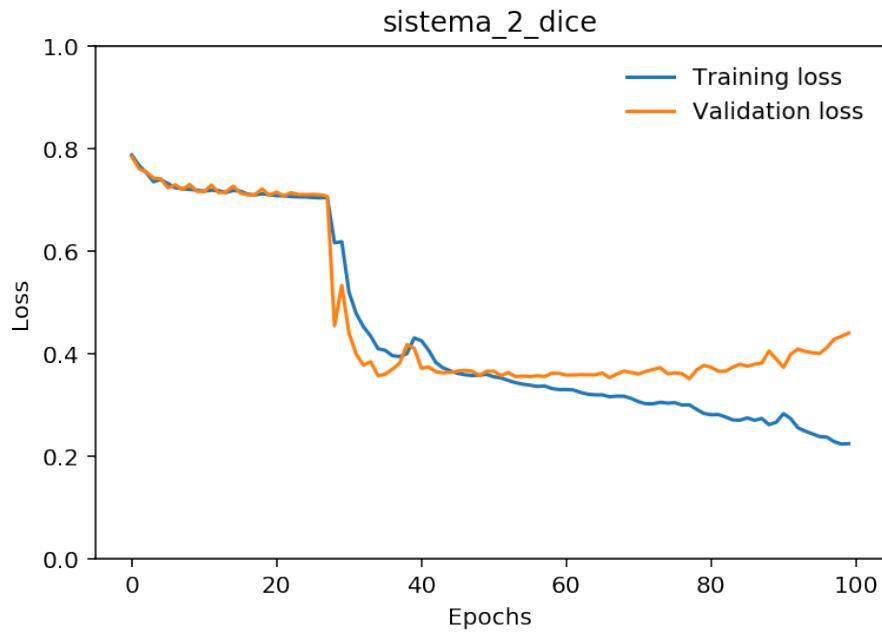


Figura 11.1: Pérdida de entrenamiento y validación en cada iteración. 100 iteraciones. Se mide la pérdida media del conjunto de entrenamiento y el conjunto de validación.

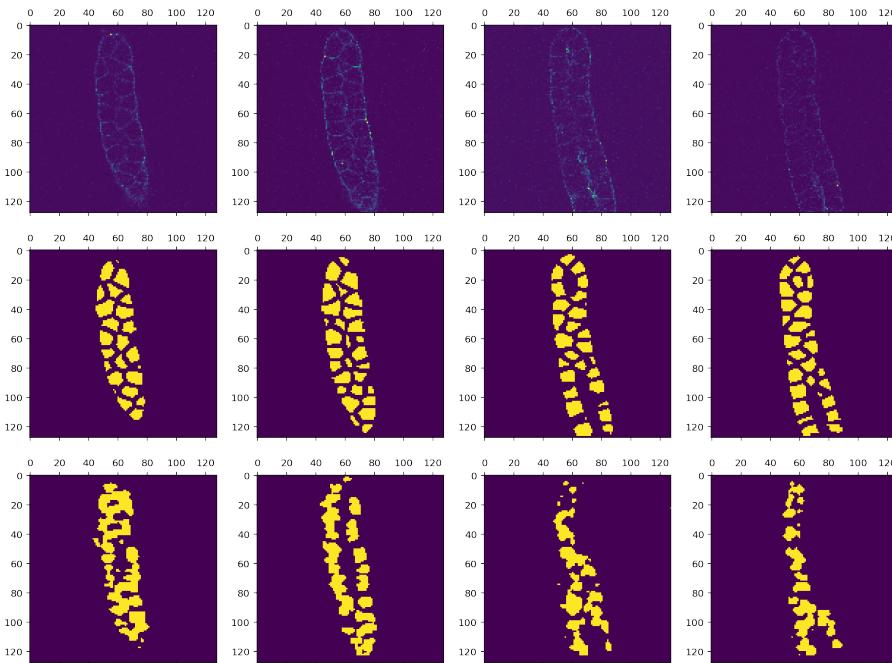


Figura 11.2: Ejemplo de segmentación del conjunto de test. Fila 1 imagen original, fila 2 segmentación objetiva, fila 3 segmentación predicha. Columnas Z=20, Z=25, Z=45, Z=50.

CAPÍTULO 12

Mejora 2: Data augmentation

En la gráfica de la función de pérdida de los resultados anteriores se puede ver algo preocupante: la pérdida de validación ha empezado a subir mientras que la pérdida de entrenamiento baja.

En cada iteración del algoritmo de entrenamiento se comparan todos los ejemplos del conjunto de entrenamiento para obtener la pérdida de entrenamiento. A menos que haya un error grave esta pérdida debería bajar ya que el modelo es entrenado con ejemplos de este mismo conjunto de entrenamiento. Sin embargo no se utilizan ejemplos del conjunto de validación para el entrenamiento, sólo se utilizan para calcular la pérdida de validación en cada iteración.

Si el modelo mejora en predecir el conjunto de entrenamiento pero empeora en predecir el conjunto de validación esto es un indicativo de que se está produciendo un sobreajuste. En la figura 12.1 se ha entrenado 100 iteraciones más (200 en total) el modelo anterior para comprobar si este sobreajuste se está produciendo realmente o era un fenómeno que ocurría durante un tramo pequeño. Claramente se puede ver que hay sobreajuste.

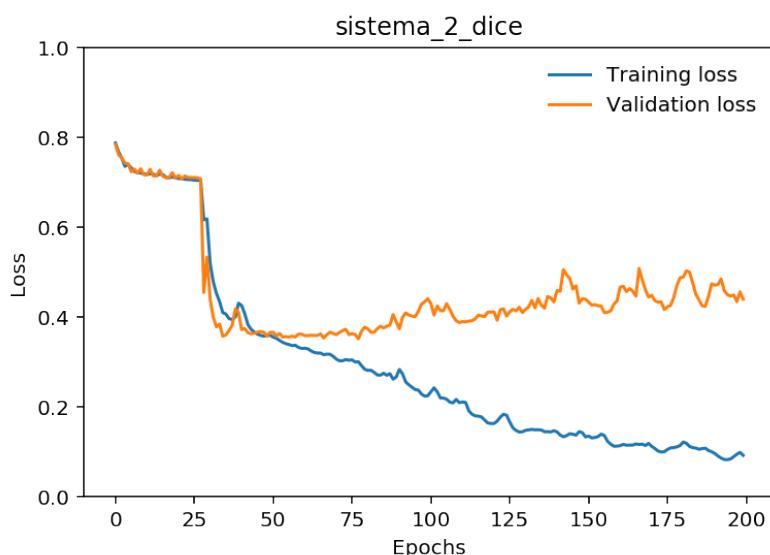


Figura 12.1: Pérdida de entrenamiento en cada iteración. 200 iteraciones. Se puede ver claramente cómo se produce sobreajuste.

