Detección de mitocondrias en células mediante Deep Learning

Tesis de Máster

Autor
Daniel Franco Barranco

Directores
Ignacio Arganda Carreras
Arrate Muñoz Barrutia

Konputazio Ingenieritza eta Sistema Adimentsuak
Ingeniería Computacional y Sistemas Inteligentes

Resumen

Este proyecto se basa en la realización de la segmentación de mitocondrias en imágenes celulares obtenidas mediante microscopia electrónica. Para ello se hace uso principalmente de las redes neuronales, y concretamente una arquitectura de red convolucional llamada U-Net. Se presenta un estudio pormenorizado para la búsqueda de la mejor combinación de parámetros y técnicas que permitan obtener una buena segmentación. En el estudio, se han probado diferentes conjuntos de datos públicos utilizados por la comunidad, donde finalmente se han alcanzado resultados comparables al estado del arte.

Índice general

Capítulo 1

Introducción y objetivos del proyecto

Al comienzo de la aparición de la inteligencia artificial (IA), los ordenadores resolvían ya fácilmente problemas que para los humanos eran difíciles (problemas que se pueden definir matemáticamente de manera formal). Sin embargo, aquellos problemas que para los seres humanos son fáciles de realizar, pero difíciles de describir formalmente, como el hecho de reconocer en una imagen una cara, un objeto etc., para los ordenadores son un verdadero reto. Los humanos podemos llegar a resolver estas tareas porque digamos, estamos "entrenados" para ello. De hecho, llevamos toda una vida haciendo diariamente esas tareas de manera inconsciente. Los sistemas de IA necesitan la habilidad de ser entrenados de la misma manera, y a este aprendizaje es al que se le conoce como aprendizaje automático o *Machine Learning* (ML) [1].

Diferentes algoritmos han sido propuestos a la largo de los años, cada uno con su manera de "aprender" o "entrenarse" y teniendo su campo de aplicación en el que obtienen buenos resultados (un pequeño resumen se puede encontrar en [2]).

El ML depende mucho de la representación que se realice de los datos. Si se quiere que el ordenador aprenda de ciertas acciones o hechos, habrá que decirle en qué cosas se tiene que fijar y qué es importante de todo aquello. Básicamente, tendremos que modelar esas acciones reales extrayendo las "características", también llamadas "variables", de las que hará uso la IA para aprender [1].

Estos hechos de los que la IA hará uso para aprender pueden estar representados también en imágenes. De hecho, una de las áreas de investigación del ML se centra en el aprendizaje basado en imágenes, como hemos comentado al principio de esta sección, intentando resolver automáticamente tareas tales como la de reconocer cierta zona dentro de las imágenes (objetos, caras, personas, animales, etc.) y acotarla, dividirlas en trozos o segmentos, o intentar "clasificar" o "etiquetar" cada imagen en función de su contenido.

Muchos de los algoritmos de ML funcionan muy bien en una amplia variedad de campos y problemas importantes. No obstante, no tienen éxito en problemas centrales de IA como el que se ha presentando antes con las imágenes. De esta manera, entra en juego el uso de nuevas técnicas conocidas como *Deep Learning*, o aprendizaje profundo, el cual trata de resolver los problemas que los algoritmos de ML "comunes" tienen, además de otros, con funciones complejas en dimensiones espaciales grandes, como son las imágenes, reconocimiento automático del habla, etc [1].

1.1. Estructura de la documentación

Esta documentación se centrará en el aprendizaje mediante imágenes para poder realizar la segmentación de ciertas zonas de la imagen. Concretamente, se trabajará con imágenes obtenidas con un microscopio que contienen diversas células y, en este caso, lo que se quiere detectar son las mitocondrias de las células. Una introducción al respecto se realizará en este primer capítulo.

En el segundo capítulo de esta documentación se explicarán los conceptos básicos de las redes neuronales artificiales, pilar central de este proyecto, junto con ciertos conceptos de todos los algoritmos de ML en general. A continuación, se llegará al tercer capítulo donde se abordará el tema de las redes convolucionales.

Una vez se hayan establecido las bases de las redes neuronales, en el cuarto capítulo se abordará el estado del arte en la segmentación de imagen biomédica, y concretamente de la segmentación de mitocondrias. Aquí se introducirá la arquitectura *U-Net*, de la cual podríamos decir que representa el estado del arte en segmentación de imágenes biomédicas.

En el quinto capítulo se introducirá la experimentación o entrenamiento previo que se ha realizado para familiarizarse con los conceptos de las redes neuronales. También se explicarán las métricas más generales con las que evaluar los modelos de ML.

En el sexto capitulo se desarrollará el caso de uso de la segmentación de mitocondrias, donde se explicarán los diferentes experimentos que se han hecho y sus resultados.

Por último, se presentarán las conclusiones del trabajo y la comparación de los resultados obtenidos con el estado del arte.

1.2. Conceptos básicos

Antes de empezar con el proyecto se ha decidido dedicar esta sección para exponer ciertos conceptos básicos. Se explicarán, de manera superficial, los tipos de microscopios de hoy en día y se hará un repaso sobre el funcionamiento de las células, ya que son temas de los que se hablará continuamente y tienen un gran impacto en esta documentación.

1.2.1. Microscopios

En esta sección se hará una breve descripción de los tipos de microscopios que existen y sus diferencias. No se entrará en mucho detalle, ni se abarcarán todos los tipos, ya que es un campo muy amplio. Sin embargo, aquí se pretende situar al lector en la compleja tarea de la obtención de imágenes con microscopio y el cómo afecta eso en el trabajo realizado.

A continuación se explican, *grosso modo*, los tipos de microscopios que hay:

Ópticos

Estos microscopios son los que primero se inventaron y los más comunes en casas y escuelas. Utilizan una combinación de lentes y luz visible (iluminando el objeto desde la parte superior a través del objetivo o lateralmente) para aumentar imágenes de pequeñas estructuras. Se puede alcanzar hasta una resolución de unos 250 nm y, dependiendo del número de lentes, pueden establecerse dos tipos de microscopios ópticos: simples o compuestos.

Dentro de este tipo de microscopios también están los de fluorescencia, que utilizan rayos de diferentes longitudes de onda, desde la ultravioleta hasta la visible. El principio se basa en iluminar la muestra mediante una luz con una determinada longitud de onda, haciendo que ésta la absorba, e irradie luz en otra longitud de onda diferente a la absorbida. De esta manera, se puede separar mediante filtros ópticos la luz fluorescente de la luz excitada para producir la imagen.

Un tipo de microscopia entre la óptica y la fluorescente es la confocal, la cual escanea secciones ópticas muy finas donde finalmente puede componerse una imagen tridimensional.

Otro tipo de microscopio óptico es el de cambio de fase, en el que se utilizan las diferentes refracciones de las distintas partes de la muestra para reconstruir la imagen.

Electrónicos

Otro tipo de microscopios son los electrónicos, o *electron microscope* (EM) en inglés. A diferencia de los ópticos donde se utilizan fotones para iluminar la muestra, estos lo hacen mediante electrones. Gracias a la dualidad de los electrones como partícula y onda, se pueden acelerar los electrones mediante un potencial eléctrico haciendo que estos tengan longitudes de onda mucho menores que los de la luz visible, aumentando así la resolución con la que se pueden obtener las imágenes. Se pueden subdividir en dos tipos principales: microscopios electrónicos de transmisión (TEM) y microscopios electrónicos de barrido (SEM)

En estos dos tipos de microscopios electrónicos, TEM y SEM, se proyecta un haz de electrones sobre la muestra, formando la imagen a través de la interacción de los electrones con ésta. En SEM, el microscopio cuenta los electrones dispersos, mientras que en TEM se miden los electrones que pasan a través de la muestra. En SEM se centra en la superficie de la muestra y su composición, mientras que TEM está más centrado en el interior de la muestra. SEM proporciona una imagen tridimensional mientras que la de TEM es bidimensional. Por último, decir que los microscopios TEM obtienen resoluciones de 0,1 nm, mientras que los SEM rondan los 0,5 nm.

Otros

Existen otros tipos de microscopios de fuerza atómica o *atomic force microscopes* (AFM) en inglés, que están basados en, como su propio nombre indica, las fuerzas atómicas. Se basan en medir y mapear las fuerzas de interacción entre una sonda en la punta del microscopio y la muestra, pasando por toda la superficie.

Otro tipo sería el microscopio de efecto túnel, o *scanning tunnelling microscope* (STM) en inglés, donde se mide el flujo de la corriente entre una punta conductora y la muestra (que también debe de serlo). Si no es conductora, se suele cubrir la muestra con un material que sí lo sea.

Para finalizar, estaría el microscopio de rayos X, donde se utilizan los rayos X para poder obtener imágenes de la muestra, ya que los rayos X tienen menor longitud de onda que la visible.

Tinción

A parte de todos los tipos de microscopios que hay y la gran cantidad de técnicas para mejorar la imagen resultante, es importante mencionar la tinción de las muestras dependiendo del tipo de microscopía que se esté haciendo y de la muestra a analizar,

diferentes técnicas de tinción son utilizadas, lo que hace más complejo aún este área de la microscopía.

Relación con el proyecto

Como ya se verá a lo largo de este documento, la manera en la que se han obtenido las imágenes y la técnica de tinción utilizada jugarán un papel fundamental en los entrenamientos de las redes neuronales.

Algo común en el área del *Deep Learning* es el llamado *transfer learning*, el cual se basa en aprovechar, para unos datos nuevos, modelos entrenados previamente con otros datos (aunque normalmente estarán relacionados de alguna manera con los nuevos datos sobre los que se quiere aplicar el modelo). De ahí la importancia de que, si el modelo se ha entrenado bien, pueda ser capaz de generalizar incluso con datos que se han obtenido de manera diferente a los usados en el entrenamiento.

En cualquier caso, y como se verá en los últimos experimentos realizados en la sección 6.3.15, habrá que ser meticulosos a la hora de querer hacer esta transferencia, ya que se tendrá que tener en cuenta la resolución con la que han obtenido las imágenes los autores de los diferentes *datasets* con sus microscopios, ajustando el tamaño del píxel en cada caso. Además, dependiendo del tipo de tinción utilizado, el contraste también será un reto con el que la red neuronal tendrá que lidiar.

1.2.2. Células

Todos los seres vivos están formados por células, las cuales son conocidas como la unidad de vida más pequeña. Todas las células tienen una estructura en común: la membrana plasmática, el citoplasma y el material genético o ADN. Además, tienen la capacidad de desarrollar tres funciones vitales: nutrición, relación y reproducción.

Las células se pueden dividir en dos grupos: aquellas que no contienen un núcleo llamadas procariotas (como bacterias y arqueas), y aquellas células muchos más evolucionadas que sí contienen núcleo, citoesqueleto en el citoplasma y distintos orgánulos, las llamadas células eucariotas (divididas como animales y plantas originalmente).

En la Figura 1.1 se muestra la estructura de una célula eucariota de un animal donde se pueden apreciar todas las partes que ésta contiene.

Entre todos los orgánulos que contiene una célula se encuentran las mitocondrias, que son las encargadas de la obtención de la mayor parte de la energía para la respiración celular. Son consideradas las centrales de energía de la célula, sintetizando trifosfato de adenosina (Adenosine triphosphate, ATP) en un proceso llamado quimi-ósmosis [3].

Esta documentación se centrará en estos orgánulos, ya que son muy importantes y es una área que está siendo explotada en el campo de la imagen biomédica. Estos orgánulos, aparte de la producción de energía también tienen un rol crucial en la comunicación, diferenciación, crecimiento y muerte celular [4]. Su detección automática está en continuo desarrollo ya que las mitocondrias están ligadas a ciertas enfermedades como cáncer o Parkinson [5, 6, 7, 8]. Sin embargo, como se verá a lo largo de la documentación, esta identificación no es trivial ya que tienen formas muy diversas, especialmente si el tejido observado no ha sido cortado como se debería [9].

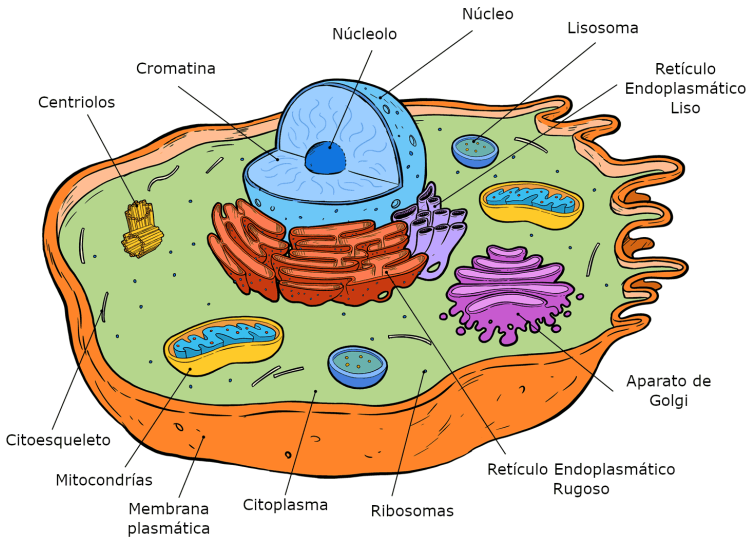


Figura 1.1: Estructura de una célula de animal. Fuente: [10].

Capítulo 2

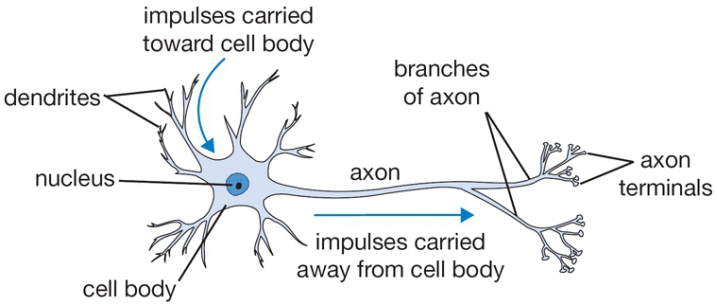
Redes Neuronales

Las *ANN* son un caso especial de algoritmos de aprendizaje automático (o *machine learning* (ML) en inglés) inspirados en las redes neuronales de los sistemas nerviosos de seres vivos. Aunque en la actualidad se han desviado de ese camino hasta convertirse en verdaderas labores de ingeniería, obteniendo muy buenos resultados en *ML* [11]. La unidad más pequeña y básica de cómputo de las redes neuronales son las neuronas, que será el primer concepto que se describirá a continuación.

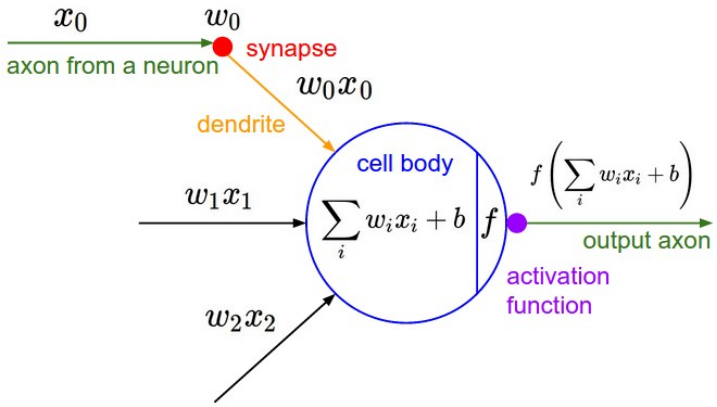
2.1. Modelado de una neurona

La unidad básica de cálculo del cerebro es la neurona. Un ser humano dispone de unas 86 mil millones de neuronas interconectadas entre sí, mediante alrededor de 10^{14} conexiones sinápticas. Se comunican entre ellas a base de impulsos que les llegan por las dendritas (*input*) y son eventualmente expulsados o propagados por los axones (*output*). Además, estos impulsos a veces tiene más fuerza y otras veces menos. Todo ello se puede ver en la Figura 2.1 presentada a la izquierda.

El modelo matemático que la representa tiene, al igual que su homóloga biológica, una entrada o *input* ($\mathbf{x}$) y una salida o *ouput*. La fuerza de los impulsos se modela como "pesos", o *weights* en inglés, y estará representada como $\mathbf{w}$. El *input* puede ser más de una señal, en cuyo caso cada señal llevaría un peso asociado. Por lo tanto, se definirá $\mathbf{x}$ como un vector $x = [x_1, x_2, ..., x_N]$ y $\mathbf{w}$ como otro vector $w = [w_1, w_2, ..., w_N]$ donde N es el número de señales de entrada.



(a)



(b)

Figura 2.1: (a) Neurona biológica; (b) Modelo matemático de la neurona. Fuente: [11].

La operación matemática más sencilla que realiza la neurona puede ser vista como una operación lineal expresada de la la siguiente forma:

$$y = \sum_{i=1}^{N} w_i x_i + b, \tag{2.1}$$

donde b es el término de sesgo, o *bias* en inglés, que es el punto de intersección de la recta en el eje.

Para finalizar, la frecuencia de los impulsos se modelará con la llamada función de activación. Es decir, sólo si la suma anterior supera cierto umbral se propagará la señal. Esto permite añadir la no-linealidad al modelo. De este modo, la función quedaría definida como:

$$z = f_{act}\left(\sum_{i=1}^{N} w_i x_i + b\right). \tag{2.2}$$

Hay diferentes tipos de funciones de activación como se verá en la sección 2.2.

2.1.1. La neurona

Supóngase un ejemplo simple de clasificación binaria como el representado en la Figura 2.2 en el que se quiere clasificar un nuevo elemento, marcado con una cruz, como clase cuadrado o clase triángulo.

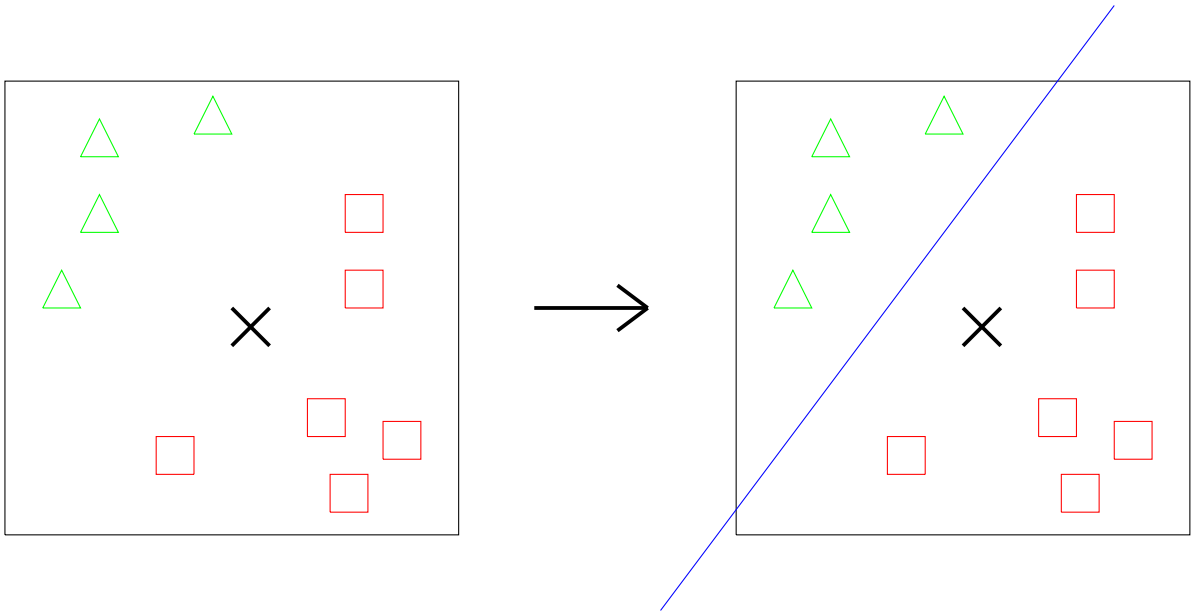


Figura 2.2: Izquierda: representación de las dos clases y el nuevo elemento a insertar señalado con una cruz. Derecha: separación que realiza la neurona con la recta en azul.

En este caso, las muestras están representadas en un espacio de dos dimensiones, así que el *input* constará de dos coordenadas. Por lo tanto, se podría trazar una recta (representada en azul) que separase los dos conjuntos que estaría definida como $y = w_1x_1 + w_2x_2 + b$ que sigue la ecuación 2.1. De este modo, la neurona solo necesitaría "aprender" los valores w_1, w_2 y el sesgo b para poder clasificar el nuevo elemento.

Resumiendo todo lo expuesto hasta ahora, la neurona aplicará el vector $\mathbf{w}$ al *input* $\mathbf{x}$ y le sumará el sesgo b. El resultado lo pasará a través de la función de activación y devolverá un 1 o un 0 si supera cierto umbral. La manera formal de definirlo sería la siguiente [12]:

$$z = f_{act}\left(\sum_{i=1}^{N} w_i x_i + b\right)$$
$$y = \begin{cases} 1 & \text{si } z =>0 \\ 0 & \text{si } z <0 \end{cases} \tag{2.3}$$

El trabajo explicado hasta ahora y la Figura 2.1b, definen la arquitectura más simple que tiene una red neuronal, conocida como "Perceptrón" o *Linear Threshold Unit* (LTU) [12]. Fue inventada por Frank Rosenblatt en 1957 [13] inspirado por el trabajo anterior de Warren S. McCulloch y Walter Pitts [14].

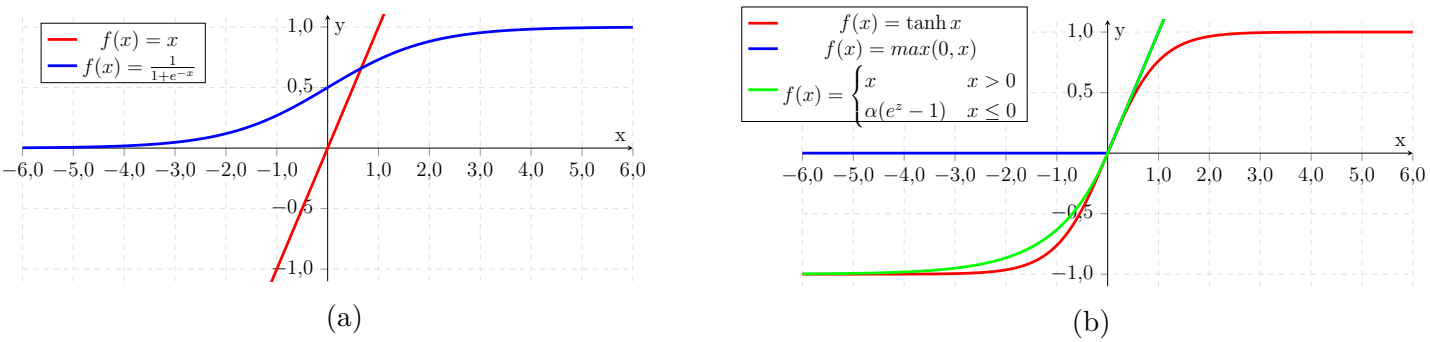
2.2. Funciones de activación

En la sección anterior se ha visto el ejemplo de la red neural más simple: el perceptrón. Allí la función de activación era bastante simple: una función a trozos donde se realizaba una clasificación binaria de 0 o 1. Sin embargo, existen bastantes tipos de funciones de activación. Aquí se presentarán algunas de las más conocidas:

- Lineal, esta función de activación hace que la señal no cambie. Está representada en rojo en la Figura 2.3a.
- Sigmoide, que corresponde matemáticamente con $\sigma(x) = 1/(1 + e^{-x})$. Los valores grandes negativos serán 0 y valores grandes positivos serán 1. Aunque actualmente ya no se usa tanto, históricamente se ha utilizado con frecuencia ya que su comportamiento es muy parecido a la frecuencia con la que las neuronas propagan las señales: de no propagar la señal (un 0) a propagarla completamente (un 1) [11]. En la Figura 2.3a se ha presentado está función en color azul.
- Tanh, representa la relación entre el seno hiperbólico y coseno hiperbólico: $tanh(x) = sinh(x)/cosh(x)$. Básicamente es una función sigmoide escalada siguiendo la fórmula $tanh(x) = 2\sigma(2x) - 1$. Sin embargo, en este caso el rango estará entre $[-1, 1]$ y estará centrada en el 0. Está representada en rojo en la Figura 2.3b.
- Rectified linear unit (ReLU), matemáticamente expresada como $f(x) = max(0, x)$. Se ha vuelto muy popular en estos últimos años [11] y su forma es similar a la función linear pero estableciendo un umbral mínimo de 0. Está representada en azul en la Figura 2.3b.
- Exponential linear unit (ELU) presentada en [15] es parecida a ReLU salvo que es exponencial en la parte negativa. Con $\alpha = 1$ esta función ELU está representada en verde en la Figura 2.3b y sigue la fórmula:

$$f(x) = \begin{cases} x & \text{si } x > 0 \\ \alpha(exp(x) - 1) & \text{si } x <= 0 \end{cases}$$

Estas funciones de activación son importantes para añadir no-linearidad a la red, haciendo así que las redes sean capaces de representar funciones más complejas, ya que sin esto las redes sólo serían modelos de regresión lineal.



(a) (b)

Figura 2.3: Ilustración de las funciones de activación más utilizadas: (a) Rojo: función lineal. Azul: función sigmoide. (b) Rojo: función Tanh. Azul: función ReLU. Verde: función ELU.

2.3. Función de coste

Definido el perceptrón y las diferentes funciones de activación, se guiará al lector a entender cómo se puede definir si una clasificación es o no acertada. Para ello, en esta sección, se introducirá el concepto de función de coste, o *loss* en inglés, mediante un ejemplo relacionado con la clasificación lineal vista anteriormente.

En la Figura 2.4 se han desglosado los pasos que se realizan en la clasificación lineal. Para poder visualizar el ejemplo de manera más sencilla se ha considerado la imagen de entrada de 4 píxeles (monocromos, sin considerar ningún canal de color). Aquí se han centrado los valores de las intensidades a un rango de $[-127, 127]$. Hay 3 clases representadas: "Gato", "Perro" y "Barco", y cada una está marcada con un color (que no tienen nada que ver con los canales RGB). Los pesos están representados como la matriz $\mathbf{W}$, que tiene una fila por cada clase, es decir, lo que correspondería con la neurona vista anteriormente. Además, también hay un vector de sesgos, uno por cada clase, representado como $\mathbf{b}$. Al hacer la multiplicación de matrices y la suma con el sesgo se obtiene un resultado al que posteriormente se le aplica la función de activación (en este ejemplo se ha elegido usar la sigmoide). Al final se obtienen los resultados señalados como "Score", uno por cada clase.

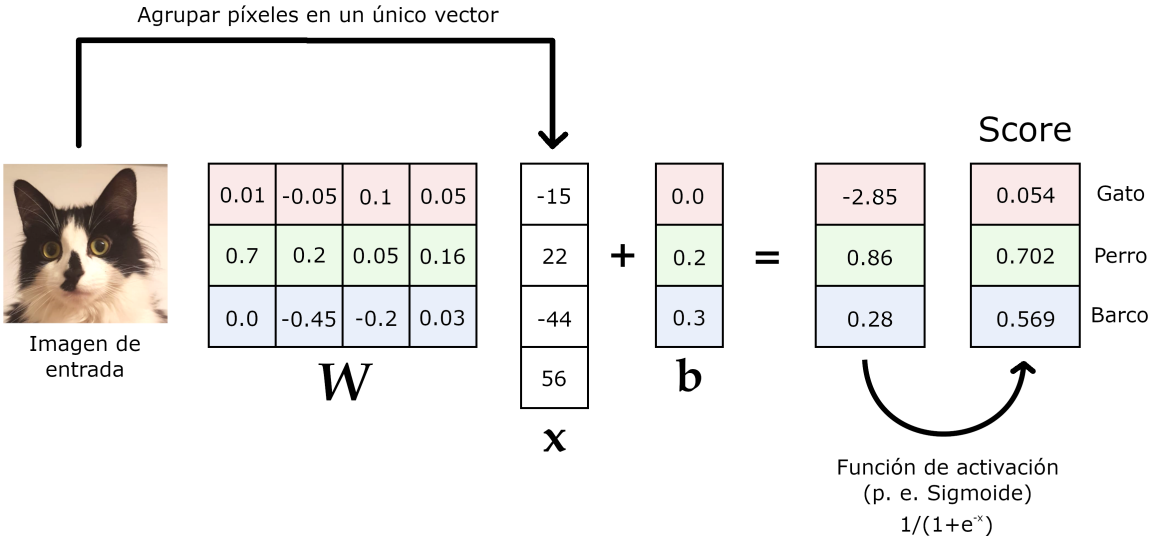


Figura 2.4: Ejemplo de clasificación lineal con tres clases.

Los *scores* obtenidos dan una idea de cómo se parece la imagen de entrada a cada etiqueta. En este ejemplo, si se considera que un alto *score* corresponde con una buena clasificación, los resultados no han sido buenos ya que el *score* obtenido para la etiqueta "Gato" es el peor. Al conjunto de las verdaderas etiquetas o clases de las muestras se les llama comúnmente como *ground truth*, y es usado para comparar cómo de buena ha sido una clasificación basándose en las clases predichas.

Por lo tanto, el resultado de los scores está parametrizado por los pesos W y el sesgo b. Como no se tiene control sobre el *input*, ya que es algo fijo con lo que se tiene que trabajar[1], la única manera de obtener buenos resultados que sean consistentes con el *ground truth* es poder ajustar esos valores $\mathbf{W}$ y b.

Se ha dicho que no se han obtenido unos buenos resultados, ya que el *score* de "Gato" era bastante bajo. Así pues, se definirá la función de coste u objetivo, más conocida como *loss function* en inglés, para medir lo "descontento" que se está con los resultados. Si este coste es alto significará que los resultados obtenidos han sido malos mientras que si es bajo se dirá que se ha obtenido una buena clasificación de los datos.

