



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
1.1 Motivación	3
1.2 Visita a Biología Molecular	3
2 Objetivos	6

II Organización del Proyecto

3 Planificación	9
3.1 Enfoque	9
3.2 Planificación inicial	9
3.3 Qué no se tuvo en cuenta	10
3.4 Nueva metodología	10
3.5 Planificación actualizada	11
3.5.1 Resumen	11
3.5.2 Fases	11
4 Costes	12
4.1 Resumen	12
4.2 Costes directos	12
4.2.1 Costes de personal	12
4.2.2 Costes de servidores	12
4.3 Costes indirectos	13

III Estudio previo

5 Red Neuronal Artificial	16
5.1 Introducción a Redes Neuronales Artificiales	16
5.2 Neurona artificial	17

5.2.1	Sigmoide	18
5.2.2	Unidad Lineal Rectificada (ReLU)	18
5.3	Red Neuronal Artificial	19
5.4	Descenso por Gradiente	19
6	Red Neuronal Convolutacional	21
6.1	Tipos de capas	22
6.2	Capa convolutacional	23
6.2.1	Capa Pooling	25
6.2.2	Capa ReLU	25
6.2.3	Capa FC	26
6.3	Funciones de Pérdida	26
6.3.1	Binary Cross-Entropy	26
6.3.2	Weighted Binary Cross-Entropy	26
6.3.3	Dice Loss	26
7	Arquitecturas para segmentación semántica	27
7.1	Fully Convolutional Network	28
7.2	Deconvnet	28
7.2.1	Unpooling	29
7.2.2	Deconvoloución	29
7.3	U-Net	29
8	Aplicaciones recientes	31
8.1	U-Net para conteo, detección y morfometría, detección y morfometría de células. (2019)	31
8.1.1	Proyecto	31
8.1.2	Pipeline	31
8.1.3	Resultados	32
8.1.4	Software	32
8.2	Segmentación 3D precisa y versátil de tejido vegetal a resolución celular. (2020)	33
8.2.1	Proyecto	33
8.2.2	Pipeline	33
8.2.3	Resultados	34
8.2.4	Software	35

IV Desarrollo

9	Guía de capítulos de desarrollo	38
10	De computación local a computación en la nube	39
10.1	Introducción	39
10.2	Computación local	39
10.3	Computación en la nube	40
11	Diseño	42
11.1	Esquema general	42
11.2	Preprocesado local	44
11.3	Entrenamiento	46
11.4	Inferencia	47
11.5	Arquitectura de las Redes Neuronales Convolucionales	48

12 Implementación	50
12.1 Lenguaje y framework	50
12.2 Preprocesado local	51
12.2.1 Consideraciones	51
12.3 Carga de datos	52
12.3.1 Consideraciones	53
12.4 U-Net	54
12.5 Postprocesado	56
13 Representación 3D	57

V Cierre

14 Resultados	60
15 Análisis temporal	68
16 Conclusiones	69
17 Manual	70
Referencias	71

Índice de cuadros

3.1	Tabla de resumen de la planificación.	11
4.1	Tabla de resumen de los costes.	12
8.1	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.	31
8.2	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.	33
8.3	Pruebas relevantes realizadas.	35
15.1	Tabla de tiempos	68

Índice de figuras

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.	4
1.2	Esta figura muestra el resultado de aplicar el procesado a la imagen 3D obtenida por microscopía, la imagen tiene las dimensiones $1024 \times 1024 \times 234$, 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 $Z=100$	5
5.1	Neurona artificial genérica	17
5.2	Representación de una Neurona Artificial con $\sum x_i w_i$ como función de integración	18
5.3	Red Neuronal Artificial completamente conectada.	19
5.4	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.	20
6.1	Imagen de 4×4 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.	21
6.2	Imagen de 4×4 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 4×4 como entrada. Imágenes del curso Intro to Deep Learning with PyTorch de Udacity.	22
6.3	Red Neuronal Convolutiva simple en la que la imagen de un barco es clasificada.	22
6.4	Capa de convolución de profundidad 2 (se usan dos filtros) con kernel de tamaño $3 \times 3 \times 3$ aplicado a un volumen de entrada de tamaño $7 \times 7 \times 3$, volumen generado al aplicar padding de 1 px a una imagen de tamaño 5×5 con 3 canales. El resultado es un volumen de $3 \times 3 \times 2$	24
6.5	Imagen 4×4 a la que se ha aplicado un pooling de $F = 2, S = 2$. Cada color de la imagen de la izquierda indica las entradas del para la operación MAX, cada color de la imagen de la derecha indica la salida de dicha operación. Figura tomada del curso CS231n de Stanford University.	25
7.1	Arquitectura de FCN32, FCN16 y FCN8.	28
7.2	Arquitectura Deconvnet.	29
7.3	Figura que ilustra las operaciones unpooling y deconvolución.	29

7.4	Arquitectura original U-Net propuesta por Ronneberger et al en U-Net: Convolutional Networks for Biomedical Image Segmentation.	30
8.1	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.	33
8.2	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.	34
8.3	Interfaz gráfica del programa PlantSet. Se pueden ver las distintas opciones para cada paso del procesado.	35
10.1	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.	40
10.2	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.	41
11.1	Diseño general	43
11.2	Preprocesado llevado a cabo en la máquina local.	44
11.3	Pasos llevados a cabo en el entrenamiento	46
11.4	Se muestran dos métodos distintos para obtener la imagen correctamente etiquetada.	47
11.5	Arquitectura U-Net completa.	48
11.6	Arquitectura U-Net reducida. Esta arquitectura no tiene mucho interés excepto para completar un primer sistema.	49
14.1	Modelo miniunet2.1. 100 epochs. Se usa MiniUnet3D. Imagen objetivo con espaciado entre células. Sin data augmentation.	61
14.2	Modelo unet4.1. 200 epochs. Imagen objetivo con espaciado entre células. Sin data augmentation.	61
14.3	Modelo unet4.13. 200 epochs. Imagen objetivo con espaciado entre células. Sin data augmentation.	62
14.4	Modelo unet4.8. 200 epochs. Imagen objetivo con espaciado entre células. Con data augmentation.	62
14.5	Modelo unet4.8. 200 epochs. Imagen objetivo con espaciado entre células. Con data augmentation. Sin Apex	63
14.6	Modelo unet5.2. 200 epochs. Imagen objetivo con espaciado entre células. Con data augmentation. Con Apex.	63
14.7	Modelo unet6t. 100 epochs. Imagen objetivo con espaciado entre células. Con data augmentation. Pérdida Dice. Sin postprocesado. Primera fila imagen de entrada. Segunda fila segmentación objetivo. Tercera fila predicción. Z=20,25,45,50 en las columnas. IoU 0.72	64
14.8	Modelo unet6t. 100 epochs. Imagen objetivo con espaciado entre células. Con data augmentation. Sin postprocesado.	64
14.9	Modelo unet6b. 100 epochs. Imagen objetivo bordes de células. Con data augmentation. Pérdida Dice. Sin postprocesado. Primera fila imagen de entrada. Segunda fila segmentación objetivo. Tercera fila predicción. Z=20,25,45,50 en las columnas. IoU 0.84	65
14.10	Modelo unet6b. 100 epochs. Imagen objetivo bordes de células. Con data augmentation. Pérdida Dice. Sin postprocesado.	65
14.11	DT Watershed aplicado a la salida del modelo unet6b. Primera fila bordes. Segunda fila transformación de la distancia. Filtro gaussiano. Cuarta fila watershed con mínimos locales como semilla. Z=20,25,45,50 en las columnas.	66

14.12	Modelo unet6b. 100 epochs. Imagen objetivo bordes de células. Con data augmentation. Pérdida Dice. Componentes conexas con etiquetas distintas. IoU 0.72 . . .	67
14.13	DT Watershed aplicado a la salida del modelo unet6b. IoU 0.70	67

Parte I

Introducción

Nacimiento del proyecto

1.1– Motivación

Uno de mis mayores intereses en el mundo del software siempre ha sido el uso de técnicas de inteligencia artificial para resolver problemas actuales. Ya tuve un poco de experiencia en las prácticas extracurriculares con el procesamiento de lenguaje natural, aunque tenía un rol más centrado en la aplicación de determinadas técnicas más que en el desarrollo de estas.

Aproveché que estaba cursando la asignatura de Procesamiento de Imágenes Digitales con 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. Mi principal motivación era que nunca había trabajado con imágenes digitales en el ámbito de la inteligencia artificial (que más tarde descubrí que es más correcto hablar de deep learning en vez de inteligencia artificial en estos casos).

María José me habló sobre un proyecto muy interesante dirigido por 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. El reto 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.

1.2– Visita a Biología Molecular

Tras ver que este proyecto era interesante planeamos una primera visita al Departamento de Biología Celular en el que fuimos recibidos por Luis María Escudero, Pablo Vicente-Manuera y Pedro Gómez-Gálvez. En esta reunión nos hablaron en más detalle sobre su investigación y el problema que queríamos abordar.

Estudiaban 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 este. Aunque las células de este tejido siempre se han representado en forma de prisma (cuando las células de la misma capa están en el mismo plano) o en forma de tronco (cuando se produce un pliegue), han demostrado que en realidad tienen siempre una nueva forma geométrica que han nombrado escutoide (Gómez-Gálvez y cols., 2018). En la figura 1.1 (Gómez-Gálvez y cols., 2018) se pueden ver estas formas.

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 manualmente utilizando herramientas de MATLAB y Fiji (ImageJ) para conseguir una imagen con todas las células etiquetadas correctamente, proceso conocido como segmentación.

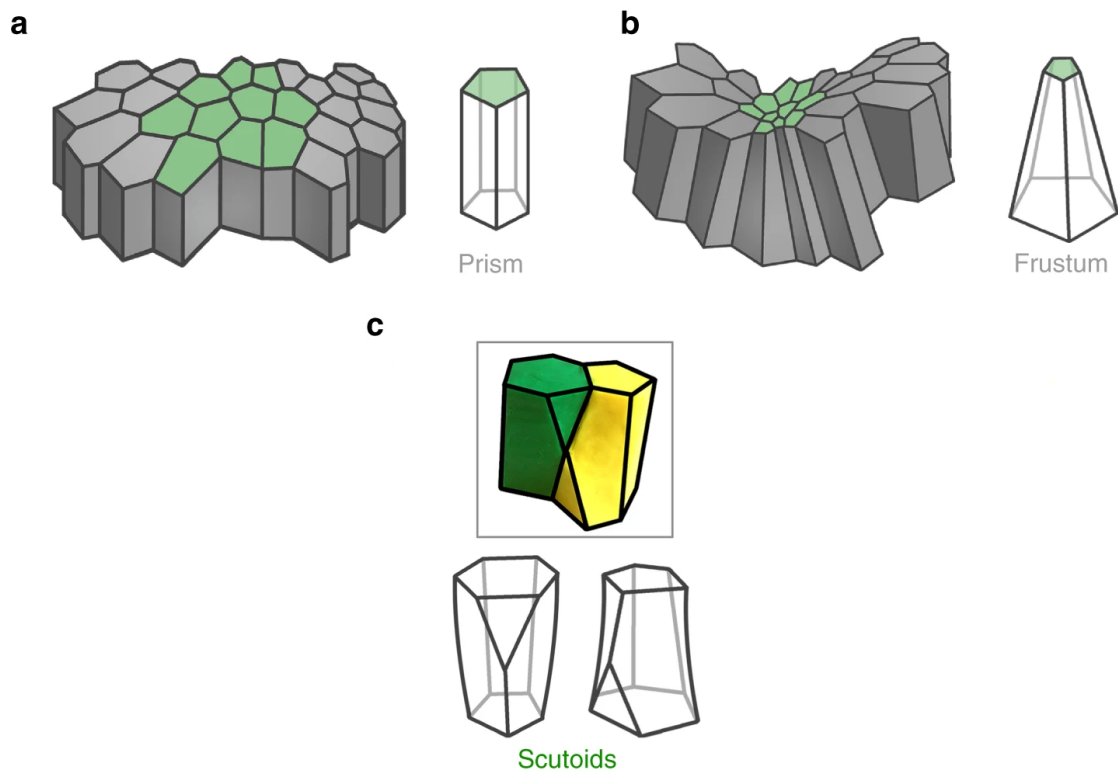


Figura 1.1: **a** Representación de las células del tejido epitelial cuando están en una estructura plana. Se suele representar en forma de prisma. **b** Representación de las células del tejido epitelial cuando se produce un pliegue. Se suele representar en forma de tronco. **c** Geometría propuesta 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.

El resultado final sería una imagen en la que cada vóxel del fondo tiene valor 0 y los vóxeles pertenecientes a cada célula tienen valor 1, 2, 3... n , donde n es el nº de células en la imagen. Un ejemplo de la imagen original y la imagen etiquetada puede verse en la figura 1.2

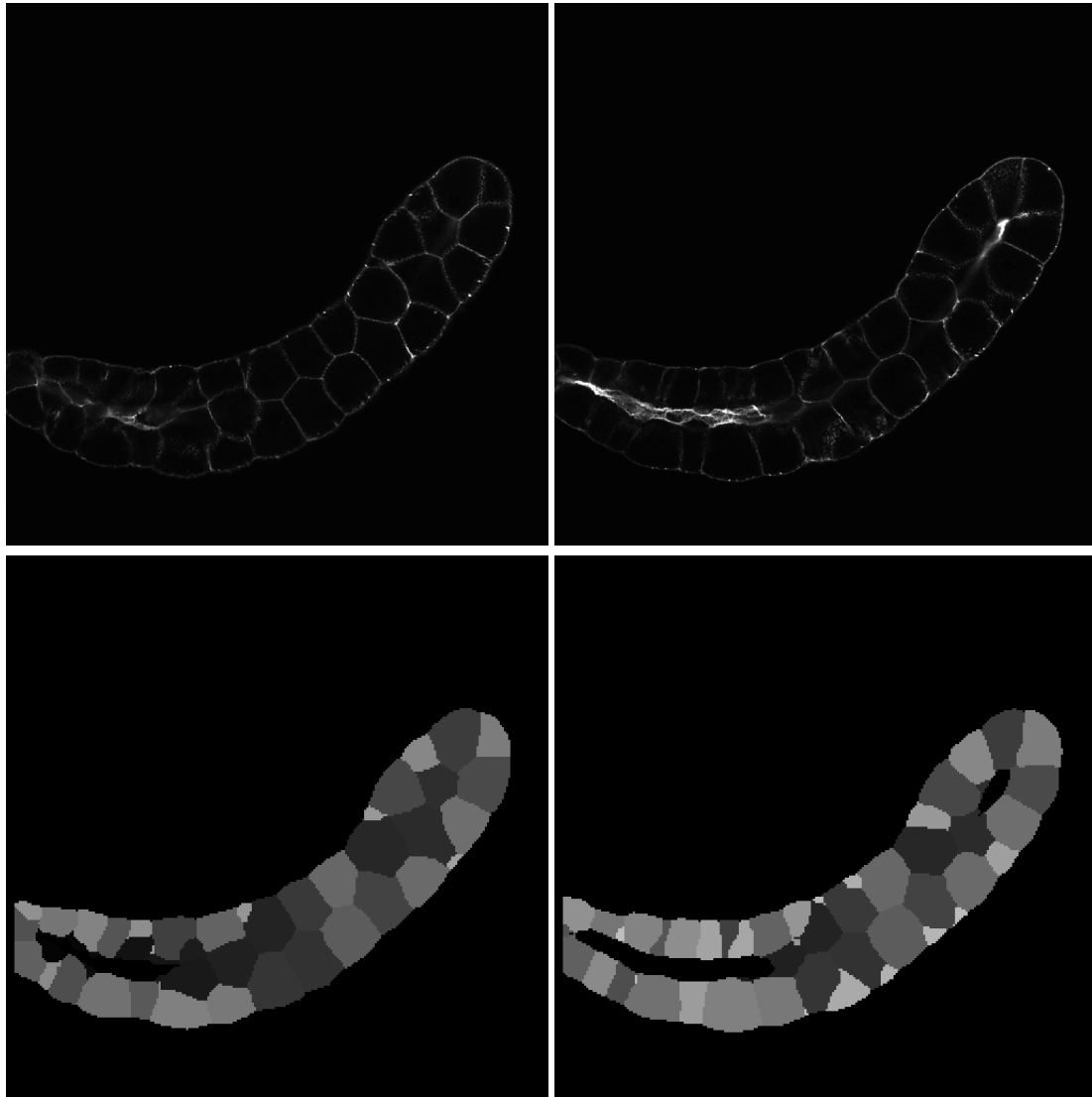


Figura 1.2: Esta figura muestra el resultado de aplicar el procesamiento a la imagen 3D obtenida por microscopía, la imagen tiene las dimensiones $1024 * 1024 * 234$, seleccionándose para su representación $Z=77$ y $Z=100$. En la primera fila se muestra la imagen obtenida por microscopía y la segunda 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 $Z=100$.