PyTorch ofrece varias técnicas de data augmentation pero no están adaptadas para datos volumétricos, por lo que es necesario recurrir a librerías externas, como Kornia.

Kornia. Kornia (Riba, Mishkin, Ponsa, Rublee, y Bradski, 2020) es una librería especializada en visión por ordenador con PyTorch como backend. Aquí es usada para voltear la imagen en cada una de las 3 dimensiones de forma aleatoria. Gracias a esto se pasa de 15 ejemplos de entrenamiento a 120. Esta diferencia en cantidad de ejemplos de entrenamiento hace que el modelo no se sobreajuste tan rápido.

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

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

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

12.1– Resultados

IoU fondo	IoU células
0.9031	0.3165

Cuadro 12.1: Métricas data augmentation.

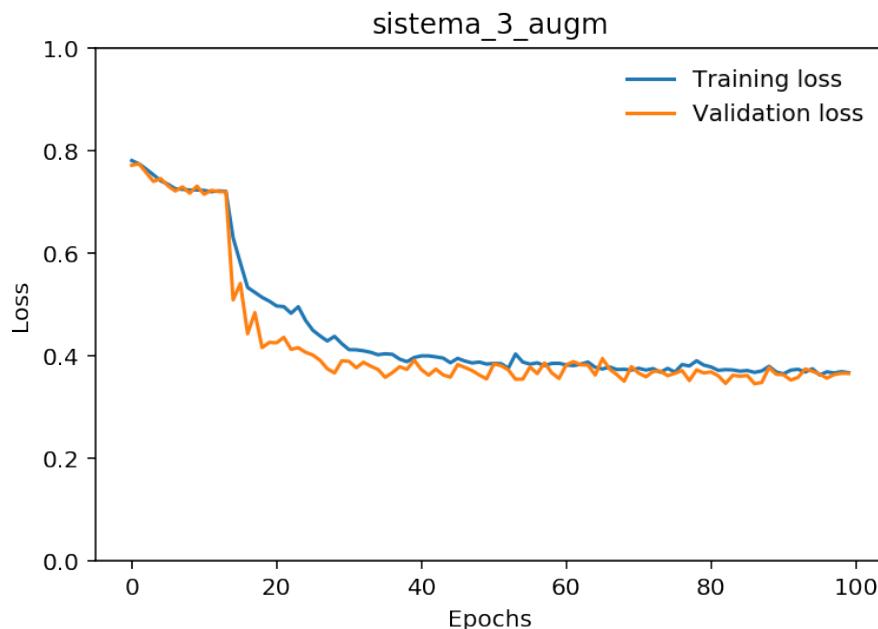


Figura 12.2: Pérdida de entrenamiento y validación. 100 iteraciones.

Gracias a este cambio se ha eliminado el sobreajuste, tal y como se puede ver en la gráfica 12.2. Se ha obtenido un $IoU = 0,3165$, mejor que sin data augmentation. Aunque como se puede ver en la figura 12.3 este valor de IoU está muy lejos de proporcionar una segmentación de células válida.

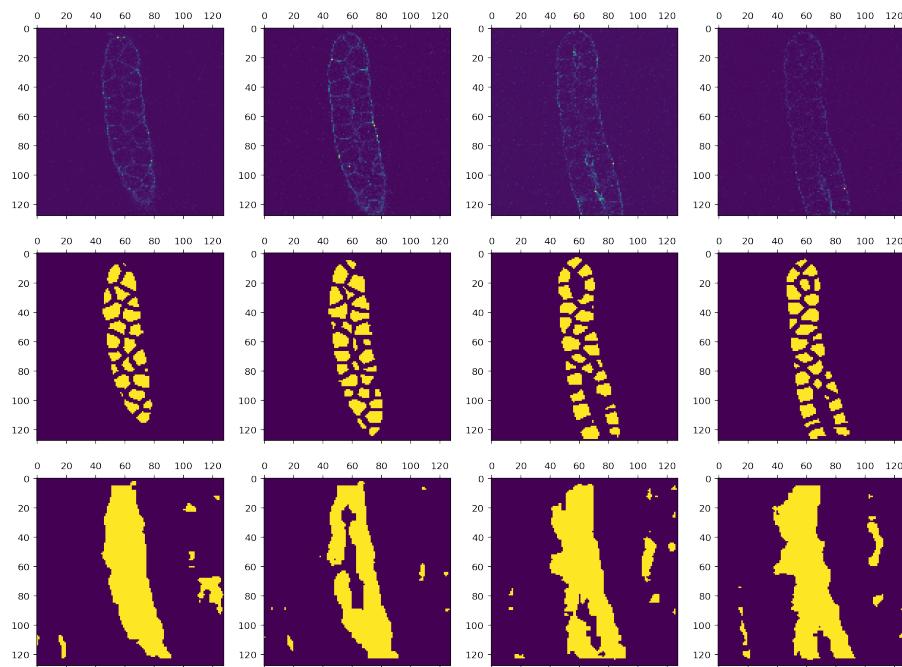


Figura 12.3: Ejemplo de segmentación del conjunto de test. Fila 1 imagen original, fila 2 segmentación objetiva, fila 3 segmentación predicha. Columnas Z=20, Z=25, Z=45, Z=50.

CAPÍTULO 13

Mejora 3: Arquitectura U-Net completa

Al utilizar una arquitectura U-Net para hacer pruebas se conseguía un tiempo bajo de entrenamiento. Esto ha sido útil ya que se ha podido mejorar el sistema con iteraciones rápidas, pero estaba lejos de producir una segmentación adecuada, algo que era de esperar.

Con la arquitectura U-Net reducida cada epoch tardaba 11s (en esos 11 segundos se hace entrenamiento y validación), durando el entrenamiento de 100 iteraciones $1100s = 18m20s$. Con la arquitectura implementada en este capítulo se tardará 37s por iteración, durando el entrenamiento de 100 iteraciones $3700s = 63m20s$,

En la figura 13.1 se puede ver la arquitectura completa de la red U-Net utilizada. Esta arquitectura ha sido usada con éxito para segmentación volumétrica densa (Özgün Çiçek, Abdulkadir, Lienkamp, Brox, y Ronneberger, 2016) y está basada en la arquitectura U-Net propuesta por Ronneberger et al (Ronneberger y cols., 2015). Más detalle en el capítulo 5 y en la sección 6.3.

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 representa 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.