La función de coste será la suma de los costes de todos los datos y se definirá de la siguiente manera [11]:

[1]Realmente que el *input* no se puede cambiar no es del todo cierto ya que puede que se requiera realizar un preprocesado antes de empezar la clasificación. Hay diversa variedad de técnicas para ello que pueden mejorar la clasificación en ciertos casos, aún así, para que el ejemplo quede más claro se dirá que el *input* es fijo.

$$L = \frac{1}{N}\sum_{i=1}^{N} L_i, \tag{2.4}$$

donde L_i es el coste de cada dato y N es el número de datos total en el entrenamiento. Se pueden definir muchas funciones de coste para los datos de entrenamiento en función de los datos y del tipo de problema, como regresión o clasificación [11]. Para problemas de clasificación una opción podría ser *cross-entropy loss* formulada de la siguiente manera [11, 16]:

$$L_i = -\sum_{i=1}^{M} y_{o,c} log(p_{o,c}), \tag{2.5}$$

donde M es el número total de clases, $y_{o,c}$ es un indicador binario 0 o 1 si el elemento o se ha clasificado correctamente como la clase c, y $p_{o,c}$ es la probabilidad de que el elemento o sea de la clase c. Nótese que únicamente el valor de la clase real será el que cuente, ya que en los demás casos la multiplicación será siempre por 0.

En problemas de regresión se podría usar por ejemplo la función *mean squared error (MSE)* formulada de la siguiente manera [11, 16]:

$$L_i = \frac{1}{2}\sum_{i}(\hat{y} - y_i)^2, \tag{2.6}$$

donde y_i es la salida obtenida del elemento i y $\hat{y}$ es el valor esperado para el elemento i. Nótese que el resultado de aplicar la fórmula siempre es positivo, obteniendo números más bajos cuanto menor sea la diferencia entre lo obtenido y lo esperado.

2.4. Estructura de una red neuronal

Ahora que ya se ha presentado una visión general del perceptrón y la manera en la que se pueden evaluar los resultados obtenidos de esa clasificación con la función de coste, toca adentrarse en el siguiente paso: la estructura de una red neuronal.

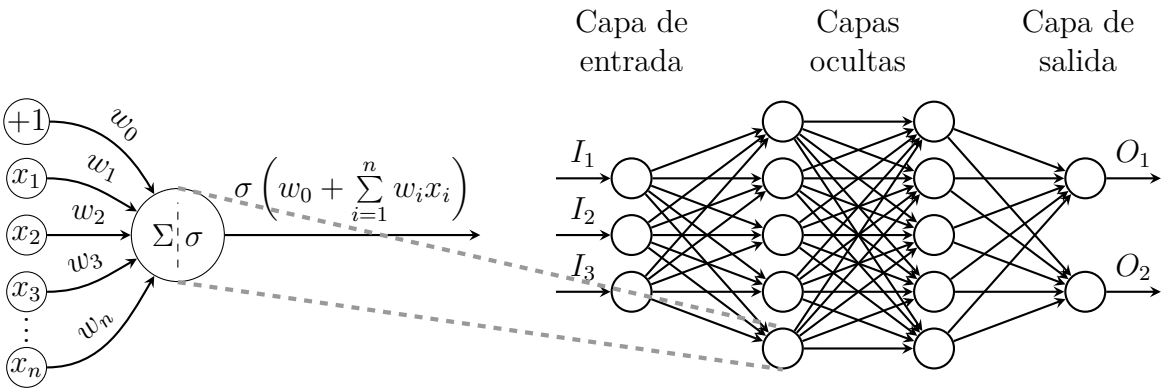


Figura 2.5: Estructura de una *Multi-Layer Perceptron* (MLP). Fuente: [17].

Una red neuronal, también nombrada como *Multi-Layer Perceptron* (MLP) o *Artificial Neural Networks* (ANN), está formada por muchos perceptrones agrupados en capas. Como se puede ver en la Figura 2.5, la red está formada por una capa de entrada, en este caso con tres *inputs*, una o más capas intermedias llamadas capas ocultas o *hidden layers* en inglés, y una capa de salida. Cada capa está conectada con la siguiente y la anterior. Las capas no tienen por qué estar completamente conectadas como en el ejemplo (aunque para las redes más comunes es lo habitual [11]).

Realmente la funcionalidad de una MLP será la de aproximar una función f^*. Por ejemplo, para una clasificación, $y = f^*(x)$ mapea un *input* x en una categoría y [1]. De hecho, una MLP con al menos una capa oculta podrá ser vista como un "aproximador universal" de funciones [11]. Dada cualquier función continua $f(x)$ y $\epsilon > 0$, existe una red neuronal $g(x)$, con al menos una capa oculta y una no-linealidad definida, como por ejemplo la activación sigmoide, que cumpla que $\forall x, |f(x) - g(x)| < \epsilon$, o dicho de otro modo, esto querrá decir que una MLP podrá aproximar cualquier función continua [11]. Puede verse una explicación visual de esta afirmación en el libro *online* en el que se ha apoyado esta documentación en varias ocasiones, concretamente en el capítulo 4 [2], y en [18].

El *input* x será un vector de números pero realmente esto engloba muchos tipos de datos que pueden verse de esa manera, como por ejemplo texto, que podría interpretarse como un vector 1D de números, espectrogramas, que podrían interpretarse como vectores 2D, imágenes como vectores 3D o incluso vídeos, que serían vectores 4D.

El *input* irá pasando a través de las capas hasta que finalmente se devuelva en la última capa la clase a la que pertenece dicho *input*. Esta última capa de salida no tiene una función de activación ya que, para problemas de clasificación, representa los *scores* de las clases, o un valor real objetivo en el caso de regresión [11]. Por lo tanto, esta última capa no tiene por que ser un único valor, también podrá devolver más de uno (véase

[2] http://neuralnetworksanddeeplearning.com/chap4.html

softmax que se explicará en la siguiente sección 2.4.1).

Para finalizar, también se quiere recalcar que, a medida que se añaden capas y neuronas a la red, el número de pesos y sesgos a calcular aumenta considerablemente, lo que implica un gasto de memoria mayor a la hora de crear la red.

2.4.1. Softmax

Una función que se suele implementar en la última capa es la denominada *softmax*. Esta capa depende de las salidas de todas las otras neuronas de la capa anterior y tendrá tantas salidas como clases a predecir. Se propondrá seguir con el ejemplo de querer clasificar una imagen como "Gato", para conectar de manera más fácil con lo anterior y por ser un ejemplo visual.

Siguiendo ese ejemplo, si se supone que hay otras tres clases más, como por ejemplo "Perro", "Barco" y "Coche", esta capa *softmax* tendría como resultado 4 probabilidades, que sumarán 1 y cada una representará la probabilidad de que el *input* pertenezca a esa clase.

Se podría decir que *softmax* se basa en calcular las "evidencias" de que un *input* dado pertenezca a una clase. En este ejemplo con imágenes, el cálculo de evidencias podría hacerse mediante una suma ponderada de la pertenencia de cada uno de los píxeles a la clase [12]. Un ejemplo visual sencillo podría ser el siguiente: uno se podría imaginar una especie de imagen a modo de "plantilla" que la red puede llegar a aprender basada en los pesos, sesgos y los *scores* de los que se ha hablado en este capítulo. Realmente esta "plantilla" será el modelo que la red ha aprendido para determinar si una imagen es de una determinada clase. En la Figura 2.6 están representadas varias plantillas con las que *softmax* podría decidir la pertenencia a una clase. Por ejemplo, se puede ver que para clasificar el *input* como "Barco" la plantilla contiene muchos píxeles de color azul (como era de esperar). Por lo tanto, si le pasara como *input* una foto de un barco y se comparase con la primera plantilla, se podría decir que habrá un mejor "matching" entre ellas que si se compara con la segunda plantilla de "Coche", por ejemplo.

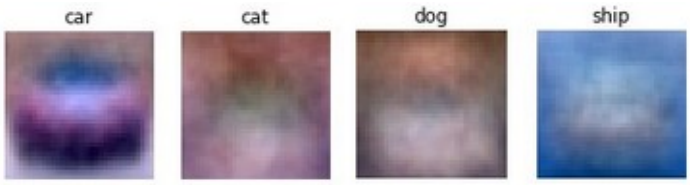


Figura 2.6: Ejemplos de plantillas en representación a como *softmax* podría indicar la clase a la que pertenece su *input*. Fuente [11].

Una vez calculadas las evidencias (ev) de pertenencia a cada una de las clases al

comparar la imagen con la plantilla, la función *softmax* podrá determinar la pertenencia a una clase i siguiendo la siguiente fórmula [12]:

$$Softmax_i = \frac{e_i^{ev}}{\sum_j e_j^{ev}} \tag{2.7}$$

Nótese que mediante esta formula las buenas predicciones tendrán una probabilidad cercana a uno en su respectiva clase, mientras que tendrá números cercanos al 0 en el resto.

2.5. Descenso del gradiente

Como se ha mencionado anteriormente, los pesos y sesgos son los que determinan la función de coste, reflejando cómo de buena es la predicción realizada por la red, ya que el *input* es fijo. Por lo tanto, el objetivo del entrenamiento de la red será el conseguir la combinación de pesos y sesgos con la que se obtenga el coste más bajo. La enorme cantidad de parámetros que se pueden llegar a tener en una red suele ser del orden de millones, como se verá en las secciones posteriores, por lo que es necesario la definición de una estrategia que converja hacía la solución. Uno de los algoritmos más usados en este problema de optimización para reducir el coste es el descenso del gradiente, o *gradient descent* (GD) en inglés.

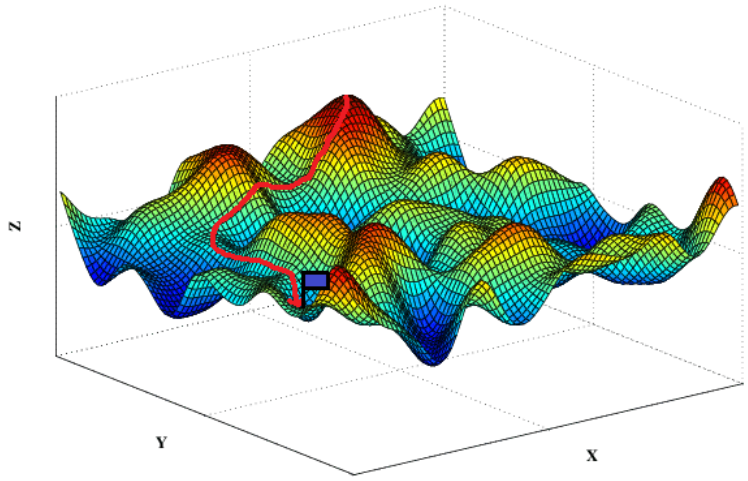


Figura 2.7: Ejemplo del algoritmo *gradient descent* (GD). Modificación de la imagen obtenida en [19].

Si uno se imagina el espacio de soluciones posibles como una cadena montañosa, siendo el valor del coste la altitud de ella, la punta de la montaña correspondería con el coste alto y la base con valor bajo. El GD se basa en ir acercándose a la mejor solución situada en la base de la montaña paso a paso. Para saber en que dirección avanzar se hace uso del gradiente, lo que indica la dirección en la que se debe descender para llegar a una solución mejor. Traducido al campo de los pesos y sesgos de las redes, el gradiente indica cómo han de modificarse estos valores para asegurarse de que nos aproximamos progresivamente hacia el mínimo coste. Un ejemplo de este algoritmo se puede visualizar en el gráfico de la Figura 2.7, donde se puede observar el camino recorrido en rojo, mediante el uso del gradiente, hasta llegar al mínimo marcado con una bandera azul.

Con el gradiente se tiene la dirección en la que hay que actualizar los pesos, pero no determina cuánto hay que moverse en esa dirección. Para ello se introduce el término de ratio de aprendizaje o *learning rate*, que es el encargado de establecer el paso/salto que se dará en la montaña, o traducido a la jerga de las redes, cuánto se van a cambiar los pesos/sesgos para conseguir mejorar el coste. Con un *learning rate* demasiado bajo el algoritmo tardará en buscar el mínimo, ya que los pasos a dar serán muy pequeños, mientras que si este valor es muy grande, los saltos que se harán pueden ser demasiado bruscos, entorpeciendo así el aprendizaje. Un ejemplo de ello se puede observar en la Figura 2.8.

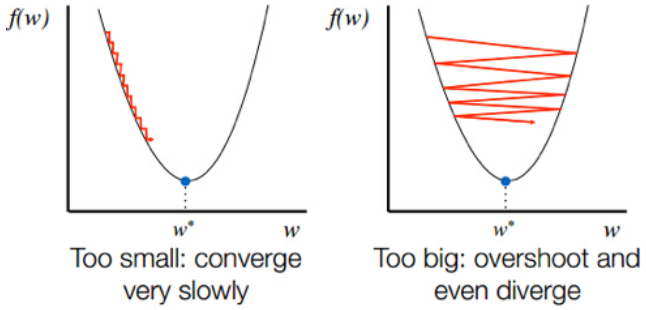


Figura 2.8: Ejemplo del diferentes valores de *learning rate*. Un *learning rate* bajo hace que se converja muy despacio (izquierda), mientras que si es muy alto los saltos serán demasiado grandes y entorpecerán el entrenamiento. Fuente [20].

Matemáticamente el GD puede ser definido de esta manera [21]:

El gradiente es la generalización multivariable de la derivada: el gradiente de C, siendo C una función de m variables $v_1, ..., v_m$, es el vector que contiene todas las derivadas parciales denotado como:

$$\nabla C = (\frac{\partial C}{\partial v_1}, ..., \frac{\partial C}{\partial v_m}) \tag{2.8}$$

donde el i-ésimo elemento del gradiente es la derivada parcial de C respecto de v_i. Por lo

tanto, un cambio pequeño en C, denotado como ΔC tendría la forma:

$$\Delta C \approx \nabla C * \Delta v \tag{2.9}$$

La ecuación anterior se refiere a que un pequeño cambio en v realizará pequeños cambios en C, por lo tanto se puede seleccionar aquellos cambios que hagan C negativo:

$$\Delta v = -\eta \nabla C \tag{2.10}$$

El símbolo η representa el *learning rate* comentado anteriormente. Además, se puede demostrar que estos cambios asegurarán siempre que C será más pequeño:

$$\Delta C \approx -\eta \nabla C * \nabla C = -\eta \|\nabla C\|^2 \tag{2.11}$$

Ya que, como $\|\nabla C\|^2 >= 0$, esto garantiza que $\Delta C <= 0$, reduciendo C si se cambian las variables v tal y como aparece en la ecuación 2.10.

Por lo tanto, se podría repetir continuamente la siguiente ecuación para ir descendiendo sobre el gradiente:

$$v \rightarrow v' = v - \eta \nabla C \tag{2.12}$$

Traducido al campo de los pesos y sesgos, las fórmulas para actualizar sus valores serían:

$$\begin{aligned} w_k &\rightarrow w'_k = w_k - \eta \frac{\partial C}{\partial w_k} \\ b_l &\rightarrow b'_l = b_l - \eta \frac{\partial C}{\partial b_l} \end{aligned} \tag{2.13}$$

A pesar de que está técnica ofrece un camino seguro hacía un coste más pequeño también tiene la desventaja de que su coste computacional es alto. Esto es por que en la ecuación 2.4 el coste se calcula para los N datos que hay, por lo que en una red en la que el *input* sea de millones de imágenes habrá que calcular el coste para cada una de ellas únicamente para realizar un paso en el GD. Además, este algoritmo de optimización no garantiza siquiera que llegue a un mínimo local en un tiempo razonable de tiempo, pero a menudo es capaz de encontrar valores de coste suficientemente bajos como para ser útiles [1].

Por todo ello, el GD no se suele usar y se prefiere una versión modificada de él llamada descenso del gradiente estocástico.

2.6. Descenso del gradiente estocástico

El descenso del gradiente estocástico, o *stochastic gradient descent* (SGD) en inglés, es una versión extendida del GD que intenta resolver los problemas que éste tenía. SGD

es la espina dorsal de muchos otros algoritmos planteados hasta la fecha para realizar el entrenamiento de las redes neuronales.

Por un lado el SGD utiliza solamente parte del conjunto de datos para actualizar los pesos en vez del conjunto entero de N elementos. Para ello utiliza conjuntos de tamaño M conocidos como *minibatches* en inglés, denotados como $X = x_i, ..., x_M$. Con un valor de M lo suficientemente grande se espera que ∇C_{x_i} sea aproximadamente la media de todos los demás ∇C_x [21]:

$$\frac{\sum_{j=1}^{M} \nabla C_{X_j}}{M} \approx \frac{\sum_x \nabla C_X}{N} = \nabla C \tag{2.14}$$

Donde la segunda suma es sobre el *dataset* completo, por lo que se podría decir que se cumple:

$$\nabla C \approx \frac{1}{M} \sum_{j=1}^{M} \nabla C_{X_j} \tag{2.15}$$

Entonces, seleccionando aleatoriamente un *minibatch* y calculando su gradiente se podrá estimar el gradiente de todo el conjunto.

Llevado al campo de los pesos y los sesgos, se tendrá entonces que actualizar esos valores con el gradiente calculado con ese *minibatch* de la siguiente manera:

$$\begin{aligned} w_k \rightarrow w'_k &= w_k - \frac{\eta}{M} \frac{\sum_j \partial C_{X_j}}{\partial w_k} \\ b_l \rightarrow b'_l &= b_l - \frac{\eta}{M} \frac{\sum_j \partial C_{X_j}}{\partial b_l}, \end{aligned} \tag{2.16}$$

donde la suma se realiza en todos los elementos X_j del *minibatch*. Entonces, esto se realiza repetidas veces con *minibatches* distintos elegidos de manera aleatoria cada vez hasta que se acabe con todos los datos. En ese momento se completará una época, o *epoch* en inglés, que es un parámetro que habrá que definir a la hora de querer entrenar una red[3]. Finalmente, el criterio de parada del aprendizaje llegará cuando se cumplan todas las épocas que se han definido (véase sección 2.10.3 para más detalle sobre los criterios de parada).

Actualmente, se han realizado muchos trabajos en torno a la optimización del SGD. Existen métodos para mejorar el tiempo de aprendizaje como el de *momentum* [22], acumulando en cada cálculo del gradiente una "velocidad" a modo de analogía con la

[3]Realmente no se tiene por que ajustar siempre el número de *epochs* con el que se quiere entrenar ya que se podrán definir otros criterios de parada como, por ejemplo, parar el entrenamiento cuando la función de coste no se haya mejorado en un número de *epochs* que se defina.

gravedad y acorde a las leyes de movimiento de Newton, que se aplicará a la hora de actualizar los pesos y sesgos [1].

Recientemente, se presentó el *Nesterov momentum* [23], que es una variante del método anteriormente presentado donde la diferencia está en que el cálculo del gradiente en este caso se realiza después de aplicar la velocidad y no antes, como se realizaba en el *momentum*. Uno puede interpretar este método de *Nesterov momentum* como un intento de añadir un factor de corrección al *momentum* original [1].

2.7. Algoritmos con *learning rates* adaptativos

Por lo general, el parámetro más difícil de configurar en las redes neuronales es el *learning rate* [1]. Existen problemas a la hora de entrenar la red como la optimización de funciones convexas ("ill-Conditioning"), mínimos locales, puntos de silla, regiones planas, etcétera, que métodos como el *momentum* mitigan en cierta manera, aunque a costa de añadir un parámetro más a la red [1]. Por ello surge la idea de realizar cambios en los *learning rates* para intentar mejorar el entrenamiento de las redes neuronales.

En 1988, se presentó el algoritmo *delta-bar-delta* [24] para adaptar individualmente el *learning rate* a cada parámetro de la red. La idea es simple: si la derivada parcial del coste, respecto a un parámetro de la red dado, permanece con el mismo signo debería de incrementarse el valor del *learning rate*, de lo contrario, si cambia de signo, entonces el *learning rate* tendrá que ser más pequeño. Esta técnica sólo se puede aplicar a un *batch* completo, es decir, a todos los datos [1]. Más recientemente, otros métodos aplicables a *minibatches* han sido presentados con la misma filosofía que este algoritmo [25][26][27]. En esta documentación, se explicará unos de los más conocidos y en los que otros trabajos se han basado mayormente.

AdaGrad

El algoritmo *AdaGrad* [25] adapta individualmente el *learning rate* de todos los parámetros del modelo escalándolos de manera inversamente proporcional a la suma de todos los valores de los gradientes elevados al cuadrado, obtenidos desde que empezó el entrenamiento. Los parámetros con mayor derivada parcial del coste corresponderán con un descenso rápido en el *learning rate*, mientras que los parámetros con derivada parcial pequeña tendrán un decrecimiento del *learning rate* más lento.

Empíricamente, la acumulación de los gradientes al cuadrado desde que se empieza a hacer el entrenamiento realiza un excesivo decrecimiento del *learning rate*, por ello, esta técnica funciona bien solo para algunos modelos de *deep learning* [1].

RMSProp

El algoritmo *RMSProp* propuesto en [26] modifica *AdaGrad* para desempeñar un mejor papel en las funciones no convexas cambiando la acumulación de gradiente por una media móvil exponencial ponderada. Con el algoritmo *AdaGrad*, en funciones no convexas, la trayectoria del aprendizaje puede llegar a pasar por diferentes estructuras hasta que finalmente llegará a un mínimo local de forma convexa del que no pueda salir por haber acumulado demasiado gradiente (lo que habrá reducido muchísimo el *learning rate*). En cambio, el *RMSProp* descarta los gradientes más antiguos para hacer la media, pudiendo así converger después de haber encontrado una zona convexa. Este algoritmo también añade un nuevo hiperparámetro a la red, ρ, que controla la longitud de la media móvil [1].

Empíricamente, *RMSProp* es utilizado con frecuencia por la comunidad de *deep learning* por ser un algoritmo efectivo y práctico en la optimización de las redes neuronales [1].

Adam

Adam, presentado en [27], es también otro algoritmo que adapta el *learning rate*. Se podría ver como una mezcla del *momentum* con el *RMSProp*, pero con alguna diferencia significativa. El nombre de Adam viene de "adaptative moment estimation" y no solo utiliza el momento de primer orden[4], sino que usa también el de segundo orden para ajustar el *learning rate*. Específicamente, el algoritmo calcula una media móvil exponencial del gradiente y del gradiente al cuadrado. Además, *Adam* incluye una corrección de sesgo de las dos estimaciones, del momento de primer orden (el término *momentum*) y del momento del segundo orden, pero para ello el algoritmo hace uso de dos nuevos hiperparámetros como pesos para ponderar en la fórmula. Esto es algo que *RMSProp* no implementó.

En general, *Adam* es bastante robusto en cuanto a la selección de los hiperparámetros que se hayan hecho para la red, no obstante hay casos en los que hay que modificar el *learning rate* del que se propone por defecto [1].

Este algoritmo, junto con el SGD con *momentum*, es uno de los más usados en el entrenamiento de las redes neuronales [28]. Por ello, son los algoritmos que más se han usado en este trabajo, como se verá en el capítulo 6.

[4]El enésimo momento de una variable aleatoria se define como el valor esperado de esa variable elevado a la n, es decir: $m_n = E[X^n]$. El primer momento o el momento de primer orden es la media y el segundo es la varianza (no centrada).

2.8. Propagación hacía atrás

Hasta ahora se han presentado diferentes algoritmos para poder actualizar los pesos y los sesgos de la red, siguiendo la dirección correcta mediante el cálculo del gradiente. En esta sección se hablará del método más usado para calcular dichos gradientes de una manera eficiente y sencilla llamado *backpropagation* en inglés, o propagación hacía atrás, presentado en [29].

El objetivo del *backpropagation* es el de calcular las derivadas parciales $\frac{\partial C}{\partial w}$ y $\frac{\partial C}{\partial b}$ de la función de coste C respecto a cualquier peso w o sesgo b en la red. Esto permitirá saber cómo cambia el coste C cuando se produce un cambio en los pesos o los sesgos. Dicho de otro modo, y como se verá a continuación, permitirá saber la "responsabilidad" o en cuánto error ha tomado parte la neurona en el valor de C. Para ello se empezará a analizar la red desde el final hacía atrás, de ahí el nombre del algoritmo.

Para una mejor comprensión se marcará con un superíndice L el número de la capa al que pertenece el parámetro. De esta manera, si se quieren identificar las derivadas de la última capa se denotarían como $\frac{\partial C}{\partial w^L}$ y $\frac{\partial C}{\partial b^L}$, donde L es el número de capas de la red.

Para calcular las derivadas de esta última capa es importante saber el camino que conecta el valor del parámetro (w y b) con el coste de la red. En la última capa el valor del coste será el siguiente:

$$C(a(z^L)), \tag{2.17}$$

donde C es la función de coste, como por ejemplo la MSE presentada en la ecuación 2.6, a representa a la función de activación y z^L el resultado de la suma ponderada de una neurona de la última capa compuesta por w y b, presentada anteriormente en la ecuación 2.1, y que puede ser vista como $z^L = W^L a^{L-1} + b^L$. Esta expresión es una composición de funciones y se resuelve mediante la llamada *chain rule* o regla de la cadena. Esta regla lo que dice es que para calcular la derivada de una composición de funciones se deben calcular cada una de las derivadas intermedias. Por lo tanto, los cálculos que se deberían hacer para poder obtener $\frac{\partial C}{\partial w^L}$ y $\frac{\partial C}{\partial b^L}$ serían:

$$\begin{aligned} \frac{\partial C}{\partial w^L} &= \frac{\partial C}{\partial a^L} \cdot \frac{\partial a^L}{\partial z^L} \cdot \frac{\partial z^L}{\partial w^L} \\ \frac{\partial C}{\partial b^L} &= \frac{\partial C}{\partial a^L} \cdot \frac{\partial a^L}{\partial z^L} \cdot \frac{\partial z^L}{\partial b^L} \end{aligned} \tag{2.18}$$

Se empezará por analizar el término $\frac{\partial C}{\partial a^L}$, que es la derivada de la función de coste respecto al *output* de la red neuronal, que expresa cuánto cambia la función de coste cuando varía el *output* de la función de activación. Por lo tanto, teniendo como función

de coste, por ejemplo, la MSE presentada anteriormente, la derivada quedará así:

$$C(a_i^L) = \frac{1}{2}\sum_i (\hat{y} - a_i^L)^2$$
$$\frac{\partial C}{\partial a_i^L} = (a_i^L - y_i), \tag{2.19}$$

donde y_i es lo que representaba $\hat{y}$ en la ecuación 2.6, es decir, la salida esperada para el elemento i, mientras que a_i^L es la salida de función de activación de una neurona de la capa L para el elemento i.

Por otro lado, estaría la derivada de la función de activación respecto a la suma ponderada, $\frac{\partial a^L}{\partial z^L}$, que expresa en cuánto cambia la salida de la neurona mediante la función de activación variando la suma ponderada de esa misma neurona. Si como función de activación se ha elegido, por ejemplo, una sigmoide, se tendrá lo siguiente:

$$a^L(z^L) = \frac{1}{1 + e^{-z^L}}$$
$$\frac{\partial a^L}{\partial z^L} = a^L(z^L)(1 - a^L(z^L)) \tag{2.20}$$

Por último, estarían las derivadas $\frac{\partial C}{\partial w^L}$ y $\frac{\partial C}{\partial b^L}$ que expresan cómo varia la suma ponderada respecto a los parámetros w y b. En este caso, las ecuaciones serían las siguientes:

$$z^L = \sum_i a_i^{L-1} w_i^L + b^L$$
$$\frac{\partial z^L}{\partial b^L} = 1 \qquad \frac{\partial z^L}{\partial w^L} = a_i^{L-1} \tag{2.21}$$

Si uno se fija bien, la última derivada $\frac{\partial z^L}{\partial w^L} = a_i^{L-1}$ depende de la salida de la capa anterior $L - 1$.

A los dos términos $\frac{\partial C}{\partial a^L} \cdot \frac{\partial a^L}{\partial z^L}$ mostrados en las ecuaciones presentadas en 2.18 se le suele llamar comúnmente "error de la neurona" y está denotado por δ^L. Esta expresión dirá en qué grado ha afectado esta neurona en el coste de la red o la "responsabilidad" que ha tenido, como se ha comentado al inicio de esta sección. Con este nuevo término y las derivadas calculadas en 2.21, las ecuaciones del inicio quedarán de la siguiente

manera:

$$\delta^L = \frac{\partial C}{\partial a^L} \cdot \frac{\partial a^L}{\partial z^L}$$
$$\frac{\partial C}{\partial b^L} = \delta^L \qquad \frac{\partial C}{\partial w^L} = \delta^L a_i^{L-1} \tag{2.22}$$

Hasta aquí se tendrían las ecuaciones necesarias para poder realizar el *backpropagation* en la última capa. Para poder extender este método al resto de $L-1$ capas se necesitará una ecuación más. Si se extienden los cálculos a la capa $L-1$ y se aplicara la regla de la cadena sobre el término $C(a^L(W^L a^{L-1}(W^{L-1}a^{L-2}+b^{L-1})+b^L))$, que sería la representación del coste en esa capa, se obtendrían las siguientes ecuaciones:

$$\frac{\partial C}{\partial w^{L-1}} = \frac{\partial C}{\partial a^L} \cdot \frac{\partial a^L}{\partial z^L} \cdot \frac{\partial z^L}{\partial a^{L-1}} \cdot \frac{\partial a^{L-1}}{\partial z^{L-1}} \cdot \frac{\partial z^{L-1}}{\partial w^{L-1}}$$
$$\frac{\partial C}{\partial b^{L-1}} = \frac{\partial C}{\partial a^L} \cdot \frac{\partial a^L}{\partial z^L} \cdot \frac{\partial z^L}{\partial a^{L-1}} \cdot \frac{\partial a^{L-1}}{\partial z^{L-1}} \cdot \frac{\partial z^{L-1}}{\partial b^{L-1}} \tag{2.23}$$

De todas estas derivadas, únicamente se tendría que calcular $\frac{\partial z^L}{\partial a^{L-1}}$ ya que el resto, o bien ya han sido calculadas, o se calculan de la misma forma que lo explicado anteriormente: $\frac{\partial C}{\partial a^L} \cdot \frac{\partial a^L}{\partial z^L}$ es δ^L, $\frac{\partial a^{L-1}}{\partial z^{L-1}}$ es la derivada de la función de activación que se realiza como en 2.20, $\frac{\partial z^{L-1}}{\partial w^{L-1}}$ es a^{L-2} y $\frac{\partial z^{L-1}}{\partial b^{L-1}}$ sería 1. Entonces, la derivada $\frac{\partial z^L}{\partial a^{L-1}}$ que es la única que hay que calcular, será básicamente la matriz de pesos W^L.