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

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. En las etapas iniciales del proyecto, tal y como se verá en el capítulo ?? hubo un cambio de rumbo en el alcance del proyecto, es por ello que se añadió un objetivo relacionado con el coste de computación. En definitiva, los objetivos son los siguientes:

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 data-sets.
3. Debido a las dificultades encontradas en el primer acercamiento al problema, 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.

Parte II

Organización del Proyecto

Planificación

3.1– Enfoque

Uno de los motivos por los que me pareció interesante este proyecto era porque nunca había aplicado técnicas de machine learning para resolver problemas reales, muchos menos de imágenes 3D. Desgraciadamente al principio no abordé el proyecto teniendo esto en cuenta, por lo que hubo una corrección en el plan que se verá en la sección 3.3. Con esto en mente, mis razones para efectuar este plan inicial son las siguientes:

- En algunas asignaturas y en mi tiempo libre he utilizado técnicas sencillas de inteligencia artificial.
- Tengo conocimiento sobre el funcionamiento de una red neuronal. Conozco los conceptos de neurona artificial, capas de la red, pérdida o descenso por gradiente.
- Necesito encontrar una arquitectura de red neuronal adecuada para resolver el problema dado.
- El uso de la GPU aumentará mucho la velocidad con la que se realizan las operaciones.

3.2– Planificación inicial

En este primer plan dividí 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 ???. El uso de Python se explica en la sección 12.1.

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 Spark.

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

3.3– Qué no se tuvo en cuenta

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 utilizado 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 como se verá en la sección [REDACTED] esto no es viable.

3.4– Nueva metodología

Cuando se estaba siguiendo la **Fase 2** del plan inicial se vieron los problemas antes mencionados, por lo que 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 tus objetivos. Los pasos a seguir propuestos por esta metodología son:

1. Determinar tus 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* y constituirá la mayor parte del proyecto.

3.5– Planificación actualizada

3.5.1. Resumen

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 3.1: Tabla de resumen de la planificación.

3.5.2. Fases

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 3.4.

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.

Costes

4.1– 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 4.1: Tabla de resumen de los costes.

4.2– Costes directos

4.2.1. Costes de personal

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

Tras investigar salarios actuales he 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 $25000€ \cdot 1,299 = 32475€$ anual.

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

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

4.2.2. Costes de servidores

Al tratarse este proyecto de una aplicación de visión artificial con imágenes de alta definición y deep learning, el coste computacional es muy alto. Es por ello que para el desarrollo de este sistema es necesario el uso de una GPU con 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 (especialmente ya que este proyecto ha sido desarrollado en 2020 con las limitaciones que eso conlleva), se ha optado por alquilar servidores virtuales.

El desarrollo se hará en una máquina con la tarjeta gráfica P5000, con un valor actual de mercado de 1890,63€ (*PNY Quadro P5000*, s.f.) que se puede alquilar por 0,78\$/hora (*Instance Types*, s.f.). Aunque tenga estos costes, se aprovechará que en la web 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 hará el cálculo sin tener esta capa gratuita.

Suponiendo que se usará este servidor durante 100 horas, ya sea para desarrollar, entrenar o hacer inferencia, el coste sería de $0,78 * 100 = 78\$$, que a día de hoy son 65,81€.

4.3– 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 $10 * (15,09 + 150) = 1650,9\text{€}$.

Parte III

Estudio previo

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: perceptrón, sigmoide y unidad lineal rectificada (ReLU). Estos serán los tipos de neurona artificial más frecuentes en los algoritmos usados en este proyecto. Por último se hablará sobre el entrenamiento de la ANN.

5.1– 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 hicieron importantes contribuciones como la representación distribuida (G. E. Hinton, 1986), donde

se habla sobre representar las entradas de un sistema en base a sus características, reconocidas por patrones de actividad en redes neuronales. Otra gran contribución fue la popularización del algoritmo de propagación hacia atrás, **backpropagation** (Rumelhart, Hinton, y Williams, 1986) para entrenar redes neuronales artificiales y actualizar sus pesos, siendo este algoritmo el más usado en la actualidad.

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

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

5.2– Neurona artificial

En la figura 5.1 se puede ver una neurona artificial genérica con la que pueden ser descritos los distintos tipos que se verán en esta sección.

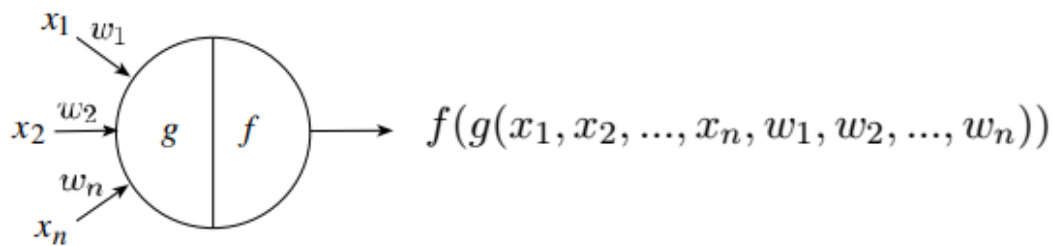


Figura 5.1: Neurona artificial genérica

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 5.3 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 $n + 1$.

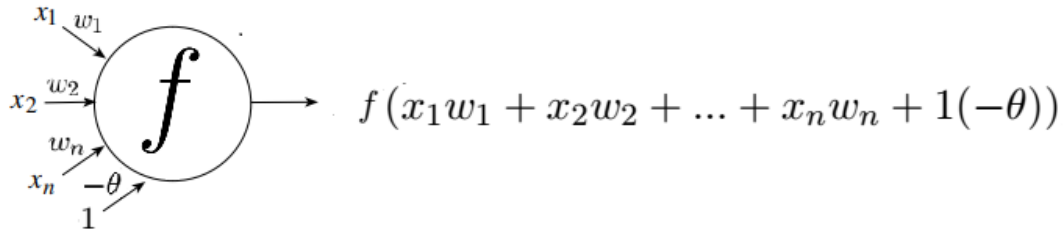


Figura 5.2: Representación de una Neurona Artificial con $\sum x_i w_i$ como función de integración

- 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.

5.2.1. Sigmoide

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

$$f(x) = \sigma(x) = \frac{1}{1 + e^{-x}} \quad (5.1)$$

5.2.2. 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 Sigmoide, ofrece un mejor rendimiento y es más generalista (Nwankpa y cols., 2018).

$$f(x) = \max(0, x) = \begin{cases} x_i & \text{si } x_i \geq 0 \\ 0 & \text{si } x_i < 0 \end{cases} \quad (5.2)$$

5.3– Red Neuronal Artificial

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

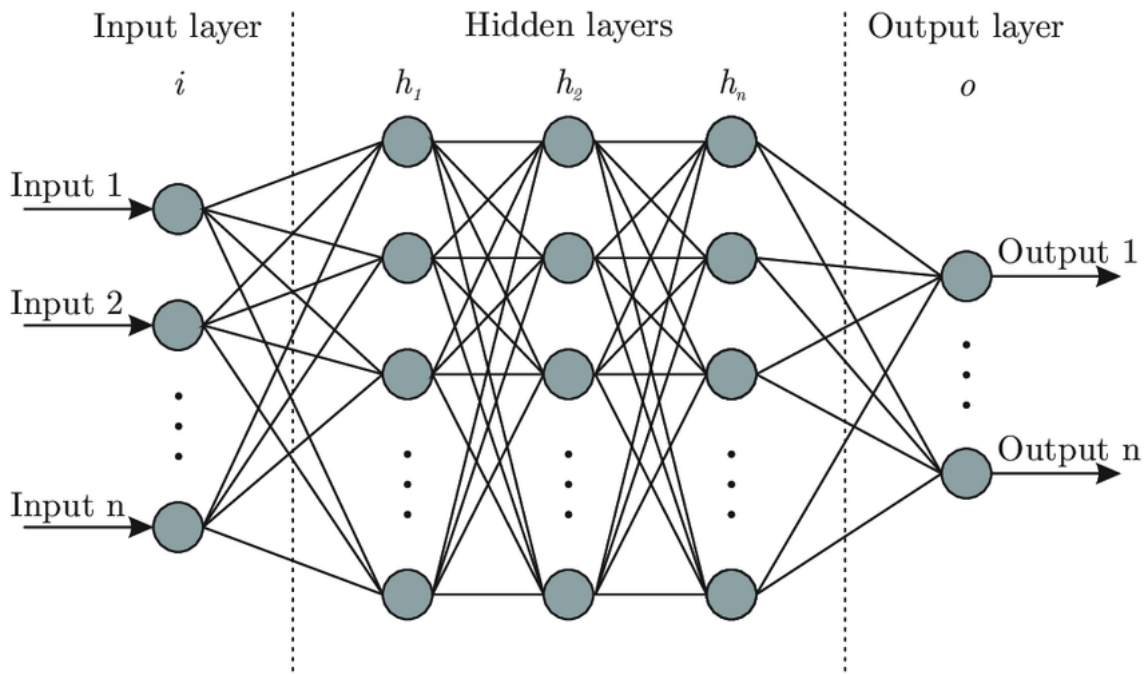


Figura 5.3: Red Neuronal Artificial completamente conectada.

En la figura 5.3 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 n° de nodos distintos. Es en estas capas donde se reconocen los patrones del conjunto de datos y tiene el mayor coste computacional. La última capa oculta está conectada con las entradas de la capa de salida.
- Capa de salida o (*output layer*) con c nodos. La última capa de la red neuronal, la salida de esta capa nos dará un vector de tamaño c como salida.

5.4– Descenso por Gradiente

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

suponemos que tenemos una función que nos da el error cometido, nuestro objetivo será minimizar dicha función.

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

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

No es extraño en deep learning encontrar una red con millones de pesos pero, para facilitar la visualización, se ha usado como ejemplo una ANN con 2 pesos (θ_0 y θ_1). En la figura 5.4 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.

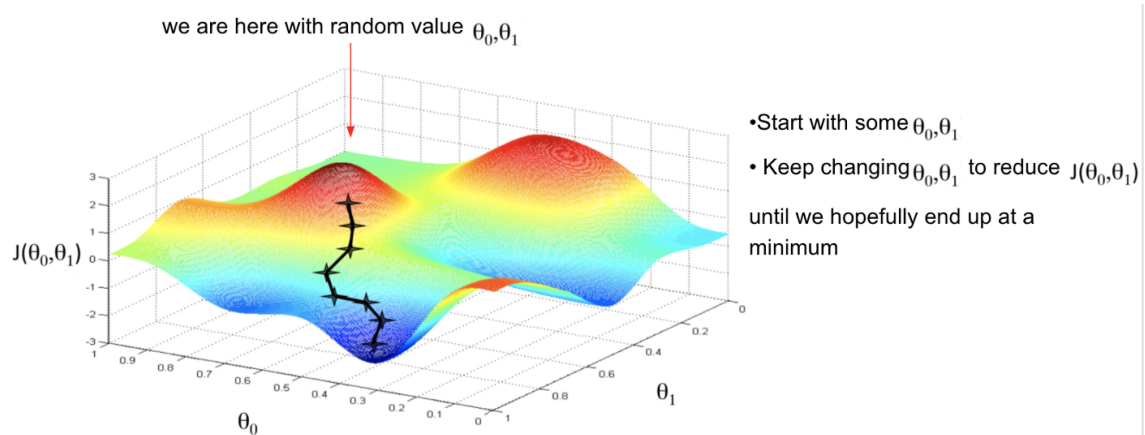


Figura 5.4: Ilustración del algoritmo de descenso por gradiente. θ_0 y θ_1 son los pesos de la ANN y J la función de pérdida. Se puede observar cómo se produce un "descenso" hacia un mínimo de la función.

Red Neuronal Convolutacional

Hasta ahora hemos supuesto que la entrada a una ANN es un vector, algo válido para un gran número de aplicaciones en deep learning. El problema está cuando la entrada de la red neuronal es una imagen. En este caso antes de utilizar la imagen como entrada hay que aplanarla para contener la imagen en un vector, de esta forma cada píxel (o vóxel) será un elemento del vector de entrada y estará conectado a cada neurona de la capa siguiente. Para el caso de imágenes pequeñas (como la de la figura 6.1) 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 $124 * 124 * 70 = 1076320 \text{ vox}$, se necesitarían más de un millón de pesos por cada neurona que haya en la primera capa oculta. Esto hace que sea completamente inviable usar este tipo de redes para imágenes a partir de cierto tamaño. En esta sección se presentan las redes neuronales convolucionales (CNN), que reducirán en gran medida el n° de pesos necesarios en la red neuronal y aprovecharán técnicas del procesamiento de imágenes para encontrar patrones.

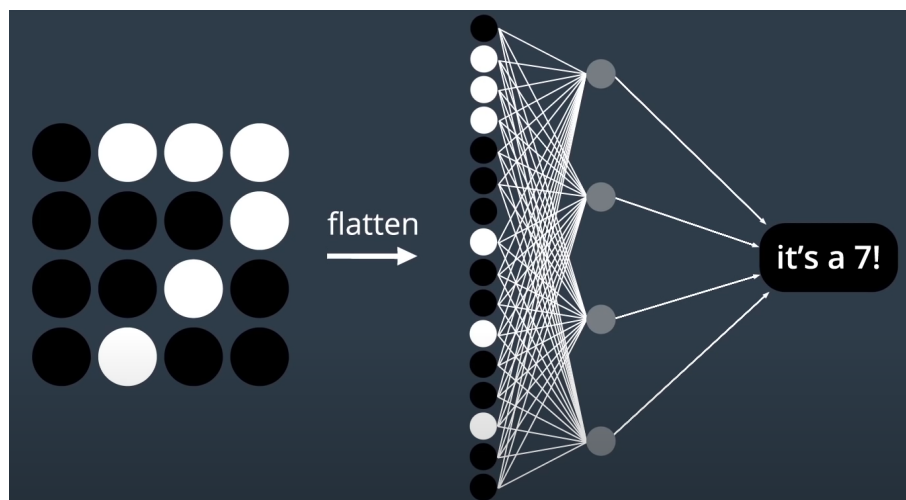


Figura 6.1: Imagen de $4 \times 4 \text{ px}$ aplanada y usada como entrada en una FCNN con 4 neuronas en su única capa oculta. No se muestra su capa de salida. Imagen del curso Intro to Deep Learning with PyTorch de Udacity.