13.1– Resultados

IoU fondo	IoU células
0.8454	0.2085

Cuadro 13.1: Métricas.

Por una parte, la pérdida ha llegado a un valor de ~ 0.2 (gráfica 13.2 mientras que con la arquitectura reducida llegó a un valor de ~ 0.4 . Es la misma función de pérdida que ya se ha validado que representa correctamente un acercamiento a la solución del problema de segmentación, por lo que podemos suponer que la segmentación es mejor.

Que la segmentación de las células es mejor se puede verificar en la figura 13.3. Se puede apreciar cómo las células están mejor definidas. Aún así, se mantiene el problema de etiquetar como célula zonas perteneciente al fondo.

A causa de etiquetar incorrectamente como célula el fondo, se siguen sin obtener resultados adecuados en la métrica IoU (tabla 13.1). En este caso la métrica IoU indica resultados inferiores

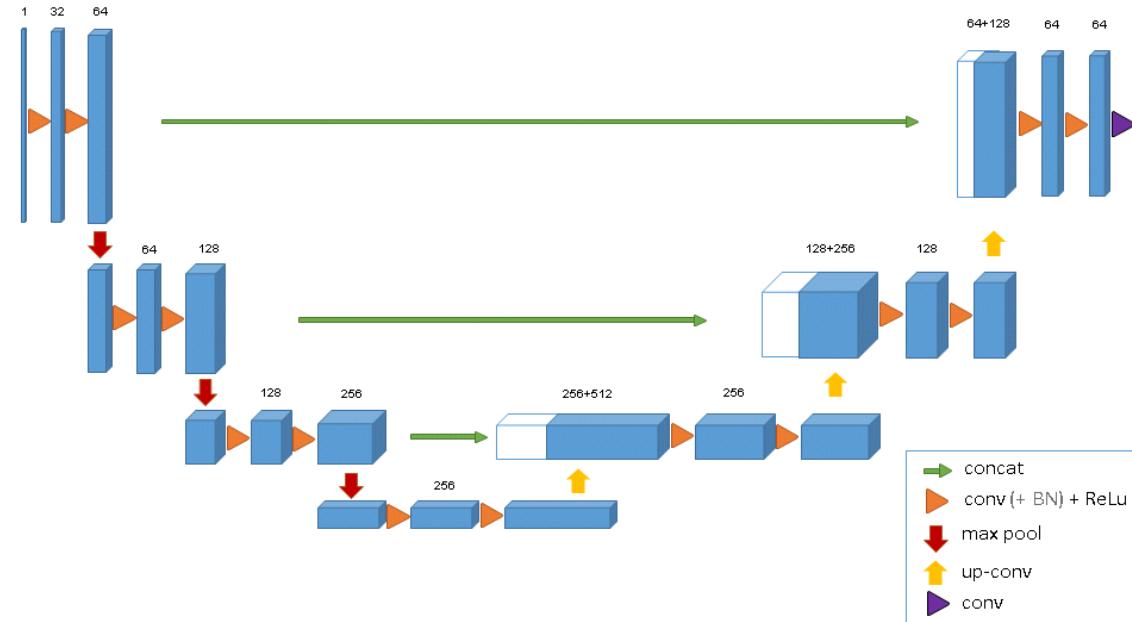


Figura 13.1: Arquitectura U-Net completa.

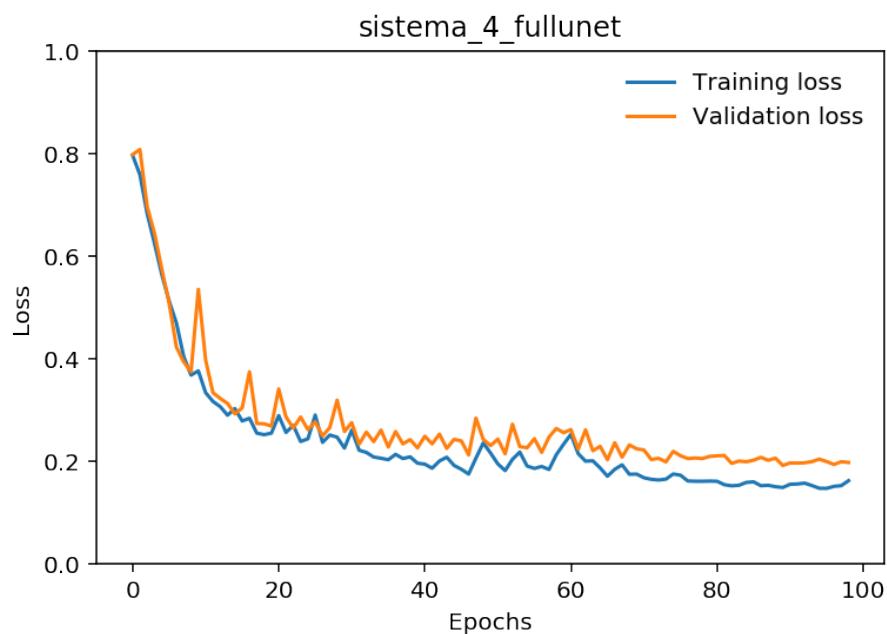


Figura 13.2: Pérdida de entrenamiento. 100 iteraciones.

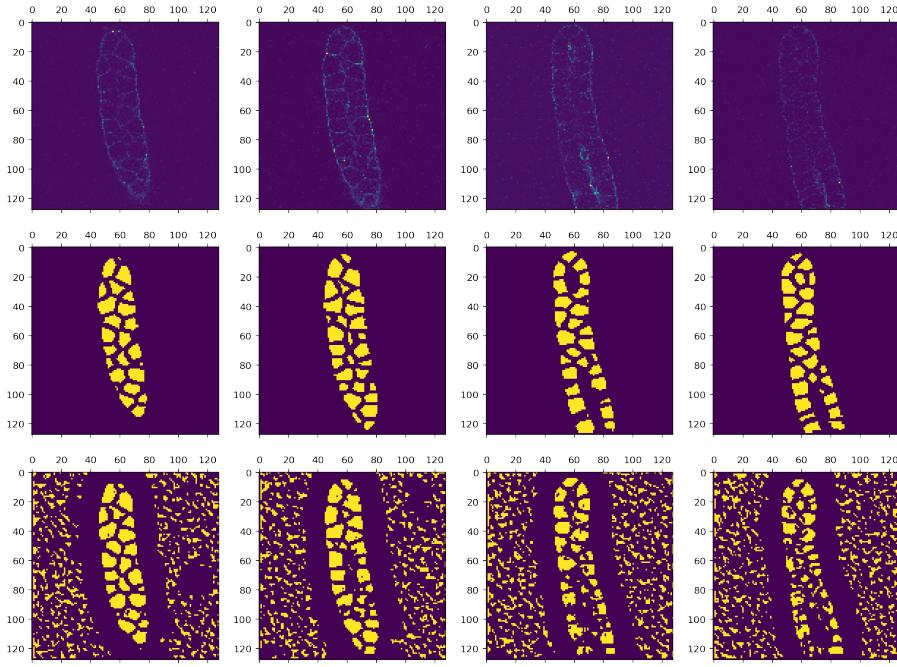


Figura 13.3: Ejemplo de segmentación del conjunto de test. Fila 1 imagen original, fila 2 segmentación objetivo, fila 3 segmentación predicha. Columnas Z=20, Z=25, Z=45, Z=50.

al anterior, pero por primera vez se ha podido ver claramente una segmentación de las células, por lo que se mantendrá este cambio y el siguiente capítulo se centrará en eliminar el sobre etiquetado actual.