De esta manera, el error de una neurona de la capa $L-1$ se convertirá en:

$$\delta^{L-1} = \frac{\partial C}{\partial a^L} \cdot \frac{\partial a^L}{\partial z^L} \cdot \frac{\partial z^L}{\partial a^{L-1}} \cdot \frac{\partial a^{L-1}}{\partial z^{L-1}}. \tag{2.24}$$

Una vez obtenidas todas las ecuaciones, el algoritmo de *backpropagation* realizará esto mismo de manera recursiva para el resto de $(L-2)$ capas, donde las cuatro expresiones

que se utilizarán serán las siguientes:

$$\begin{aligned}
\delta^L &= \frac{\partial C}{\partial a^L} \cdot \frac{\partial a^L}{\partial z^L} \\
\delta^{L-1} &= W^L \delta^L \cdot \frac{\partial a^{L-1}}{\partial z^{L-1}} \\
\frac{\partial C}{\partial b^{L-1}} &= \delta^{L-1} \\
\frac{\partial C}{\partial W^{L-1}} &= \delta^{L-1} a^{L-2}
\end{aligned} \tag{2.25}$$

Finalmente, con todas las derivadas parciales se podrá calcular el vector gradiente, para poder hacer uso del SGD y calcular los nuevos valores de los pesos/sesgos, de manera que se reduzca el coste de la red.

2.9. Generalización y sobreajuste

El reto central de los algoritmos de *Machine Learning* (ML) es que se comporten adecuadamente frente a nuevos datos que no se han visto a la hora de hacer el entrenamiento. A esta habilidad se le es llamada comúnmente como generalización, o *generalization* en inglés [1].

A la hora de realizar el entrenamiento con el conjunto que se ha asignado para ello (conjunto de entrenamiento), se puede definir el error que se produce en él como "error de entrenamiento". La idea está en reducir este error. Aquí únicamente se ha definido un problema de optimización, pero lo que separa realmente al ML de un problema de optimización es el llamado "error de generalización" o "error de test". Este error se define como el error realizado en los nuevos datos que no se han visto hasta ahora [1].

Dada la asunción de que el conjunto de entrenamiento y el conjunto de test han sido generados bajo la misma distribución de probabilidad, uno espera que al bajar el error de entrenamiento el error de test baje también, aunque en la realidad esto no siempre ocurre así. Entonces, el error de test esperado será mayor o igual que el error esperado de entrenamiento. En general, los factores que determinan cómo de bien se comporta un algoritmo de ML son los siguientes [1]:

1. Reducir el error de entrenamiento a un valor pequeño.

2. Hacer que la diferencia entre el error de test y de entrenamiento sea baja.

Estos dos factores se corresponden con los retos centrales del ML llamados *underfitting* y *overfitting* en inglés. El *underfitting* ocurre cuando el modelo no es capaz de obtener un error de entrenamiento bajo, es decir, que no se haya entrenado lo suficiente. El caso contrario, el *overfitting*, ocurre cuando la diferencia entre el error de entrenamiento y el de test es demasiado alta, dicho de otro modo, cuando el modelo se haya "sobreentrenado" con los datos que se le han pasado.

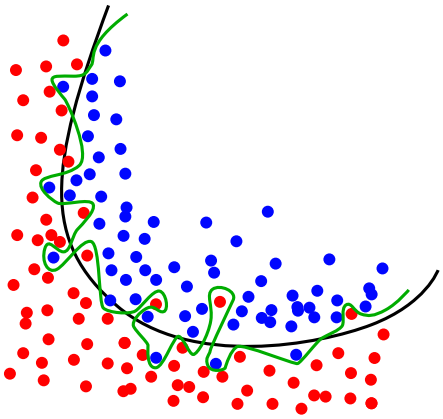


Figura 2.9: Ejemplo de *overfitting* donde la línea verde se ha ajustado demasiado a los datos. En general, el error de test generado por la línea negra será menor que el de la línea verde. Fuente: Wikipedia [5]

La línea verde de la Figura 2.9 sería un ejemplo de *overfitting*, donde el modelo ha aprendido demasiado de los datos de entrenamiento hasta, llevado al extremo, ser capaz de memorizar la posición de los datos, adaptándose demasiado a ellos. A pesar de que el error de entrenamiento en este caso resultará ser muy bajo, a la hora de tener que trabajar con datos nuevos probablemente el error de test será mucho mayor que si se hubiera usado la línea negra.

Se denota como capacidad, *capacity* en inglés, la habilidad de un modelo de ajustarse a una amplia variedad de funciones, es decir, servirá para controlar su *overffiting* o *underffiting*. Modelos con capacidad baja tendrán dificultades a la hora de ajustarse a los datos de entrenamiento, mientras que modelos con capacidad alta podrá memorizar las propiedades del conjunto de entrenamiento, tal y como se ha comportado la línea verde en la Figura 2.9 [1].

Por lo tanto, la labor será la de encontrar esa capacidad óptima en la que el modelo haya aprendido lo suficiente de los datos, teniendo un error de entrenamiento bajo para no hacer *underfitting*, y a la vez sea capaz de realizar una buena generalización, lo que

[5]Explicación de *Overfitting* en Wikipedia: https://en.wikipedia.org/wiki/Overfitting

supone un error de test cercano al de entrenamiento. En la Figura 2.10 se puede ver un ejemplo de ello donde en la parte izquierda el modelo tiene aún un error de entrenamiento alto y la parte derecha un error de test o generalización alto. El mejor punto está marcado como "optimal capacity" y a medida que uno se aleja de ese punto el modelo cada vez generalizará peor, lo que corresponderá con una diferencia mayor entre el error de entrenamiento y el de test.

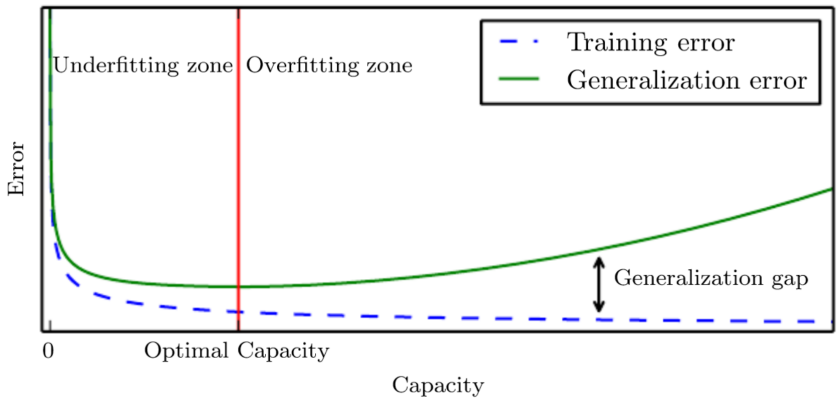


Figura 2.10: Relación típica entre la capacidad y el error. A medida que uno se aleja de la capacidad óptima la generalización se pierde, correspondiendo con una diferencia mayor entre el error de entrenamiento y el de test. Fuente: [1]

2.10. Regularización

Uno de los problemas con los que cualquier red neuronal tendrá que lidiar es el *overfitting* descrito anteriormente (véase el apartado 2.9). Para evitar el *overfitting* se han diseñado estrategias de regularización. Estas estrategias tratan de reducir el error de test, aún teniendo que aumentar el error de entrenamiento. La búsqueda de nuevo métodos de regularización ha sido una de las líneas de investigación más extensa en este campo del *Deep Learning* [1].

Existe bastantes métodos de regularización pero en esta documentación se presentarán solamente algunos de ellos, especialmente los que han sido usados en la realización de este trabajo.

2.10.1. *Data augmentation*

La mejor manera de que un modelo generalice mejor es que pueda tener más datos de entrenamiento. La cantidad de datos de entrenamiento es finita, pero gracias a la técnica

conocida como *data augmentation* (DA) se puede hacer crecer el conjunto con el que se entrena, creando nuevos casos de entrenamiento ficticios, basados en las imágenes de entrenamiento reales.

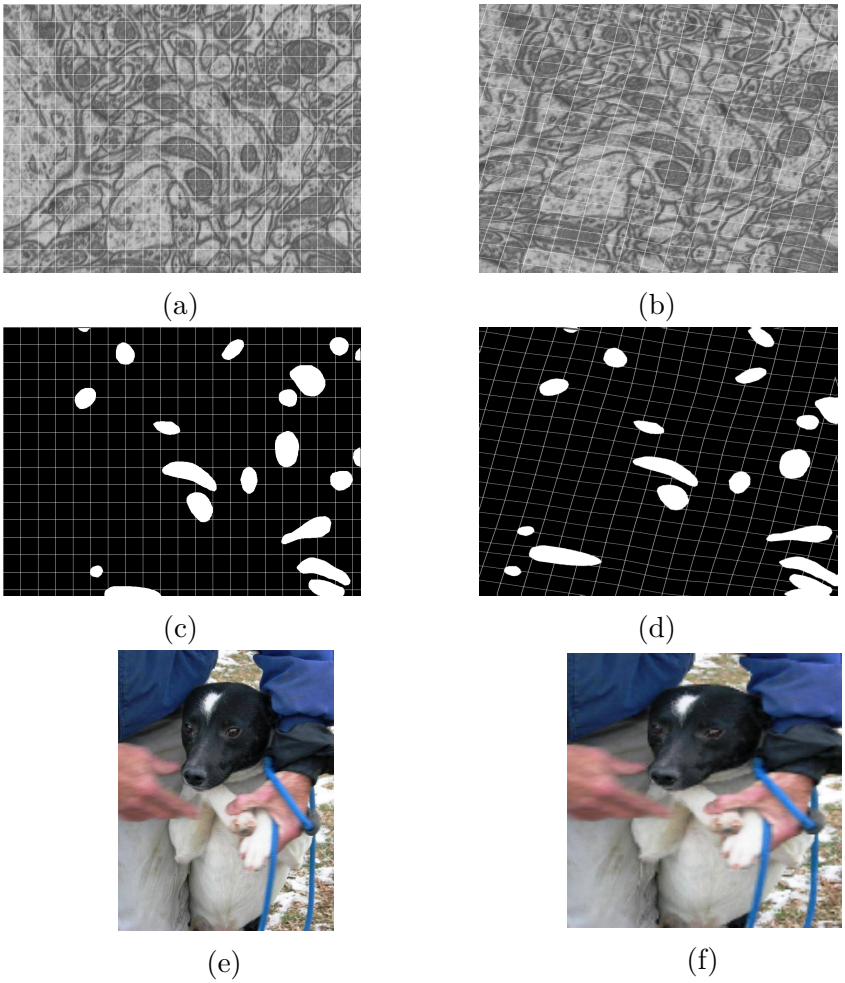


Figura 2.11: Data augmentation: (a) y (b) corresponden a una transformación elástica de una imagen de microscopio. (c) y (d) corresponde a la misma imagen que (a) y (b), realizando de nuevo una transformación elástica, pero de la máscara de las mitocondrias de la imagen. Más abajo, (f) corresponde a la redimensión de la imagen de un perro (e) extraída del *dataset* "Dogs vs Cats" [30].

Esta técnica se hace aún más efectiva cuando los datos con los que se trabajan son imágenes, ya que las imágenes realmente incluyen una enorme cantidad de transformaciones entre sí que pueden ser simuladas fácilmente. Eso sí, habrá que tener cuidado de realizar según qué transformaciones a según qué datos. Si por ejemplo, se está trabajando en un conjunto de datos en los que hay animales como gatos, perros, caballos, etc. una transformación de girar la imagen 90 grados o 180 puede que no tenga sentido, ya que los animales estarían boca rriba, sin embargo, un pequeño zoom o una pequeña deformación

sí que podrán ser beneficiosos.

Además de aumentar el conjunto de entrenamiento, como uno ya habrá podido imaginar, esta técnica ayuda a que el modelo haya tenido un conjunto de entrenamiento más heterogéneo, lo que mejora aún más la clasificación. Un ejemplo de DA está presentado en la Figura 2.11, donde las primeras cuatro son imágenes con las que se ha trabajado en el proyecto y a las que se les ha realizado una deformación elástica (se ha dibujado una rejilla o *grid* para que ayude al lector a ver la transformación, pero en las imágenes originales no está presente). Las dos últimas imágenes han sido obtenidas de las pruebas realizadas en el *dataset* de "Perros y Gatos" (véase la sección 5.3), donde se ha redimensionado la imagen de un perro.

2.10.2. Dropout

Otra de los métodos de regularización más usados es el llamado *dropout* presentado por Srivastava *et al.* [31]. El término de "dropout" se refiere al hecho de realizar un descarte, *drop* en inglés, de ciertas neuronas de la red. Como una aproximación, este método puede ser visto como la realización de un *bagging*[6] sobre las redes neuronales.

La combinación de muchos modelos siempre mejora el rendimiento de los algoritmos de ML [31]. Sin embargo, el realizar esto mismo con las redes neuronales, especialmente si estas son grandes, es demasiado costoso. La combinación de muchos modelos es útil cuando los modelos son diferentes entre sí, lo que supone o bien que los datos con los que se hayan entrenado sean diferentes o que, para el caso de las redes neuronales, que cada red tenga diferente arquitectura. El utilizar redes con diferentes arquitecturas es computacionalmente muy costoso ya que la optimización de sus hiperparámetros es una tarea difícil. Por otro lado, normalmente las redes neuronales requieren de muchos datos de entrenamiento, por lo que no es una buena opción el tener que dividir estos datos en subconjuntos para luego poder aplicarlos a cada red.

La aplicación de esta técnica de *dropout* resuelve esos dos problemas anteriores mediante el descarte de ciertas neuronas de manera aleatoria de las capas ocultas o de *input*, un ejemplo de ello se encuentra en la Figura 2.12. En el caso más simple, cada neurona tendrá asignada una probabilidad p de ser descartada. De esta manera, si se ha decidido descartar cierta neurona, ella y todas sus conexiones también son descartadas.

[6] El algoritmo de *bagging* [32] es utilizado para reducir la varianza en problemas de clasificación o regresión, ayudando así a prevenir el *overfitting*, mediante la combinación de muchos modelos. Por ello, esta técnica es considerada por si misma una técnica de regularización. Se basa en crear subconjuntos de entrenamiento (m), de tamaño menor que el conjunto de entrenamiento entero (n), mediante un muestreo uniforme y con reemplazos. Con estos m subconjuntos, se realiza el entrenamiento de m modelos y finalmente se combinan los resultados de todos para producir la respuesta final.

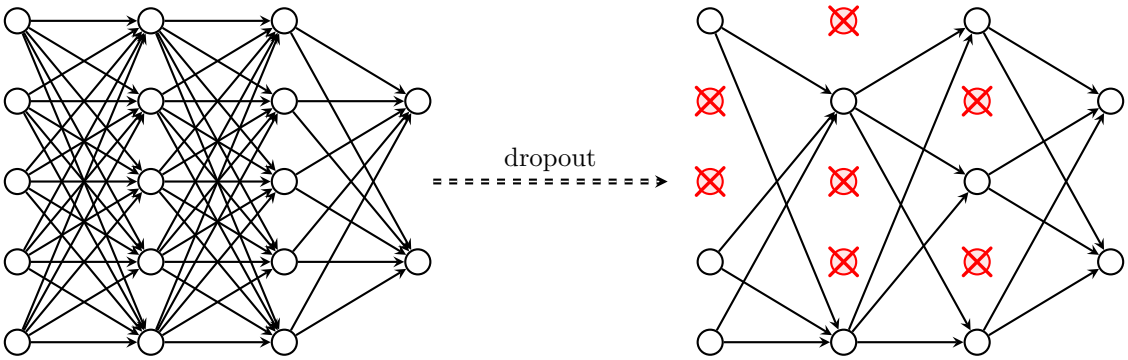


Figura 2.12: Ejemplo de *dropout* donde se han descartado de la red de la izquierda neuronas al azar y sus conexiones, teniendo como resultado la red de la derecha. Fuente: Github [7].

Una red de n neuronas tendrá entonces 2^n posibles redes alternativas que se pueden construir a partir de ella mediante el *dropout*. Por lo tanto, la aplicación de esta técnica se realiza en cada iteración que actualiza los pesos de la red, utilizando una nueva red "reducida", que está basada en la red original y que ha descartado las neuronas con su probabilidad p.

Como se ha mencionado, el *dropout* se utiliza a la hora de realizar el entrenamiento de la red, y no se aplica a la hora de trabajar con los datos de test, ya que sería muy costoso el tener que promediar todos los posibles modelos reducidos. Sin embargo, se realiza una aproximación simple que funciona muy bien en la práctica: la idea se basa en utilizar una red sin *dropout* a la hora de usar el conjunto de test y escalar los pesos de la red haciendo que sean versiones de los pesos que se han utilizado a la hora de hacer el entrenamiento. Para ello, lo que se hace, es escalar esos pesos con cada probabilidad p de cada neurona, asegurando que la salida "esperada" de cada neurona en la fase de test sea la misma, o tenga el mismo comportamiento, que la salida que se ha obtenido a la hora de realizar el entrenamiento. Realizando este escalado, las 2^n redes pueden ser combinadas en una sola a la hora de trabajar en el conjunto de test, reduciendo el error de test y consiguiendo que la red generalice mucho mejor [31].

2.10.3. *Early stoppping*

Antes de empezar, se realizará la siguiente aclaración para el lector: en general en los algoritmos de ML se utiliza el llamado conjunto de validación para poder saber cómo de bien se está comportando el algoritmo en cuestión frente a datos no vistos en el entrenamiento pero sin llegar a usar los datos de test. Este conjunto de validación será

[7] Imagen creada a partir del proyecto de Github de Petar Veličković: https://github.com/PetarV-/TikZ

realmente una parte del conjunto de entrenamiento que se ha separado antes de empezar a entrenar el modelo. Así pues, se llamará error de validación al error cometido en este conjunto de datos, de la misma manera que el error de entrenamiento y de test definidos anteriormente.

A la hora de entrenar un modelo con suficiente capacidad de representación para sobreajustarse a los datos, se observará que casi siempre el error de entrenamiento cada vez será más y más bajo, mientras que el error de validación, después de haber bajado con el de entrenamiento, empezará a subir en un momento determinado. Lo que tendrá como resultado un curva convexa, como la que aparece en la Figura 2.13.

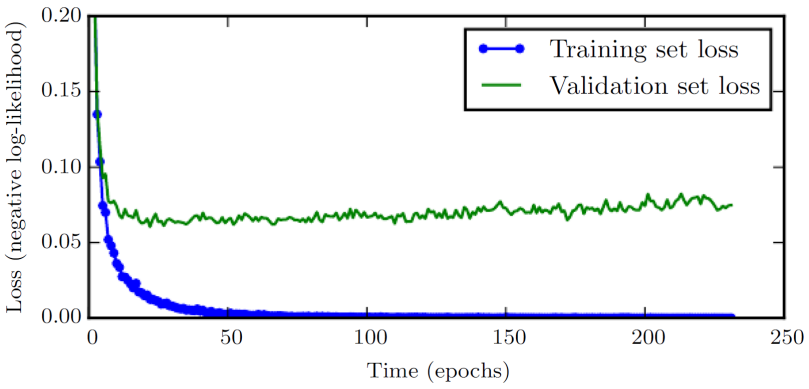


Figura 2.13: Ejemplo de comportamiento de los errores de entrenamiento y validación a la hora de entrenar un modelo a lo largo de las épocas que se hayan definido. Fuente: [1].

Para evitar esto se hace uso de una técnica de regularización muy común y sencilla llamada *early stopping* que, como su propio nombre indica, se trata de parar el entrenamiento del modelo antes de que finalice.

En la sección 2.6 ya se habló de que el criterio de parada más común en el aprendizaje de la red era el definir un número de épocas de la red. No obstante, hay que elegir este parámetro con delicadeza ya que, como se ha visto a lo largo de esta sección, un entrenamiento largo puede llevar al modelo a "sobreaprender" demasiado, y de manera inversa, un entrenamiento corto puede llegar a ser insuficiente para que aprenda.

Si se guardan en cada iteración los parámetros del modelo resultante, se podrá devolver el mejor de ellos, es decir, el que tenga el error de validación más bajo. Con ello se espera que el error de test también sea más bajo, ya que, hay que recordar que los dos conjuntos están bajo las mismas condiciones de no haber sido utilizados en el entrenamiento. De esta manera, cuando el algoritmo termine, se devolverá el mejor modelo de entre todos los que se han obtenido, en vez de el último. El algoritmo terminará, o bien cuando se haya alcanzado el número máximo de épocas definidas, o cuando no se haya mejorado en un número de iteraciones predefinido, comparándolo con el mejor modelo guardado con

error de validación más bajo. A este número de iteraciones predefinidas sin obtener una mejora más adelante se le hará referencia como paciencia o *patience* en inglés.

Capítulo 3

Redes neuronales convolucionales

Las redes neuronales convolucionales o redes de convolución, *convolutional neural network* (CNN o ConvNet) en inglés, presentadas en [33], hacen referencia a unos tipos de redes muy parecidas a las redes neuronales vistas en el capítulo 2. En este caso, las CNNs hacen la presunción de que el tipo de datos que van a procesar tiene una tipología tipo "grid'. Dentro de esta topología se incluirían por ejemplo las series temporales, que pueden ser vistas como un *grid* 1D, y las imágenes, que pueden ser vistas como un *grid* de 2D[1]. El nombre de las CNNs viene por dado por realizar las operaciones matemáticas llamadas "convoluciones" que se explicarán más adelante.

3.1. Arquitectura

Recordando el funcionamiento de las redes neuronales convencionales explicadas en el capítulo 2 se tiene que, una red está formada por un grupo de neuronas, agrupadas en las llamadas capas de la red, y que están interconectadas de alguna manera con las anteriores y/o posteriores capas, pero no hay conexiones entre las neuronas de una misma capa.

Por otro lado, cabe destacar que las ANNs no escalan bien con imágenes ya que, por ejemplo, utilizando imágenes tan pequeñas como de $32 \times 32 \times 3$ píxeles (siendo la tercera coordenada el número de canales, en este caso 3 por ser RGB), una neurona de la primera capa tendrá $32 * 32 * 3 = 3072$ pesos. Con una imagen un poco más grande, como de $200 \times 200 \times 3$ píxeles, los pesos serían $200 * 200 * 3 = 120000$. Por lo tanto, cuando se quiera formar una red con multitud de neuronas y capas, lo que es habitual, el número de parámetros a entrenar puede ser demasiado alto y tener como resultado un sobreajuste u *overfitting* sobre los datos [11].

Dado que la entrada de la red son imágenes, las CNNs tienen tres dimensiones en

cada capa: ancho, alto y fondo. Por ejemplo, en las imágenes de $32 \times 32 \times 3$ píxeles de antes, la neurona de la primera capa no estará completamente conectada con la imagen, sino que, como se explicará más adelante, se conecta solo con una región de ella. Además, si las imágenes de ejemplo pertenecen al *dataset* CIFAR-10[1][34], la capa final de la red será un vector de tamaño igual al número de clases que se quieren clasificar, que en este caso será $1 \times 1 \times 10$.

3.2. Tipos de capas

En esta sección se describirán los tipos de capas que contienen las CNNs. Por simplificar, se seguirá con el ejemplo anterior, con imágenes de $32 \times 32 \times 3$ del dataset CIFAR-10. Las capas principales son las siguientes:

- INPUT, de tamaño $32 \times 32 \times 3$, se corresponde con la primera capa de la red, es decir, la entrada, por lo que representará una imagen de CIFAR-10.
- CONV, representará las capas de convolución. Capa que realizará la operación de convolución por la que las neuronas estarán conectadas mediante la multiplicación de sus pesos con una pequeña región de la imagen en vez de con la imagen completa. Si se decide usar, por ejemplo, 12 filtros, el resultado de esta capa será de tamaño $32 \times 32 \times 12$. Se explicará más detalladamente su funcionamiento en la siguiente sección.
- RELU, capa que representará la función de activación, ReLU en este caso (véase la sección 2.2 para más información).
- POOL, capa que reducirá las dimensiones de la imagen mediante un *downsampling*. Si se opta, por ejemplo, por reducir la dimensionalidad a la mitad, la salida de esta capa será $16 \times 16 \times 12$.
- FC, capa que representará la conexión completa con la capa anterior y es usada como última capa que dará los resultados de las clases. En este caso será una capa de tamaño $1 \times 1 \times 10$ donde el 10 corresponde con el número de clases existentes y cada valor del vector será la probabilidad que tiene la imagen de representar a esa clase.

De esta manera, las CNNs consisten en una unión de este tipo de capas seguidas una detrás de otra hasta que, finalmente, de la imagen *input* se pase a un vector de 10 posiciones representado las probabilidades que tiene esa imagen de pertenecer a cada

[1] CIFAR-10 es un *dataset* comúnmente usado que está compuesto por imágenes de $32 \times 32 \times 3$ píxeles representando 10 clases distintas: *airplane*, *automobile*, *bird*, *cat*, *deer*, *dog*, *frog*, *horse*, *ship* y *truck*.

una de las clases. Un ejemplo de esto se puede observar en la Figura 3.1, donde se han visualizado únicamente 5 clases por simplificar (recuérdese que CIFAR-10 contiene 10 clases) pero realmente la última capa marcada como FC representaría la probabilidad de cada clase. En este ejemplo, la red parece haber acertado clasificando la imagen como un coche (clase "car").

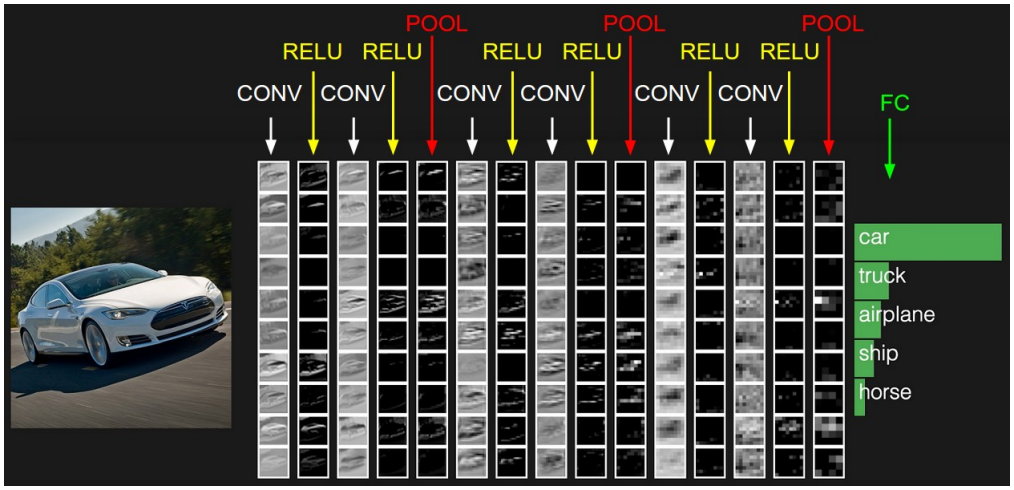


Figura 3.1: Estructura de capas de una *convolutional neural network* (CNN). Fuente: [11].

3.3. Capa de convolución

Entender el funcionamiento de la capa de convolución (CONV) es crucial para poder comprender el funcionamiento de las CNNs. Esta capa es la que realizará el cálculo más costoso de todos. Como se ha mencionado antes, las neuronas solamente estarán conectadas a una región de la imagen y esta conexión se realizará con un filtro. Por ejemplo, siguiendo con las imágenes de tamaño $32 \times 32 \times 3$, un filtro podría ser definido como una matriz de tamaño $4 \times 4 \times 3$ (el ancho y largo del filtro puede variar según se quiera pero siempre tendrá que tener en cuenta toda la tercera dimensión, por eso en este caso también se termina con un 3). Este filtro se irá deslizando (o mejor dicho, realizando una convolución) por la imagen, empezando por la esquina superior izquierda, por todo lo ancho y largo de la imagen, multiplicando elemento a elemento los valores del filtro y los de la propia imagen. Esto producirá una matriz de valores 2D que será la respuesta de la imagen a ese filtro. Entonces, la red aprenderá a partir de los filtros que se activarán con formas específicas de las imágenes como bloques de color, rectas, círculos, esquinas etc. Se definirá la cantidad de filtros a aprender por cada capa y se agruparán todos ellos en la dimensión z.

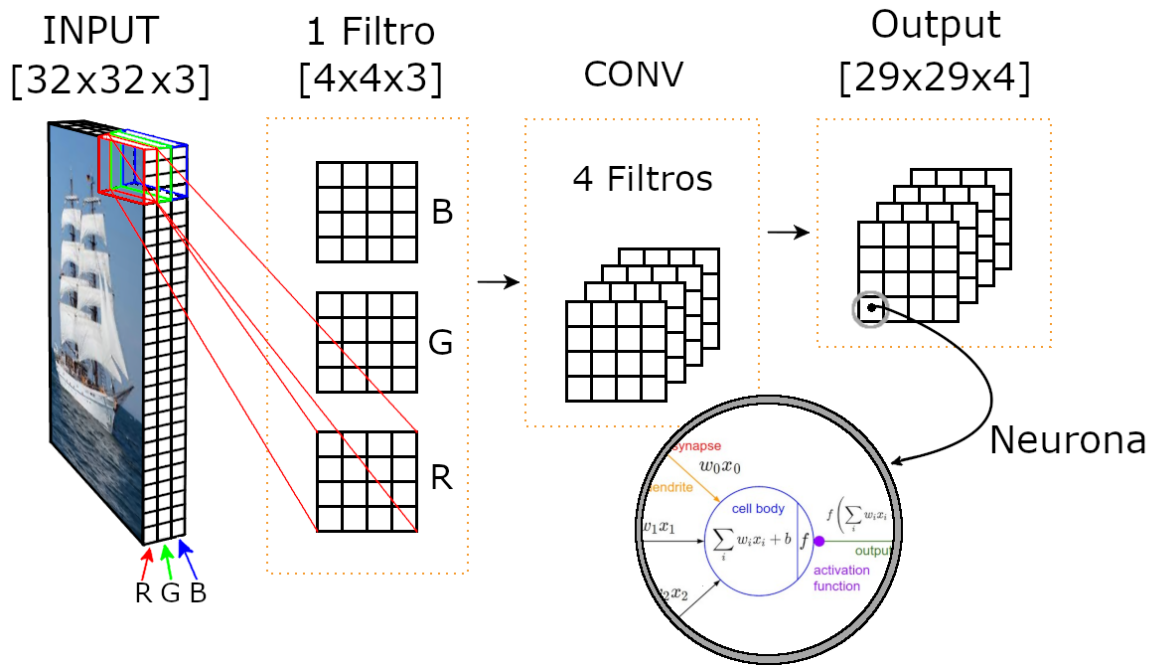


Figura 3.2: Estructura de capas de una *convolutional neural network* (CNN).