Cambiando la arquitectura de la red vista en la figura 6.1 por una CNN, obtendríamos una arquitectura similar a la vista en la figura 6.2. En esta CNN se ha reducido el n° de conexiones de la capa de entrada a la capa oculta de 64 a 16, además, como veremos más adelante, los pesos de las 4 neuronas son compartidos, esto significará que sólo necesitamos 4 pesos distintos.

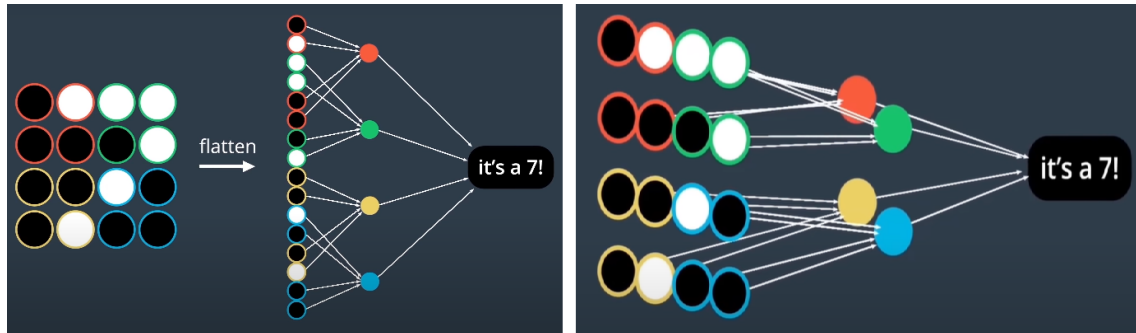


Figura 6.2: Imagen de 4x4 px usada como entrada en una CNN. Ambas figuras muestran la misma arquitectura, estando en la figura de la izquierda la imagen aplanada y en la figura de la derecha se muestra la matriz 4x4 como entrada. Imágenes del curso Intro to Deep Learning with PyTorch de Udacity.

6.1– Tipos de capas

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

Las capas más comunes usadas en una CNN son: Capa Convolucional, Capa de Pooling, Capa ReLU y Capa Completamente Conectada (FC). En la figura 6.3 (missinglink, 2020) se puede ver un ejemplo en el que la imagen de un barco pasa por varias capas de convolución + ReLU, Pooling y por último FC, dando la predicción de la clase de la imagen.

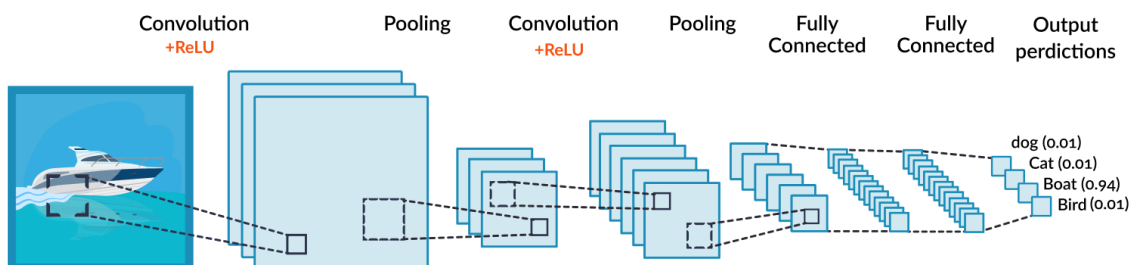


Figura 6.3: Red Neuronal Convolucional simple en la que la imagen de un barco es clasificada.

6.2– Capa convolutacional

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

- Número de filtros, **K**.
- Tamaño del kernel, **F**.
- Paso, **S**.
- Padding, **P**.

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

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

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

En la figura 6.4(Li, Krishna, y Xu, 2020) se muestra un ejemplo de una capa de convolución de profundidad 2 con imagen de entrada 5×5 con $1px$ de padding y profundidad 3, siendo los filtros de tamaño 3 y aplicándose con un paso de 3. Se obtendrá una imagen 3×3 con profundidad 2.

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

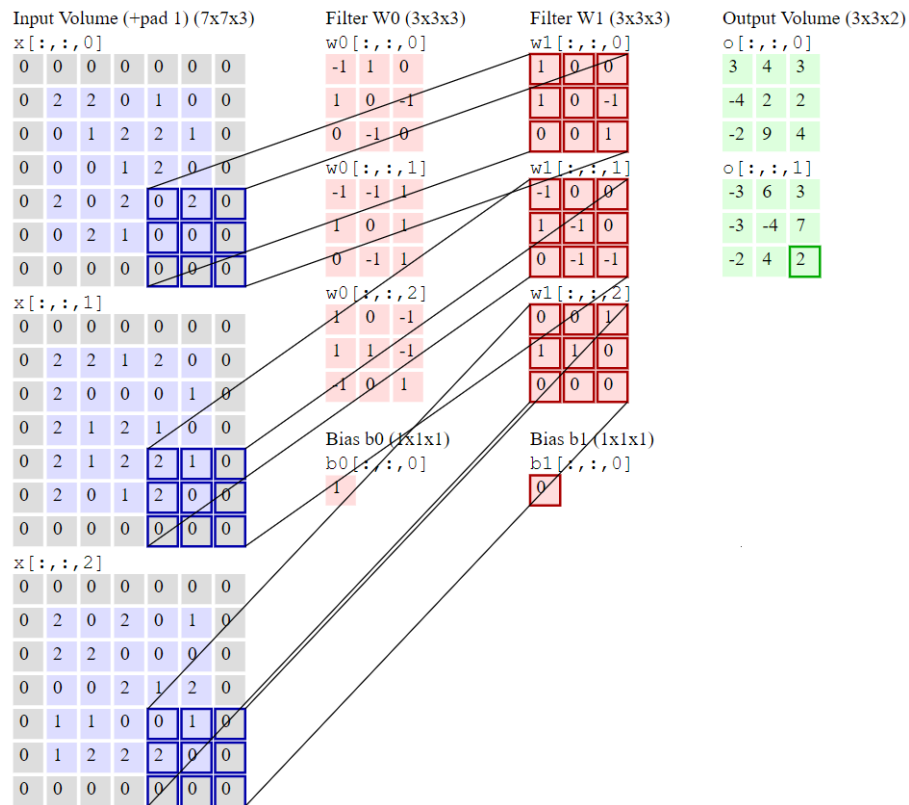


Figura 6.4: Capa de convolución de profundidad 2 (se usan dos filtros) con kernel de tamaño $3 \times 3 \times 3$ aplicado a un volumen de entrada de tamaño $7 \times 7 \times 3$, volumen generado al aplicar padding de 1 px a una imagen de tamaño 5×5 con 3 canales. El resultado es un volumen de $3 \times 3 \times 2$

6.2.1. Capa Pooling

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

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

Los parámetros necesarios para definir una capa de pooling son:

- Tamaño del kernel, F .
- Paso, S .

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

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

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.

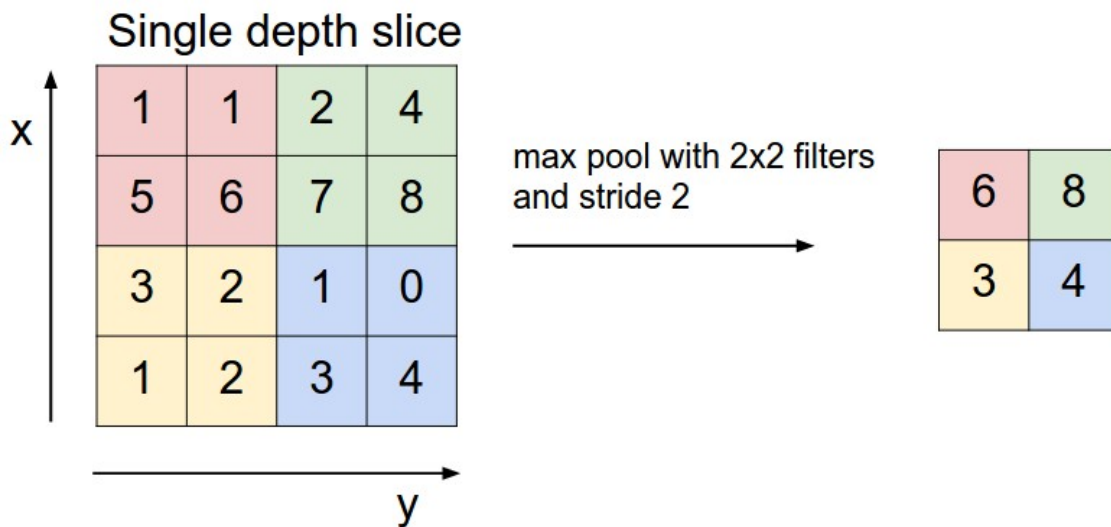


Figura 6.5: Imagen 4×4 a la que se ha aplicado un pooling de $F = 2, S = 2$. Cada color de la imagen de la izquierda indica las entradas del para la operación MAX, cada color de la imagen de la derecha indica la salida de dicha operación. Figura tomada del curso CS231n de Stanford University.

6.2.2. Capa ReLU

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

6.2.3. Capa FC

Se usa para obtener la puntuación de clase de cada píxel/vóxel de la capa anterior, a la que está completamente conectada (cada nodo de la capa anterior está conectado a todos los de esta capa). Es la salida de la red ya que es la última capa. En problemas de clasificación esta capa tendrá C nodos, siendo C el n° de clases. En problemas de segmentación esta capa tendrá tantos nodos como clases haya multiplicado por el n° píxeles/vóxeles que haya en la capa anterior, entendiéndose la segmentación como etiquetar cada píxel/vóxel con la probabilidad que tiene de pertenecer a cada clase.

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

6.3– Funciones de Pérdida

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

6.3.1. Binary Cross-Entropy

La entropía cruzada 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:

$$L_{BCE}(y, \hat{y}) = -(y \log(\hat{y}) + (1 - y) \log(1 - \hat{y})) \quad (6.1)$$

6.3.2. Weighted Binary Cross-Entropy

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

$$L_{WBCE}(y, \hat{y}) = -(\beta y \log(\hat{y}) + (1 - y) \log(1 - \hat{y})) \quad (6.2)$$

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

6.3.3. 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:

$$DL(y, \hat{p}) = 1 - \frac{2y\hat{p} + 1}{y + \hat{p} + 1} \quad (6.3)$$

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

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

Arquitecturas para segmentación semántica

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

- El ganador del reto de 2012 fue un equipo de la Universidad de Toronto con la arquitectura AlexNet (Krizhevsky, Sutskever, y Hinton, 2012), usando filtros 11x11 y siendo el primer modelo en usar ReLU como función de activación, algo que se ha convertido en estándar.
- En 2014 VGGNet (VGG), del Grupo de Geometría Visual de la Universidad de Oxford, fue de los que obtuvo mejor resultado en la competición con 2 versiones distintas, VGG16 con 16 capas y VGG19 con 19 capas (Simonyan y Zisserman, 2014). Ambas versiones emparejan convolución y pooling, usan filtros para las capas de convolución, 2x2 para las capas de pooling, acabando la red en 3 capas completamente conectadas. VGG fue el primer modelo en usar filtros tan pequeños, mostrando que se obtenían mejores resultados al hacerlo.
- El ganador del reto de 2015 fue ResNet, desarrollada por Microsoft Research (He, Zhang, Ren, y Sun, 2015). ResNet utiliza una misma distribución de capas repetida durante toda la red. Posee 152 capas, siendo la primera vez que se alcanzaba un número tan alto de capas de forma práctica. Era difícil tener tantas capas a causa del problema del desvanecimiento de gradiente (Hochreiter, 1998) que se da al entrenar por backpropagation y resulta en que, al hacer la propagación hacia atrás, cada capa consecutiva reduce el error propagado llegando "desvanecerse" resultando en un difícil entrenamiento. Esto es solucionado por ResNet al conectar capas lejanas para que el error pueda propagarse entre estas más rápido, llamando a estas conexiones *skip connections*.

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

7.1– Fully Convolutional Network

Fue propuesta en 2014 por investigadores de la Universidad de California en Berkeley (Long, Shelhamer, y Darrell, 2014) obteniendo los mejores resultados hasta la fecha en segmentación semántica mediante el uso de redes convolucionales. Un punto clave fue tomar una entrada de un tamaño arbitrario y producir una salida de un tamaño correspondiente. Usa como base los modelos AlexNet, VGGNet y GoogLeNet (Szegedy y cols., 2014), donde se sustituyeron las capas completamente conectadas con capas convolucionales con filtros 1x1 y añadieron una capa convolucional con filtro 1x1 y profundidad C+1 para predecir las puntuaciones de cada clase, C el n° de clases y añadiendo una más para el fondo.

Se compararon los modelos FCN-AlexNet, FCN-VGG16 y FCN-GoogLeNet, obteniendo los mejores resultados con FCN-VGG16 y convirtiéndose este en el modelo base. Con el objetivo de ganar más nivel de detalle en la salida, se aumentó la resolución de la salida en 32x usando interpolación bilineal. También se usaron *skip connections* para conectar varias capas con la capa final. En la figura 7.1 se puede ver un ejemplo de la arquitectura FCN (Sultana, Sufian, y Dutta, 2020).

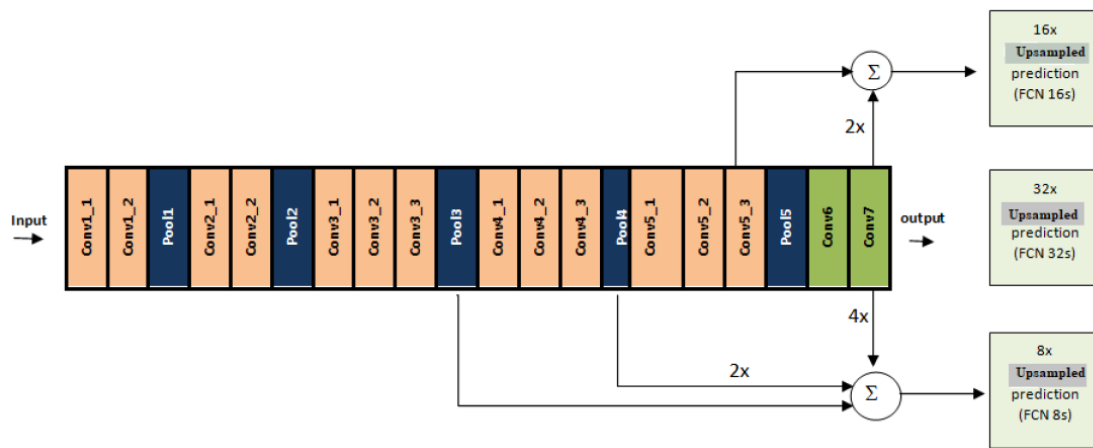


Figura 7.1: Arquitectura de FCN32, FCN16 y FCN8.