CAPÍTULO 14

Mejora 4: Normalización

Aunque en los últimos resultados ya se empieza a ver una segmentación adecuada de las células, se está etiquetando incorrectamente el fondo como célula.

Tras estudiar la imagen original se ha llegado a la conclusión de que podría deberse a la diferencia en la intensidad de los vértices. En el cuadro 14.1 se muestra la intensidad mínima y máxima de las 20 imágenes que componen el dataset. Puede ser un problema que haya vértices con valor 60000 y otros con valor 1, ya que los que tengan un valor más alto tendrán mucho más peso en la red convolucional. Lo interesante en la red convolucional será encontrar patrones estructurales, no patrones relacionados con una diferencia alta de intensidad.

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

Para ignorar esta diferencia alta de intensidad y centrarnos en las diferencias estructurales se puede recurrir a la normalización. En concreto se probarán dos normalizaciones distintas: el cambio a escala $[0, 1]$ y por distribución normal (puntuación tipificada).

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

$$v'_i = \frac{v_i - v_{min}}{v_{max} - v_{min}} \quad (14.1)$$

Donde v'_i es el nuevo valor del vértice, v_i el valor actual, v_{min} el valor mínimo en toda la imagen y v_{max} el valor máximo en toda la imagen.

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

$$v'_i = \frac{v_i - \mu}{\rho} \quad (14.2)$$

Donde v'_i es el nuevo valor del vértice, v_i es el valor actual, μ es la media de los valores de la imagen y ρ es la desviación estándar. (*Z-normalization of time series.*, s.f.)

Para decidir qué técnica usar se van a comparar los histogramas.

En la figura 14.1 se muestran los dos histogramas. En el histograma del cambio a escala se puede ver cómo los vértices sólo tienen valores entre 0 y 1. Aún así se observa que la mayoría

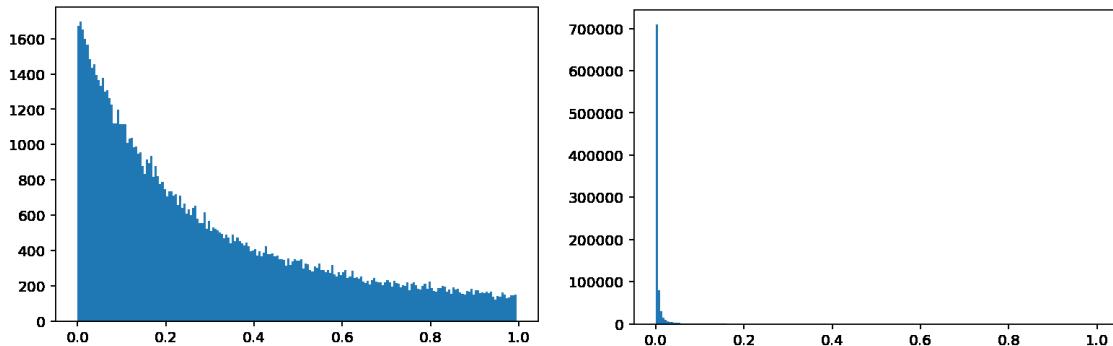


Figura 14.1: Izquierda: Normalización tipificada. Derecha: Cambio a escala [0, 1].

de los valores están en la zona de menor intensidad, lo que se soluciona con la normalización tipificada. Por lo tanto es esta última la que se implementará.

Aunque las operaciones matemáticas son simples, se utilizará TorchIO (Pérez-García, Sparks, y Ourselin, 2020) para implementar la normalización tipificada, ya que facilita el uso de otras posibles operaciones a los datos sin modificar mucho el código. TorchIO es una librería que contiene herramientas para trabajar con datos 3D especializadas para imágenes médicas. Se ha utilizado la función `transforms.ZNormalization()` en la imagen de entrada.

14.1– Resultados

IoU fondo	IoU células
0.9811	0.7224

Cuadro 14.2: Métricas normalización.

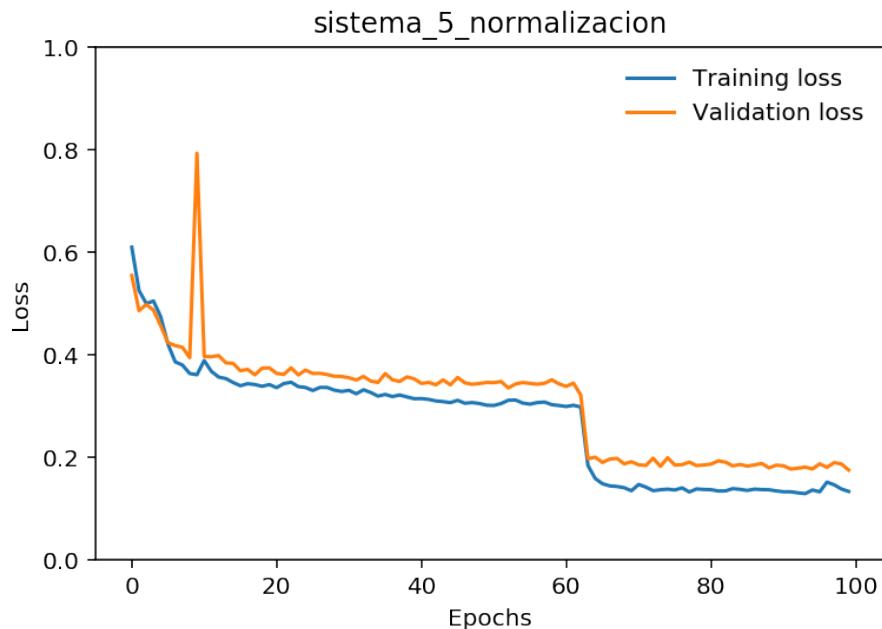


Figura 14.2: Pérdida de entrenamiento. 100 iteraciones.