En la Figura 3.2, se puede ver un ejemplo de como sería una capa de convolución y sus filtros. A la izquierda está representada, en este caso, una imagen de $32 \times 32 \times 3$ de un barco que será el *input* que recibe la red y la imagen que se quiere etiquetar (nótese que son 3 canales RGB los que tiene la imagen). A su derecha se ha dibujado el primer filtro, que como debe coger siempre toda la dimensión z, está representado con tres *grids* de 4×4, uno por cada color RGB (aunque normalmente son vistos como un único *pack* de 3). Para que sea más claro visualmente, solo se ha dibujado la conexión del *grid* de R a su respectivo trozo R de la imagen, pero realmente los otros dos también están conectados. Viendo entonces el filtro como un *pack* 3D o cubo que forman los tres colores en la esquina superior derecha del barco, uno puede apreciar que únicamente está conectado con una parte de la imagen y no con toda ella. Este filtro es el que se irá moviendo, realizando una convolución, a través de la imagen.

La siguiente estructura hacía la derecha representa la capa de convolución, que estará formada en este caso por 4 filtros (que no tiene nada que ver con el tamaño 4×4 de los filtros). Por lo tanto, esta capa realmente tendrá 4 veces el *pack* de los 3 *grids* que están representados a su izquierda, por lo que podría ser visto como una matriz de tamaño $4 \times 4 \times 4 \times 3$.

Finalmente, la convolución del filtro tendrá como resultado el *output* que está representado más a la derecha. Para ello, se realiza una multiplicación elemento a elemento entre el filtro y el trozo de la imagen, por lo que, llevado al campo de los pesos y sesgos, un filtro será realmente una matriz 3D de pesos al que además habrá que añadirle el término

de sesgo que aquí no ha sido dibujado. En este caso, se generará un *output* de tamaño $29 \times 29 \times 4$: la dimensión z será 4, una por cada filtro, las dos primeras dimensiones sin embargo, será de 29×29 porque, si se mueve el filtro un píxel hacía la derecha cada vez, se aplicará el filtro 29 veces horizontal y verticalmente. Conectando está explicación con lo visto hasta ahora, se pude decir que una neurona está representada como un valor dentro de la matriz 3D que representa el *output* de la capa de convolución.

3.3.1. Definiciones de la capa de convolución

Una vez llegados a este punto en el que el lector se ha hecho una idea general del funcionamiento de las capas de convolución, es hora de profundizar en ciertos parámetros que se pueden configurar dentro de ellas, en concreto:

- La profundidad o *Depth*, que indica el número de filtros a aplicar en la capa de convolución. Nótese que este número está relacionado directamente con el número de parámetros de la red, es decir, los pesos y sesgos, como se ha comentado anteriormente. Por lo tanto, aumentar el número de filtros indicará un aumento en los parámetros que la red tendrá que aprender.
- El campo de visión o *Receptive field*, que indica cómo de grande serán los filtros. Si por ejemplo, se decide establecer el valor de este parámetro a 5, indicará que los filtros serán de tamaño 5×5.
- La zancada o *Stride*, que indica la longitud del salto en cada deslizamiento del filtro por la imagen. En el ejemplo de arriba se ha indicado que el deslizamiento iba a ser de uno en uno pero realmente este valor puede ser mayor.
- El relleno o *Padding*, que se utiliza para establecer un marco o borde alrededor de la imagen. Hay varias opciones para decidir los valores que tendrán, pero una opción es rellenar con ceros, o realizar un "espejo" copiando los valores de los píxeles de las últimas columnas y filas (*mirroring*), etc.

Los valores de estos parámetros decidirán el tamaño del *output* de la capa de convolución con la fórmula:

$$\frac{(Receptive\_field - Depth + 2 * Padding)}{Stride} + 1 \tag{3.1}$$

Por ejemplo, para determinar que el *output* del ejemplo de la Figura 3.2 era $29 \times 29 \times 4$, se calcula de la siguiente manera: $(32 - 4 + (2 * 0))/1 + 1 = 29$.

3.3.2. Compartición de parámetros

Con un ejemplo más realista de red, como la *AlexNet* que se presentará en las secciones posteriores, uno podrá intuir que el número de parámetros, es decir, pesos y sesgos, con el que tiene que lidiar es muy grande. Por ejemplo, en la primera capa de *AlexNet* se utiliza un $receptivefield = 11$, es decir filtros de $11 \times 11 \times 3$, $padding = 0$, $stride = 4$ y $depth = 96$ (96 filtros). El resultado entonces es de $(227 - 11)/4 + 1 = 55$, es decir $55 \times 55 \times 96$, dando lugar a un total de $55 * 55 * 96 = 290400$ neuronas. Si cada una tiene $11 * 11 * 3 = 363$ pesos y 1 sesgo, el número de parámetros para únicamente la primera capa sería de $290400 * 364 = 105705600$, lo que es un número altísimo.

Por ello, se hace la siguiente asunción razonable con la que se reduce el número de parámetros: si se está aplicando un filtro a cierta región de la imagen en busca de una característica concreta, resulta interesante la aplicación de ese mismo filtro en otras regiones de la imagen para buscar exactamente el mismo patrón. En otras palabras, interesa aplicar el mismo filtro entre las neuronas de la misma dimensión z (esas 55×55 neuronas por cada una de las 96 partes), lo que implicará que compartan los mismos pesos y sesgos. De esta manera, el número de parámetros se verá reducido a $96 * 11 * 11 * 3 = 34848$.

Esta es la razón por la que se conoce como "filtro" a este conjunto de pesos por cada capa en la profundidad z. Un ejemplo visual de los filtros que se podrían aplicar a lo largo de los 96 componentes de la dimensión z están representados en la Figura 3.3. Nótese que no solo se pueden relacionar con la búsqueda de líneas en cualquier sentido, detectando bordes y/o esquinas, sino que también pueden definirse para detectar ciertas combinaciones de colores que resulten interesantes.

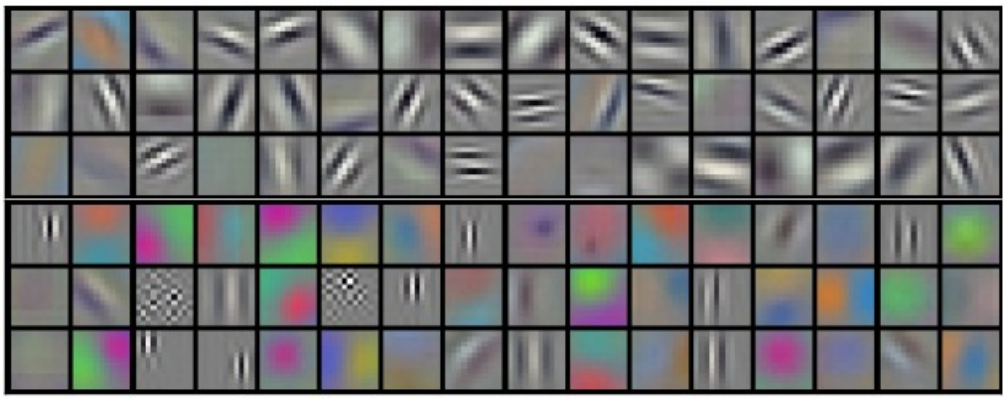


Figura 3.3: Ejemplos de los 96 filtros que se podrían aplicar en la primera capa de la red *Alexnet*. Fuente: [11].

3.3.3. Ejemplo de la capa de convolución

A continuación, en la Figura 3.4 está representado un ejemplo más real del funcionamiento de la capa de convolución, donde ya se han puesto valores reales con los que realizar las operaciones.

Los tres *grids* de la izquierda representarían la imagen del barco que se ha presentado anteriormente, donde cada canal sería un *grid*. En este caso, se ha especificado tener un *padding* de 1 píxel, dibujado en la imagen con color gris haciendo de "marco" de la imagen.

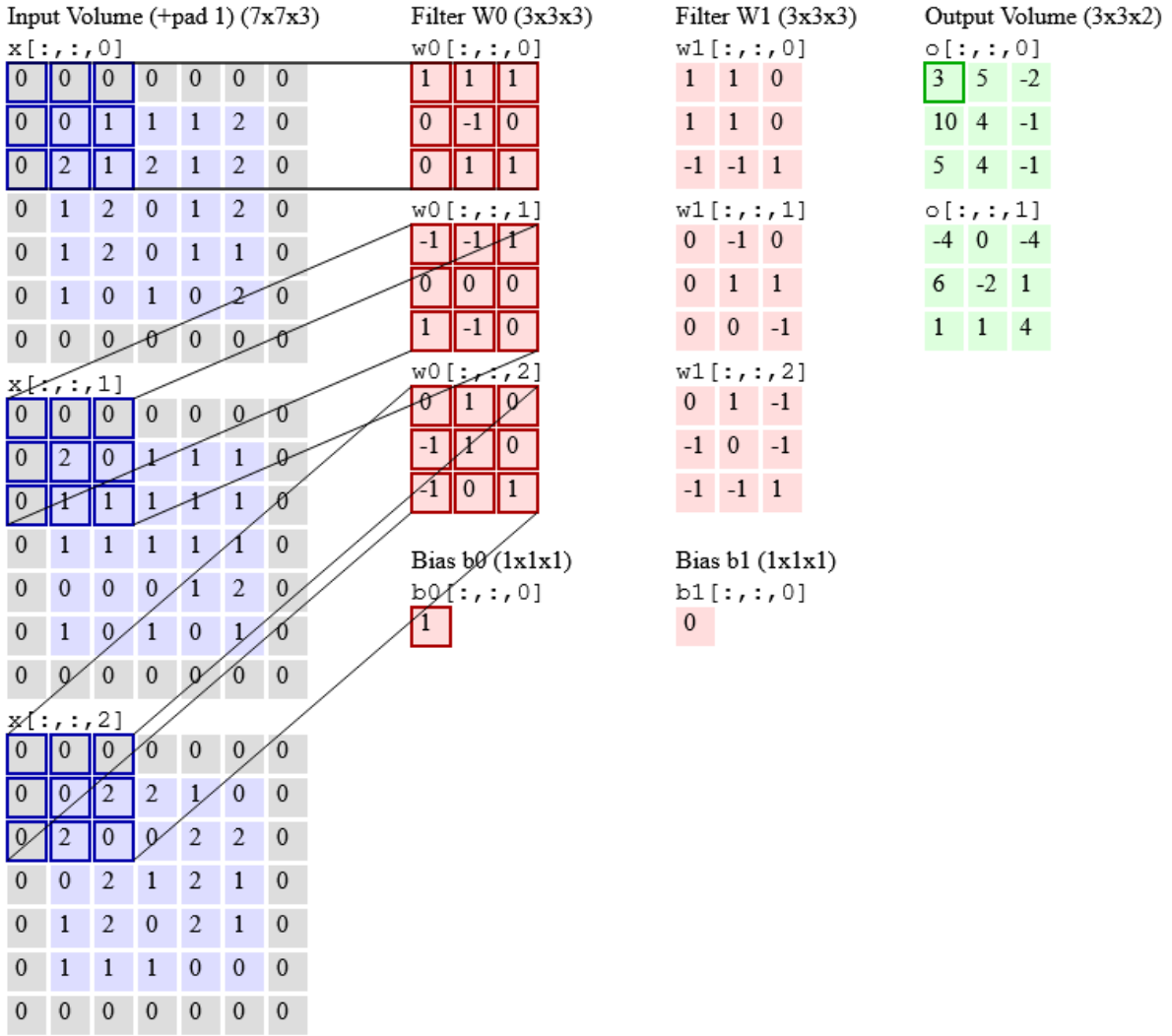


Figura 3.4: Estructura detallada de la capa de convolución de una *convolutional neural network* (CNN): primer paso. Fuente: [11].

En medio, en color rojo, se muestran 2 filtros donde se puede observar como se

representan los pesos y los sesgos. En este caso su tamaño es de $3 \times 3 \times 3$ en vez de $4 \times 4 \times 3$ como en el caso anterior, por lo que $depth = 2$ y $receptive field = 5$. Se han distinguido tres *grids* para cada filtro, como era de esperar teniendo 3 canales en la dimensión z, y cada uno está conectado con el correspondiente trozo y dimensión de la imagen.

Para finalizar, está representado en verde el resultado de la convolución. Como era de esperar, al tener únicamente dos filtros, el *output* tendrá dos *grids* de 3×3, es decir, $3 \times 3 \times 2$.

Si, por ejemplo, se establece el valor del último parámetro restante, $stride = 2$, las operaciones que tendrían que realizarse para obtener el valor 3 que aparece marcado en *output* serían las siguientes:

1. Multiplicación elemento a elemento entre $x[:, :, 0]$ y $w_0[:, :, 0]$, es decir, $(0 * 1 + 0 * 1 + 0 * 1) + (0 * 0 + 0 * (-1) + 1 * 0) + (0 * 0 + 2 * 1 + 1 * 1) = 0 + 0 + 3 = 3$.
2. Multiplicación elemento a elemento entre $x[:, :, 1]$ y $w_0[:, :, 1]$, es decir, $(0 * (-1) + 0 * (-1) + 0 * 1) + (0 * 0 + 2 * 0 + 0 * 0) + (0 * 1 + 1 * (-1) + 1 * 0) = 0 + 0 - 1 = -1$.
3. Multiplicación elemento a elemento entre $x[:, :, 2]$ y $w_0[:, :, 2]$, es decir, $(0 * 0 + 0 * 1 + 0 * 0) + (0 * (-1) + 0 * 1 + 2 * 0) + (0 * (-1) + 2 * 0 + 0 * 1) = 0 + 0 + 0 = 0$.
4. Suma de todos los resultados obtenidos más el sesgo b_0, que en este caso es 1, es decir, $3 - 1 + 0 + 1 = 3$.

Para el siguiente paso, que es el que esta representado en la Figura 3.5, siguiendo el mismo procedimiento se obtendría el valor 5. Nótese que el filtro se ha desplazado a la derecha por la imagen realizando un salto de 2, como se ha definido con el parámetro *stride*.

El mismo procedimiento se seguiría hasta terminar con el primer filtrado, se pasaría entonces al segundo, hasta terminar la capa de convolución completa. Un ejemplo interactivo de todo ello se puede ver en el curso de Stanford en el que se ha apoyado la documentación [11], más concretamente en la explicación de las CNN [2].

[2] http://cs231n.github.io/convolutional-networks/

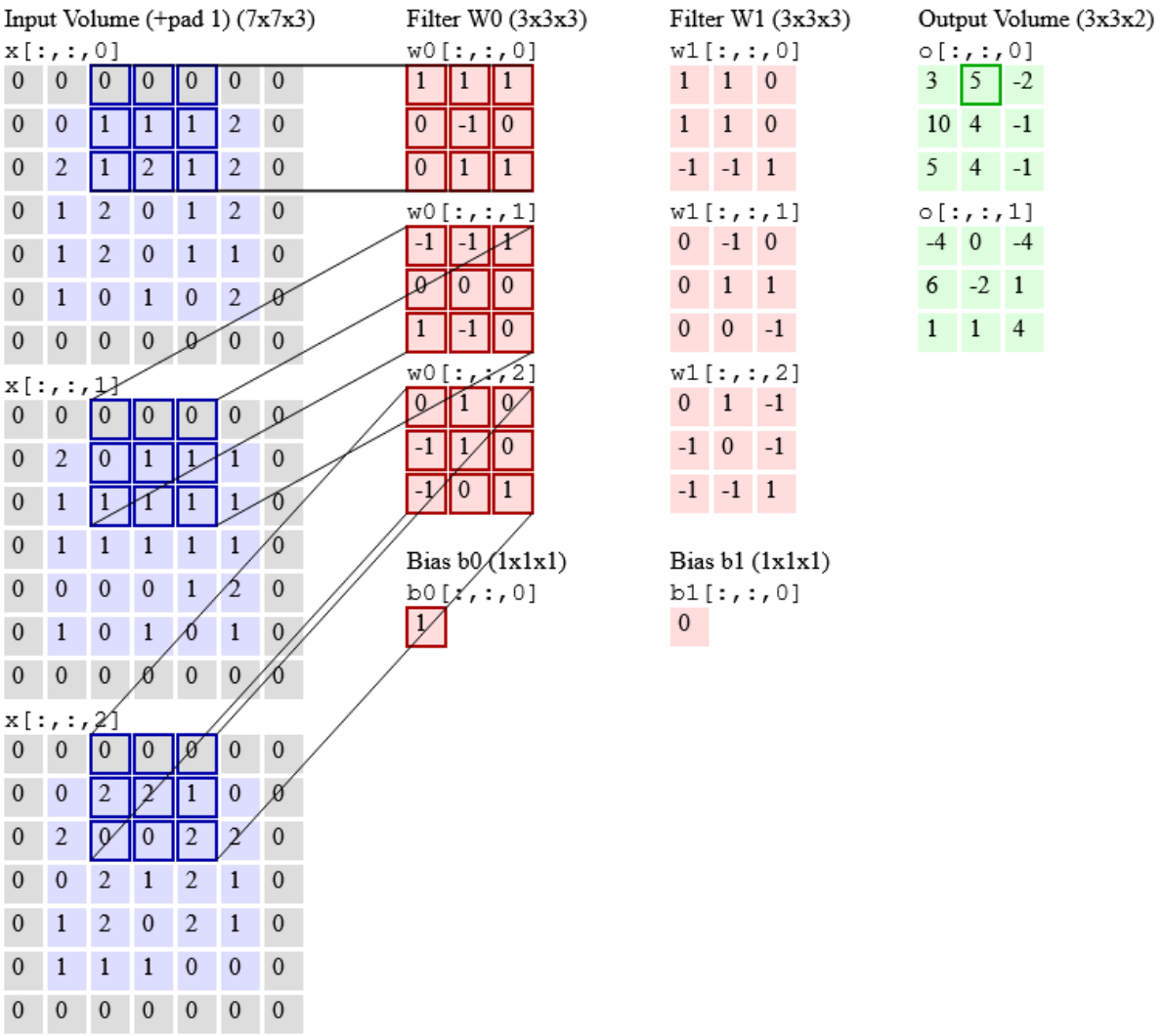
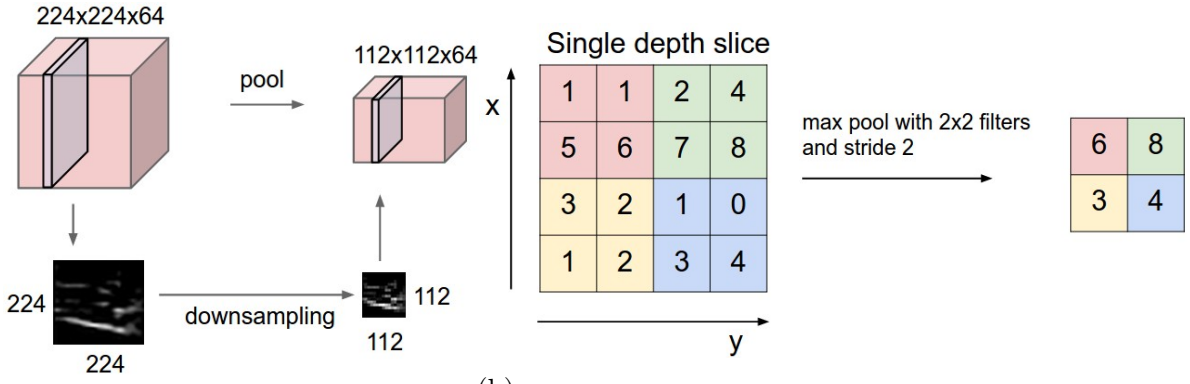


Figura 3.5: Estructura detallada de la capa de convolución de una *convolutional neural network* (CNN): segundo paso. Fuente: [11].

3.4. Capa de *pooling*

Otra capa importante dentro de las CNN es la llamada capa de *pooling* (POOL), que realizará la labor de reducir la representación espacial de las capas de convolución, reduciendo así el número de parámetros y ayudando a que no se realice un sobreajuste [11]. Uno de las capas de pooling más usadas es la denominada *max pooling* [35], la cual produce como *output* el máximo valor dentro de una vecindad rectangular definida. Otras opciones de *pooling* populares optan por realizar la media sobre todos los valores dada una vecindad rectangular, sobre la norma L^2 dada una vecindad rectangular o una media ponderada basada en la distancia de un píxel central [1].

La opción más típica de *pooling* divide las dimensiones entre dos mediante un filtro de 2×2 y un *stride* de 2. En la Figura 3.6a se puede ver un ejemplo de ello, reduciendo las dimensiones a la mitad. En la Figura 3.6b se muestra un ejemplo con valores numéricos de como sería esa misma transformación.



(a) (b)

Figura 3.6: Ejemplo de la capa de *max pooling*. En (a) se puede ver como se reduce la dimensionalidad de la imagen a la mitad (menos en la dimensión z que se mantendrá el tamaño), y en (b) se muestra un ejemplo numérico de la misma transformación. Fuente: [11].

3.5. Capa densamente conectada

Cada neurona de la capa densamente conectada (FC) tiene conexiones con cada una de las neuronas de la capa anterior. Ya se habló de esta conexión múltiple cuando se presentó la arquitectura de la red neuronal en la sección 2.4.

La diferencia de FC con CONV está en que esta última está conectada únicamente a una parte del *input*, mientras que FC está conectada completamente. Nótese que una CONV puede ser implementada por una FC, aunque la matriz de pesos en caso de FC tendrá que tener valores no cero únicamente en la región en la que está conectada la capa CONV.

De manera inversa, una FC también puede convertirse en una capa CONV. Si se establecen el mismo número de filtros y si la capa CONV tiene como tamaño del filtro la misma dimensión de su *input*. Es decir, si el *input* es, por ejemplo de tamaño $32 \times 32 \times 3$, la capa CONV tendrá que tener un filtro de tamaño 32×32, con un $stride = 1$, $padding = 0$ y 3 como número de filtros. De esta manera la salida será $1 \times 1 \times 3$ al igual que sería la capa FC.

3.6. Ejemplo real de CNN: AlexNet

Desde hace unos años, las CNNs conforman el estado del arte en cuanto a la segmentación de imágenes se refiere. A pesar de que las CNNs ya existían desde hace aproximadamente 20 años [36], no fueron realmente exitosas hasta el trabajo presentado por Krizhevsky *et al.* en [37], con la famosa red llamada *AlexNet*, ganando la competición ImageNet del año 2012 [38]. Desde ese trabajo diferentes tipos de CNN han aparecido y se ha abierto una amplia línea de investigación en lo que se refiere a la visión por computador [39, 40]. La red *AlexNet* sigue la arquitectura que se muestra en la Figura 3.7.

Anualmente, desde el 2010 hasta el 2017, se realizaba una competición llamada "ImageNet Large-Scale Visual Recognition Challenge (ILSVRC)" [38] en la que se propone un *dataset* sobre el que se deben realizar diferentes tareas como la de clasificar las imágenes, detectar objetos en imágenes etc. En el año 2012, la competición trataba de la clasificación de imágenes y para ello proponía un *dataset* de aproximadamente 1000 categorías cada una con 1000 imágenes, en total unos 1.2 millones de imágenes, donde 50.000 se usarían para validación y 150.000 para test. Estas imágenes realmente fueron obtenidas de *ImageNet*, que es una base de datos de imágenes enorme: 15 millones de imágenes clasificadas pertenecientes a aproximadamente 22.000 categorías diferentes [41]. Como las imágenes de este *dataset* tienen tamaños distintos, Krizhevsky *et al.* centraron las imágenes y entrenaron la red con recortes de 227×227.

La arquitectura de *AlexNet* está formada por 5 capas de convolución (CONV) y 3 capas densamente conectadas (FC) tal y como se muestra en la Figura 3.7. Se decribirá la arquitectura desde la primera capa de la izquierda hacía la derecha:

1. Para empezar, está la entrada a la red representada con la imagen de los gatos, que en el caso de esta red ya se ha mencionado que es de $227 \times 227 \times 3$.

2. CONV1, es la primera capa de convolución con 96 filtros de tamaño $11 \times 11 \times 3$, con un *stride* de 4, teniendo como resultado una matriz de tamaño $96 \times 55 \times 55$ (recuérdese la fórmula 3.1 donde $((227 - 11)/4 + 1 = 55)$. A partir de aquí se cambiará ligeramente la notación descartando los 3 canales RGB para que sea más claro el ejemplo.

3. A continuación hay una capa de *pooling*, representada en naranja oscuro y pegada a la capa de convolución. En este caso se ha usado un filtro 3×3 con un *stride* de 2, por ello el resultado es de $96 \times 27 \times 27$, que será representado por CONV2.

4. CONV3 representa el resultado de realizar una convolución en CONV3, que en este

[3]Imagen creada a partir del proyecto de Github de Haris Iqbal: https://github.com/HarisIqbal88/PlotNeuralNet

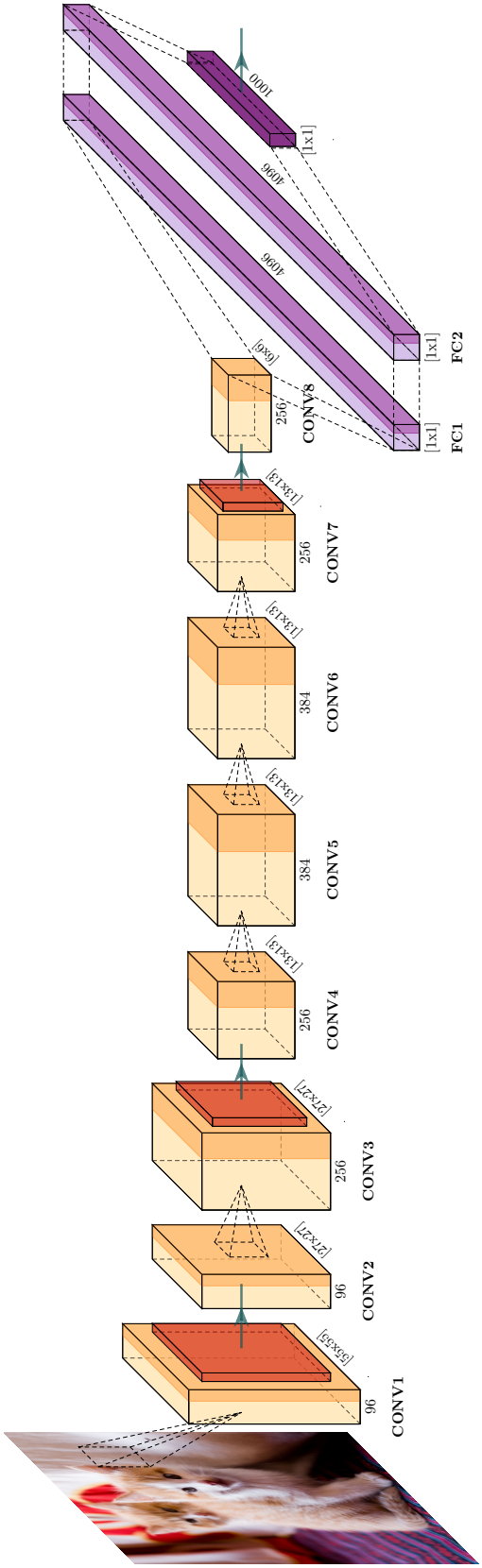


Figura 3.7: Arquitectura de AlexNet. Fuente: Github [3].

caso tiene 256 filtros de 5×5, con 2 de *padding* y *stride* 1, dando como resultado una matriz de tamaño $256 \times 27 \times 27$.

5. CONV4 surge de la aplicación de la capa de *pooling* a CONV3, que en este caso ha sido con un filtro 3×3 y *stride* 2, dando lugar a una matriz de tamañao $256 \times 13 \times 13$.

6. CONV5 es resultado de aplicar una convolución a CONV4, aplicando esta vez 384 filtros de 3×3 con *padding* de 1, obteniendo como resultado una matriz de $384 \times 23 \times 23$.

7. CONV6 es el resultado de aplicar a una convolución a CONV5 de 384 filtros de 3×3 y *padding* de 1, que como resultado se obtiene una matriz de $384 \times 13 \times 13$.

8. CONV7 es el resultado de aplicar una convolución a CONV6 de 256 filtro de 3×3 con *padding* de 1 de nuevo. Como resultado se obtiene una matriz de $256 \times 13 \times 13$.

9. Se realiza una capa de pooling a CONV7 y se obtiene como resultado CONV8. En la capa de pooling esta vez se ha usado un filtro de 3×3 con un *stride* de 2. La salida es de tamaño $256 \times 6 \times 6$.

10. Después viene la capa FC1, donde todas las neuronas de la capa CONV8 están conectadas con todas las neuronas de FC1. Por ello, el resultado es de tamaño $4096 \times 1 \times 1$. Nótese que para visualizar mejor esta capa se ha girado 90 grados.

11. FC2 es la aplicación de una misma capa FC a FC1 que tiene como resultado de nuevo un vector de tamaño $4096 \times 1 \times 1$.