7.2– 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 7.2 (Noh y cols., 2015).

A continuación se describen brevemente las operaciones de unpooling y deconvolución (más correctamente llamada convolución transpuesta). En la figura 7.3 (Noh y cols., 2015) se puede ver un ejemplo gráfico.

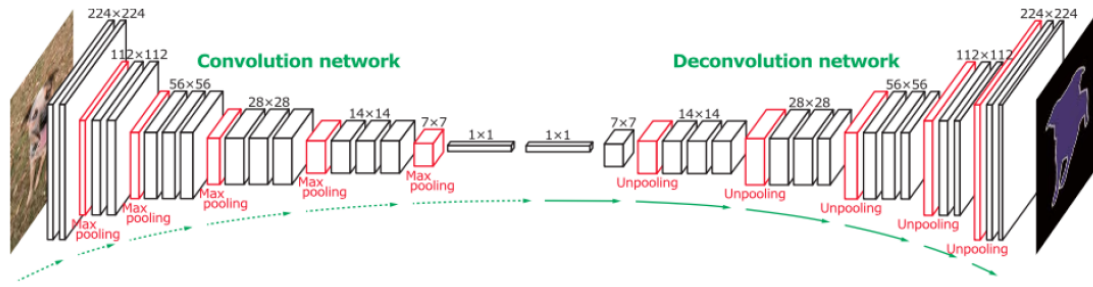


Figura 7.2: Arquitectura Deconvnet.

7.2.1. 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 7.3). 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).

7.2.2. 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 deconvolución (Noh y cols., 2015). Las operación de convolución toma varias entradas y las transforma en un único valor al aplicar un filtro. La operación deconvolución toma un único valor y lo transforma en varias salidas al aplicar el mismo filtro usado en la convolución correspondiente.

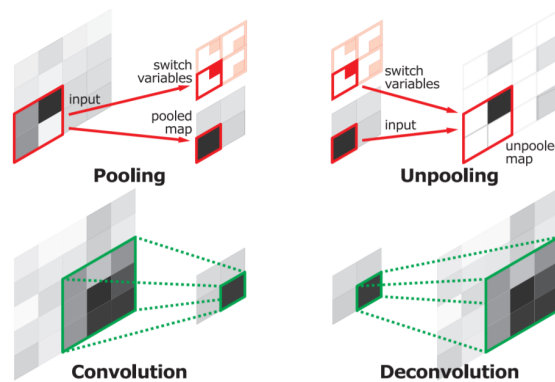


Figura 7.3: Figura que ilustra las operaciones unpooling y deconvolución.

7.3– 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 n° correspondiente de clases. En la arquitectura original se tienen en total 23 capas convolucionales. En la figura 7.4 puede verse la arquitectura original.

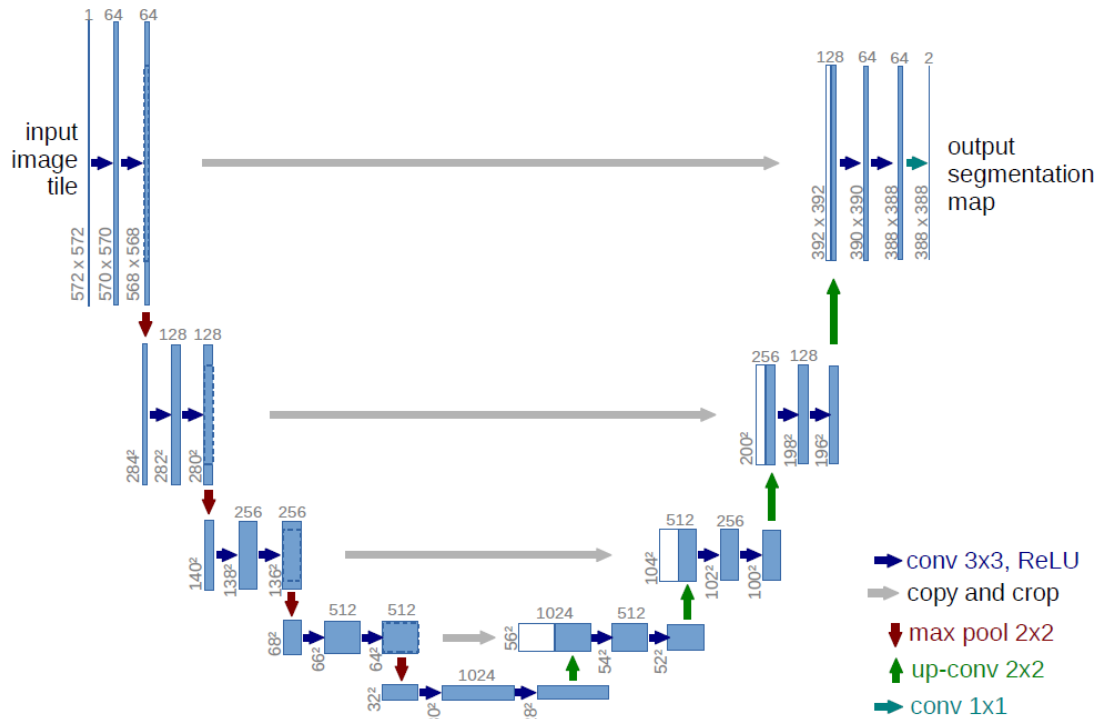


Figura 7.4: Arquitectura original U-Net propuesta por Ronneberger et al en U-Net: Convolutional Networks for Biomedical Image Segmentation.

Aplicaciones recientes

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

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

8.1.1. 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.

8.1.2. Pipeline

Preprocesado	Procesado	Postprocesado
GT:Espaciado de 1 vóxel entre células GT:Células etiqueta 1, fondo 0 IMyGT:Troceo de imagen a 236x236x100 vx IMyGT:Data Augmentation: -Rotación -Deformación suave -Incremento de intensidad	U-Net Función de pérdida: -Entropía Cruzada con Pesos -Peso alto en espaciado entre células Optimizador ADAM: -Learning rate 10^{-5} $\beta_1 : 0,9$, $\beta_2 : 0,999$ 150000 Iteraciones	Ninguno

Cuadro 8.1: 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:

$$w(x) := w'_{bal} + \lambda w_{sep} \quad (8.1)$$

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

$$w'_{bal}(x) := \begin{cases} 1 & y(x) > 0 \\ v_{bal} + (1 - v_{bal}) * \exp(-\frac{d_1^2(x)}{2\sigma_{bal}^2}) & y(x) = 0 \\ 0 & y(x) desconocido \end{cases} \quad (8.2)$$

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.

$$w_{sep}(x) := \exp(-\frac{(d_1(x) + d_2(x))^2}{2\sigma_{sep}^2}) \quad (8.3)$$

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

8.1.3. Resultados

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

$$M_{IoU}(A, B) := \frac{|A \cap B|}{|A \cup B|} \quad (8.4)$$

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.

8.1.4. 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/resources/opensource/unet>

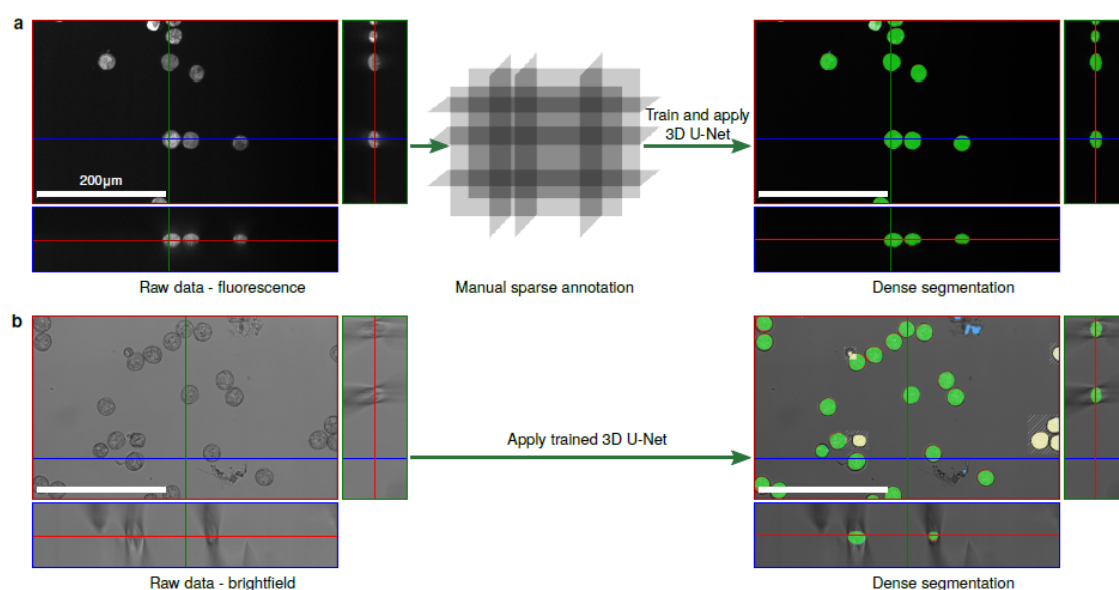


Figura 8.1: Segmentación volumétrica. (a) Segmentación de una imagen con un modelo que se ha entrenado con imágenes del mismo dataset. (b) Segmentación de una imagen con un modelo entrenado con un dataset distinto.

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

8.2.1. 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.

8.2.2. Pipeline

Preprocesado	Procesado	Postprocesado
GT: Bordes con 2 vóxeles de anchura GT: Desenfoque Gaussiano IM y GT: Data Augmentation - Volteo Horizontal y Vertical - Rotación en plano XY - Deformación elástica IM: Noise Augmentation IM, GT: Troceo de imagen a 170x170x80 vx	U-Net Funciones de pérdida probadas: - Entropía Cruzada Binaria - Pérdida Dice Optimizador ADAM: - Learning rate 10^{-5} - $\beta_1 : 0,9$, $\beta_2 : 0,999$ 100000 iteraciones	Transformada de la distancia Detectado de centroides Algoritmo watershed Grafo de adyacencia Particionamiento de grafo

Cuadro 8.2: Pipeline que siguen los datos de inicio a fin. GT significa *groundtruth* y se refiere a la imagen ya segmentada. IM es la imagen de entrada.

En el preprocesado, se parte de una imagen con etiquetado perfecto y se le aplica la función *find_boundaries* de la librería scikit (Pedregosa y cols., 2011) (van der Walt y cols., 2014) obte-

niendo los bordes de las células con 2 vóxeles de anchura. A la imagen con los bordes se le aplica un desenfoque Gaussiano para reducir los componentes de alta frecuencia, lo que ayuda a prevenir el sobreajuste. Luego se aplica varias técnicas de *data augmentation* en el espacio a la imagen de entrada y a la imagen de bordes. Tras esto se aplica *noise augmentation* a la imagen de entrada. Por último se divide la imagen en trozos de 170x170x80 vx.

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

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

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

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

8.2.3. Resultados

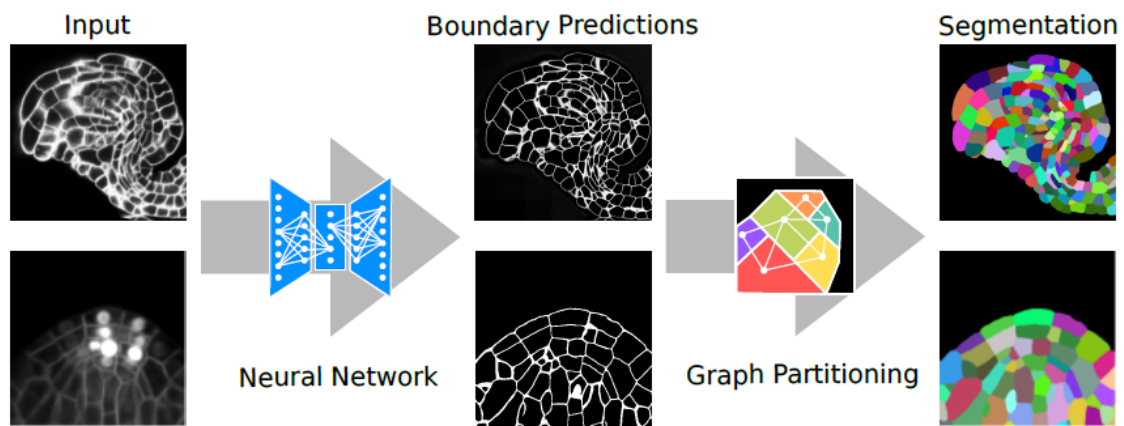


Figura 8.2: Segmentación de tejido vegetal usando PlantSeg. En el primer paso se predicen los bordes de las células usando una red U-Net 3D. En el segundo paso se aplica un algoritmo de particionamiento de grafo para segmentar cada célula.

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

- Precisión: N° vóxeles correctamente etiquetados como borde en la predicción entre el n° de vóxeles etiquetados como borde en la predicción.
- Exhaustividad: N° de vóxeles etiquetados correctamente en la predicción entre el n° de vóxeles etiquetados como bordes en la imagen con perfecto etiquetado.

- Puntuación F1: $F1 = 2 * \frac{Precision * Exhaustividad}{Precision + Exhaustividad}$

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

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

$$VOI = H(seg|GT) + H(GT|seg) \quad (8.5)$$

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 8.3 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 8.3: Pruebas relevantes realizadas.

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

8.2.4. 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>.

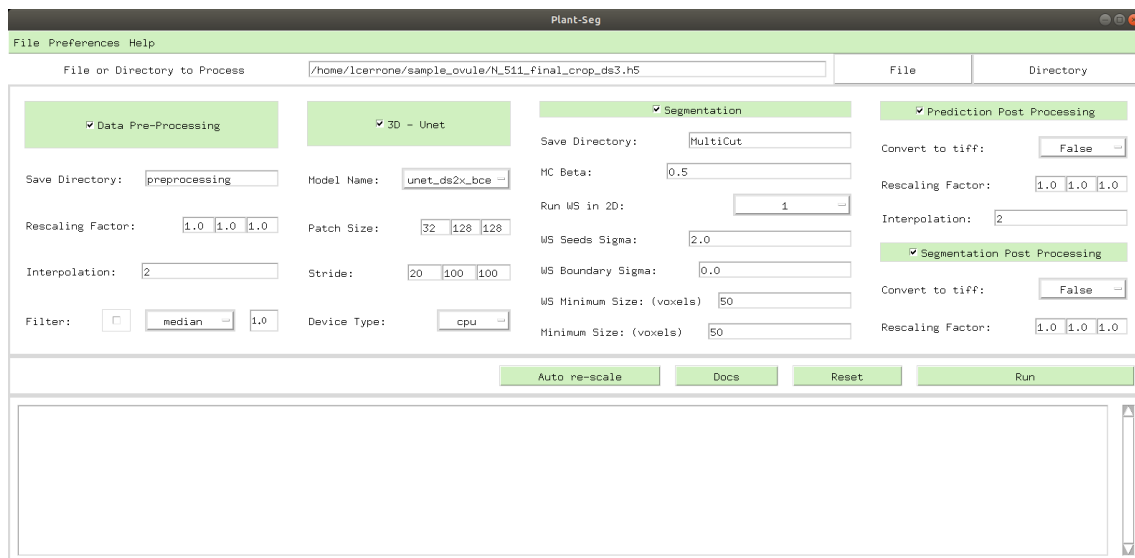


Figura 8.3: Interfaz gráfica del programa PlantSet. Se pueden ver las distintas opciones para cada paso del procesado.