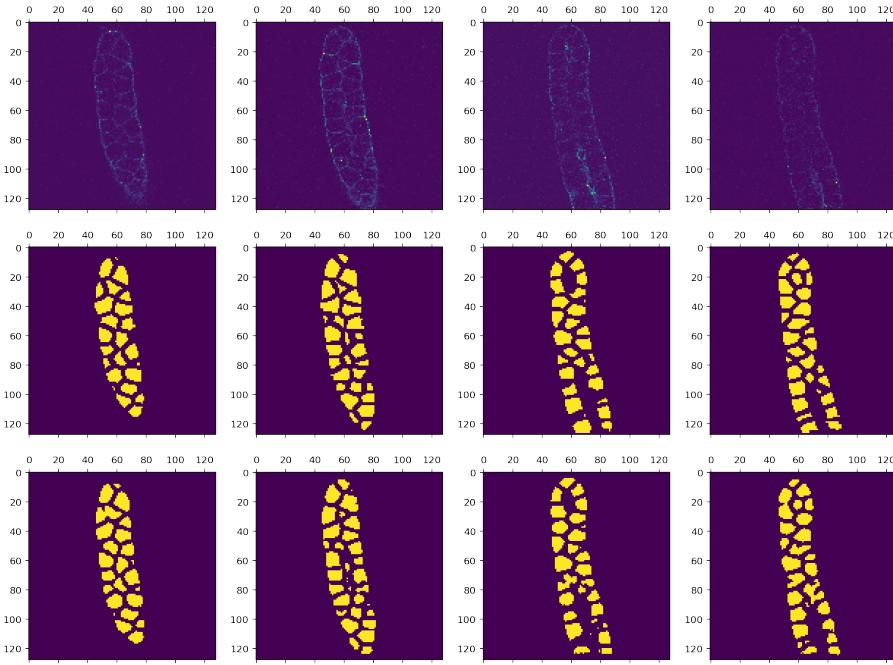


Figura 14.3: Ejemplo de segmentación del conjunto de test. Fila 1 imagen original, fila 2 segmentación objetivo, fila 3 segmentación predicha. Columnas Z=20, Z=25, Z=45, Z=50.

Por primera vez se ha conseguido una $IoU > 0,7$ para las células, siendo la IoU del fondo casi perfecto. Al normalizar los datos la red convolucional ha podido encontrar correctamente los patrones estructurales en la imagen original para poder llegar a la segmentación objetivo.

Ya se ha llegado a un $IoU > 0,7$, que era el objetivo impuesto. Los capítulos posteriores estarán orientados a reducir el consumo de memoria para aumentar la resolución de las imágenes de entrada, buscar métodos alternativos o postprocesados.

CAPÍTULO 15

Mejora 5: Apex, precisión mixta

Durante el entrenamiento se ha usado la herramienta gpustat (Wookayin, s.f.) para medir la vRAM de la GPU utilizada. Desde el sistema inicial se han utilizado imágenes de baja calidad al reescalar la original a un tamaño de $(Z_L, 126, 126)$, donde Z_L depende de cada imagen, y un batch de tamaño 1. Al entrenar se consumía 11525GB y al hacer inferencia 6063GB. Se ha intentado utilizar imágenes de mayor calidad $(Z_M, 252, 252)$ pero no había suficiente memoria. También se había intentado utilizar un batch de tamaño 2 en el sistema inicial, pero la memoria también lo limitaba.

El problema que se quiere resolver es el alto consumo de memoria. Si se reduce la memoria necesaria se podrían utilizar imágenes de mayor calidad, aumentando la cantidad de información que tienen las imágenes durante el entrenamiento. Además, usando imágenes de mayor calidad se reduciría el espaciado entre células ya que el espaciado entre dos células adyacentes seguiría teniendo el mismo nº de píxeles pero la imagen (y cada célula) sería mayor.

Si no fuera posible aumentar la calidad de imagen aún consiguiendo una reducción de memoria, se intentará aumentar el tamaño del batch para reducir el tiempo de entrenamiento.

En 2017 se publicó el artículo *Mixed Precision Training* en el se demostró que el uso de precisión mixta no influye negativamente en el entrenamiento de CNNs (Micikevicius y cols., 2018) y reduce el uso de memoria y el tiempo de entrenamiento.

En la precisión mixta se usa FP16 para almacenar los tensores involucrados en el entrenamiento y para operaciones aritméticas, aunque algunas operaciones (como sumar todos los elementos de un vector) se dejan en FP32. Además se añade una copia maestra de pesos en FP32.

- **FP32** Formato en coma flotante de precisión simple. Ocupa 32 bits, con rango $[2^{-126}, 2^{127}]$
- **FP16** Formato en coma flotante de precisión media. Ocupa 16 bits, con rango $[2^{-14}, 2^{15}]$

Las técnicas explicadas en *Mixed Precision Training* están implementadas para su uso en PyTorch por Apex (nVidia Apex, 2020). Apex es una librería desarrollada por NVidia que gestiona estas operaciones al modificar directamente PyTorch. Hay varios niveles de optimización disponibles, se ha probado a utilizarlo con una optimización de nivel 1 y de nivel 2.

- En optimización de nivel 1 (O1) se cambia la entrada de algunas funciones para que usen FP16 (float de 16 bits) y se dejan otras que puedan beneficiarse de la precisión en FP32 (float de 32 bits).
- En optimización de nivel 2 (O2) se cambian los pesos del modelo a FP16, se cambian los métodos del modelo para que acepten FP16 y se mantienen unos pesos maestros en FP32 que son usados por el optimizador. En este caso no se cambia las entradas de las funciones como en el nivel 1.

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

15.1– Resultados

A partir de este capítulo se utilizará otra herramienta para medir la mejora de un modelo y tener más información.

Por cada ejemplo del conjunto de test (2), se comparará el nº de componentes conexas de la segmentación predicha con el nº de componentes conexas que debería haber en la segmentación perfecta. Cada componente conexa es una célula, esto significa que si en la predicción hay menos componentes conexas habrá células en contacto que se interpretarán como una sola célula. La métrica utilizada será *wrongCells*:

$$wrongCells = \sum_{i=0}^{n-1} |correctCount_i - predictionCount_i| \quad (15.1)$$

Dónde i es el índice del ejemplo del conjunto de test y n es el nº de ejemplos en el conjunto de test.

Además se medirá el consumo de memoria y tiempo por epoch del entrenamiento.