12. La última capa es una capa *softmax* que calcula las probabilidades de las 1000 clases del *dataset* (véase la sección 2.4.1 para más detalle sobre *softmax*).

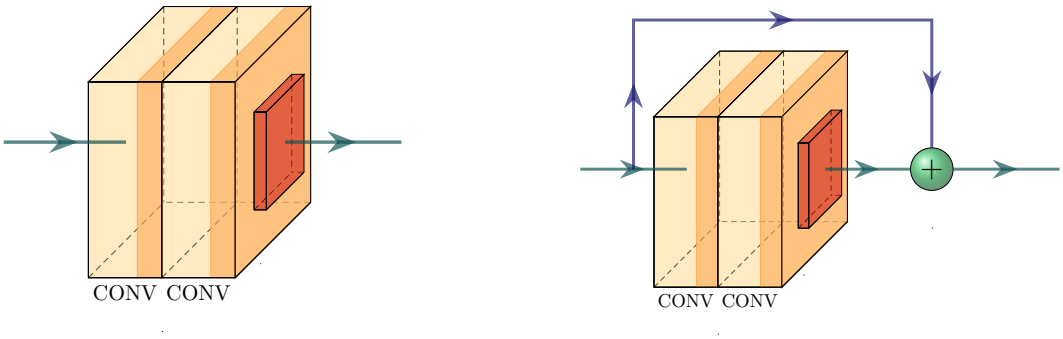
3.7. Conexiones residuales

Cuando las redes neuronales empiezan a tener cada más y más capas existe el problema del desvanecimiento del gradiente [4]. Este problema se mitiga mediante el uso de funciones de activación como ReLU o ELU, y también con las conexiones residuales, presentadas por primera vez en [42].

Estas conexiones residuales añaden conexiones directas o "saltos" entre capas, reduciendo así el problema del desvanecimiento del gradiente. La diferencia entre un bloque

[4]Como se ha explicado, la salida de una capa es la multiplicación de la salida de la capa anterior y la matriz de pesos de la capa actual, a lo que se le aplica finalmente la función de activación. Si, por ejemplo, se elige la función sigmoide como activación, al estar su derivada acotada entre 0 y $0,25$, si se tiene una red con muchas capas el valor de gradiente cada vez es más cercano a 0 ya que se está multiplicando muchas veces un valor pequeño. Por ello, los pesos/sesgos que dependen de este valor del gradiente para actualizarse, no se variarán apenas, estancando el aprendizaje de la red.

"habitual" de una CNN frente a uno residual está presentada en la Figura 3.8. Estos bloques residuales son la base de las ResNets, presentadas en el mismo trabajo [42], y con la que los autores ganaron diferentes competiciones.



(a) (b)

Figura 3.8: En (a) se muestra el bloque convolucional usado en la U-Net, mientras que en (b) se muestra el bloque residual presentado en [42]. Las capas de naranja claro representan capas de convolución, naranja oscuro para las capas de pooling y la concatenación mediante la bola con un '+' en su interior. Fuente: Github [5].

[5]Imagen creada a partir del proyecto de Github de Haris Iqbal: https://github.com/HarisIqbal88/PlotNeuralNet

Capítulo 4

Estado del arte

En los últimos años se han realizado muchos trabajos en torno a la segmentación de imágenes biomédicas para detectar regiones de interés dentro de una imagen mediante la aplicación de CNNs. El uso de las CNNs para este fin se ha vuelto habitual en el estado del arte, alcanzando resultados muy prometedores en diferentes tareas [43, 44].

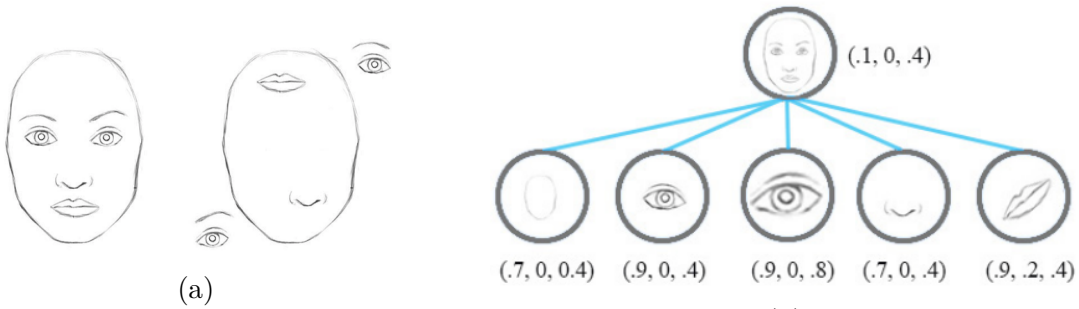
Un gran avance se produjo con el trabajo de Ronneberger *et al.* [45], logrando muy buenos resultados realizando la segmentación de células en imágenes de microscopía electrónica con una red llamada *U-Net*. Este trabajo alcanzó las primeras posiciones en la competición del 2012 propuesta en el ISBI [1] de segmentación de estructuras neuronales obtenidas mediante un microscopio electrónico [46]. A día de hoy, se han propuesto muchas implementaciones de *Deep Learning* usando la *U-Net* como referencia. En muchas ocasiones, se ha realizado la segmentación de otras áreas de imágenes de microscopía electrónica como pólipos, hígado y células [47].

Algunos trabajos introducen cambios en la arquitectura base de la *U-Net*, como por ejemplo en [48], donde se propone una red en forma de "W" en vez de "U", mediante la concatenación de varias *U-Net* conectadas entre si en forma de "puente": la primera *U-Net* para aprender las características más grandes de las imágenes, y la segunda para los detalles. En este trabajo realizan la segmentación de próstata de imágenes obtenidas mediante resonancia magnética (IRM) del *dataset* llamado PROMISE12 [49]. Otro trabajo en el que también modifican la arquitectura de la *U-Net* es el presentado en [50], donde proponen una red en forma de "V", que trabaja directamente con las imágenes 3D en vez de 2D como en la *U-Net* original. También cabe destacar que el mismo año Ronneberger *et al.* publicaron la extensión de la *U-Net* al espacio 3D [51].

Por otro lado, se han presentado trabajos combinando la *U-Net* con otro tipo de red para crear una nueva. Un ejemplo de ello está en [52], donde la *U-Net* se ha aplicado con la famosa *DenseNet* [53] para crear la red llamada "Tiramisu", utilizada para segmentación semántica, en la que obtienen muy buenos resultados.

Otro tipo de trabajo basado en la *U-Net* fue presentado en [54], donde realizan la implementación de una idea teórica presentada en [55] llamada *capsule network*, obteniendo muy buenos resultados en una competición de segmentación patológica de pulmón obtenida mediante tomografía axial computarizada (TAC).

El concepto de *capsule network* se puede ver como una técnica para poder tener en cuenta la información espacial entre las características encontradas por la red a la hora de hacer la predicción. Por ejemplo, una CNN detectará con una probabilidad alta una cara en una imagen que quizás no lo sea realmente, pero que sí contenga todos los componentes de una, es decir, que tenga ojos, nariz, boca, pelo etc. Por esta razón surge la idea de las cápsulas, de manera que para formar una cara, aparte de que sus componentes existan, tendrán que tener una relación espacial concreta. De esta manera, el rostro de la derecha que aparece en la Figura 4.1a no será tratado como cara.



(a) (b)

Figura 4.1: En (a) aparece la representación de un rostro "común" a la izquierda, mientras que a la derecha se muestra otro que tiene los mismos componentes que el anterior pero "desordenados". Una CNN podría detectar los dos como caras. En (b) está representada la idea de la *capsule network*, donde un vector de valores guardarían la información de cada característica, haciendo que para los casos en los que la cara no tenga sus componentes bien posicionados la red no lo detecte como cara. Los valores de los vectores son [probabilidad, orientación, tamaño], donde se aprecia que aunque los componentes tengan probabilidad alta de aparición, la probabilidad de detectar cara es de 0,1. Fuente [2].

Existen otras redes parecidas a la *U-Net* que tratan de localizar objetos de interés mediante su *bounding box*[3]. Las más conocidas son *R-CNN* [56], *Fast R-CNN* [57], *Faster R-CNN* [58] o *Mask R-CNN* [59].

[2] *Capsule network datascience*

[3] El *bounding box* es únicamente un rectángulo que contiene en su interior el objeto al que se quiere hacer referencia

4.1. *U-Net*

En el típico uso de las CNNs (véase el capítulo 3), el *output* de una imagen es su clase, es decir, una única etiqueta. Sin embargo, en muchas otras áreas de la visión por computador, y concretamente en las relacionadas con el tratamiento de imágenes biomédicas, el resultado requiere localización, por lo que a cada píxel se le asigna una etiqueta.

Ronneberger *et al.*[45] propusieron la red denominada *U-Net* para realizar esa misma tarea de clasificar cada píxel de la imagen. La red propuesta está formada en dos partes como puede verse en la Figura 4.2: la parte de la izquierda es la típica arquitectura de una CNN, con sus capas de convolución y reduciendo las dimensiones de la imagen con capas de *pooling* (*downsampling*), mientras que por el lado derecho estaría la reconstrucción de la imagen mediante el uso de capas de convolución habituales y otras capas convoluciones inversas (*upsampling*) [4]. Se podría decir que la red tiene una forma simétrica, reduciendo la imagen por un lado y ampliándola por el otro.

En cada paso de la fase de *downsampling* se utilizan 2 capas de convolución de filtros de 3×3 seguidos de una activación ReLU y una capa de *pooling* (concretamente *max pooling*) de 2×2 con un *stride* de 2. Cada vez que realizan el *downsampling*, multiplican por dos el número de filtros de las capas. Por otro lado, en cada paso de la fase de *upsampling* realizan una *up-convolution* de 2×2, concatenan el resultado del tamaño correspondiente obtenido en la fase de *downsampling*, y aplican dos capas de convolución 3×3 seguidas de una activación ReLU. En cada uno de estos pasos, de manera simétrica con el otro lado de la red, el número de filtros se divide por dos. Finalmente se añade una capa 1×1 para realizar la clasificación de cada píxel de la imagen. En total, la red dispone de 23 capas de convolución [45].

Para que no haya problemas en la red es importante aclarar que deberá escogerse el tamaño del *input* de manera que todas las capas de *max-pooling* trabajen con dimensiones pares.

[4] Estas capas de convolución inversas, llamadas "up-convolution", tienen el funcionamiento inverso de las capas habituales de convolución, aumentando el *output* en vez de reduciéndolo, mediante el mismo procedimiento de convolución

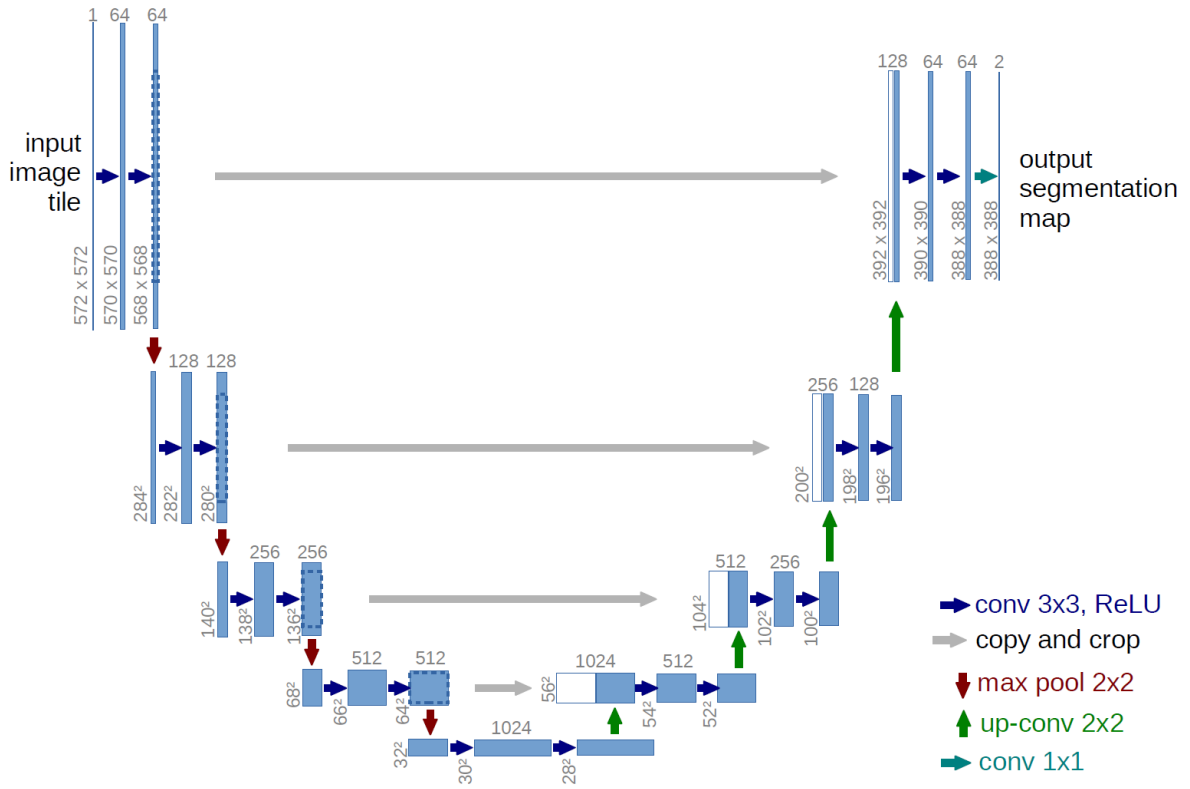


Figura 4.2: Arquitectura de la *U-Net*. Fuente: [45].

Finalmente, decir que en su artículo trabajaron principalmente con el *dataset* propuesto en la competición para la realización de segmentación de neuritas del ISBI 2012 [46], compuesto por imágenes de tamaño 512×512. Además, utilizan como función de coste la entropía cruzada o *cross-entropy* (que sigue la formula 2.5), y hacen un uso intensivo de DA realizando deformaciones elásticas, desplazamientos y rotaciones.

4.2. Segmentación de mitocondrias

En el caso particular de la segmentación de mitocondrias, muchos trabajos han sido presentados utilizando como base el *dataset* presentando por Lucchi *et al.* en [60], nombrado a partir de ahora como FIBSEM_EPFL (véase más información acerca de este *dataset* en 6.1.1). Este *dataset* se ha convertido en un estándar *de facto* para realizar este tipo de segmentaciones y un ejemplo del tipo de imágenes que lo componen se muestra en la Figura 4.3.

En el trabajo presentando junto con este *dataset*, Lucchi *et al.* proponen un método de segmentación de las mitocondrias basado en particionar un grafo sobre la salida de un algoritmo de *clustering*. Los mismos autores desarrollaron en [61] un método para

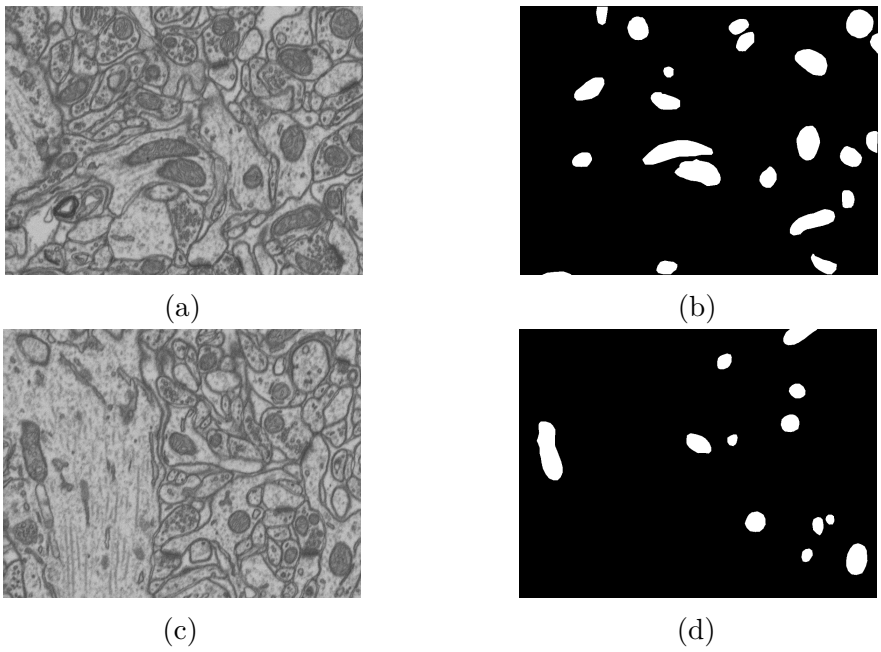


Figura 4.3: Ejemplo de imágenes del *dataset* FIBSEM_EPFL presentado por Lucchi *et al.* [60]: (a) imagen de entrenamiento, (b) su correspondiente máscara binaria o *ground truth*, (c) imagen de testeo, y (d) su correspondiente *ground truth*.

realizar la segmentación de imágenes construyendo previamente subregiones de la imagen (agrupadas por similitudes como color y forma) antes de aplicar una *Structural Support Vector Machine* (SSVM). Siguiendo con ellos, en [62] presentan un trabajo que se basa en la realización de un "subgradient descent method" mediante el uso de lo que ellos denominan "working set", que es un conjunto de restricciones extraídos del problema de inferencia conocido como *loss-augmented inference* [63], lo que mejora la convergencia y exactitud del optimizador. Después, en [64], presentaron un trabajo para intentar modelar la membrana de la mitocondrias, es decir, la propia forma de la mitocondria, para obtener mejores resultados en la segmentación. Para finalizar, en [65], obtuvieron sus mejores resultados utilizando la técnica de *working set* presentada en el anterior trabajo junto con la selección de un *step size* personalizado para cada iteración.

Oztel *et al.* propusieron en [66] una CNN (presentanda en la Figura 4.4) consiguiendo los mejores resultados reportados hasta ahora en el *dataset* de FIBSEM_EPFL, 0,907 de *score* medido con el índice de Jaccard (medida utilizada comúnmente en la visión por computador y explicada detalladamente en la sección 6.2.1). En su trabajo sugieren el entrenamiento de su CNN mediante imágenes de 32×32 píxeles extraídas de las originales de 1024×768 píxeles. No obstante, en vez de usar todas las imágenes generadas, únicamente entrenan la red con aquellas que tienen más de un 80 % de los píxeles etiquetados como una de las dos clases (*background* o mitocondria), el resto son descartadas. Realizan también DA (explicado en la sección 2.10.1) para así disponer de más imágenes para entrenar la red. Con esto, intentan compensar el desbalanceo de

datos entre las dos clases, ya que hay muchos más casos de la clase *background* que de la clase mitocondria. Para finalizar, también cabe mencionar que realizan tres pasos de post-procesado para obtener una mejor segmentación:

1. *2-D Spurious Detection Filtering*. En su caso, la salida de la CNN es una imagen binaria. Este filtrado comprueba la probabilidad de los píxeles de ser mitocondrias y descartan los que tengan la probabilidad baja, que es finalmente algo muy parecido a lo que se ha realizado en este trabajo.

2. *Boundary refinement*, que realiza una erosión de las mitocondrias y el fondo para obtener dos marcadores, uno dentro de la mitocondria y otro fuera. Posteriormente aplican el algoritmo *watershed*, presentado por primera vez en [67], dentro del área que delimitan esas dos regiones para completar los bordes de las mitocondrias.

3. *3-D filtering*, realizan un filtro a lo largo de la dimensión z aprovechando la información 3D del *dataset* para detectar más mitocondrias. Con esto también descartan aquellas que se han detectado y que realmente no lo son.

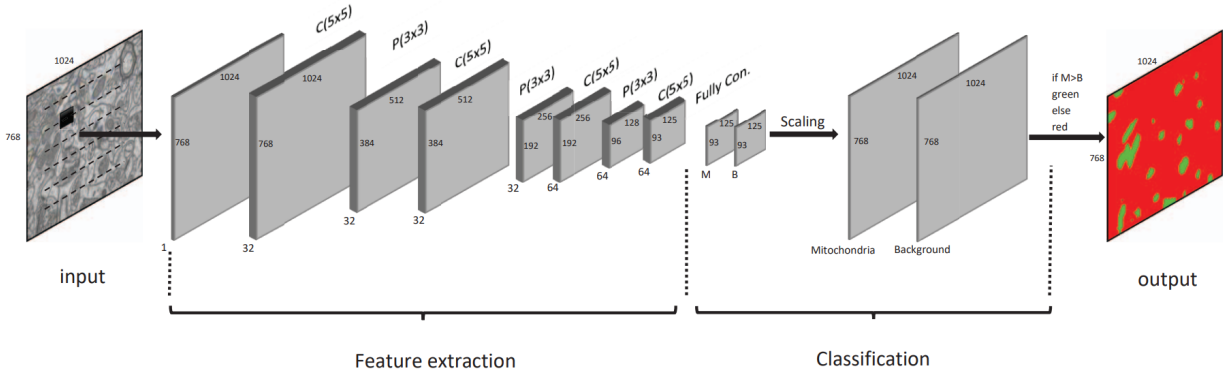


Figura 4.4: Arquitectura de la red presentada por Oztel *et al.* Fuente: [66].

En [68], Liu *et al.* proponen una combinación de ResNet [42] mezclado con la redes en forma "piramidal" presentadas en [69]. También hacen uso de DA incluyendo ruido aleatorio y un paso de post-procesado, obteniendo un Jaccard de 0,849.

Chi Xiao *et al.* proponen en [70], una combinación de una U-Net modificada con bloques residuales operando directamente sobre el volumen de datos en 3D en vez de en 2D. Realizan la ecualización del histograma de la imagen y técnicas de DA, realizando rotaciones fijas de 90 y 180 grados, y *flips* (imágenes especulares) horizontales y verticales. Obtienen mejores resultados a medida que aumentan las variaciones que realizan con DA, alcanzando el estado del arte, en sus mejores resultados, con 16 variaciones de DA.

Otro trabajo reciente ha sido presentado en [9] por Casser *et al.* donde utilizan una modificación de la U-Net original logrando resultados comparables con el estado del arte, a pesar de que enfocan el trabajo como segmentación en tiempo real de mitocondrias. En su propuesta reducen el número de parámetros de la U-Net original y aplican técnicas de DA. Por otro lado también presentan una versión "mejorada" del *dataset* de FIBSEM_EPFL al que han llamado como Lucchi++, donde corrigen el etiquetado donde la comunidad había reportado errores. Además, ajustan los bordes de forma más precisa en las membranas de las mitocondrias.

Capítulo 5

Entrenamiento previo: experimentación con otros *datasets*

Para familiarizarse con las redes neuronales se hicieron pruebas con otros *datasets* más sencillos que son los que se presentarán a continuación. En estos experimentos también se han utilizado diversos tipos de arquitecturas de redes neuronales.

5.1. Métrica de evaluación: precisión o *accuracy*

Antes de comenzar con la explicación de la experimentación que se ha realizado hay que aclarar ciertos criterios de evaluación del ML para poder realizar una correcta evaluación de los resultados obtenidos.

Se ha decido mostrar un ejemplo para luego poder definir la métrica de manera más clara. Se supone que la respuesta de un algoritmo de clasificación supervisada ha sido la que se muestra en la Tabla 5.1, que comúnmente está denominada como matriz de confusión. La tarea de este algoritmo ha sido la de clasificar ciertas imágenes como "Gato" o "Perro".

		Clasificación Algoritmo	
		Gato	Perro
Ground Truth	Gato	10	7
	Perro	5	8

Tabla 5.1: Matriz de confusión de un algoritmo de *Machine Learning* supervisado cualquiera.

Las filas de la Tabla 5.1 representarían el *ground truth* mientras que las columnas representarían el resultado que se ha obtenido con el algoritmo. Hay 30 casos en total ($10 + 7 + 5 + 8 = 30$). En la primera columna se puede observar que 15 imágenes son "Gato" ($10 + 5 = 15$), mientras que 15 son "Perro" ($7 + 8 = 15$). También se observa que el clasificador ha etiquetado 17 imágenes como "Gato" ($10 + 7 = 17$) y 13 como "Perro" ($5 + 8 = 13$).

En este caso, el número de predicciones correctas serían $10 + 8 = 18$, que son los valores en la diagonal. Con ello se podrá definir la métrica precisión o *accuracy* (en inglés) con la siguiente fórmula:

$$Accuracy = \frac{Número\_de\_predicciones\_correctas}{Número\_total\_de\_predicciones} \tag{5.1}$$

Por ello, para este ejemplo se tendría un *accuracy* de $18/30 = 0{,}6$.

Como dato también decir que esta métrica no se comporta bien ante los *datasets* que no están balanceados, es decir, cuando hay muchas más entradas de una clase que de otra. Para ello existen otro tipo de métricas más avanzadas como Kappa, curva ROC o AUC, que no se explicarán en esta documentación.

Clasificación binaria

En el caso de que la clasificación sea binaria, al igual que en el ejemplo anterior, se podría construir una tabla con las opciones disponibles, comúnmente denominada "tabla de contingencia" (véase la Tabla 5.2).

	True condiction	
Predicted condition	True positive (TP)	False positive (FP)
	False negative (FN)	True Negative (TN)

Tabla 5.2: Ejemplo de tabla de contingencia de una clasificación binaria. TP representa los aciertos que realmente son positivos, FP los detectados como positivos cuando realmente no lo son, FN los no detectados como positivos cuando realmente si lo son y TN los detectados como negativos cuando son negativos.

Por ejemplo, si se tiene una clasificación binaria donde se quiere discernir pacientes que están o no enfermos mediante un test, los resultados pueden ser de estas cuatro categorías: TP sería el caso en el que el test de enfermedad realizado a un paciente da positivo y realmente tiene la enfermedad, por lo que se habrá acertado en este caso. FP será cuando el test salga positivo pero en realidad el paciente no sufre la enfermedad. FN

cuando el test es negativo mientras que el paciente realmente tiene la enfermedad (este será el peor de los casos). Por último, TN será cuando el test de negativo y el paciente no esté enfermo.

Entendido un poco el funcionamiento de la Tabla 5.2 habrá otras métricas que se podrán utilizar para evaluar esta clasificación binaria, como pueden ser *precision* y *recall* de los que no se entrará en detalle aquí. No obstante, se va a definir la métrica *accuracy* de antes sujeta a los términos que se acaban de definir, representando lo mismo, y que sigue la fórmula:

$$Accuracy = \frac{TP + TN}{TP + TN + FP + FN} \tag{5.2}$$

5.2. Dígitos

Se ha realizado la experimentación con el *dataset* MNIST que contiene imágenes de dígitos escritos a mano por 500 escritores diferentes [71]. Los dígitos dentro de este *dataset* irán desde el 0 hasta el 9 y la variable clase es el número correspondiente. Cada imagen está en escala de grises (0-255), centrada en el origen y con un tamaño de 28×28. El *dataset* está construido por 60.000 casos para el entrenamiento y 10.000 casos para el testeo. Algunas imágenes de ejemplo se muestran en la Figura 5.1. En este caso se ha realizado la clasificación de este *dataset* utilizando una RNN y una CNN como en [12].

Primeramente, para el caso de la RNN, se ha divido el valor de los píxeles entre 255 para obtener valores entre 0 y 1. Además se ha puesto cada imagen en forma de vector con longitud 784. La red creada tiene dos capas de 10 neuronas cada una. La primera capa tiene como *input* el vector de longitud 784, y ReLU como función de activación en todas las neuronas. La última capa está densamente conectada con la anterior y utiliza *softmax* con 10 salidas, una para cada clase. Como función de coste se ha elegido *cross-entropy*, como métrica para la evaluación se ha elegido *accuracy* y como técnica de optimización SGD. Se ha obtenido como resultado un *accuracy* de $0,8957$ y un coste de $0,3726$.

En el caso de la CNN, se ha creado la red con 5 capas: 2 de convolución, 2 de *pooling* y la última *softmax* de salida. Esta vez se realizado la división de los valores de las imágenes entre 255 para mantenerlos entre 0 y 1 pero las imágenes se han dejado como 28×28. En cuanto a la red, la primera capa consta de 32 filtros, con un kernel 5×5 y ReLU como función de activación. A continuación, una capa de *pooling* para reducir a la mitad el tamaño del *input*. Más adelante, de nuevo una capa de convolución igual que la primera pero esta vez con 64 filtros. Luego, de nuevo, una capa de *pooling* para reducir a la mitad la imagen y, para finalizar, la última capa *softmax*, al igual que con la RNN, con 10 salidas (una para cada clase). Como métrica *accuracy*, como optimizador SGD y como coste *cross-entropy* de nuevo. Como resultado, se ha obtenido un coste de $0,1034$ y

un *accuracy* de $0,9708$.

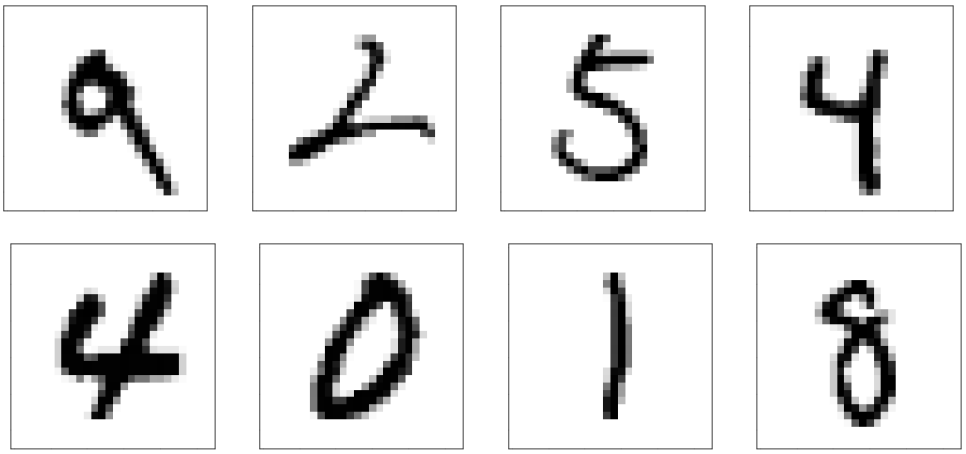


Figura 5.1: Imágenes del *dataset* MNIST [71].