Parte IV

Desarrollo

Guía de capítulos de desarrollo

En los capítulos siguientes se hablará sobre el desarrollo del sistema. Los capítulos estarán en orden cronológico explicando primero lo que se ha desarrollado antes y por qué.

En este punto el estudio previo ya se ha realizado, por lo que la mayoría de los capítulos estarán enfocados a desarrollar un sistema inicial y mejorarlo de forma iterativa.

Los capítulos son los siguientes:

- **De computación local a computación en la nube.** Se prueba un ejemplo de segmentación en local y, después de ver las limitaciones de hardware, se pasa a una opción en la nube.
- **Iteración 1: Sistema inicial.** Se construye un sistema inicial que es capaz de producir un resultado. Para esto se usa una implementación simplificada de la arquitectura U-Net, además de elegir métricas, función de pérdida y otros parámetros.
- **Iteración 2: Arquitectura U-Net completa.** Se cambia la arquitectura del modelo por la U-Net completa.
- **Iteración 3: Data Augmentation.** Se aplican técnicas de data augmentation a los datos de entrada para aumentar el nº de ejemplos de entrenamiento de forma artificial.
- **Iteración 4: Apex, precisión mixta.** Se investiga sobre la precisión de los tensores y cómo cambiarla para aumentar la velocidad de entrenamiento y reducir el consumo de memoria.
- **Iteración 5: DT Watershed.** Se utiliza un enfoque distinto en el que se aplica el algoritmo watershed a la salida.

De computación local a 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.

10.1– Introducción

Después de investigar sobre las soluciones actuales he llegado a la conclusión de que dichas soluciones están pensadas para un hardware superior al que tengo disponible. Voy a implementar redes neuronales convolucionales de forma manual. Lo más probable es que obtenga peores resultados que los que obtendría por las soluciones vistas en el estudio previo, pero voy a tener libertad a la hora de decidir la complejidad del modelo, lo que reducirá el coste de entrenarlo y usarlo.

Voy a empezar siguiendo los pasos del “Getting started” del framework fastai (?, ?) de Python. Este framework es una capa de abstracción por encima de PyTorch (Paszke y cols., 2019). Parece ser que abstrae muchos elementos del programador e incluye mejoras de optimización por debajo, haciendo que probar modelos sea rápido. También es importante que esta librería tenga implementada la arquitectura u-net (y otras de procesamiento de imágenes).

10.2– Computación local

He seguido un tutorial en el que 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>

He tenido que modificar parámetros para poder ejecutarlo en mi máquina, 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 he tenido que 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 nos muestra una gráfica de “Learning rate” vs “Loss”, en el que se podría comprobar qué ratio de aprendizaje tiene menor pérdida. No he podido ejecutarla 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 10.1: 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 10.1 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.

10.3– Computación en la nube

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

Por ello he investigado sobre el uso de GPUs en Cloud Computing. Estoy ejecutando el código de las imágenes 2D 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 (?), que tiene una tier gratuita en la que te ofrecen un notebook con la GPU P5000 con pocas limitaciones.

Gracias a esto he conseguido ejecutarlo con los siguientes parámetros:

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

En la figura 10.2 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 lo bueno que es la predicción, donde 1 sería una predicción perfecta. Las pérdidas de entrenamiento y validación también son menores.

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 como dato de entrada uno con tamaño $\frac{1}{4} * 16 = 4$ veces de la

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 10.2: 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.

imagen original. En el caso anterior se han usado imágenes de $(720, 960)/8px$ con un batch de 2, por lo que se podría decir que en cada epoch se ha usado como dato de entrada uno 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.

Iteración 1: Sistema inicial

11.1– Metas y métricas

11.2– Lenguaje y framework

11.3– Arquitectura CNN

11.4– Carga de datos

11.5– El problema del tamaño de la imagen

11.6– Segmentación semántica multiinstancia con U-Net

11.7– Guardar modelo

11.8– Resultados

CAPÍTULO 12

Iteración 2: Arquitectura U-Net completa

12.1– Problema detectado

12.2– Cambio a U-Net completa

12.3– Resultados

Iteración 3: Data augmentation

13.1– Problema detectado

13.2– Técnica de data augmentation

13.3– Implementación de data augmentation

13.4– Resultados

Iteración 4: Apex, precisión mixta

14.1– Problema detectado

14.2– Precisión de los tensores

14.3– Implementación de Apex

14.4– Resultados

CAPÍTULO 15

Iteración 5: DT Watershed

15.1– Problema detectado

15.2– Algoritmo watershed

15.3– Implementación DT Watershed

15.4– Resultados

Diseño

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

16.1– Esquema general

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

Sería ideal ser capaz de entrenar una CNN usando las imágenes provistas directamente, pero a causa de su gran resolución no es viable. Una imagen de resolución $(200, 1024, 1024) \times 3$ con una precisión de 8 bytes ocupa en memoria (siempre hablaremos de memoria de GPU) $size(imagen) = \frac{1024 \times 1024 \times 200 \times 8}{1024 \times 1024} = 1600MB$. Si bien podríamos almacenar esta imagen en memoria, las CNN se caracterizan por aplicar un gran número de filtros distintos en paralelo a una imagen de entrada, utilizando los resultados de la aplicación de estos filtros en el siguiente paso de la CNN. Tal y cómo se verá en más detalle en la sección sobre las arquitecturas seleccionadas, es completamente inviable para nosotros usar la resolución original. Es por ello que como parte del preprocesado se han simplificado los datos de entrada.

Por otro lado, también sería ideal contar con la última tecnología en GPU o TPU. En un artículo sobre segmentación de tejido celular 3D usando U-Net se explica cómo han usado 8 NVIDIA GeForce RTX 2080 Ti GPUs para realizar 100K iteraciones (Wolny y cols., 2020). El acceso a un hardware superior permite el uso de arquitecturas más complejas, datos más precisos o mayor n° de iteraciones, lo que va a influir en el resultado obtenido.

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

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

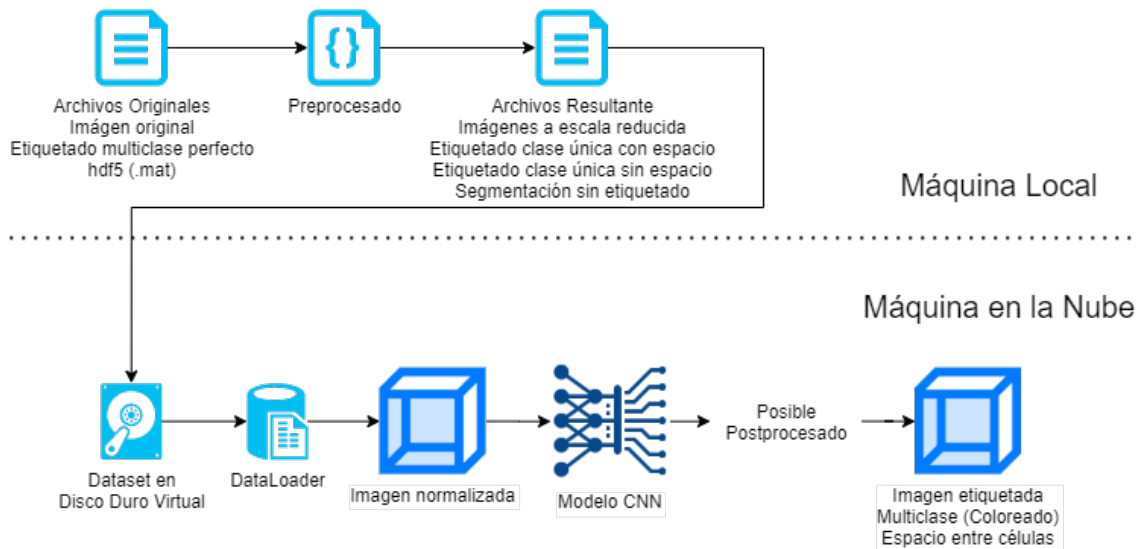


Figura 16.1: Diseño general

En la figura 11.1 se muestra el diseño general sobre el proceso en el que los datos pasan de los archivos hdf5 inicialmente provistos hasta completar la segmentación requerida.

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

En esta etapa se prepararán los datos con 4 objetivos:

1. Reducir la resolución de la imagen de entrada, reduciendo así la memoria necesaria para almacenarla en GPU.
2. Generar las imágenes etiquetadas correctamente para tenerlas como objetivo.
3. Reducir el tamaño de los archivos resultantes, ya que estos serán usados en servicios cloud y tendrán que ser subidos y descargados con frecuencia.
4. Modificar el orden de las dimensiones y añadir una *singleton dimension* para el canal. Este formato es necesario para su uso en la CNN.

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

16.2– Preprocesado local

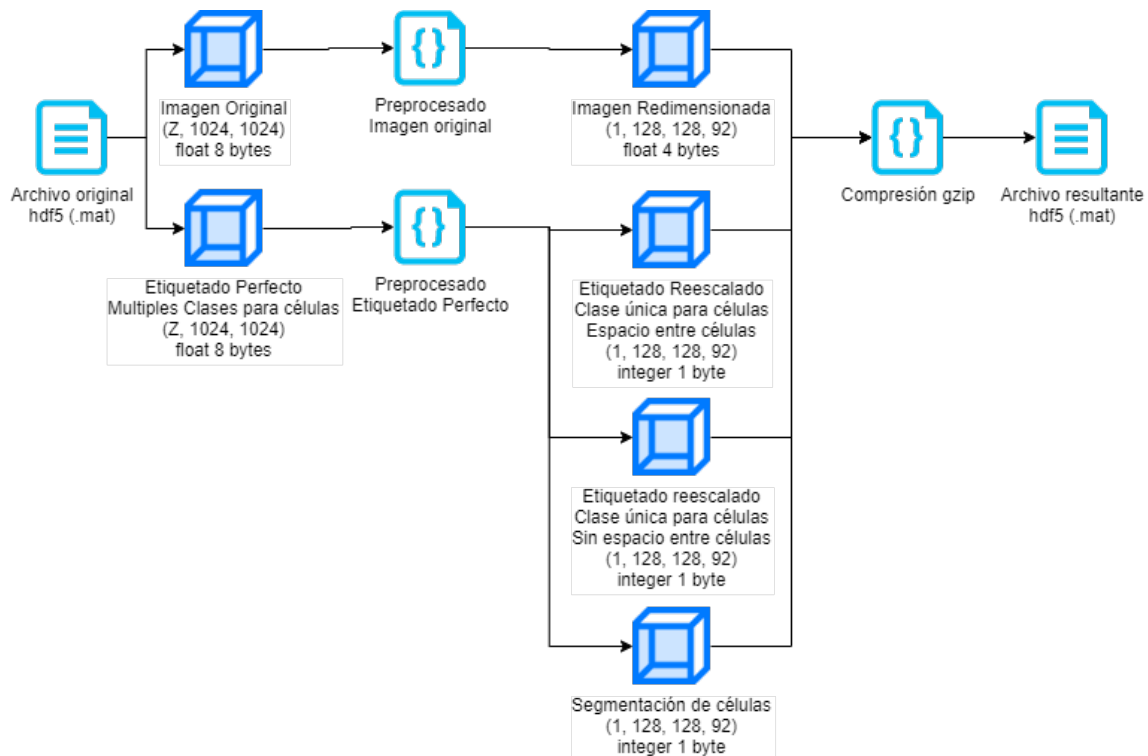


Figura 16.2: Preprocesado llevado a cabo en la máquina local.

En la figura 11.2 se puede ver con más detalle el resultado que se obtendría en esta etapa.

Reducir la resolución de la imagen de entrada

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

Generar las imágenes etiquetadas correctamente

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

1. Añadir espacio entre células para que ninguna esté en contacto y hacer que todas tengan 1 como etiqueta.
2. Etiquetar los bordes de las células con 1 para luego aplicarle DT Watershed.

Los 2 etiquetados distintos son preprocesados en este paso.

Reducir el tamaño de los archivos resultantes

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

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

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

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

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

Modificar el orden de las dimensiones

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

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

16.3– Entrenamiento

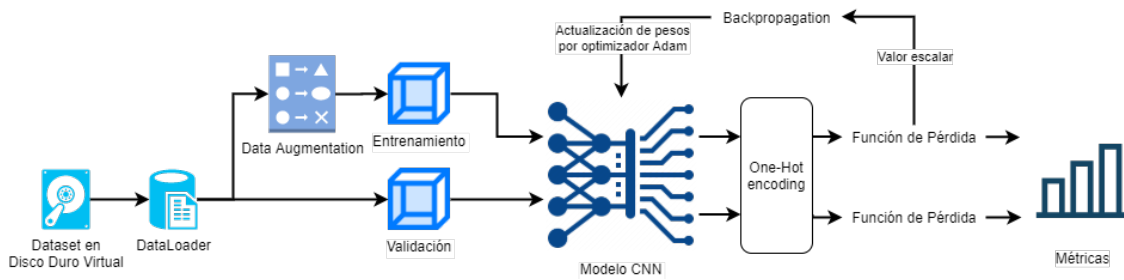


Figura 16.3: Pasos llevados a cabo en el entrenamiento

En la figura 11.3 se muestra el proceso que se lleva a cabo durante el entrenamiento de un modelo. Como ya se mencionó antes este proceso deberá realizarse con una GPU de alto rendimiento, por lo que en este proyecto se utilizará una máquina en la nube. Antes de comenzar este proceso los datos ya habrán sido cargados en un disco duro virtual.

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

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

En el entrenamiento se usará un *batch* de datos, se harán pruebas con valores en el rango $N \in [1, 4]$. Durante cada *epoch*, tanto la predicción obtenida como la imagen con el etiquetado perfecto se transformarán con el *one-hot encoding*, lo que les dará el formato (N, C) , donde N es el tamaño del *batch* y C el n° de clases, que siempre será $C = 2$. Esta disposición de datos transforma cada imagen en 2 vectores, uno por cada clase, lo cual hará que sea muy eficiente calcular diferencias entre la predicción y la imagen con etiquetado perfecto. Para calcular estas diferencias está la función de pérdida, que tendrá un valor bajo si hay poca diferencia y alto si hay mucha diferencia. El objetivo de cualquier algoritmo de optimización será disminuir esta función de pérdida. Para la función de pérdida se probarán *Dice Loss* y *Cross Entropy Loss* con pesos.