El cuadro 15.1 muestra las nuevas métricas usadas en este capítulo para la mejora de normalización del capítulo 14.

IoU fondo	IoU células	<i>wrongCells</i>	Tiempo epoch	Mem. entre.	Mem. infe.
0.9811	0.7224	30	~37s	11525GB	6063GB

Cuadro 15.1: Métricas normalización.)

Nº	IoU fondo	IoU células	<i>wrongCells</i>	Tiempo epoch	Mem. entre.	Mem. infe.
1	0.9821	0.7431	32	~37s	6243GB	6063GB
2	0.9818	0.7396	29	~37s	6243GB	6063GB

Cuadro 15.2: Métricas precisión mixta O1. Se han hecho 2 pruebas con la misma configuración.

Nº	IoU fondo	IoU células	<i>wrongCells</i>	Tiempo epoch	Mem. entre.	Mem. infe.
1	0.9805	0.7539	11	~37s	6633GB	6063GB
2	0.9805	0.7263	74	~37s	6633GB	6063GB

Cuadro 15.3: Métricas precisión mixta O2. Se han hecho 2 pruebas con la misma configuración.

El objetivo principal de esta mejora es reducir la cantidad de memoria necesaria para el entrenamiento e inferencia y así poder usar imágenes de mayor resolución o un tamaño del batch mayor que 1.

En el sistema del capítulo 14 eran necesarios 11525GB de memoria para el entrenamiento (cuadro 15.1). Al aplicar la técnica de precisión mixta se ha conseguido reducir la memoria necesaria durante el entrenamiento a 6243GB.

Hasta ahora se han utilizado imágenes con dimensión $[Z/4, X/8, Y/8]$, siendo $[Z, X, Y]$ las dimensiones originales. Tras conseguir esta disminución de memoria se ha intentado entrenar un modelo utilizando imágenes de dimensión $[Z/2, X/4, Y/4]$, pero no ha sido posible. Se estima que se necesitarán al menos 25GB para esto. La otra opción para aprovechar esta reducción de memoria podría ser aumentar el batch de entrenamiento.

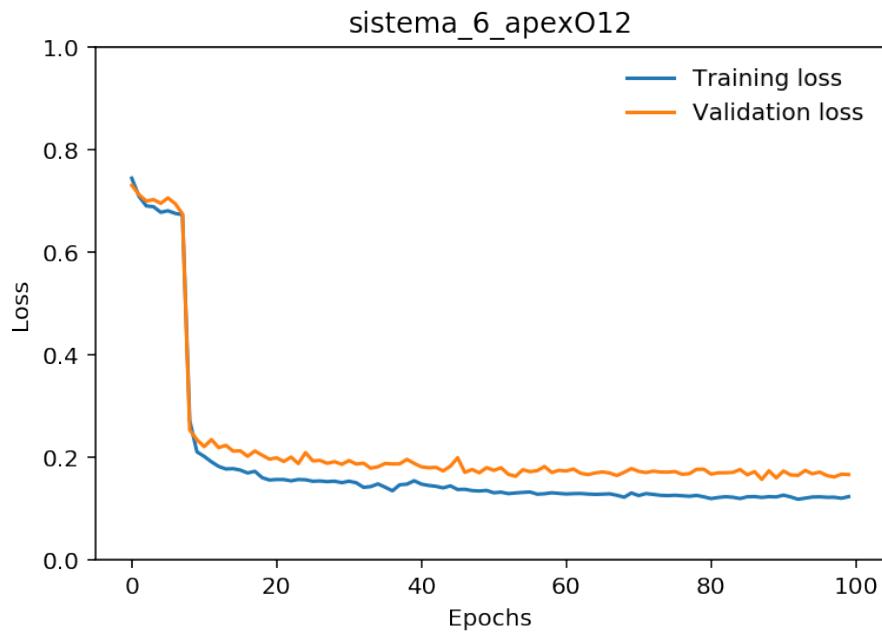


Figura 15.1: Pérdida de entrenamiento. 100 iteraciones.

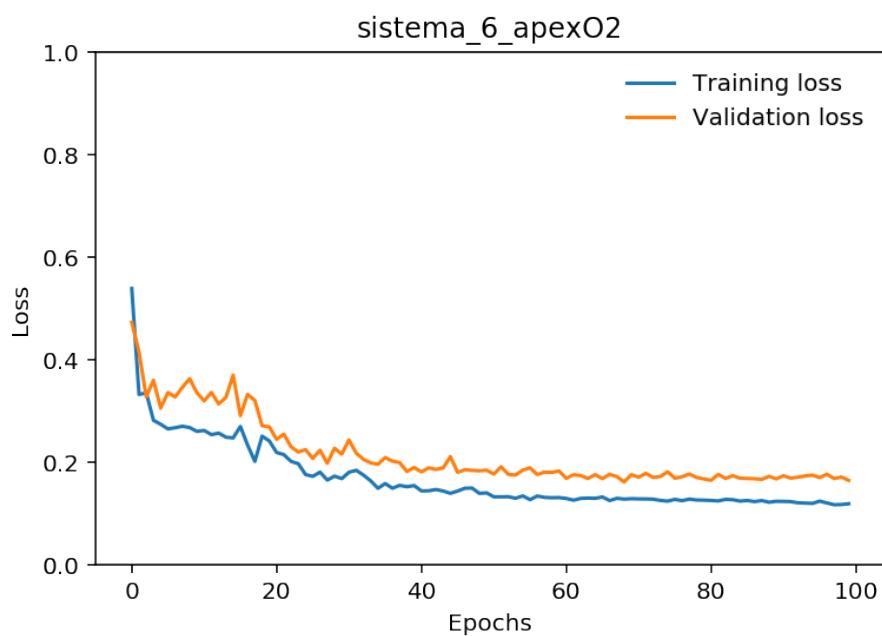


Figura 15.2: Pérdida de entrenamiento. 100 iteraciones.

De cualquier forma, el uso de un escalado dinámico de pérdida ha hecho que en 100 iteraciones se alcance un error de entrenamiento y validación menor, lo que ha provocado un IoU mayor. Aún así, gracias a la nueva métrica utilizada (*wrongCells*), se puede ver un comportamiento anómalo en las métricas de precisión mixta O2 (tabla 15.3). Pese a obtener buen IoU se ha obtenido en cada prueba un recuento de células muy distintos, etiquetando 11 células incorrectamente en la prueba 1 (el mejor resultado hasta ahora) y 74 células en la prueba 2 (el peor resultado hasta ahora). Se va a optar por la precisión mixta O1 por aportar resultados más consistentes (cuadro 15.2).