5.3. Perros y gatos

Se ha creado una CNN para realizar la clasificación de perros y gatos propuesta por una competición de Kaggle de hace algunos años [30]. Para ello, se ha creado una CNN similar a la presentada por LeCun *et al.* [36], con la diferencia de la función de activación, en este caso se ha usado ReLU. El *dataset* se compone de muchos más casos pero solamente se ha entrenado con 2.000 imágenes, 300 de validación y 12.500 como conjunto de testeo. Las imágenes de este *dataset* tienen tamaños diferentes, por ello se han tenido que escalar a un tamaño de $150 \times 150 \times 3$ para poder entrenar la CNN. En la Figura 5.2 se muestran varias imágenes extraídas de este *dataset*.

La CNN creada está compuesta por 7 capas: 3 capas de convolución, 2 de *pooling*, 2 capas densamente conectadas y una última capa *softmax* de salida. La primera capa está compuesta por 32 filtros 3×3 con una función de activación ReLU. Viene seguida una capa de *pooling* reduciendo el *input* a la mitad de tamaño y luego, de nuevo, otra capa de convolución igual que la primera. Nuevamente, una capa de *pooling* y convolución, pero esta vez con 64 filtros. A continuación, se "aplana" el *input* para que pueda ser conectado con la siguiente capa densamente conectada. Por último, se añade una capa *softmax* con una salida binaria para representar las dos clases "Perro" y "Gato".

Los resultados han sido un coste de $0,7550$ y una *accuracy* de $0,4$ lo cual son algo malos, pero normal quizás al haber utilizado sólo una pequeña parte del *dataset*. Se ha de decir que la experimentación abordada en este capítulo ha servido para familiarizarse con el uso de las redes, por lo que el conseguir resultados muy buenos no era el objetivo de estas primeras práctica.

Figura 5.2: Imágenes del *dataset* de perros y gatos de la competición de Kaggle [30].

Capítulo 6

Caso de estudio: segmentación de mitocondrias

Como se ha ido mencionando a través de la documentación, el caso de estudio se ha enfocado en la segmentación de mitocondrias en imágenes de microscopia electrónica. Luego el objetivo está en poder producir una clasificación por píxel lo más cercana posible al *ground truth*.

A lo largo del proyecto se ha trabajado principalmente con la arquitectura tipo U-Net y se han realizado varias centenas de experimentos para intentar ajustar los parámetros de la red, probando multitud de opciones, hasta encontrar la mejor configuración.

Primeramente, se presentan los *datasets* con los que se ha trabajado y las particularidades de cada uno. Después, se presentan las diferentes métricas de evaluación que se han ido añadiendo a lo largo de la experimentación. Más adelante, se van describiendo todos los experimentos que se han realizado, agrupándolos por el tipo de cambio que se haya hecho en la configuración, poniendo ejemplos si se necesitase, mostrando los resultados obtenidos y una pequeña reflexión. Por último, se presentan todos los resultados obtenidos así como los resultados de otros trabajos del estado del arte para poder realizar la comparación.

La tabla completa de resultados se puede encontrar aquí, mientras que los primeros experimentos, con los que se tuvieron problemas en la métrica usada, están recogidos en este otro enlance. Por otro lado, el código utilizado en los experimentos está disponible en Github, y ha sido desarrollado usando Keras con TensorFlow como soporte.

Casi todos los experimentos se han repetido entre 10 y 30 veces (refiriéndose a cada uno de ellos como *run* a partir de ahora). Por ello, los resultados que se muestran aquí son la media de los valores obtenidos en todos ellos. Esto se ha hecho así ya que, en el momento de la realización de este proyecto, había un problema con la reproducibilidad

de los resultados entrenando las redes neuronales en GPUs usando Keras y Tensorflow. Realmente, el problema reside en que algunos de los algoritmos de la librería cuDNN de NVIDIA no son deterministas, y producen diferentes salidas a pesar de dejar fijos tanto los parámetros como las semillas aleatorias.

Los experimentos se han realizado en un servidor con múltiples GPUs, entre las que se encuentran tres tipos de tarjetas: NVIDIA GTX 2080 Ti, NVIDIA TITAN X y NVIDIA GTX 1080 Ti.

6.1. *Datasets* utilizados

Principalmente se ha trabajado con el *dataset FIBSEM_EPFL* a lo largo de la experimentación hasta encontrar la mejor configuración. *Lucchi++* y *Kasthuri++* se han probado, una vez obtenida la mejor configuración con el primero, para poder comparar el trabajo con el estado del arte. Por último, el *dataset* de *Achucarro* es un conjunto de datos nuevos creados por el centro de investigación *Achucarro - Basque Center for Neuroscience* con el que se ha colaborado.

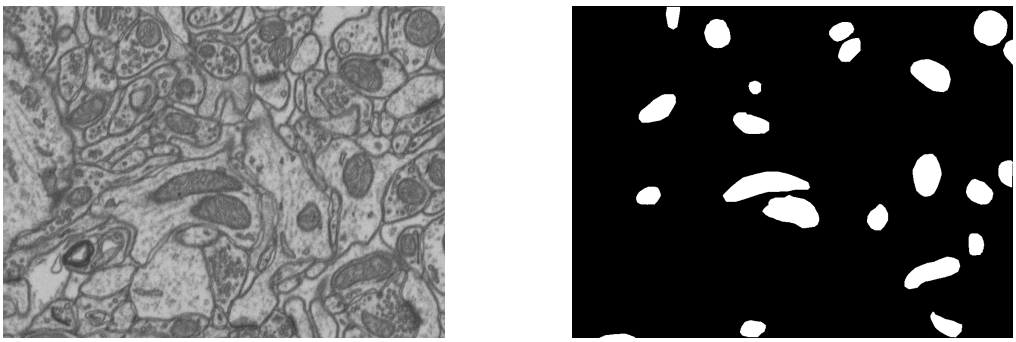
6.1.1. *FIBSEM_EPFL*

Este *dataset* se introdujo en el trabajo de Lucchi *et al.* [60] en el *Computer Vision Laboratory - EPFL*, en Suiza. Las imágenes provienen del hipocampo de un ratón y fueron adquiridas con la técnica *focused ion beam* (FIB)[1] mediante un microscopio tipo SEM (véase sección 1.2.1 para más información) obteniendo una resolución por píxel de $5 \times 5 \times 5$nm. Todo el conjunto de imágenes 3D tienen un tamaño de $2048 \times 1536 \times 1065$ vóxeles, pero se separó en dos manualmente, por ello el *dataset* lo componen 330 imágenes en total: 165 para entrenamiento y 165 para test.

Cada una de las imágenes tiene una máscara de la localización de las mitocondrias en blanco, mientras que el fondo está marcado con negro. Estas máscaras son típicas en las imágenes biomédicas para poder localizar la región de interés, como se mencionó en la explicación de la U-Net (véase sección 4.1).

Los píxeles que pertenecen a las mitocondrias están etiquetados con la clase 1 y los que pertenecen al fondo están etiquetados con un 0. En la Figura 6.1, puede verse un ejemplo de dos imágenes de este *dataset*.

[1] FIB se diferencia con SEM, explicado en la sección 1.2.1, en que en vez de usar haces de electrones se usan iones



(a) (b)

Figura 6.1: Ejemplo de imágenes del *dataset FIBSEM_EPFL* presentado por Lucchi *et al.* [60]: (a) imagen de entrenamiento y (b) su correspondiente máscara binaria o *ground truth*.

6.1.2. *Lucchi++*

Este *dataset* se introdujo en Casser *et al.* [9] como mejora del *dataset FIBSEM_EPFL*. En su trabajo argumentan que el etiquetado de las mitocondrias no está del todo bien realizado: hay algunas que faltan y otras zonas de la imagen que han sido tratadas como mitocondrias realmente no lo son. Además de eso, también han ajustado la máscara a la forma de la mitocondrias, ya que en los bordes no estaban bien definidas. Un ejemplo del cambio realizado se puede observar en la Figura 6.2.

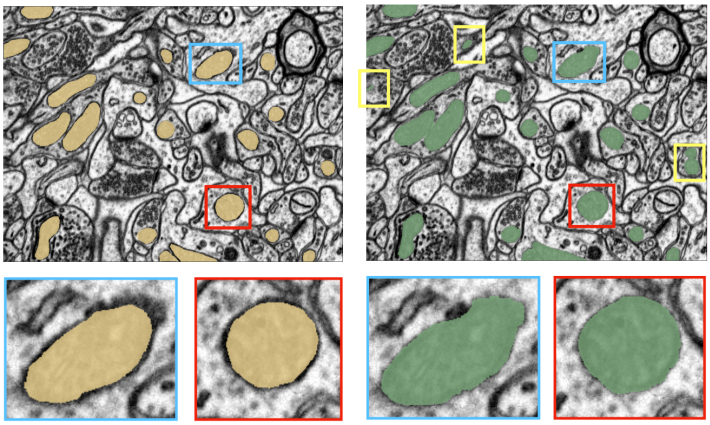
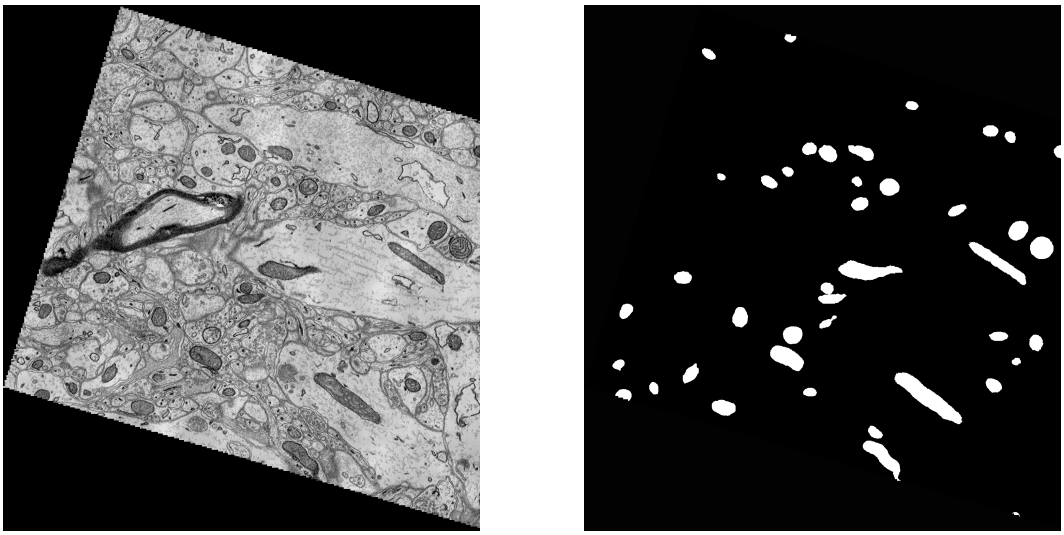


Figura 6.2: A la izquierda está representado el etiquetado de FIBSEM_EPFL mientras que a la derecha está el re-etiquetado realizado en *Lucchi++*, donde se puede apreciar que se ha ajustado mucho más el etiquetado en los bordes. Fuente: [9].

6.1.3. *Kasthuri++*

Este *dataset* se introdujo también en el trabajo de Casser *et al.* [9], como mejora de nuevo al *dataset* original introducido por Kasthuri *et al.* en [72]. Las imágenes se obtuvieron del cortex de un ratón mediante un *serial section electron microscopy* (ssME) creando un volumen de imágenes 3D con una resolución de $3 \times 3 \times 30$nm por píxel. Visto en 2D, el *dataset* está compuesto de 85 imágenes de entrenamiento, con un tamaño de 1463×1613 píxeles, y 75 de test, con un tamaño de 1334×1553 píxeles. En la Figura 6.3, se muestran dos imágenes del *dataset* de *Kasthuri++*.

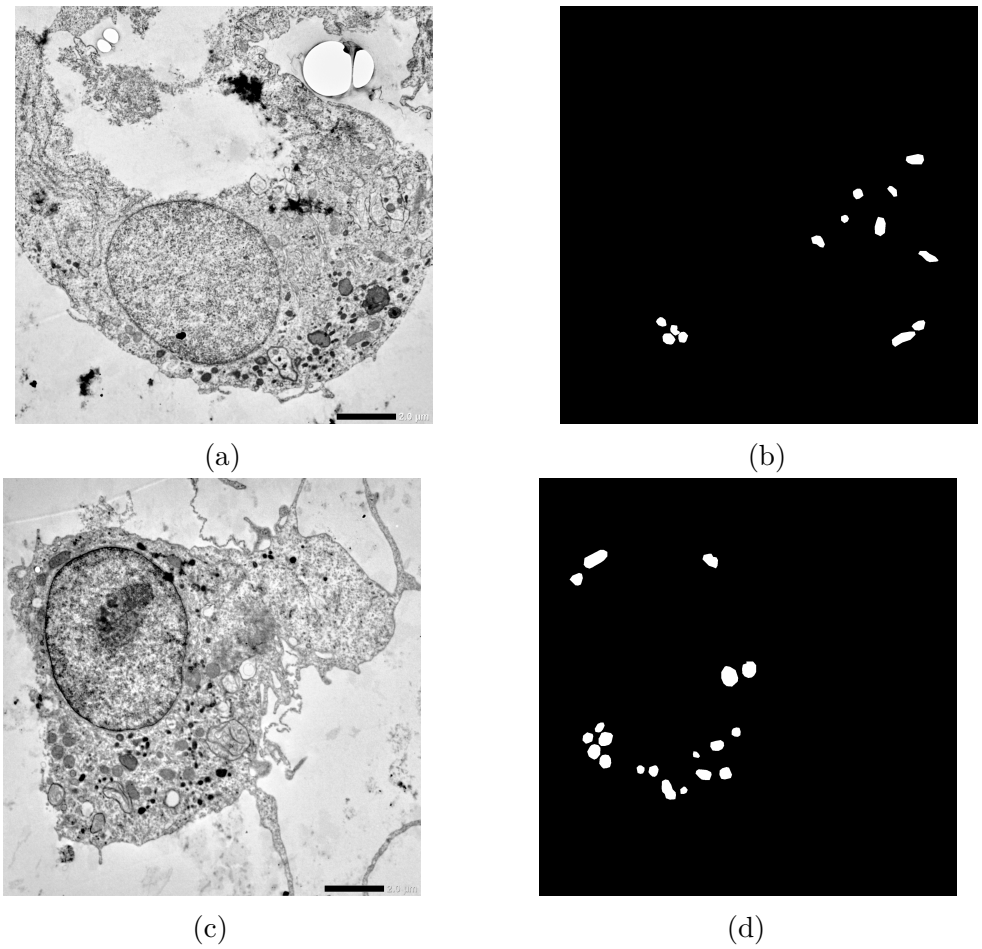


(a) (b)

Figura 6.3: Ejemplo de imágenes del *dataset Kasthuri++* presentado por Casser *et al.* [9] y que originalmente se introdujo en [72]: (a) imagen de entrenamiento y (b) su correspondiente máscara binaria o *ground truth*.

6.1.4. *Achucarro*

El *dataset* de *Achucarro* es un conjunto de datos nuevos creados en el *Achucarro - Basque Center for Neuroscience*. El conjunto de datos en este caso es pequeño, disponiendo únicamente de 8 imágenes que se han dividido en 7 para entrenamiento y 1 para test. Las imágenes son de 2048×2048 píxeles y tienen una resolución de $6,85 \times 6,85$nm por píxel. Un ejemplo de las imágenes que componen este conjunto de datos se puede ver en la Figura 6.4, donde se puede apreciar que en este caso se dispone de la célula completa.



(a) (b)

(c) (d)

Figura 6.4: Ejemplo de imágenes del *dataset Achucarro*: (a) y (c) Imágenes de ME de un corte de célula completa; (b) y (d) El *ground truth* correspondiente.

6.2. Métricas de evaluación

En la sección 5.1 se comentaron varias métricas utilizadas para evaluar modelos de ML en general, y son métricas que se pueden usar también con las redes neuronales (como se vio en el capítulo de experimentación 5). No obstante, cuando se trabaja con imágenes, hay otro tipos de métricas más adecuadas que se ajustan más a este campo.

Por otro lado, a lo largo de la documentación, también se ha hablado de que el coste de la red era la manera de evaluar su rendimiento. No obstante, este coste es algo que está relacionado con la redes y no con los datos. Aunque un valor bajo en el coste suponga una buena clasificación, muchas veces se requiere de otro tipo de métricas más conectadas con los datos con los que se está trabajando para poder medir el rendimiento. Además, muchos de los trabajos están relacionados con competiciones que se proponen en Internet, y muchas de ellas traen consigo sus propias métricas de evaluación para medir el rendimiento de los modelos que se proponen, ya que no tiene por qué ocurrir que

todos los modelos que se presenten sean redes neuronales y tengan ese "coste" definido.

Aquí se presentan varias métricas con las que se ha trabajado a lo largo de este proyecto, por ser unas de las más usadas en este campo del tratamiento de imágenes biomédicas y, también, para poder comparar el trabajo realizado con otros que utilizan estas métricas de evaluación.

6.2.1. Índice de Jaccard

La métrica Jaccard se introdujo en [73] y también es comúnmente referida a ella como *Intersection over Union* (IoU) o *Jaccard similarity*. La fórmula que sigue es la siguiente:

$$J(A, B) = \frac{|A \cap B|}{|A \cup B|} \tag{6.1}$$

Donde A y B son dos conjuntos, como por ejemplo los que están definidos en verde y rojo en la Figura 6.5.

Lo realmente mide esta métrica es la similitud o diversidad de dos conjuntos. Por ejemplo, hay redes neuronales, como la R-CNN [56], Fast R-CNN [57], Faster R-CNN [57] o Mask R-CNN [59], que se basan en localizar los objetos de las imágenes mediante un *bounding box*. El cálculo del índice de Jaccard en estos casos se ilustra en la Figura 6.5.

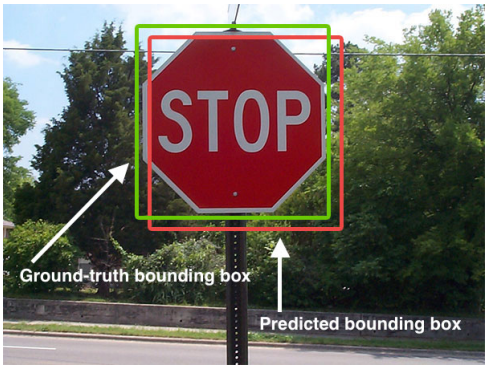


Figura 6.5: Ejemplo de uso del índice de Jaccard mediante dos *bounding boxes*: el *ground truth* marcado en verde frente al que se ha predicho en rojo. Fuente: Wikipedia.[2]

Esta misma métrica también puede ser representada mediante las definiciones realizadas en la clasificación binaria (explicada en la sección 5.2). La ecuación del índice de Jaccard en ese caso sería la siguiente:

[2] https://en.wikipedia.org/wiki/Jaccard_index

$$Jaccard\ index = \frac{TP}{TP + FP + FN} \tag{6.2}$$

Donde TP es el número de *true positives*, FP el número de *false positives* y FN el número de *false negatives*.

Como en este trabajo el objetivo es realizar la clasificación de todos los píxeles de la imagen en dos clases, se implementará esta última ecuación para llevar a cabo la evaluación de los experimentos.

6.2.2. Visual Object Classes

Esta métrica se presentó en *The PASCAL Visual Object Classes (VOC) Challenge* [74] y es comúnmente conocida como VOC. Esta métrica es definida como el valor de Jaccard promedio de la clase negativa y positiva, que corresponde con el *background* y *foreground* (el fondo y las mitocondrias, respectivamente). Como se menciona en [9], esta métrica ha llegado a causar confusión en algunos trabajos al identificarla erróneamente con el índice de Jaccard. El VOC sigue la siguiente fórmula:

$$VOC\ score = \frac{(IoU_{foreground} + IoU_{background})}{2} \tag{6.3}$$

6.2.3. Detection

Esta métrica es una de las tres métricas que ofrece el *Cell Tracking Challenge*[3] para poder medir con cuánta precisión se han identificado los objetos. La fórmula que sigue esta métrica (*detection accuracy*, DET) es la siguiente:

$$DET = 1 - \frac{min(AOGM\text{-}D, AOGM\text{-}D0)}{AOGM\text{-}D0} \tag{6.4}$$

donde *AOGM-D* representa el coste para convertir los resultados obtenidos al *ground truth*, mientras que *AOGM-D0* representa el coste para recrear el *ground truth* desde cero. Al estar normalizada, está métrica devolverá resultados entre 0 y 1.

Concretamente, AOGM se refiere a *Acyclic oriented graphs matching*, por lo que se utiliza un grafo para poder construir esta y otras métricas presentadas en [75].

[3]Presentando en http://celltrackingchallenge.net/

Al estar las métricas basadas en un grafo, el cálculo de ellas se basan en la suma de las operaciones que hay que hacer para convertir el grafo que representa la predicción realizada con el grafo que representa el *ground truth*. La fórmula que define AOGM sería la siguiente:

$$AOGM = w_{NS}NS + w_{FN}FN + w_{FP}FP + w_{ED}ED + w_{EA}EA + w_{EC}EC \tag{6.5}$$

Donde NS representa el número de operaciones de división de vértices, FN el número de operaciones de añadir un vértice, FP las de borrado de un vértice, ED las de borrado de aristas, EA las de añadir una arista y EC las de operaciones semánticas de las aristas. Por otro lado, las w son los pesos que se les asigna a cada una de las operaciones.

Nótese que con la fórmula presentada cuantas menos operaciones haya que realizar para convertir el grafo predicho en el *ground truth* más bajo será este valor (por lo que el valor de DET será más alto según la ecuación 6.4). En la figura 6.6 se muestra un ejemplo de como transformar un grafo en otro mediante las operaciones que se han comentado.

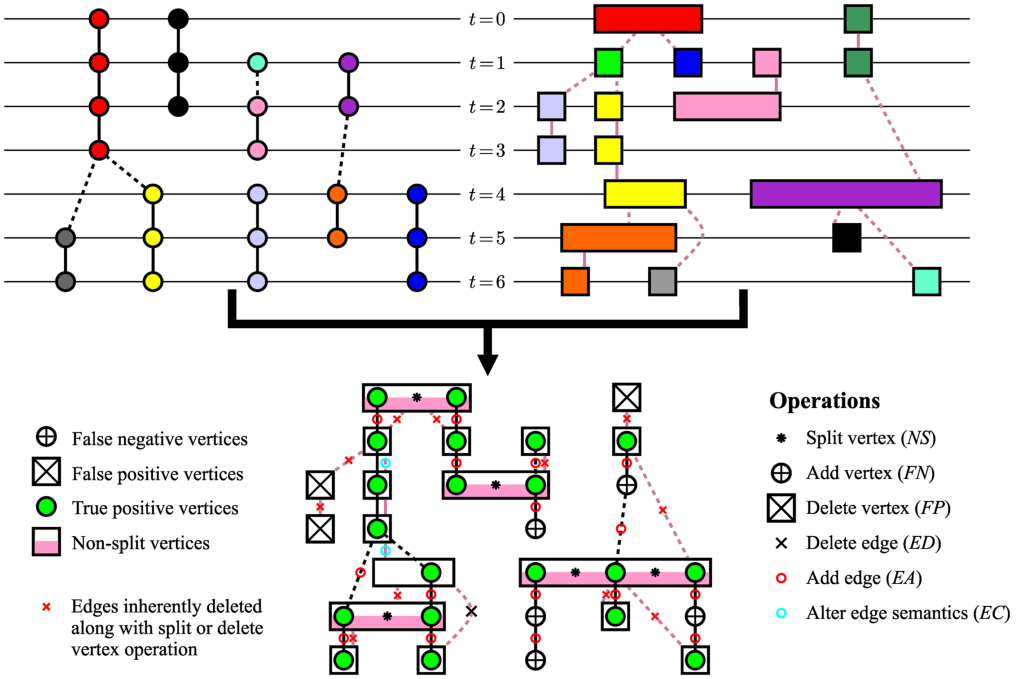


Figura 6.6: Ejemplo de la transformación de un grafo predicho (izquierda) con el que representa el *ground truth* (derecha). En los dos grafos, el eje vertical representa el dominio temporal mientras que el horizontal representa el dominio espacial. En la parte de abajo están representadas las operaciones que se han tenido que realizar para transformar el grafo de la izquierda en el de la derecha. Fuente: [75].

Para el caso concreto de la métrica de DET que ocupa esta sección, las operaciones de los vértices no se tienen en cuenta, ya que se utilizan para las otras métricas definidas

en su trabajo como el *tracking* o seguimiento de objetos. De esta manera, ED, EA y EC tendrán sus pesos w a cero, por lo que *AOGM-D* quedará así:

$$AOGM\text{-}D = w_{NS}NS + w_{FN}FN + w_{FP}FP \tag{6.6}$$

Esta métrica se utilizará principalmente para comprobar si, a pesar de haber hecho una buena o mala clasificación, los elementos (en este caso mitocondrias) se han detectado correctamente. Esta métrica se usa en la segmentación de imágenes biomédicas donde el localizar los elementos puede llegar a tener más importancia que ajustar con precisión su forma.

6.3. Experimentación

Se parte de los experimentos mostrados en la siguiente Tabla. Realmente se hicieron unos cuantos experimentos antes (*link*) donde hubo un error con la implementación de la métrica de evaluación utilizada, por ello no se tendrán en cuenta en la descripción presentada en el documento.

6.3.1. Configuración base

La arquitectura de red base utilizada ha sido la U-Net pero con alguna pequeña modificación. En el trabajo presentado por Ronneberger *et al.* [45] se utilizan 64 filtros en las capas de convolución y luego se van multiplicando por dos a medida que se desciende en la red. En la arquitectura base de este trabajo se ha reducido la cantidad de filtros a 16, obteniendo así una red mucho menos pesada, con casi 2 millones de parámetros (cuando la U-Net ronda los 31 millones). Además, se ha utilizado ELU como función de activación en lugar de ReLU, ya que acelera el entrenamiento de las redes (entre otras ventajas) como se menciona en [15].

En la U-Net original el *input* (572×572) tenía diferente tamaño que el *output* (388×388). A pesar de utilizar imágenes de 512×512, ellos realizaban una estrategia de troceado con solapamiento, además de "mirroring" [4] para tratar los bordes, por ello tienen 30 píxeles más a cada lado, llegando así a 572×572. En el caso de este trabajo, y para el caso de esta primera configuración base, el *input* tendrá el mismo tamaño que el *output*, evitando el uso de estrategias como las utilizadas en el trabajo original de U-Net, y teniendo como referencia los buenos resultados obtenidos en [9] siguiendo esta misma estrategia.

[4] La técnica de *mirroring* se utiliza para extrapolar los píxeles de los bordes de las imágenes como si se pusiera un espejo justo en el borde de la imagen, copiando simétricamente los valores de los píxeles.

Realizados estos cambios, la arquitectura de la red base se muestra en la Figura 6.7.

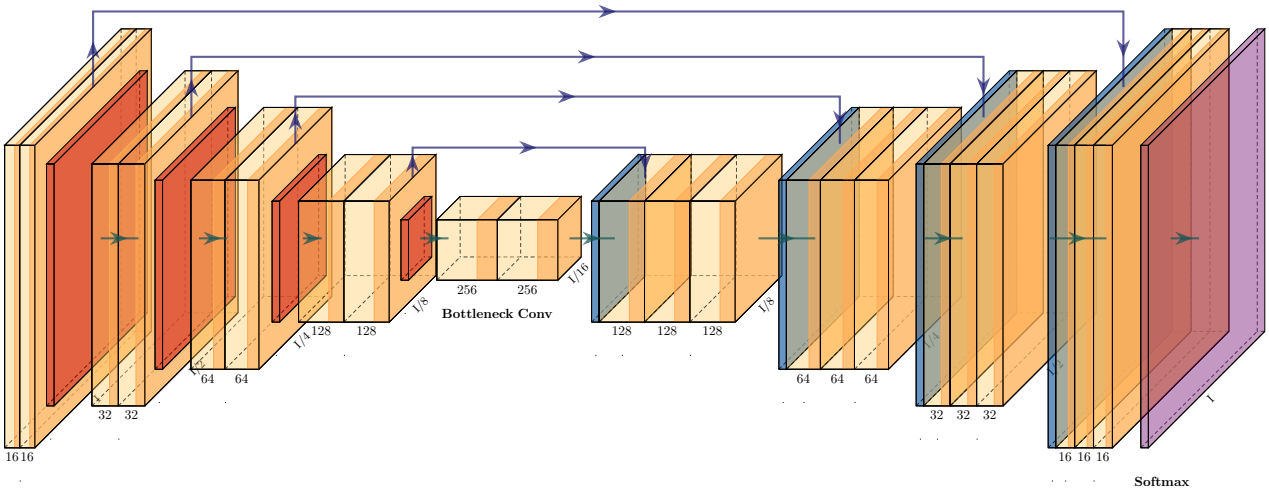


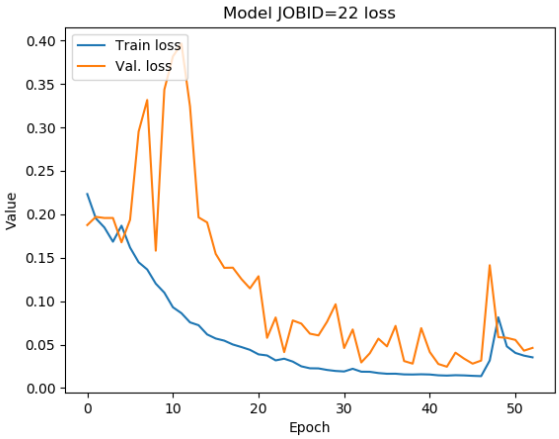
Figura 6.7: Arquitectura de la U-Net modificada que se ha usado como base para realizar los experimentos. Las cubos naranjas claros representan las capas de convolución, el naranja oscuro para las capas de *pooling* y las azules para los *upsamplings* o *up-convolutions*. Fuente: Github [5].