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

16.4– Inferencia

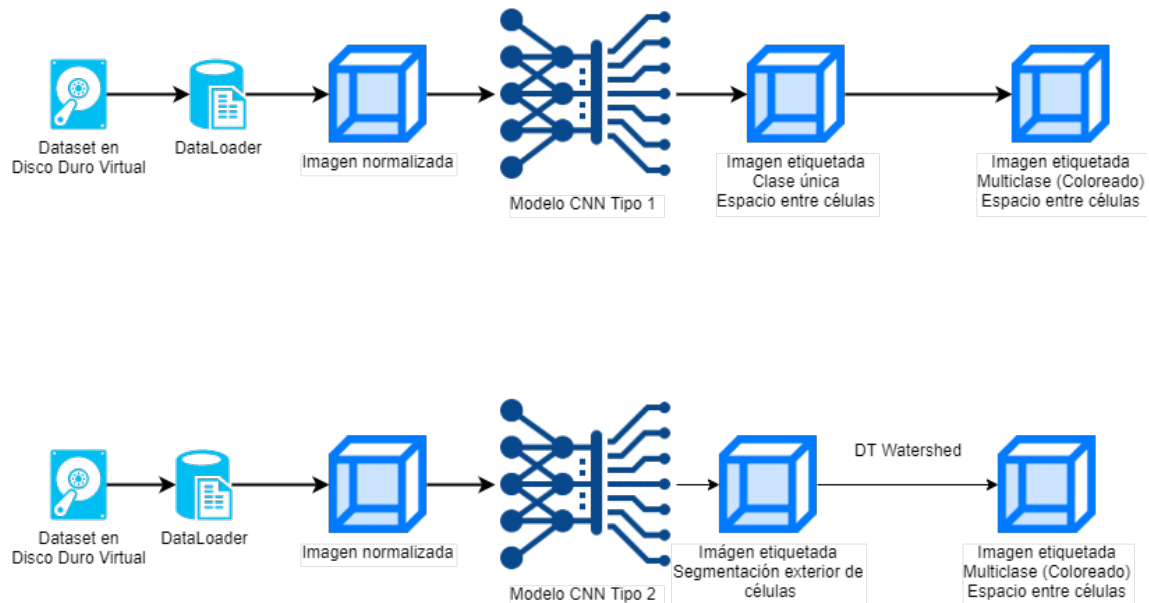


Figura 16.4: Se muestran dos métodos distintos para obtener la imagen correctamente etiquetada.

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

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

1. Enfoque 1: Usar un sólo modelo que llamaremos de **Tipo 1** que ha sido entrenado usando como objetivo imágenes en la que todas las células están etiquetadas con el valor 1 y hay un espaciado (valor 0) entre ellas. Una vez hecho esto se asignará una etiqueta distinta para cada célula.
2. Enfoque 2: El modelo de **Tipo 2** dará como predicción un etiquetado con los bordes de las células de valor 1. A la predicción se le aplica del algoritmo DT Watershed.

16.5– Arquitectura de las Redes Neuronales Convolucionales

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

En la figura 11.5 se puede ver la arquitectura completa de la red U-Net utilizada, arquitectura usada con éxito para segmentación volumétrica densa (Özgün Çiçek, Abdulkadir, Lienkamp, Brox, y Ronneberger, 2016), basada en la arquitectura U-Net propuesta por Ronneberger et al (Ronneberger y cols., 2015). Se ha omitido el tamaño de las imágenes ya que la red acepta imágenes de cualquier tamaño, Lo único a tener en cuenta es el nº de canales que tiene la imagen. Los nº mostrados en las capas determinan la profundidad de cada capa o, lo que es lo mismo, el nº de filtros distintos usados en cada capa. En la capa inicial, que simboliza la imagen de entrada, el número 1 indica que la imagen debe tener tan sólo 1 canal de entrada (escala de grises). Este tipo de arquitectura también es válida para imágenes 2D o 3D, la diferencia estaría en que las convoluciones y el pooling sean sobre 2 dimensiones o sobre 3 dimensiones.

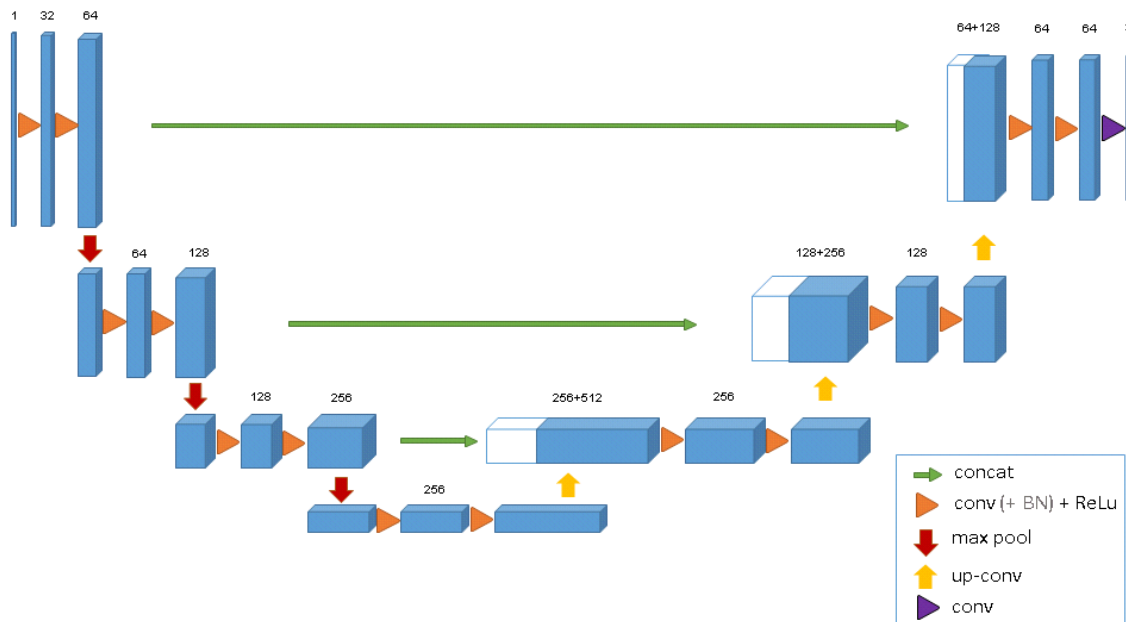


Figura 16.5: Arquitectura U-Net completa.

En la figura 11.6 se muestra la arquitectura anterior eliminando varias capas de convolución y una de pooling. Los modelos generados por esta arquitectura serán menos fiables, pero el entrenamiento será más rápido, por lo que es útil para realizar comparaciones al cambiar técnicas de preprocesado o de entrenamiento.

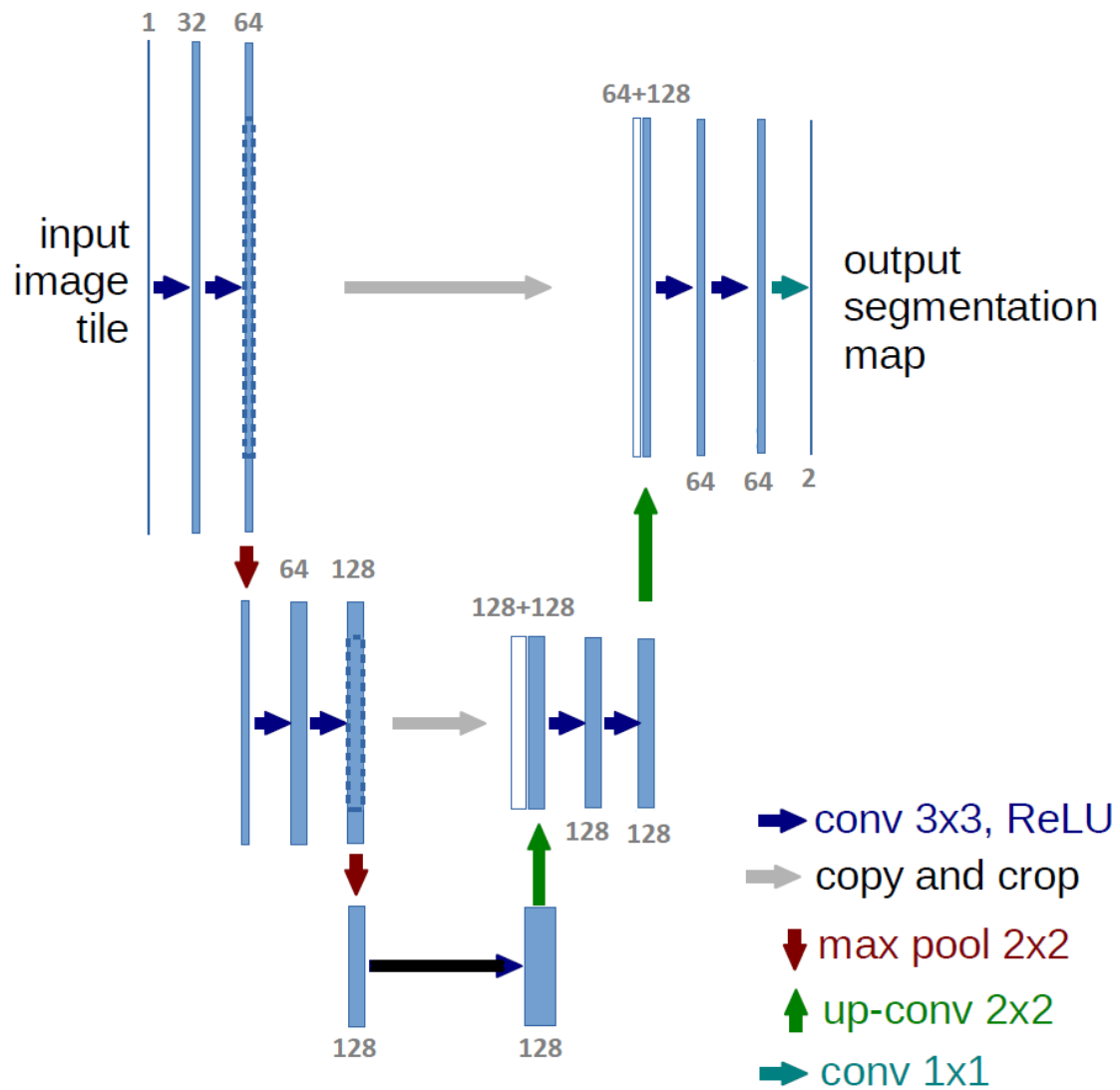


Figura 16.6: Arquitectura U-Net reducida. Esta arquitectura no tiene mucho interés excepto para completar un primer sistema.

Implementación

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

17.1– Lenguaje y framework

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

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

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

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

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

1. Es interpretado, no compilado. Al compilar un programa escrito en un lenguaje compilado el compilador optimiza el programa con una multitud de técnicas, como el desenrollado de bucles, que elimina o reduce las instrucciones de control de un bucle. En Python el intérprete sólo accede a la instrucción a realizar, no realiza ninguna tarea de optimización.
2. Es de tipado dinámico. Esto quiere decir que el intérprete no sabe de qué tipo es una variable hasta que intenta acceder a ella, por lo que antes de acceder a su valor debe comprobar el tipo de variable. Esto hace que las variables en Python sean fáciles de declarar y asignar, pero reduce el rendimiento.

Estas desventajas se pueden reducir gracias a que Python tiene la capacidad de llamar a subrutinas 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 una gran multitud de operaciones de forma eficiente.

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

17.2– Preprocesado local

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

- **scikit-image** (van der Walt y cols., 2014). Es una librería de procesamiento de imágenes. Se ha usado para encontrar los bordes de la imagen con etiquetado multiclase sin espaciado entre células. Esta imagen no podía ser usada directamente en el entrenamiento de la CNN, por lo que se ha seguido la estrategia de encontrar los bordes de cada célula con la función (*segmentation.search_boundaries()*) (Wolny y cols., 2020), después de encontrar los contornos se substraen de la imagen original y se cambian todas las etiquetas a 1, consiguiendo así un espaciado entre células, tal y como se indica en (Falk y cols., 2019). También se ha usado esta librería (*measure_label()*) para asegurarnos de que el nº de células no varía al realizar las operaciones de preprocesado (por ejemplo, podría ser que dos células con etiqueta 1 eliminaran el espacio entre sí, convirtiéndose en la misma instancia de célula).
- **SciPy** (Virtanen y cols., 2020). Contiene diversos módulos con algoritmos útiles en el ámbito científico. En el submódulo *ndimage* se pueden encontrar los algoritmos relativos al procesamiento de imagen. Se ha utilizado la función *ndimage.zoom()* para reescalar las imágenes de entrada, haciendo que ocupen menos memoria y sea viable usarlas para el entrenamiento.
- **h5py** (Collette, 2013). Aporta una interfaz para trabajar con el formato de datos HDF5 (The HDF Group, 2000-2010), útil para usar en python ficheros generados en programas como MATLAB. El tratamiento de datos con esta librería es muy intuitivo ya que imita la sintaxis de NumPy. Ha sido usado para archivos .mat generados en MATLAB con los datos iniciales. Tras el preprocesado, se ha usado esta librería para generar un nuevo archivo con todas las imágenes en un nuevo archivo .mat.

17.2.1. Consideraciones

Reescalado

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

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

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

- **Escalado hacia arriba.** Después de escalar hacia abajo, si se quiere hacer entrenamiento por batch con la imagen completa es necesario que todas las imágenes tengan las mismas dimensiones. Para esto se comprueba las mayores dimensiones de las imágenes después de escalar hacia abajo y se elige como objetivo $(X_{obj}, Y_{obj}, Z_{obj})$. Para cada imagen (X, Y, Z) el procedimiento es crear una nueva matriz de ceros de tamaño $(X_{obj}, Y_{obj}, Z_{obj})$ y asignar a la imagen a los valores $([0 : X - 1], [0 : Y - 1], [0 : Z - 1])$. De esta forma el volumen de las células mantienen el mismo ratio entre todas las imágenes.
- **Orden.** Se ha utilizado orden 3, ya que es el que está por defecto y se han obtenido buenos resultados con él.

Encontrar bordes

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

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

Crear nuevo fichero

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

- **Tipo de dato.** Tanto la imagen de entrada como la imagen etiquetada tienen el tipo de datos *f8*. Es posible pasar la imagen de entrada al tipo *f4* y la imagen etiquetada al tipo *i1* sin perder información. Esto va a reducir el tamaño en disco en más de la mitad. Por lo escribir las imágenes en el fichero se harán con los tipos *f4* y *i1*.
- **Compresión.** Se ha optado por usar la compresión *gzip*, que ofrece una buena compresión a una velocidad no muy alta. Tiene 10 niveles de compresión $[0, 9]$. Se ha probado a comprimir con el nivel 9 consiguiendo una buena compresión pero es excesivamente lento, al final se ha optado por usar una compresión de nivel 4 que, aunque tiene $\sim 10\%$ más tamaño que con el nivel 9, el tiempo de la compresión se reduce a más de la mitad y de todas formas el resultado ya es muy bueno.

17.3– Carga de datos

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

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

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

Además, después de la ZNormalization, se han normalizado los valores al rango $[0, 1]$.

17.3.1. Consideraciones

Tamaño del batch

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

Normalización

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

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

$$v'_i = \frac{v_i - v_{min}}{v_{max} - v_{min}} \quad (17.1)$$

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

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

$$v'_i = \frac{v_i - \mu}{\rho} \quad (17.2)$$

Donde v'_i es el nuevo valor del vóxel, v_i es el valor actual, μ es la media de los valores de la imagen y ρ es la desviación estándar.

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

Data Augmentation

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

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

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

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

17.4– U-Net

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 el capítulo anterior. Se ha desarrollado una versión de esta arquitectura llamada MiniUNet3D con menos capas. Esta versión se ha usado para iterar rápidamente y tomar decisiones como el uso de *data augmentation* o el tamaño del batch, así como comprobar que las funciones de pérdida se han implementado correctamente.