Parte IV

Cierre

CAPÍTULO 16

Análisis temporal y de costes

16.1– Análisis temporal

Fecha de inicio	20/02/2020
Fecha de fin prevista	14/06/2020
Fecha de fin real	4/12/2020
Horas de trabajo previstas	300
Horas de trabajo real	

Cuadro 16.1: Tabla de resumen de la planificación.

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

Cuadro 16.2: Tabla de tiempos

16.2– Resumen

Concepto	Coste previsto	Coste real
Coste de personal	5074,22	
Coste de servidores	65,81	
Costes indirectos	660,36	1650,9
Total	5800,39	

Cuadro 16.3: Tabla de resumen de los costes.

16.2.1. Costes directos

Costes de personal

En los costes de personal se ha incluido el salario de los trabajadores involucrados en el proyecto.

Tras investigar salarios actuales se ha concluido que un junior en el ámbito de inteligencia artificial (Ingeniero de Machine Learning, Ingeniero de Inteligencia Artificial, Ingeniero de Datos, Ingeniero de Big Data) puede cobrar 25000€ brutos anuales. Según el BOE (*Bases y tipos de cotización 2019*, s.f.) la empresa deberá pagar un 29,90% (23,60% contingencias comunes + 5,50% desempleo + 0,20% FOGASA + 0,60% formación profesional) del salario como coste a la Seguridad Social. Haciendo un total de $25000\text{€} * 1,299 = 32475\text{€}$ anual.

Suponiendo que el sueldo se cobra en 12 pagas y que la jornada laboral es de 160 horas, el coste por hora para la empresa sería: $32475\text{€} / (12 * 160) = 16,914\text{€}$.

Al ser un proyecto de 300 horas, los costes de personal en total son $16,914\text{€} * 300 = 5074,22\text{€}$

Costes de servidores

Al tratarse este proyecto de una aplicación de visión artificial con imágenes de alta definición y deep learning, el coste computacional ha sido muy alto. Es por ello que para el desarrollo de este sistema ha sido necesario el uso de una GPU con al menos 16GB de memoria (VRAM) para poder llegar a unos resultados mínimos. Al no disponer de ninguna máquina en físico que cumpla este requisito, se ha optado por alquilar servidores virtuales.

La computación se ha realizado en una máquina con la GPU P5000, con un valor de mercado de 1890,63€ (*PNY Quadro P5000*, s.f.) pudiéndose alquilar por 0,78\$/hora (*Instance Types*, s.f.). Aunque tenga estos costes, se ha aprovechado que la web en la que se puede alquilar también ofrece una capa gratuita en la que se puede usar esa GPU de forma indefinida con algunas limitaciones. Aún así se ha hecho el cálculo sin tener esta capa gratuita.

El servidor se ha usado aproximadamente durante 100 horas, usándose para desarrollar, entrenar y hacer inferencia. El coste sería de $0,78 * 100 = 78\$$, que en el momento de escribir esto equivale a 65,81€.

16.2.2. Costes indirectos

Se tomará como costes indirectos el uso de material informático en físico y costes varios.

Se ha utilizado el ordenador portátil MSI GP62 7RE Leopard Pro con un valor actual de 905,59€ (*MSI GP62 7RE*, s.f.). Suponiendo que un portátil tiene una esperanza de vida de 5 años, cada mes que se use en el proyecto supondrá un coste de $905,59 / (12 * 5) = 15,09\text{€}$.

Como costes varios se tomará la electricidad, el uso de periféricos, el coste de la oficina y otros. Debido a la incertidumbre al hacer este cálculo se supondrá un gasto de 150€ al mes.

Según el plan en el que se estimaban 4 meses, el coste sería de $4 * (15,09 + 150) = 660,36\text{€}$. En la realidad el proyecto ha tomado 10 meses, por lo que el coste real sería $10 * (15,09 + 150) = 1650,9\text{€}$.

Referencias

- Apachespark - unified analytics engine for big data.* (s.f.). Descargado de <https://spark.apache.org/>
- Bases y tipos de cotización 2019.* (s.f.). Descargado de <http://www.seg-social.es/wps/portal/wss/internet/Trabajadores/CotizacionRecaudacionTrabajadores/36537#top>
- Cardoso, M. J., Arbel, T., Carneiro, G., Syeda-Mahmood, T., Tavares, J., Moradi, M., ... Lu, Z. (2017). *Deep learning in medical image analysis and multimodal learning for clinical decision support: Third international workshop, dlmia 2017, and 7th international workshop, ml-cds 2017, held in conjunction with miccai 2017, québec city, qc, canada, september 14, proceedings.* doi: 10.1007/978-3-319-67558-9
- Collette, A. (2013). *Python and hdf5.* O'Reilly.
- Convolution.* (2020, Nov). Wikimedia Foundation. Descargado de <https://en.wikipedia.org/wiki/Convolution>
- 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
- Gómez-Gálvez, P., Vicente-Munuera, P., Tagua, A., Forja, C., Castro, A. M., Letrán, M., ... Escudero, L. M. (2018, 27 de Jul). Scutoids are a geometrical solution to three-dimensional packing of epithelia. *Nature Communications*, 9(1), 2960. Descargado de <https://doi.org/10.1038/s41467-018-05376-1> doi: 10.1038/s41467-018-05376-1
- 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
- Howard, J., y cols. (2018). *fastai.* <https://github.com/fastai/fastai>. GitHub.
- Instance types.* (s.f.). Descargado de <https://docs.paperspace.com/gradient/instances/instance-types>
- Jadon, S. (2020). *A survey of loss functions for semantic segmentation.*
- Jeong, J., Yoon, T., y Park, J. (2018, 08). Towards a meaningful 3d map using a 3d lidar and a camera. *Sensors*, 18, 2571. doi: 10.3390/s18082571