El *dataset* inicial utilizado ha sido *FIBSEM_EPFL*, por lo que se dispone de 165 imágenes de entrenamiento y 165 imágenes de test. De las 165 de entrenamiento se ha reservado el 10 % para validación, lo que supone 17 imágenes, por lo que finalmente quedan 148 para entrenamiento, 17 para validación y 165 para test.

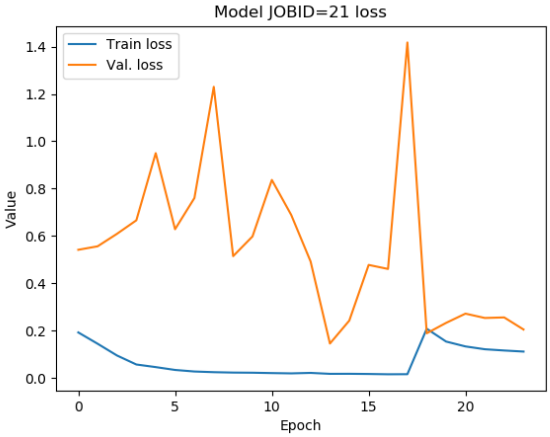
Como es habitual, se ha hecho uso de DA (implementación de Keras) para poder mejorar el rendimiento de la red. En este caso, se han aplicado *flips* horizontales y verticales, y una rotación de 180 grados de manera aleatoria.

Se ha definido un número de épocas máximo de 230 y una *patience* de 50, haciendo que cuando el modelo no mejora en 50 épocas se pare el entrenamiento, aún no habiendo alcanzado esas 230 definidas. La métrica de evaluación escogida es el índice de Jaccard (mientras que VOC y DET se introducirán más adelante).

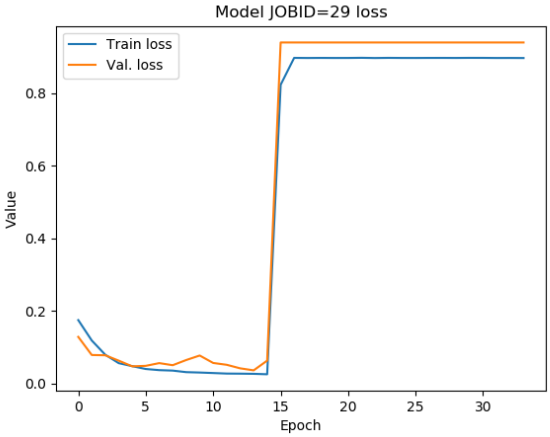
El algoritmo que realizará la optimización en este caso ha sido el SGD, con un momentum de $0,99$. Para el valor del *learning rate* se ha escogido $0,001$, que es algo menor del valor por defecto de SGD en Keras $(0,01)$. En un principio, se hicieron pruebas con Adam como optimizador, pero muchos de los experimentos tenían saltos muy grandes en el coste (incluso con un *learning rate* bajo), desde unas pequeñas variaciones justo al final como se muestran en la Figura 6.8a, hasta saltos enormes como se muestran en la Figura 6.8c.



(a)



(b)



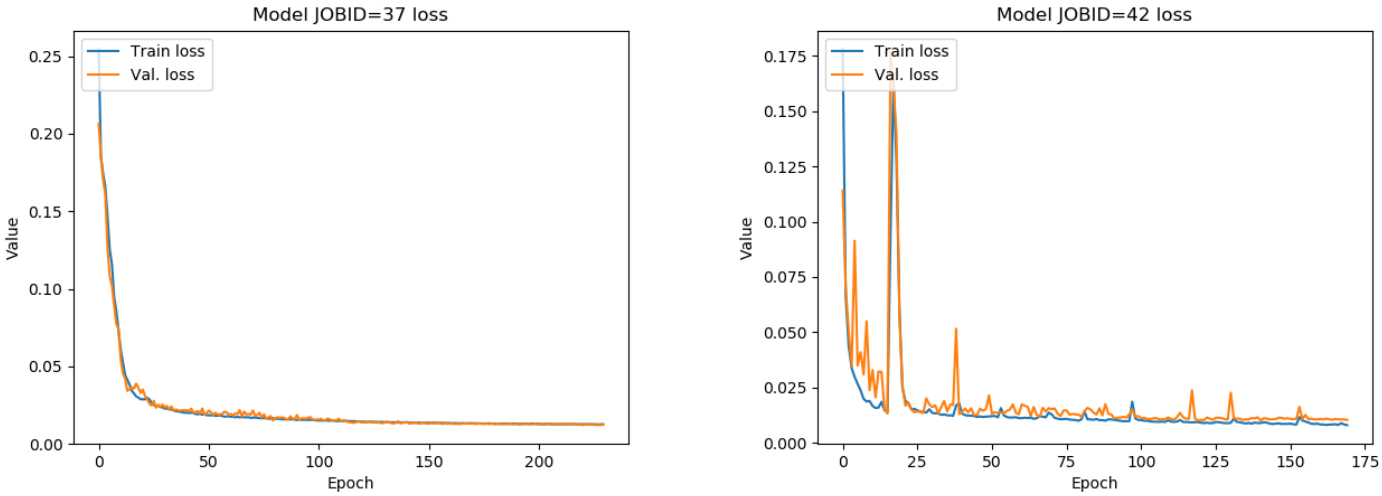
(c)

Figura 6.8: Variación del coste (*loss*) a lo largo de la épocas de entrenamiento de la red utilizando el optimizador Adam. En (a) se puede ver como hay un pequeño salto del coste justo al final. En (b) los saltos son mucho más pronunciados, mientras que en (c) es salto que se da es enorme, anulando completamente el entrenamiento de la red.

Por ello, se ha decidido utilizar SGD con *momemtum* en lugar de Adam, ya que se obtienen mejores resultados y además las diferencias de los valores finales entre los diferentes *runs* se reducen muchísimo. Para que se vea claramente, la diferencia en el comportamiento de los dos algoritmos se muestra en la Figura 6.9, donde se ve que en SGD (Figura 6.9a), la curva es mucho más suave que en Adam (Figura 6.9b). Además, en Adam se ha definido el *learning rate* como la mitad que el usado para SGD, para que las variaciones fueran aún menores.

Como función de coste se ha elegido *binary cross entropy*, comúnmente usada en estos problemas de clasificación de imagen biomédica. Su fórmula está definida en la Ecuación 2.5.

Como se explicó en la sección 2.6 sobre el SGD, se trabajará con los llamados *minibatches* en vez de realizar los cambios de los pesos/sesgos con todo el conjunto de datos. La variable que controla el número de *batches* se llama *batch size*. A la hora de entrenar en GPU, resulta necesario la definición de un *batch size* adecuado, ya que esto también decidirá el número de imágenes que se llevará a la memoria de la GPU para realizar los cálculos del *batch*, y posteriormente actualizar los valores de los pesos/sesgos. En esta configuración base se ha escogido un valor de 1 como *batch size*.



(a) (b)

Figura 6.9: (a) Reducción del coste a lo largo de las épocas de entrenamiento mediante: (a) El optimizador SGD con momentum; (b) El optimizador de Adam.

[5]Imagen creada a partir del proyecto de Github de Haris Iqbal: https://github.com/HarisIqbal88/PlotNeuralNet

Configuración	Media Coste entren.	Media Coste val.	Media Coste test	Media Jaccard test
Base	0,0125	0,0120	0,0312	0,8431

Tabla 6.1: Resultados obtenidos con la configuración base: *batch size*=1, *epoch*=230, condición de parada: *loss patience*=50, uso de *data augmentation* (DA) y como optimizador SGD.

Finalmente, la red devolverá como resultado una probabilidad por cada píxel, la cual representa la probabilidad de pertenecer a la clase mitocondria. De esta manera, se establece un umbral de 0,5[6] y se binariza la imagen, para así poder compararla con el *ground truth*. Los resultados obtenidos son los que se muestran en la Tabla 6.1.

6.3.2. Pesos en las clases

El problema que se afronta está bastante desbalanceado ya que sólo alrededor de un 5 % de los píxeles están etiquetados con la clase mitocondria, el resto es *background*. Por ello, se propone ponderar el peso que se les da a las clases, de manera que se puede equilibrar el problema, dándole más peso a la clase mitocondria que a la de *background*. Realmente estos pesos modifican la función de coste para que el coste de equivocarse en una de las clases sea mayor o menor.

Configuración	Pesos de las clases	Media Coste entren.	Media Coste val.	Media Coste test	Media Jaccard test
Base	0,9 ; 1,1	0,0125	0,0121	0,0307	0,8447
Base	**0,7 ; 1,3**	**0,0125**	**0,0120**	**0,0307**	**0,8448**
Base	0,5 ; 1,5	0,0125	0,0121	0,0308	0,8444
Base	0,3 ; 1,7	0,0125	0,0121	0,0310	0,8438
Base	0,1 ; 1,9	0,0125	0,0121	0,0312	0,8427
Base	1,1 ; 0,9	0,0125	0,0121	0,0313	0,8423
Base	1,3 ; 0,7	0,0125	0,0121	0,0311	0,8429
Base	1,5 ; 0,5	0,0125	0,0121	0,0310	0,8434
Base	1,7 ; 0,3	0,0125	0,0121	0,0308	0,8442
Base	1,9 ; 0,1	0,0125	0,0120	0,0314	0,8416

Tabla 6.2: Mejores resultados obtenidos dándole más peso a la clase mitocondria que a la clase *background* (mejor resultado en negrita). Dentro del vector de pesos denotado como [a ; b], el primer valor a representaría el peso dado a la clase *background*, mientras que b representaría el peso de la clase mitocondria.

[6]Normalmente es el umbral que se usa en el estado del arte

Los mejores resultados obtenidos están mostrados en Tabla 6.2. Aquí, como se puede observar, al no tener resultados claros dándole más peso a la clase mitocondria que a la de *background*, se hicieron pruebas invirtiendo los valores. A pesar de que no se esperaban buenos resultados, los valores que se han obtenido rondan las mismas cifras. Por ello, se deduce que está técnica no ha tenido efecto alguno en el entrenamiento de la red y se descartará en futuros experimentos.

6.3.3. Ajuste de *epochs* y *learning rate*

En este caso, se ha intentando ajustar los parámetros *epochs* y *learning rate* en busca de la mejor combinación de ellos respecto a la configuración base presentada. En la Tabla 6.3 se muestra un resumen de todos los mejores experimentos que se han realizado entorno a ello.

ID	***Learning rate***	**Media epochs**	***Max. epochs***	**Media Coste entren.**	**Media Coste val.**	**Media Coste test**	**Media Jaccard test**
38	0,001	400,0	393	0,0115	0,0115	0,0349	0,8407
37	0,001	230,0	230	0,0125	0,0120	0,0312	0,8431
39	0,01	345,1	400	0,0085	0,0101	0,0386	0,8556
76	0,01	302,1	600	0,0089	0,0102	0,0374	0,8570
69	0,01	179,3	180	0,0101	0,0107	0,0330	0,8571
70	0,01	222,4	240	0,0096	0,0105	0,0338	0,8575
74	0,01	335,0	480	0,0087	0,0103	0,0358	0,8581
75	0,01	335,0	520	0,0087	0,0103	0,0358	0,8581
73	0,01	359,7	420	0,0083	0,0100	0,0360	0,8590
68	0,01	120,0	120	0,0110	0,0112	0,0293	0,8592
71	0,01	255,2	300	0,0092	0,0103	0,0336	0,8604
72	**0,01**	**311,1**	**360**	**0,0087**	**0,0102**	**0,0346**	**0,8614**
40	0,05	51,0	400	1,3531	2,3476	2,2796	0,0053
41	0,1	51,0	400	2,1925	2,3476	2,2796	0,0053

Tabla 6.3: Selección de resultados obtenidos combinando diferentes *learning rates* y *epochs* respecto a la configuración base (mejor resultado en negrita). Cada fila de la tabla representa un experimento, que tiene asociado un identificador, y se han separado los experimentos por bloques de valores de *learning rates*. Dentro de esos grupos se han ordenado por los valores de Jaccard.

Se puede observar que con *learning rates* muy bajos y altos, es decir, con $0,001$, $0,05$ y $0,1$ los resultados son peores, ya que el índice de Jaccard que se consigue es demasiado bajo respecto a la media de los otros experimentos que se presentan.

Los mejores resultados se han obtenido con el valor de *learning rate* a $0,01$ y dejando

a la red entrenar como máximo hasta 360 épocas. El índice de Jaccard obtenido en este caso ha sido de $0,8614$, y se establece como el mejor valor obtenido hasta el momento.

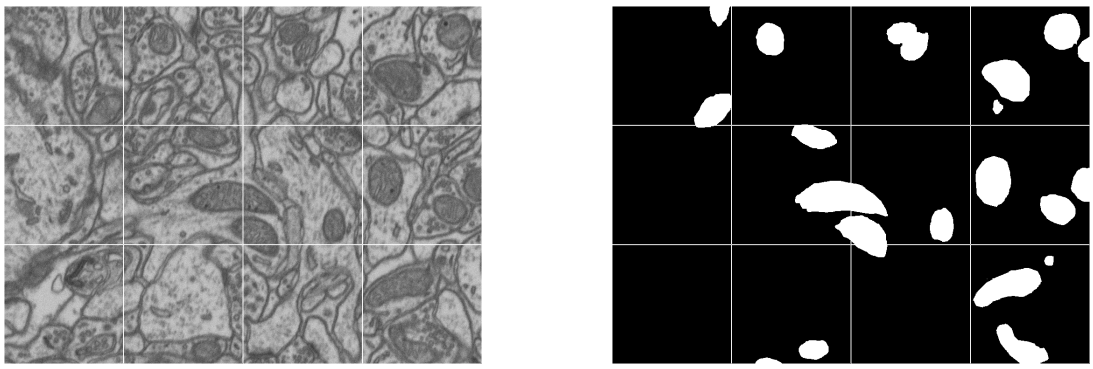
La tabla 6.4 muestra un resumen de los mejores experimentos obtenidos en este apartado.

Configuración	***Learning rate***	***Epochs***	**Media Coste entren.**	**Media Coste val.**	**Media Coste test**	**Media Jaccard test**
Base	0,1	400	2,1925	2,3476	2,2796	0,0053
Base	0,05	400	1,3531	2,3476	2,2796	0,0053
Base	0,001	230	0,0125	0,0120	0,0312	0,8431
Base	**0,01**	**360**	**0,0087**	**0,0102**	**0,0346**	**0,8614**

Tabla 6.4: Resumen de los mejores resultados obtenidos sobre la configuración base combinando diferentes *learning rates* y *epochs* (mejor resultado en negrita).

6.3.4. Troceado de las imágenes

Cuando se trabaja con imágenes biomédicas, suele ser habitual dividirlas en trozos más pequeños para su procesamiento. La mayor razón para hacer esto está en que el aprovechamiento de memoria de la GPU es mayor, ya que, como se comentó anteriormente, se cargan las imágenes en la memoria de la GPU para posteriormente trabajar con ellas en función del *batch size* definido. El que se reduzca el tamaño de la imagen de entrada a la red hará que se pueda jugar con estos valores de *batch size* para poder decidir cual es el mejor de ellos (cosa que hasta ahora no se ha podido hacer por trabajar con la imagen completa).



(a) (b)

Figura 6.10: Ejemplo de troceado de las imágenes del *dataset FIBSEM_EPFL*, donde las rayas blancas delimitan los trozos que se han realizado.

En este caso de *FIBSEM_EPFL*, como las imágenes son de tamaño 1024×768 se dividirán en pequeños trozos de 256×256 píxeles. Al ser las dimensiones de la imagen

divisibles por esos valores, no se ha necesitado ninguna técnica para tratar el caso de los bordes. Aún así, esto es algo que se tiene que tener en cuenta, y se deberá pensar en cómo rellenar el borde, o bien estableciendo un mismo valor para todo el relleno, o haciendo "mirroring", o solapando los trozos etc. Un ejemplo de troceado se presenta en la Figura 6.10.

Mediante este troceado los mejores resultados que se consiguen se muestran en la Tabla 6.5.

Configuración	**Batch size**	**Media Coste entren.**	**Media Coste val.**	**Media Coste test**	**Media Jaccard test**
Troceado	2	0,0103	0,0120	0,0380	0,8407
Troceado	**4**	**0,0113**	**0,0122**	**0,0324**	**0,8487**
Troceado	8	0,0123	0,0122	0,0353	0,8398

Tabla 6.5: Mejores resultados obtenidos mediante la realización de recortes de las imágenes y probando diferentes *batch sizes* (mejor resultado en negrita).

Comparando los resultados con la configuración base, donde el mejor Jaccard obtenido era de $0,8431$, se ha mejorado sustancialmente ese valor. Por ello, se ha decidido explotar este troceado de las imágenes en futuros experimentos para intentar alcanzar mejores resultados que los obtenidos en la anterior sección.

6.3.5. Troceado: ajuste de *epochs*, *learning rate* y *batch size*

Al haber obtenido mejores resultados con el troceado de las imágenes, se ha decidido realizar el mismo estudio que se hizo en la sección 6.3.3, para ajustar los parámetros *epochs*, *learning rate* y, también en este caso, *batch size*, en busca de la mejor combinación de ellos. En la Tabla 6.6 se muestra un resumen de todos los mejores experimentos que se han realizado entorno a ello y que siguen la misma ordenación que la Tabla 6.3.

Se puede observar que, con *learning rates* muy bajos, es decir, con $0,0001$ y $0,0005$, los resultados son malos. Aún habiéndole dejado margen a la red para realizar el entrenamiento, el índice de Jaccard que se consigue es demasiado bajo respecto a la media de los otros experimentos que se presentan.

Learning rate $= 0,001$

En este grupo con learning rate de $0,001$, se puede observar que los peores resultados se han obtenido con valores de *batch size* más altos y, a medida que ese valor decrece, los valores de Jaccard mejoran.

Los mejores resultados se obtienen en el experimento 45, con un *batch size* de 4, entrenando con 230 *epochs*, donde casi todos los *runs* de ese experimento han llegado a entrenar todas las épocas que se han definido. El mejor Jaccard obtenido ha sido de $0,8487$.

Learning rate $= 0,005$

ID	***Learning rate***	***Batch size***	Media ***epochs***	Max. ***epochs***	Media Coste entren.	Media Coste val.	Media Coste test	Media Jaccard
82	0,0001	6	360,0	360	0,0172	0,0172	0,0358	0,8130
83	0,0005	6	360,0	360	0,0121	0,0121	0,0349	0,8409
49	0,001	64	230,0	230	0,0239	0,0245	0,0381	0,7862
48	0,001	32	230,0	230	0,0165	0,0149	0,0349	0,8206
98	0,001	32	360,0	360	0,0145	0,0138	0,0352	0,8276
91	0,001	3	347,4	360	0,0099	0,0112	0,0413	0,8277
47	0,001	16	230,0	230	0,0140	0,0137	0,0352	0,8332
96	0,001	9	347,7	360	0,0118	0,0119	0,0354	0,8341
43	0,001	1	221,7	230	0,0093	0,0120	0,0417	0,8381
95	0,001	8	357,6	360	0,0115	0,0116	0,0399	0,8382
97	0,001	16	360,0	360	0,0126	0,0126	0,0364	0,8387
94	0,001	7	357,4	360	0,0113	0,0118	0,0363	0,8396
46	0,001	8	230,0	230	0,0123	0,0122	0,0353	0,8398
84	0,001	6	342,1	360	0,0112	0,0119	0,0363	0,8407
44	0,001	2	225,1	230	0,0103	0,0120	0,0380	0,8407
93	0,001	5	289,3	360	0,0112	0,0118	0,0354	0,8416
92	0,001	4	312,5	360	0,0106	0,0119	0,0363	0,8441
45	**0,001**	**4**	**225,7**	**230**	**0,0113**	**0,0122**	**0,0324**	**0,8487**
86	0,005	3	291,9	360	0,0079	0,0109	0,0442	0,8247
88	0,005	5	256,4	360	0,0090	0,0113	0,0383	0,8406
87	0,005	4	321,7	360	0,0080	0,0107	0,0404	0,8407
89	0,005	7	310,7	360	0,0092	0,0109	0,0398	0,8438
90	0,005	8	222,1	360	0,0102	0,0121	0,0369	0,8455
85	0,005	6	291,1	360	0,0088	0,0108	0,0380	0,8473
78	0,01	3	258,7	360	0,0078	0,0108	0,0433	0,8268
79	0,01	4	232,1	360	0,0084	0,0113	0,0398	0,8348
80	0,01	5	267,0	360	0,0084	0,0108	0,0405	0,8372
81	0,01	6	268,4	360	0,0083	0,0107	0,0405	0,8430

Tabla 6.6: Mejores resultados obtenidos combinando diferentes valores de *learning rates*, *epochs* y *batch sizes* (mejor resultado en negrita). Se han separado los experimentos por bloques de valores de *learning rate* y dentro de esos grupos se han ordenado por el índice Jaccard obtenido.

Para este subgrupo los mejores resultados se han obtenido con *batch size* más altos que para el anterior. En general, haber aumentado el *learning rate* no parece haber mejorado los resultados ya obtenidos en el anterior grupo, ya que, a pesar de que los valores del

coste para el entrenamiento y validación desciendan, no lo hacen para el conjunto de test (lo que indica un posible *overfitting* sobre datos).

Learning rate $= 0,01$

En este grupo los resultados obtenidos han sido generalmente malos, lo que parece indicar que un aumento del *learning rate* más allá de $0,01$ no mejora el entrenamiento (algo que también se produjo en la Tabla 6.3).

La Tabla 6.7 muestra un resumen de los mejores experimentos obtenidos en este apartado, con un valor máximo de $0,8487$ de índice de Jaccard, mediante el uso de un *learning rate* de $0,001$, *batch size* de 4 y entrenando la red con 230 épocas como máximo.

Configuración	*Learning rate*	*Batch size*	*Epochs*	Media Coste entren.	Media Coste val.	Media Coste test	Media Jaccard test
Troceado	0,0001	6	360	0,0172	0,0172	0,0358	0,8130
Troceado	0,0005	6	360	0,0121	0,0121	0,0349	0,8409
Troceado	**0,001**	**4**	**230**	**0,0113**	**0,0122**	**0,0324**	**0,8487**
Troceado	0,005	6	360	0,0088	0,0108	0,0380	0,8473
Troceado	0,01	6	360	0,0083	0,0107	0,0405	0,8430

Tabla 6.7: Resumen de los mejores resultados obtenidos con el troceado y mediante la combinación de diferentes valores de *learning rates*, *epochs* y *batch sizes* (mejor resultado en negrita).

6.3.6. Evaluación de *overfitting*

A pesar de haber intentado ajustar los parámetros de *batch size*, *learning rate* y *epochs* en la anterior sección, no se han visto mejoras en el resultado mediante el troceado de las imágenes. En busca del problema, se ha visto que hay una correlación negativa entre el coste de entrenamiento y el de test: a medida que se mejora en el coste de entrenamiento, el coste del conjunto de test empeora, lo que indica un posible overfitting sobre los datos (como se muestra en la Figura 6.11).

Además de eso, a medida que mejora el coste de entrenamiento también lo hace el de validación, como es de esperar, hasta que en un punto tiende de nuevo a subir, tal y como la curva de tendencia sugiere.

Por esto, y para averiguar si se está haciendo o no *overfitting*, se han realizado múltiples experimentos: se ha fijado un valor de épocas máximas con las que la red podrá entrenarse y se omite el *early stopping* de la variable *patience*, el cual ha estado fijado en 50, tal y como se especificó en la configuración base (recuérdese que esta variable servía para parar el entrenamiento de la red si en un número de épocas definido no se había mejorado el resultado). De este modo, se ha podido medir en qué momento la red

empieza a realizar unas peores predicciones por haber aprendido “demasiado” el conjunto de entrenamiento.

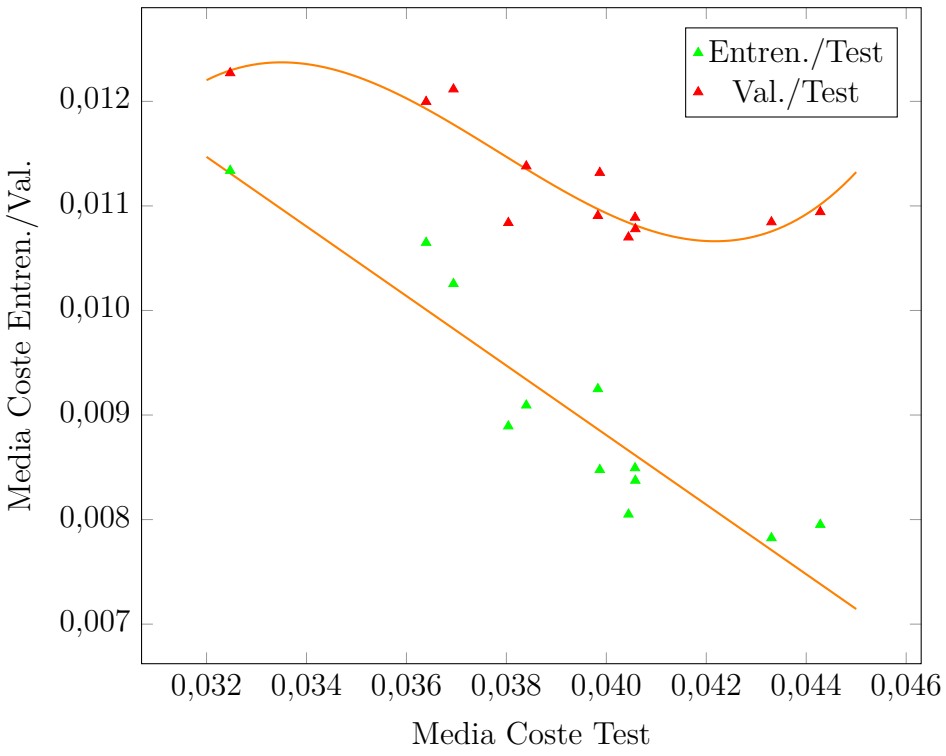
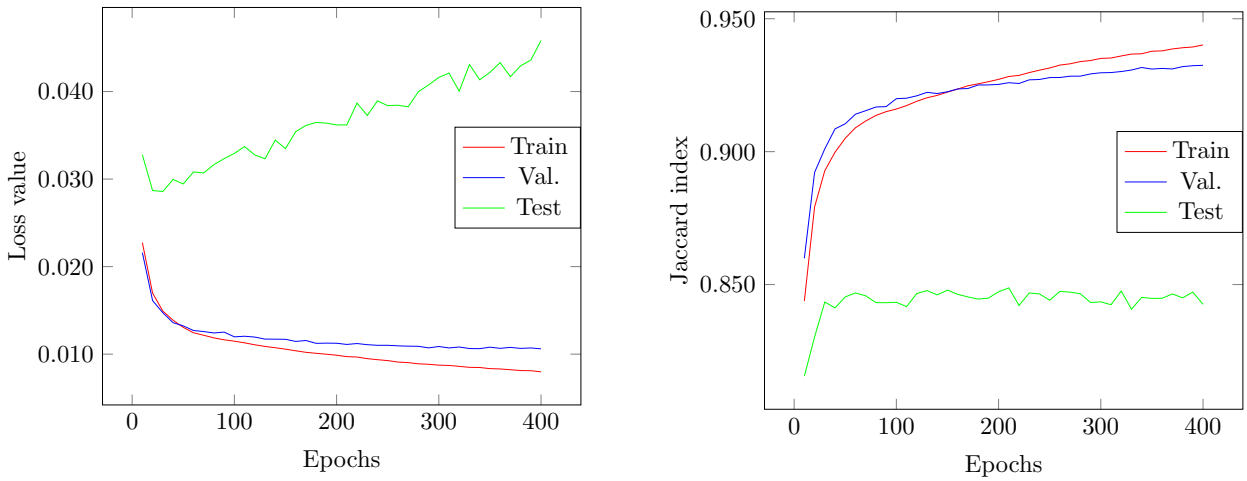


Figura 6.11: Correlación entre el coste de entrenamiento frente al de test (en verde) y entre el coste de validación y test (en rojo). Se han dibujado las líneas de tendencia en naranja.

Primera prueba

En esta primera prueba se decidió experimentar sobre el mejor resultado obtenido con un *learning rate* de $0,005$ y *batch size* de 6. Como se puede observar en la Figura 6.12a, el coste de entrenamiento baja con el número de épocas, y el de validación también hasta cierto punto. El coste de test, sin embargo, parece que tiende a subir después de unas pocas épocas.

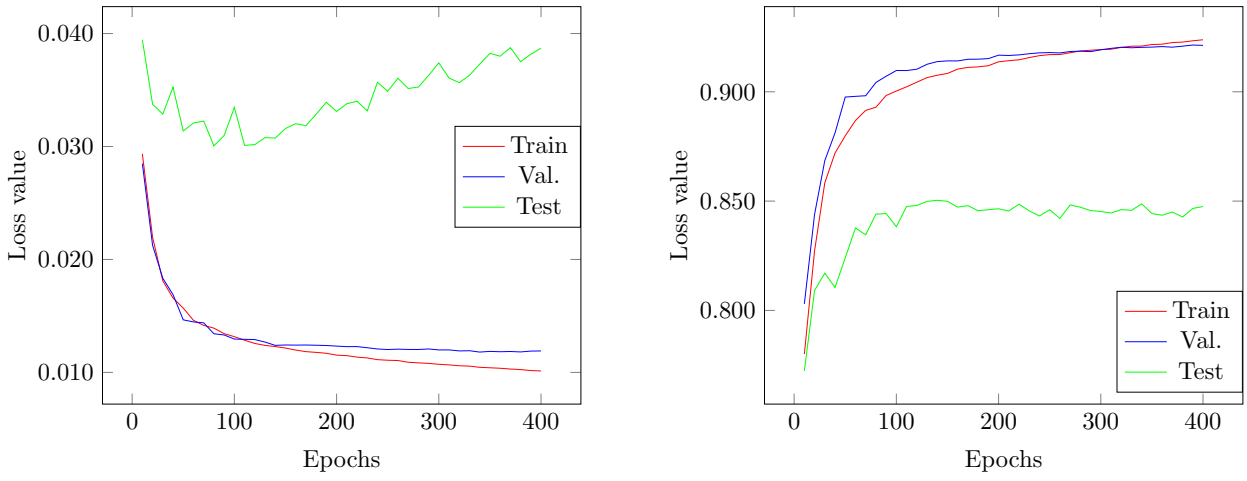
Por otro lado, si uno se fija en la Figura 6.12b, a pesar de que el coste de test aumente a lo largo de las épocas, el valor del Jaccard parece mantenerse. Esto resulta bastante extraño ya que tendría que tender a subir también a medida que sube el coste. En cuanto al índice de Jaccard de entrenamiento y validación, parecen tender a incrementarse con el número de épocas.