Funciones de pérdida

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

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

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

Precisión Mixta

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

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

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

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

Ficheros generados

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

- **epochs:** N° de epochs entrenados.
- **best_model_state.dict:** Pesos del modelo con menor error de validación.
- **model_state.dict:** Pesos del modelo en la última iteración.
- **optimizer_state.dict:** Pesos del optimizador en la última iteración. Al usar el optimizador Adam se guarda el *learning rate* de cada parámetro de forma individual y este se va modificando a lo largo del aprendizaje, por lo que es beneficioso usar el último estado de estos pesos si se desea seguir con el entrenamiento.
- **train.losses:** Lista con todos los valores de pérdida en el entrenamiento.
- **valid.losses:** Lista con todos los valores de pérdida en la validación. `train.losses` y `valid.losses` son usados para dibujar las curvas de aprendizaje.

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

17.5– Postprocesado

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

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

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

Representación 3D

Hacía falta cambiar el formato usado hasta ahora para poder usar las imágenes con la mayoría de software que tratan imágenes biomédicas.

Se prueba el programa Icy <http://icy.bioimageanalysis.org/>, un programa gratuito Open Source que cumple la función de representación 3D de imágenes, además tiene muchas otras funciones que se podrán probar. +Imagen del programa Todas las imágenes se encuentran almacenadas en archivos con formato hdf5 y el programa no los carga por defecto, pero se puede instalar el plugin "H5 Importer" para ello. Por desgracia un plugin base llamado "BigDataViewerLib", necesario para que "H5 Importer" funcione, ya no se encuentra disponible para su instalación, por lo que me resulta imposible cargar archivos hdf5.

Se decide convertir todas las imágenes al formato tiff ya que es más usado que el hdf5 por defecto en software biomédico, además de también permitir guardar las imágenes con compresión sin pérdida. Para ello se escribe un pequeño script como parte de un preprocesado opcional.

Pruebas de tamaños para comparar. tiff tiene la desventaja de que hay que leer la imagen entera mientras que en hdf5 sólo se leen las porciones de la imagen que se vayan a usar, aunque para el software de representación visual esto no es relevante. En el formato tiff no se pueden guardar varias imágenes, por lo que hace falta un fichero por imagen (original, imagen objetivo y las que haya). Se modifica el código que se encarga de la carga del dataset para permitir también imágenes en formato tiff. Se decide que la mejor forma de guardar las imágenes en tiff es en 3 carpetas: train, validation, test y que en cada carpeta la imagen original tenga el nombre habitual acabado en .tif y la imagen objetivo tenga el nombre de la imagen original correspondiente acabado en _target.tif.

Parte V

Cierre

Resultados

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

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

MiniUnet3D vs Unet3D

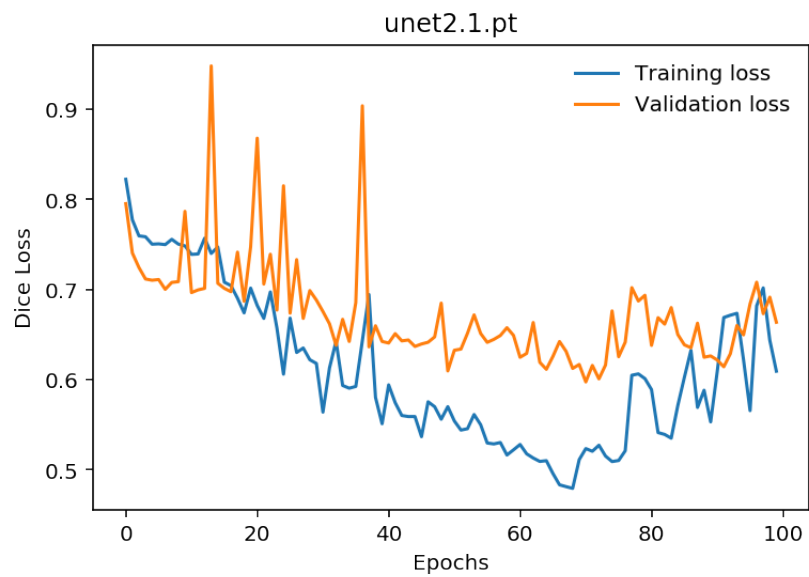


Figura 19.1: Modelo miniunet2.1. 100 epochs. Se usa MiniUnet3D. Imagen objetivo con espaciado entre células. Sin data augmentation.

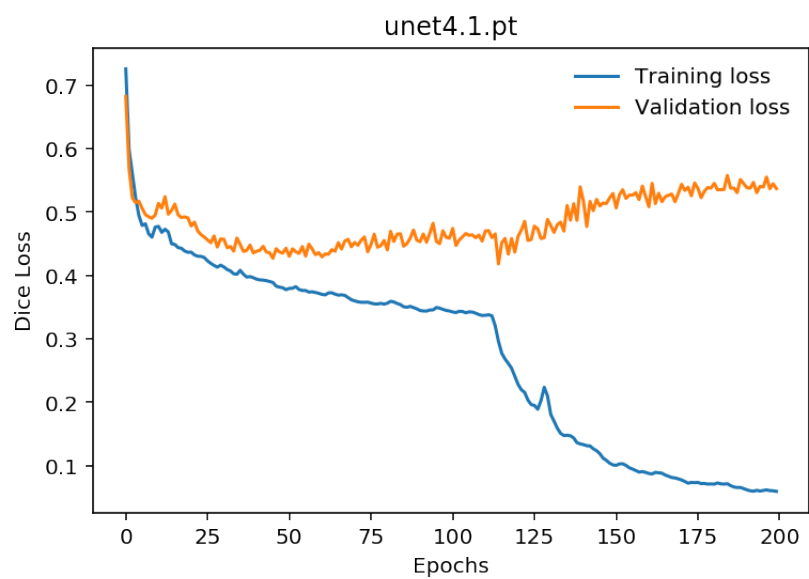


Figura 19.2: Modelo unet4.1. 200 epochs. Imagen objetivo con espaciado entre células. Sin data augmentation.

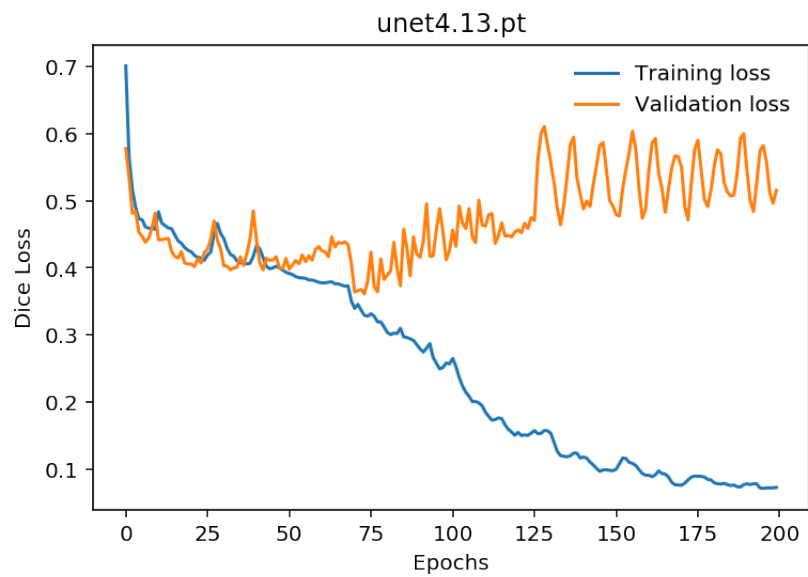
Sin data augmentation vs Con data augmentation

Figura 19.3: Modelo unet4.13. 200 epochs. Imagen objetivo con espaciado entre células. Sin data augmentation.

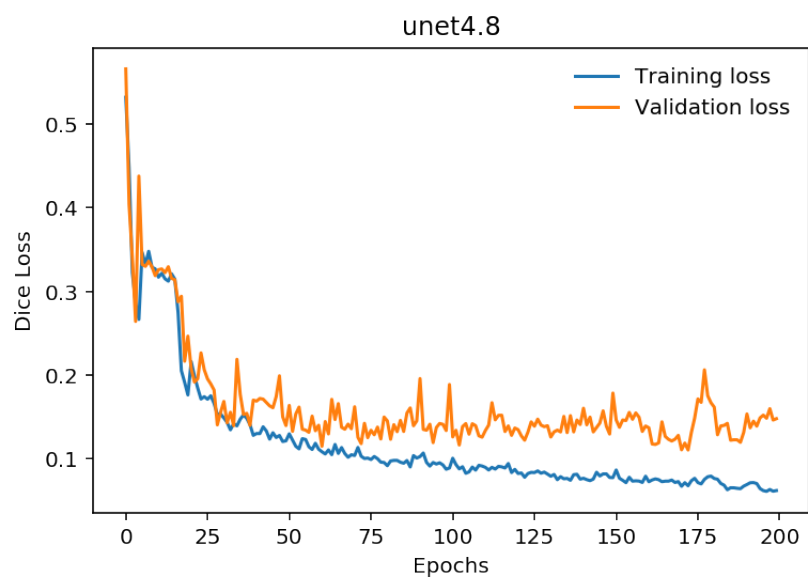


Figura 19.4: Modelo unet4.8. 200 epochs. Imagen objetivo con espaciado entre células. Con data augmentation.

Sin Apex vs Con Apex

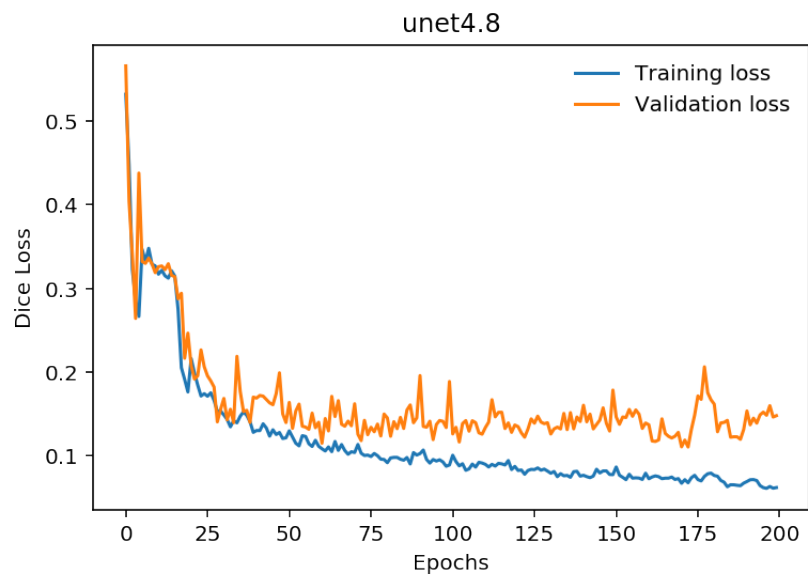


Figura 19.5: Modelo unet4.8. 200 epochs. Imagen objetivo con espaciado entre células. Con data augmentation. Sin Apex

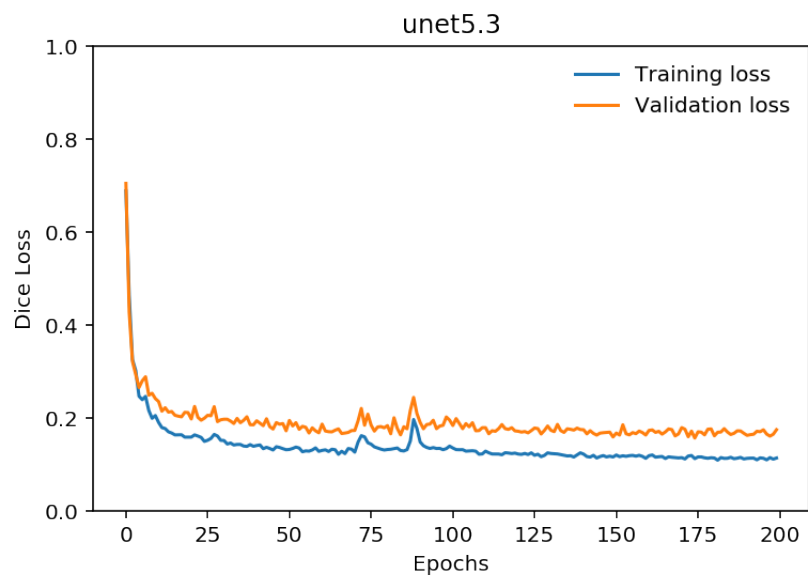


Figura 19.6: Modelo unet5.2. 200 epochs. Imagen objetivo con espaciado entre células. Con data augmentation. Con Apex.

Prueba de espaciado entre células.

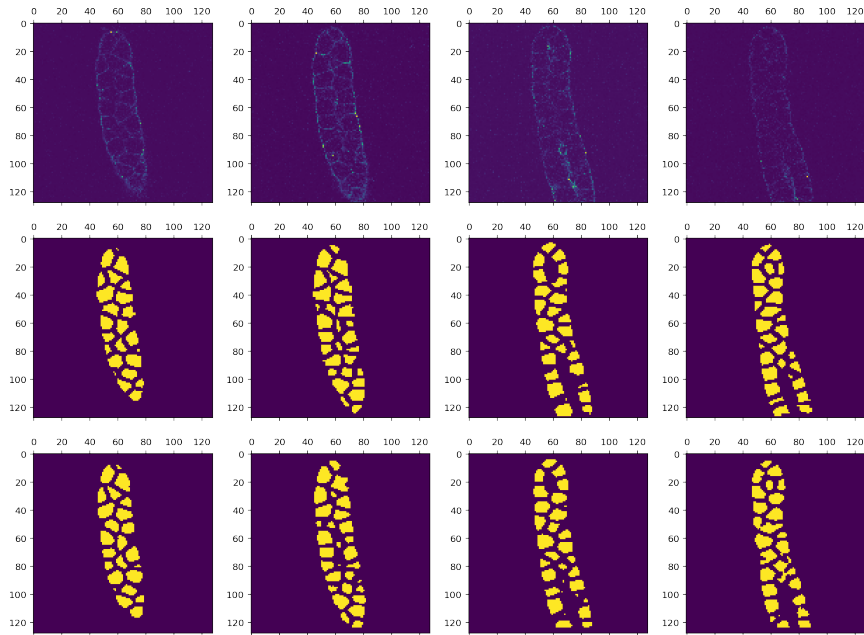


Figura 19.7: Modelo unet6t. 100 epochs. Imagen objetivo con espaciado entre células. Con data augmentation. Pérdida Dice. Sin postprocesado. Primera fila imagen de entrada. Segunda fila segmentación objetivo. Tercera fila predicción. $Z=20,25,45,50$ en las columnas. IoU 0.72

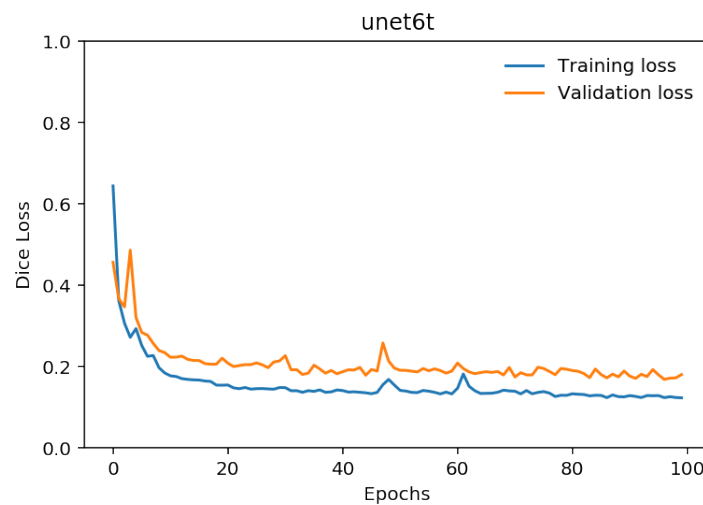


Figura 19.8: Modelo unet6t. 100 epochs. Imagen objetivo con espaciado entre células. Con data augmentation. Sin postprocesado.