- Jia, Y., Shelhamer, E., Donahue, J., Karayev, S., Long, J., Girshick, R., ... Darrell, T. (2014). Caf- fe: 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
- Micikevicius, P., Narang, S., Alben, J., Diamos, G., Elsen, E., Garcia, D., ... Wu, H. (2018). *Mixed precision training*.
- Minsky, M., y Papert, S. (1969). *Perceptrons: An introduction to computational geometry*. Cambridge, MA, USA: MIT Press.
- missinglink.ai. (2020). *missinglink.ai*. Descargado 2020-09-03, de <https://missinglink.ai/guides/convolutional-neural-networks/convolutional-neural-network-tutorial-basic-advanced/>
- Msi gp62 7re.* (s.f.). Descargado de <https://www.pcccomponentes.com/msi-gp62-7re-431xes-leopard-pro-intel-core-i7-7700hq-8gb-1tb-gtx1050ti-156>
- Nair, V., y Hinton, G. E. (2010). Rectified linear units improve restricted boltzmann machines. En *Proceedings of the 27th international conference on international conference on machine learning* (p. 807–814). Madison, WI, USA: Omnipress.
- Noh, H., Hong, S., y Han, B. (2015). *Learning deconvolution network for semantic segmentation*.
- nVidia Apex. (2020). *A PyTorch Extension: Tools for easy mixed precision and distributed training in Pytorch*. Descargado 2020-08, de <https://github.com/NVIDIA/apex>
- Nwankpa, C., Ijomah, W., Gachagan, A., y Marshall, S. (2018). *Activation functions: Comparison of trends in practice and research for deep learning*.
- Paperspace. (2020). *Paperspace*. Descargado de <https://gradient.paperspace.com/free-gpu>
- Paszke, A., Gross, S., Massa, F., Lerer, A., Bradbury, J., Chanan, G., ... Chintala, S. (2019). Pytorch: An imperative style, high-performance deep learning library. En *Advances in neural information processing systems* 32 (pp. 8024–8035). Curran Associates, Inc. Descargado de <http://papers.neurips.cc/paper/9015-pytorch-an-imperative-style-high-performance-deep-learning-library.pdf>
- Pedregosa, F., Varoquaux, G., Gramfort, A., Michel, V., Thirion, B., Grisel, O., ... Duchesnay, E. (2011). Scikit-learn: Machine learning in Python. *Journal of Machine Learning Research*, 12, 2825–2830.
- Pérez-García, F., Sparks, R., y Ourselin, S. (2020, marzo). TorchIO: a Python library for efficient loading, preprocessing, augmentation and patch-based sampling of medical images in deep learning. *arXiv:2003.04696 [cs, eess, stat]*. Descargado 2020-03-11, de <http://arxiv.org/abs/2003.04696> (arXiv: 2003.04696)
- Pny quadro p5000.* (s.f.). Descargado de <https://www.amazon.es/PNY-Quadro-P5000-16GB-GDDR5X/dp/B01LWVP480>
- Ramachandran, P., Zoph, B., y Le, Q. V. (2017). *Searching for activation functions*.
- Riba, E., Mishkin, D., Ponsa, D., Rublee, E., y Bradski, G. (2020). Kornia: an open source differentiable computer vision library for pytorch. En *Winter conference on applications of computer vision*. Descargado de <https://arxiv.org/pdf/1910.02190.pdf>
- Ronneberger, O., Fischer, P., y Brox, T. (2015). *U-net: Convolutional networks for biomedical*

- image segmentation.
- Rosenblatt, F. (1958). The perceptron: A probabilistic model for information storage and organization in the brain. *Psychological Review*, 65–386.
- Rumelhart, D. E., Hinton, G. E., y Williams, R. J. (1986, 01 de Oct). Learning representations by back-propagating errors. *Nature*, 323(6088), 533-536. Descargado de <https://doi.org/10.1038/323533a0> doi: 10.1038/323533a0
- Schindelin, J., Arganda-Carreras, I., Frise, E., Kaynig, V., Longair, M., Pietzsch, T., ... Cardona, A. (2012, 01 de Jul). Fiji: an open-source platform for biological-image analysis. *Nature Methods*, 9(7), 676-682. Descargado de <https://doi.org/10.1038/nmeth.2019> doi: 10.1038/nmeth.2019
- Serrano, L. (2020). *Deep Learning with PyTorch*. Descargado 2020-09-03, de <https://classroom.udacity.com/courses/ud188>
- shiba24. (2017). *3d unet implementation*. <https://github.com/shiba24/3d-unet>. GitHub.
- Simonyan, K., y Zisserman, A. (2014). *Very deep convolutional networks for large-scale image recognition*.
- Sultana, F., Sufian, A., y Dutta, P. (2020, agosto). Evolution of image segmentation using deep convolutional neural network: A survey. *Knowledge-Based Systems*, 201-202, 106062. Descargado de <http://dx.doi.org/10.1016/j.knosys.2020.106062> doi: 10.1016/j.knosys.2020.106062
- Szegedy, C., Liu, W., Jia, Y., Sermanet, P., Reed, S., Anguelov, D., ... Rabinovich, A. (2014). *Going deeper with convolutions*.
- The HDF Group. (2000-2010). *Hierarchical data format version 5*. Descargado de <http://www.hdfgroup.org/HDF5>
- Tiobe. (2020). *Tiobe index*. Descargado de <https://www.tiobe.com/tiobe-index/>
- Van Der Walt, S., Colbert, S. C., y Varoquaux, G. (2011). The numpy array: a structure for efficient numerical computation. *Computing in Science and Engineering*, 13(2), 22.
- van der Walt, S., Schönberger, J. L., Nunez-Iglesias, J., Boulogne, F., Warner, J. D., Yager, N., ... the scikit-image contributors (2014, 6). scikit-image: image processing in Python. *PeerJ*, 2, e453. Descargado de <https://doi.org/10.7717/peerj.453> doi: 10.7717/peerj.453
- Virtanen, P., Gommers, R., Oliphant, T. E., Haberland, M., Reddy, T., Cournapeau, D., ... Contributors, S. . . (2020). SciPy 1.0: Fundamental Algorithms for Scientific Computing in Python. *Nature Methods*, 17, 261–272. doi: <https://doi.org/10.1038/s41592-019-0686-2>
- Widrow, B., y Hoff, T. (2015). Adaptive linear neuron..
- Wolny, A., Cerrone, L., Vijayan, A., Tofanelli, R., Barro, A. V., Louveaux, M., ... Kreshuk, A. (2020). Accurate and versatile 3d segmentation of plant tissues at cellular resolution. *bioRxiv*. Descargado de <https://www.biorxiv.org/content/early/2020/01/18/2020.01.17.910562> doi: 10.1101/2020.01.17.910562
- Wookayin. (s.f.). *wookayin/gpustat*. Descargado de <https://github.com/wookayin/gpustat>
- Zeiler, M. D., Taylor, G. W., y Fergus, R. (2011). Adaptive deconvolutional networks for mid and high level feature learning. En *2011 international conference on computer vision* (p. 2018-2025).
- Özgün Çiçek, Abdulkadir, A., Lienkamp, S. S., Brox, T., y Ronneberger, O. (2016). *3d u-net: Learning dense volumetric segmentation from sparse annotation*.
- Z-normalization of time series. (s.f.). Descargado de <https://jmotif.github.io/sax-vsm-site/morea/algorith/znorm.html>