Prueba de bordes.

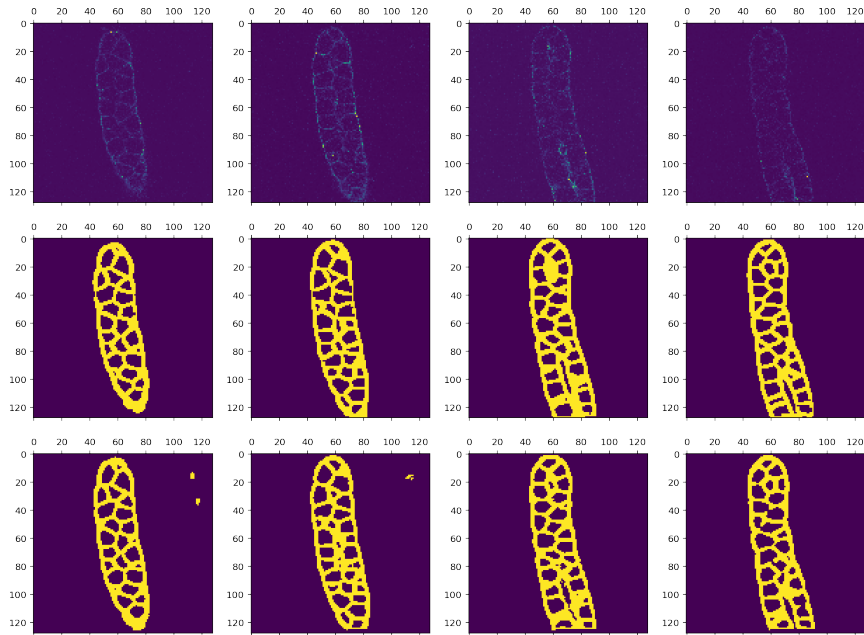


Figura 19.9: Modelo unet6b. 100 epochs. Imagen objetivo bordes de células. Con data augmentation. Pérdida Dice. Sin postprocesado. Primera fila imagen de entrada. Segunda fila segmentación objetivo. Tercera fila predicción. $Z=20,25,45,50$ en las columnas. IoU 0.84

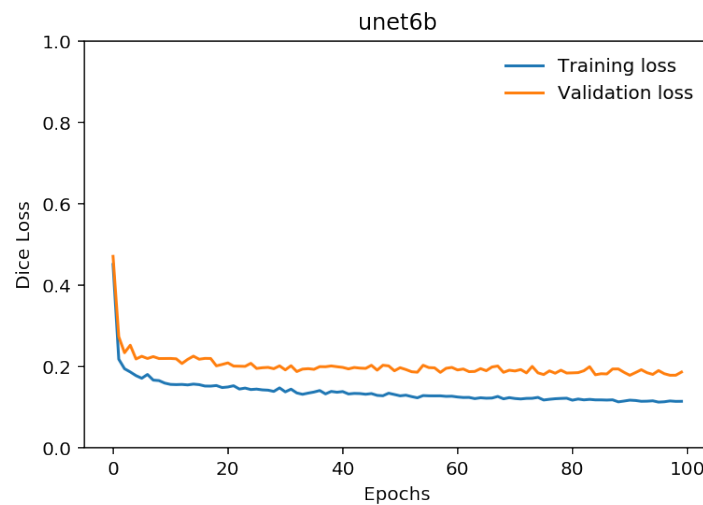


Figura 19.10: Modelo unet6b. 100 epochs. Imagen objetivo bordes de células. Con data augmentation. Pérdida Dice. Sin postprocesado.

DT Watershed a bordes.

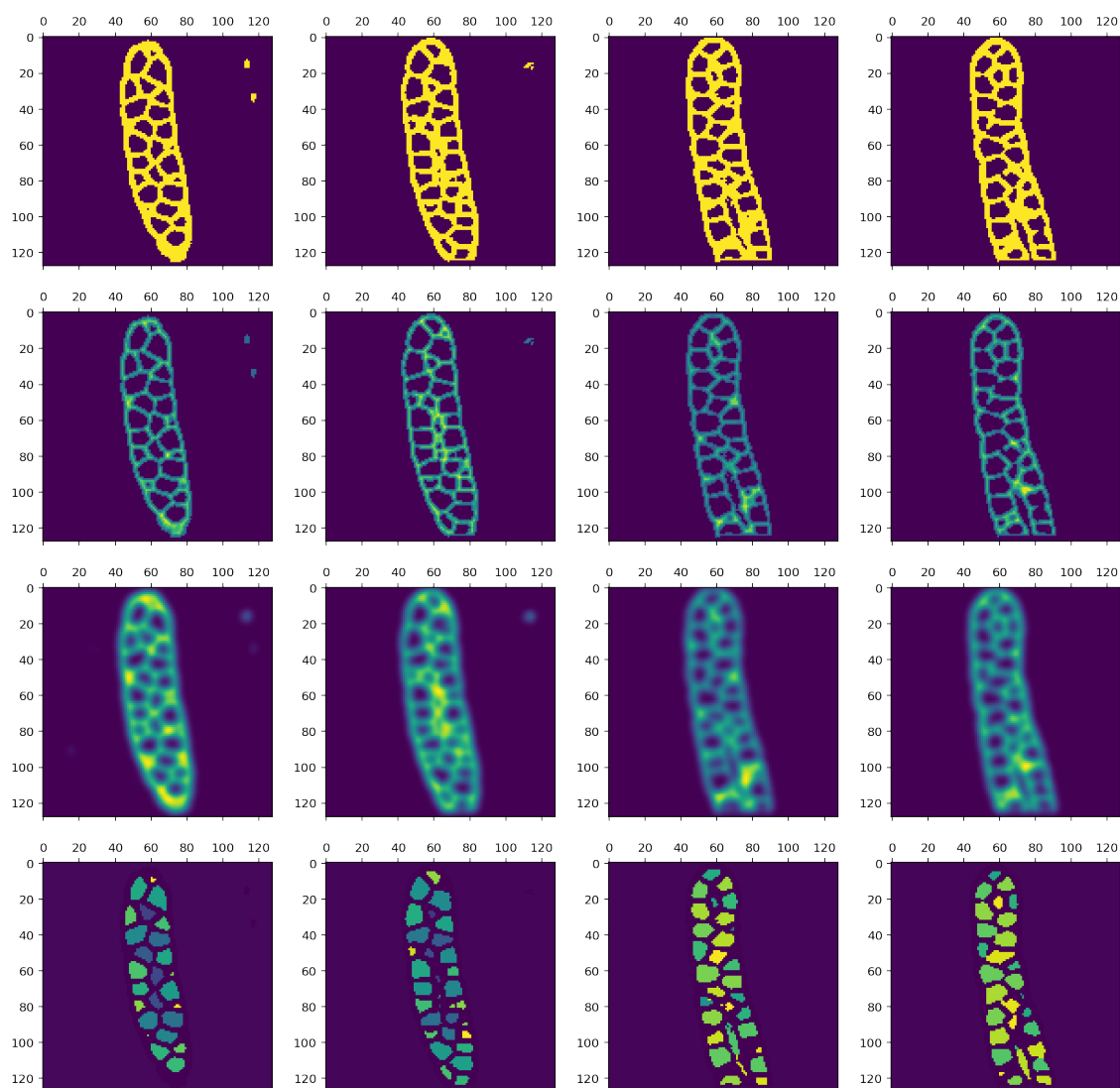


Figura 19.11: DT Watershed aplicado a la salida del modelo unet6b. Primera fila bordes. Segunda fila transformación de la distancia. Filtro gaussiano. Cuarta fila watershed con mínimos locales como semilla. $Z=20,25,45,50$ en las columnas.

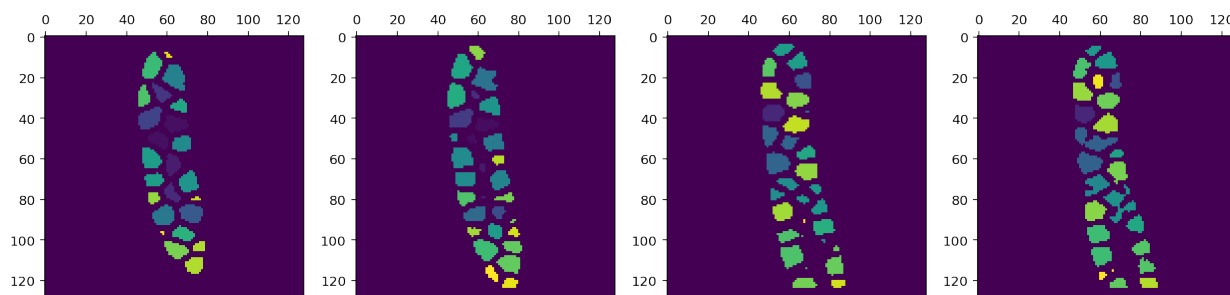
Comparación DT Watershed con espaciado entre células.

Figura 19.12: Modelo unet6b. 100 epochs. Imagen objetivo bordes de células. Con data augmentation. Pérdida Dice. Componentes conexas con etiquetas distintas. IoU 0.72

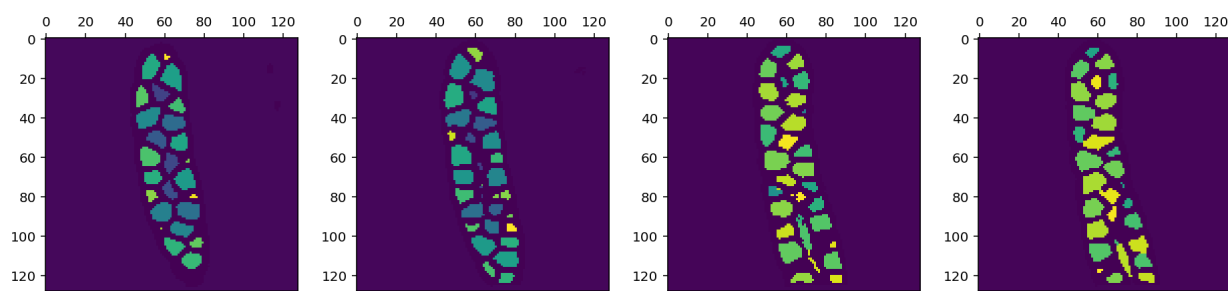


Figura 19.13: DT Watershed aplicado a la salida del modelo unet6b. IoU 0.70

CAPÍTULO 20

Análisis temporal

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

Cuadro 20.1: Tabla de tiempos

Conclusiones

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

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

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

Manual

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

Prerequisitos:

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

Cómo usarlo:

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

Referencias

- 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.
- 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., Besch, 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
- 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*.
- Jia, Y., Shelhamer, E., Donahue, J., Karayev, S., Long, J., Girshick, R., ... Darrell, T. (2014). Caffe: Convolutional architecture for fast feature embedding. *arXiv preprint arXiv:1408.5093*.
- Krizhevsky, A., Sutskever, I., y Hinton, G. (2012, 01). Imagenet classification with deep convolutional neural networks. *Neural Information Processing Systems*, 25. doi: 10.1145/3065386
- Li, F.-F., Krishna, R., y Xu, D. (2020). *CS231n convolutional neural networks for visual recognition*. Descargado 2020-09-03, de <https://cs231n.github.io/convolutional-networks/>

- Long, J., Shelhamer, E., y Darrell, T. (2014). *Fully convolutional networks for semantic segmentation*.
- McCulloch, W. S., y Pitts, W. (1943, 01 de Dec). A logical calculus of the ideas immanent in nervous activity. *The bulletin of mathematical biophysics*, 5(4), 115-133. Descargado de <https://doi.org/10.1007/BF02478259> doi: 10.1007/BF02478259
- Minsky, M., y Papert, S. (1969). *Perceptrons: An introduction to computational geometry*. Cambridge, MA, USA: MIT Press.
- missinglink. (2020). *missinglink.ai*. Descargado 2020-09-03, de <https://missinglink.ai/guides/convolutional-neural-networks/convolutional-neural-network-tutorial-basic-advanced/>
- Msi gp62 7re. (s.f.). Descargado de <https://www.pccomponentes.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 machine learning* (p. 807–814). Madison, WI, USA: Omnipress.
- Noh, H., Hong, S., y Han, B. (2015). *Learning deconvolution network for semantic segmentation*.
- Nwankpa, C., Ijomah, W., Gachagan, A., y Marshall, S. (2018). *Activation functions: Comparison of trends in practice and research for deep learning*.
- Paszke, A., Gross, S., Massa, F., Lerer, A., Bradbury, J., Chanan, G., ... Chintala, S. (2019). Pytorch: An imperative style, high-performance deep learning library. En *Advances in neural information processing systems* 32 (pp. 8024–8035). Curran Associates, Inc. Descargado de <http://papers.neurips.cc/paper/9015-pytorch-an-imperative-style-high-performance-deep-learning-library.pdf>
- Pedregosa, F., Varoquaux, G., Gramfort, A., Michel, V., Thirion, B., Grisel, O., ... Duchesnay, E. (2011). Scikit-learn: Machine learning in Python. *Journal of Machine Learning Research*, 12, 2825–2830.
- Pérez-García, F., Sparks, R., y Ourselin, S. (2020, marzo). TorchIO: a Python library for efficient loading, preprocessing, augmentation and patch-based sampling of medical images in deep learning. *arXiv:2003.04696 [cs, eess, stat]*. Descargado 2020-03-11, de <http://arxiv.org/abs/2003.04696> (arXiv: 2003.04696)
- Pny quadro p5000. (s.f.). Descargado de <https://www.amazon.es/PNY-Quadro-P5000-16GB-GDDR5X/dp/B01LWVP480>
- Riba, E., Mishkin, D., Ponsa, D., Rublee, E., y Bradski, G. (2020). Kornia: an open source differentiable computer vision library for pytorch. En *Winter conference on applications of computer vision*. Descargado de <https://arxiv.org/pdf/1910.02190.pdf>
- Ronneberger, O., Fischer, P., y Brox, T. (2015). *U-net: Convolutional networks for biomedical image segmentation*.
- Rosenblatt, F. (1958). The perceptron: A probabilistic model for information storage and organization in the brain. *Psychological Review*, 65–386.
- Rumelhart, D. E., Hinton, G. E., y Williams, R. J. (1986, 01 de Oct). Learning representations by back-propagating errors. *Nature*, 323(6088), 533-536. Descargado de <https://doi.org/10.1038/323533a0> doi: 10.1038/323533a0
- Schindelin, J., Arganda-Carreras, I., Frise, E., Kaynig, V., Longair, M., Pietzsch, T., ... Cardona, A. (2012, 01 de Jul). Fiji: an open-source platform for biological-image analysis. *Nature Methods*, 9(7), 676-682. Descargado de <https://doi.org/10.1038/nmeth.2019> doi: 10.1038/nmeth.2019
- shiba24. (2017). *3d unet implementation*. <https://github.com/shiba24/3d-unet>. GitHub.
- Simonyan, K., y Zisserman, A. (2014). *Very deep convolutional networks for large-scale image recognition*.

- 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
- 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*.