(a) (b)

Figura 6.12: (a) Variación del coste en función al número de épocas máximas que se le dan a la red para entrenar; (b) Índice de Jaccard obtenido a lo largo de las épocas.

Segunda prueba



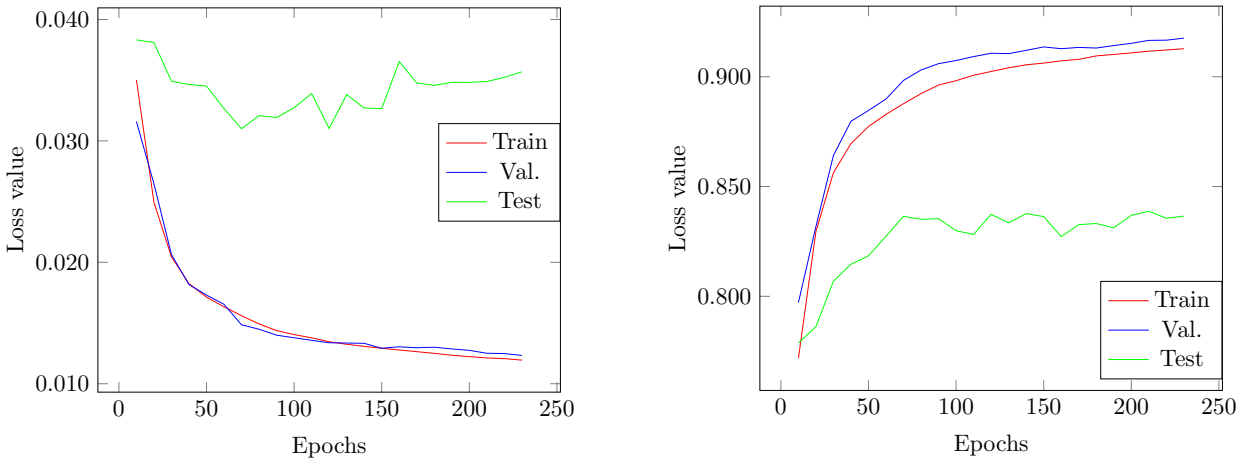
(a) (b)

Figura 6.13: (a) Variación del coste en función al número de épocas máximas que se le dan a la red para entrenar; (b) Índice de Jaccard obtenido a lo largo de las épocas.

En esta segunda prueba se ha utilizado otro de los mejores valores obtenidos también en la sección anterior: *learning rate* de $0,001$ y *batch size* de 4. Los resultados obtenidos son los que se muestran en la Figura 6.13b.

El comportamiento en este caso es parecido al del la primera prueba, donde se aprecia que el coste de entrenamiento y validación bajan a la vez hasta un punto, donde el de validación parece estancarse. El coste de test sobre las 100 épocas empieza a subir de nuevo. Por otro lado, el resultado del Jaccard parece seguir la misma pauta de nuevo, manteniéndose el de test estancado a pesar de que su coste empeore a partir de las 100 épocas.

Tercera prueba



(a) (b)

Figura 6.14: (a) Variación del coste en función al número de épocas máximas que se le dan a la red para entrenar; (b) Índice de Jaccard obtenido a lo largo de las épocas.

En esta última prueba se ha escogido como *learning rate* $0,001$, al igual que en la segunda prueba, pero esta vez se ha aumentado el *batch size* a 6. Los resultados se muestran en la Figura 6.14b, y esta vez se han realizado menos experimentos, por ello los gráficos se paran a las 230 épocas. El comportamiento de los costes y el índice de Jaccard parecen seguir el mismo patrón que en las anteriores pruebas.

Conclusión

La conclusión que se puede obtener de este estudio es que el coste de test parece aumentar pasadas ciertas épocas, lo que parecía una muestra de *overfitting* sobre los datos. No obstante, esto parece no influir en la métrica de Jaccard. Realmente el foco se tiene en la métrica de Jaccard, por lo que se concluye que, a pesar de este comportamiento, con estos datos y esta red, no parece que se esté haciendo *overfitting* sobre los datos.

6.3.7. Implementación propia de *data augmentation*

Hasta ahora se ha realizado un DA mínimo con la implementación de Keras, realizando *flips* verticales y horizontales además de una rotación aleatoria de 180 grados. En esta sección, se aborda una implementación de DA propia en la que se introducen nuevas transformaciones a fin de poder aumentar la generalización de la red.

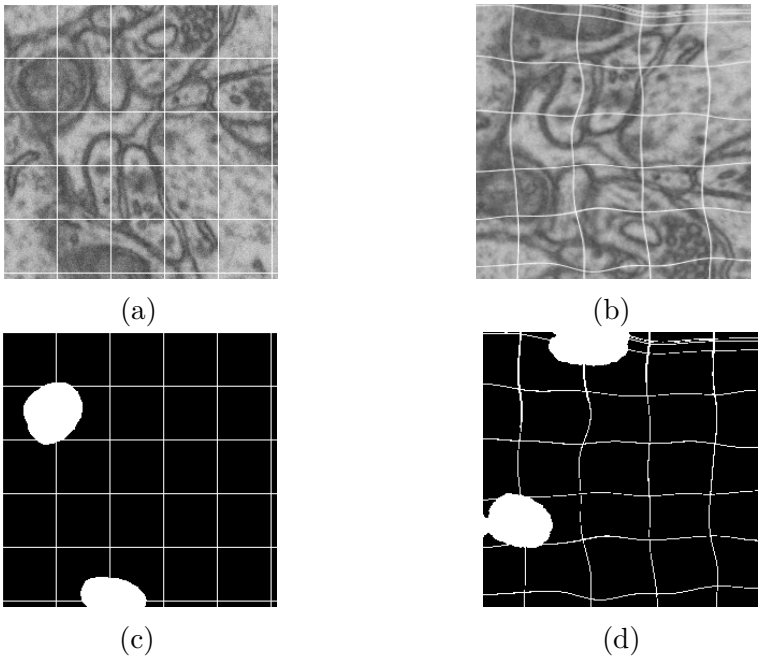


Figura 6.15: Ejemplo de imágenes obtenidas mediante la implementación propia de *data augmentation* (DA). En este caso las imágenes son todas trozos, donde (a) y (c) son la imagen original y su máscara respectivamente, y (b) y (d) son los resultados de aplicar a esas dos primeras imágenes un *flip* horizontal, una rotación de 180º y una deformación elástica.

Realizar una implementación propia de DA supondrá tener libertad de desarrollar la transformaciones que se quieran a la hora de entrenar, incluso alguna que Keras no cubre.

Esta implementación propia también se ha visto promovida por el hecho de querer sacarle más partido al troceado. Haber dividido la imagen en trozos más pequeños y cuadrados abre un abanico de posibilidades nuevas con las que poder hacer DA, ya que se pueden hacer rotaciones de ángulos múltiplos de 90 sin tener que preocuparse por el tamaño de la imagen. Esto es algo que antes no se podía hacer, ya que al no ser cuadrada y girar la imagen 90 grados, el ancho y alto de la imagen cambia (algo que la red no se espera ya que se le ha especificado un ancho y alto que no se cumple en ese caso).

Después de probar varias opciones, la última implementación realiza las transformaciones que se muestran a continuación, cada una de ellas con una probabilidad que se especifica con independencia del resto:

1. Flip horizontal, vertical, los dos a la vez, o ninguno (cada opción con un 25 % de probabilidad).
2. Rotación de 90º, 180º, 270º o 0º (cada una con 25 % de probabilidad).
3. Deformaciones elásticas con una probabilidad que se establezca previamente en el experimento, ya que es una transformación muy costosa.

De esta manera, podrá haber imágenes de entrenamiento que realicen varias transformaciones a la vez, mientras que otras puede que no sufran ninguna.

Configuración	***Data Augmen.***	***Learning rate* entren.**	***Batch size***	***Epochs***	**Media Coste entren.**	**Media Coste val.**	**Media Coste test**	**Media Jaccard test**
Base	**ela=0,7**	**0,01**	**1**	**360**	**0,0121**	**0,0098**	**0,0287**	**0,8660**
Troceado	ela=0	0,001	6	360	0,0127	0,0119	0,0307	0,8641
Troceado	ela=0,4	0,001	6	360	0,0150	0,0122	0,0262	0,8655

Tabla 6.8: Resumen de los mejores resultados obtenidos mediante la implementación de un *data augmentation* propio, seleccionando una probabilidad para las transformaciones elásticas (ela=x), probando diferentes *learning rates*, *batch sizes* y *epochs* (mejor resultado en negrita). La primera fila representa el mejor resultado logrado hasta el momento, y el mejor de toda la tabla se ha marcado en negrita.

Además de probar esta implementación con las imágenes troceadas, también se han realizado experimentos con las imágenes originales, produciendo rotaciones de 180º con un 50 % de probabilidad. Los mejores resultados en los dos casos son los que se muestran en la Tabla 6.8 que, como se puede observar, la realización del troceado junto con el DA propio ya alcanza los valores máximos obtenidos hasta el momento con las imágenes originales. Por los resultados obtenidos, esta técnica se posiciona como la que mayor mejora ha producido hasta el momento.

Aparte de todo ello, cabe mencionar el hecho de que con las imágenes originales la red ha tardado alrededor de 77 horas en realizar el entrenamiento, mientras que con el troceado el entrenamiento ha sido de unas 13 horas.

En la Figura 6.15 se presentan varios ejemplos de DA extraídos mediante la implementación propia realizada.

6.3.8. Descartes

Inspirado por el trabajo de Oztel *et al.* [66], se probará en esta sección a descartar aquellas imágenes que tengan menos de un porcentaje de píxeles etiquetados como mitocondria. De nuevo, es posible la realización de esta técnica gracias al troceado de las imágenes, siendo posible descartar aquellos trozos en los que no haya mitocondrias etiquetadas.

Configuración	Descartes	*Learning rate*	*Batch size*	*Epochs*	Media Coste entren.	Media Coste val.	Media Coste test	Media Jaccard test
Troceado	**5 %**	**0,001**	**1**	**360**	**0,0148**	**0,0160**	**0,0297**	**0,8642**
Troceado	10 %	0,001	1	360	0,0148	0,0160	0,0297	0,8642
Troceado	15 %	0,001	1	360	0,0147	0,0178	0,0292	0,8634
Troceado	20 %	0,001	1	360	0,0152	0,0147	0,0294	0,8617
Troceado	25 %	0,001	1	360	0,0150	0,0140	0,0288	0,8631
Troceado	30 %	0,001	1	360	0,0152	0,0138	0,0290	0,8619

Tabla 6.9: Resumen de los mejores resultados obtenidos mediante el descarte de aquellas imágenes que no contienen un porcentaje de píxeles etiquetados como mitocondria (marcando el x % mínimo de píxeles etiquetados como mitocondria que tiene que tener la imagen para no ser descartada). Se ha utilizado la implementación personalizada de DA (sin transformaciones elásticas) y se han probado diferentes *learning rates*, *batch sizes* y *epochs*.

Se han probado diferentes porcentajes para descartar, empezando desde un 5 %, siendo suficiente para descartar aquellas imágenes cuyas máscaras no tengan nada de información sobre la clase mitocondria, hasta un 30 %, en el que la cantidad de imágenes descartadas ya empieza a penalizar el entrenamiento de la red.

Los resultados obtenidos se han presentado en la Tabla 6.9 y sugieren que, con estos datos y esta arquitectura de red, no hay una mejoría aparente al realizar esta técnica. La comparación habría que hacerla con la segunda fila de la anterior sección (donde se obtuvo un índice de Jaccard de $0,8641$), ya que no se han realizado transformaciones elásticas al igual que en ese experimento.

6.3.9. Cambios en la arquitectura de la U-Net

En esta sección se presentan varias modificaciones en la arquitectura base U-Net que se han estudiado:

1. Función de activación a ReLU en vez de ELU.

2. Cambiar las capas de *pooling* por capas llamadas *average pooling*. Estas capas tienen la misma manera de funcionar que las capas de *max pooling* presentadas en la sección 3.4 pero, en vez de coger el máximo valor entre los píxeles, hacen una media de todos ellos.

3. Cambio de *dropout* a *spatial dropout* presentado por Tompson *et al.* en [76]. Este tipo de *dropout* presentado para las CNN funciona de la misma forma que el *dropout* original salvo que, en vez de descartar una neurona a la vez con su probabilidad dada, lo que se hace es descartar todas las neuronas a lo largo de la dimensión z a la vez. Por ejemplo, si el *input* es [[1, 1, 1], [2, 2, 2]], el *dropout* original elegirá descartar individualmente cada neurona, por lo que una posible opción sería esta: [[1, 0, 1], [2, 2, 2]], donde se ha reemplazado con un 0 la neurona que se ha descartado. En el caso de *spatial dropout* ese mismo descarte quedaría como [[1, 0, 1], [2, 0, 2]], descartando todas las neuronas a lo largo de la dimensión z.

De estos cambios ninguno ha mejorado la clasificación que ya se había logrado, no obstante, se muestra en la Tabla 6.10 los resultados obtenidos al igual que en las secciones anteriores.

Configuración	**Mod. arquitectura base**	***Learning rate***	***Batch size***	***Epochs***	**Media Coste entren.**	**Media Coste val.**	**Media Coste test**	**Media Jaccard test**
Troceado	ReLU	0,001	6	360	0,0133	0,0123	0,0275	0,8583
Troceado	*average pooling*	0,001	6	360	0,0126	0,0117	0,0312	0,8617
Troceado	***spatial dropout***	**0,001**	**6**	**360**	**0,0133**	**0,0119**	**0,0284**	**0,8623**

Tabla 6.10: Resumen de los mejores resultados obtenidos realizando cambios en la arquitectura de la U-Net (mejor resultado conseguido hasta el momento en negrita). Se ha hecho uso de la implementación de *data augmentation* propia para entrenar la red.

6.3.10. Funciones de coste diferentes

En este apartado se describen las pruebas de las diferentes funciones de coste que se han realizado. Recuérdese que se ha estado trabajando hasta ahora con la función de coste *binary cross entropy* formulada en la sección 2.5. No obstante, se probará con la función denominada *Dice loss* o *Dice-coefficient* (que es comúnmente usada en la segmentación de imágenes biomédicas) y con *Jaccard loss* (que sigue la fórmula ya presentada en las secciones 6.1 y 6.2).

La fórmula del *Dice loss* es muy parecida a la del Jaccard loss y puede ser definida así:

$$Dice = \frac{2TP}{2TP + FP + FN} = \frac{2|A \cap B|}{|A| + |B|} \tag{6.7}$$

Donde TP es el número de *true positives*, FP el número de *false positives* y FN el número de *false negatives*.

Los resultados se muestran en la Tabla 6.11. Como se puede observar, no se han mejorado los resultados ya obtenidos hasta el momento con *binary cross entropy*.

Configuración	**Función de coste**	***Learning rate***	***Batch size***	***Epochs***	**Media Coste entren.**	**Media Coste val.**	**Media Coste test**	**Media Jaccard test**
Troceado	*Jaccard loss*	0,001	6	360	0,6544	0,6375	0,6630	0,3523
Troceado	*Dice loss*	0,001	6	360	0,0331	0,0287	0,0611	0,8624
Troceado	***Binary cross entropy***	**0,001**	**6**	**360**	**0,0150**	**0,0122**	**0,0262**	**0,8655**

Tabla 6.11: Resumen de los resultados obtenidos probando diferentes funciones de coste (mejor resultado hasta el momento en negrita). Se ha hecho uso de la implementación de *data augmentation* propia para entrenar la red.

6.3.11. Corrección del Jaccard y VOC

En cierto momento durante los experimentos, se descubrió que la manera en la que se estaba reportando el índice de Jaccard y el VOC podía no ser del todo correcta al realizar el troceado de las imágenes. El problema radicaba en que el valor del índice de Jaccard (y consecuentemente del VOC) se calculaba como media del valor de cada uno de los trozos. Esto es problemático ya que en trozos en los que no hay ninguna mitocondria, es decir, que todos sus píxeles son *background*, una predicción correcta produce un valor de Jaccard igual a 0.

Por ello, se realizó una nueva implementación que calcula los valores de Jaccard y VOC por imagen completa (donde siempre hay mitocondrias). De manera retrospectiva se calculó este nuevo valor de Jaccard/VOC para los mejores resultados obtenidos, y son estos resultados los que finalmente se han usado para comparar el trabajo realizado aquí con otros del estado del arte.

A los valores de esta nueva implementación se les añadirá el sufijo "por imagen", mientras que los anteriores aparecerán como "por trozo". Nótese que existe cierta incertidumbre sobre las métricas de evaluación reportadas en los trabajos publicados, ya que muchas veces su código no está disponible.

6.3.12. ResUNet y DA de Keras

El uso de conexiones residuales [42] (véase la sección 3.7) puede mejorar la clasificación en determinadas tareas. Trabajos del estado del arte citados aquí los utilizan, como en [70] y [52], obteniendo buenos resultados.

Hasta ahora se estaban utilizando los bloques de convolución clásicos, compuestos por dos capas de convolución y una de *pooling*. Sin embargo, en estos experimentos se han sustituido por los bloques residuales, con el fin de poder mejorar la clasificación.

Además, también se han probado diferentes valores de filtros, los cuales se multiplican por dos a medida que se hace *downsampling* y se dividen entre dos en el *upsampling*. Hasta ahora se empezaba con 16 filtros en la primera etapa, hasta llegar a los 256. Se probará entonces con valores como 32, 48 y hasta 64 (tal y como la U-Net original tenía en [45]).

Para finalizar, con los cambios de este grupo de experimentos, se han realizado pruebas cambiando de nuevo a la implementación de DA realizada por Keras, en vez de seguir utilizando la implementación propia. La implementación propia dio margen para poder realizar el DA de manera personalizada, aplicando nuevas transformaciones, eligiendo el orden de su aplicación, permitiendo que fueran o no excluyentes entre ellas, con diferentes probabilidades etc. Sin embargo, las opciones con las que finalmente se ha seguido trabajando también las implementa Keras, con sus propios criterios y optimizaciones, por lo que se decide probar de nuevo esa implementación para ver si hay o no mejora. Finalmente, en la Tabla 6.12, se presentan los mejores resultados obtenidos que, como se puede observar, la implementación del DA de Keras ha mejorado sustancialmente los valores del índice de Jaccard obtenidos y siempre mediante el uso de 32 filtros de base.

Configuración	Red	*Data Augmen.*	Numero de filtros	*Learning rate*	*Batch size*	*Epochs*	Media Coste entren.	Media Coste val.	Media Coste test	Media Jaccard test por trozo	Media Jaccard test por imagen
Troceado	ResUNet	**Propia ela=0**	32	0,001	6	360	0,0118	0,0114	0,0320	-	0,8577
Troceado	ResUNet	Keras	32	0,001	6	360	0,0298	0,0138	0,0231	0,8704	0,8679
Troceado	U-Net	**Propia ela=0**	32	0,001	6	360	0,0138	0,0114	0,0262	-	0,8665
Troceado	**U-Net**	**Keras**	**32**	**0,001**	**6**	**360**	**0,0275**	**0,0149**	**0,0222**	**0,8735**	**0,8700**

Tabla 6.12: Resumen de los resultados obtenidos mediante el uso de la U-Net, ResUNet y la implementación de DA de Keras (mejor resultado en negrita).

6.3.13. Efecto borde

A pesar de que a lo largo de la experimentación se ha dicho que la realización del troceado ha beneficiado en varios aspectos los resultados obtenidos, también tiene sus inconvenientes. Analizando las imágenes resultantes, se puede comprobar que hay un "efecto borde" con la clasificación de algunos píxeles de las imágenes (como se muestra en

la Figura 6.16).

Esto es algo de lo que se habla en [77], donde se expone el mismo problema pero en imágenes satelitales. Para aminorar este error, el autor de ese mismo artículo propone una solución y además aporta el código en Github [7]. La solución consiste en realizar un suavizado en la predicción combinando los diferentes trozos de la imagen. Se realiza una superposición de trozos rotados y *flips*, para realizar una predicción final. Los resultados que se obtienen son bastante buenos en su caso, por eso se ha decidido intentar hacer uso de este suavizado como técnica de post-procesado en este proyecto, calculando de nuevo el Jaccard para los mejores experimentos que se han obtenido, como se puede observar en la Tabla 6.4, concretamente en las columnas etiquetadas con el prefijo "post-proc".

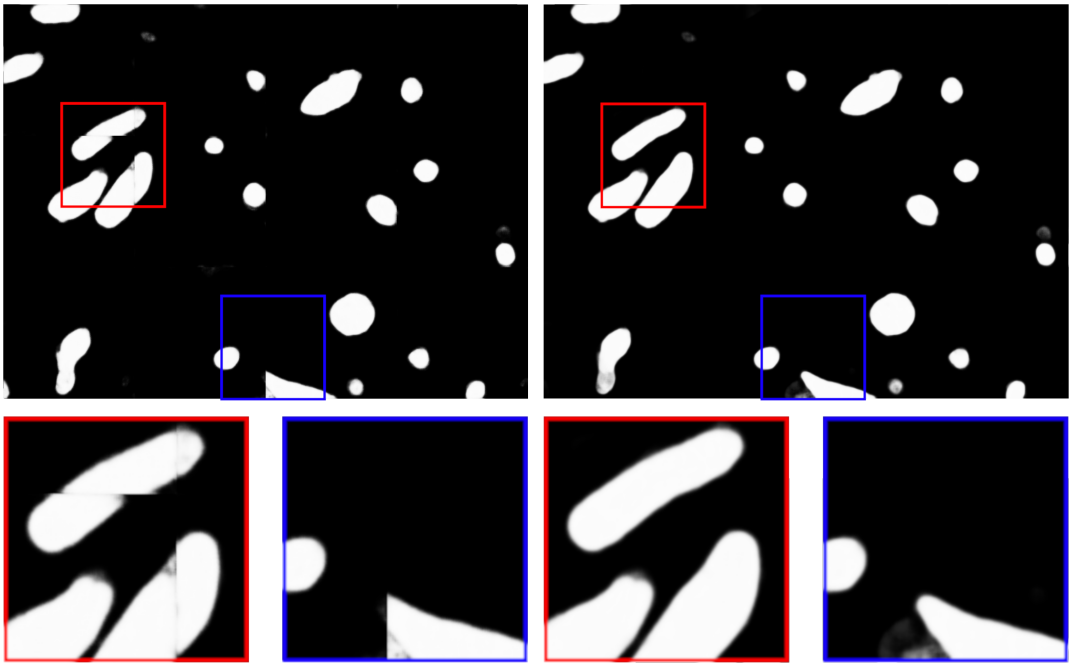


Figura 6.16: Ejemplo del "efecto borde" producido por realizar el troceado de las imágenes. Se puede apreciar que ciertas mitocondrias, que se sitúan en los bordes de algún trozo, están mal etiquetadas. A la izquierda se muestra la predicción "original" realizada por la red, mientras que a la derecha estaría el resultado de la aplicación de la técnica de post-procesado propuesta en [77].

6.3.14. Pruebas con otros *datasets*

Hasta ahora se ha estado trabajando únicamente con el *dataset FIBSEM_EPFL* pero, para poder comparar el trabajo realizado aquí con el estado del arte, se han realizado

[7] https://github.com/Vooban/Smoothly-Blend-Image-Patches

pruebas con otros *datasets* diferentes: *Lucchi++*, *Kasthuri++* y *Achucarro* (véase la sección 6.1).

Los mejores resultados con cada uno de los distintos *datasets* se han mostrado en la Tabla 6.13.

Configuración	*Dataset*	*Descartes*	Número de filtros	*Learning rate*	*Batch size*	*Epochs*	Optim.	Media Coste entren.	Media Coste val.	Media Coste test	Media Jaccard test por trozo	Media Jaccard test por imagen
Troceado	Lucchi++	-	32	0,001	6	360	SGD	0,0272	0,0139	0,0232	0,8946	0,8927
Troceado	Kasthuri++	5 %	32	0,001	6	360	SGD	0,0126	0,0124	0,0069	0,7764	0,9066
Troceado	Achucarro	5 %	16	0,0001	1	360	Adam	0,0654	0,0762	0,0511	-	0,5998

Tabla 6.13: Resumen de los mejores resultados obtenidos en otros *datasets* distintos de *FIBSEM_EPFL* que se ha venido usando hasta ahora (sin post-procesado y haciendo uso del *data augmentation* de Keras).

Como se puede observar en los resultados, para poder entrenar la red con los datos de *Achucarro* se han tenido que modificar bastantes parámetros de la red: realizar descartes ya que muchos trozos de este *dataset* no tienen información de mitocondria, se ha tenido que volver a utilizar 16 filtros en vez de 32, bajar el *learning rate* a 0,0001, un *batch size* de 1 y hasta cambiar el optimizador de SGD a Adam. Decir que, tal y como se comentó en la sección 6.1, este *dataset* tiene pocas imágenes, por ello los resultados obtenidos no han sido muy buenos. Además, la resolución es bastante peor para el caso de este *dataset* comparado con el resto, y la tinción utilizada no logra un contraste muy alto entre las diferentes partes de la célula.

6.3.15. Pruebas *inter-datasets*

En la sección 1.2.1, ya se comentó el concepto de *transfer learning*, donde se busca reusar un modelo entrenado previamente en otros datos. En estos últimos experimentos se aborda esto mismo: predecir los resultados de un *dataset* utilizando los mejores modelos entrenados en el resto. Es decir, el mejor modelo que se ha obtenido en *FIBSEM_EPFL* se utilizará para predecir los datos de *Lucchi++*, *Kasthuri++* y *Achucarro*, y de la misma manera se prueban todo el resto de combinaciones, produciendo así una matriz de resultados entre los diferentes *datasets* (mostrada en la Tabla 6.14).

Para poder realizar estas pruebas a hecho falta ajustar la resolución de las imágenes entre los *datasets*, de manera que la predicción se realice sobre imágenes con la misma resolución que las usadas para el entrenamiento. Por ejemplo, si el modelo a usar se ha entrenado en *FIBSEM_EPFL*, que tiene una resolución de 5×5nm, y se va a predecir sobre el conjunto de test de *Kasthuri++*, con una resolución de 3×3nm, habrá que reducir las imágenes de *Kasthuri++* 0,6 veces ($3/5 = 0,6$). De esta manera, se logrará que un píxel que antes representaba un trozo de 3×3nm ahora represente uno de 5×5nm.

Por otro lado, se han normalizado los datos de cada *dataset* antes de aplicarlo en

los datos de otro, de manera que estén las intensidades de los píxeles estén centradas en cierto punto. Además se ha añadido como técnica de DA un pequeño *zoom* en las imágenes, de manera que la red tenga más variabilidad en el tamaño de las mitocondrias que ve a lo largo de su entrenamiento y haciéndola más flexible a los cambios de tamaño de píxel.

		***Dataset* de entrenamiento**			
		FIBSEM_EPFL	**Lucchi++**	**Kasthuri++**	**Achucarro**
Dataset de test	**FIBSEM_EPFL**	0,8700	0,7717	0,0000	0,4197
	Lucchi++	0,7578	0,8927	0,0000	0,4963
	Kasthuri++	0,0010	0,0044	0,9066	0,0730
	Achucarro	0,0009	0,0024	0,0006	0,5998

Tabla 6.14: Resumen de los mejores resultados (índice de Jaccard) *inter-dataset* (sin post-procesado). Las columnas representan el *dataset* con el que se ha entrenado el modelo, mientras que en las filas representan el origen de los datos a predecir.

Un ejemplo de lectura de la Tabla 6.14 sería la siguiente: la fila 1, la segunda columna, correspondería con el índice de Jaccard obtenido en la predicción realizada sobre los datos de *FIBSEM_EPFL* utilizando el modelo entrenado con *Lucchi++*.

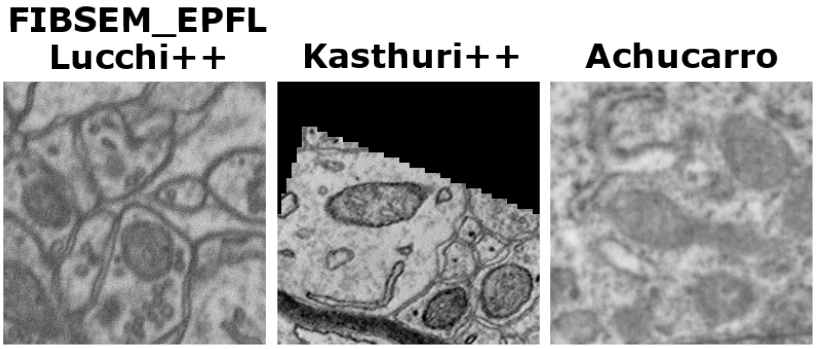


Figura 6.17: Diferencias entre los diferentes *datasets* utilizados donde se muestran una imagen de cada uno de ellos representando el mismo tamaño de píxel, 5×5nm.

En general, los resultados de esta experimentación no han sido muy exitosos, ya que no se ha podido generalizar más allá de los resultados entre *FIBSEM_EPFL* y *Lucchi++* (los cuales eran de esperar ya que los *datasets* son iguales a excepción del etiquetado).

Un análisis más detallado de las imágenes, con el tamaño de píxel ajustado, se muestra en la Figura 6.17, donde se puede apreciar que los *datasets* son bastante diferentes en cuanto a la tinción utilizada y el ME utilizado en la adquisición de las imágenes.

Finalmente, parece ser que las diferencias entre los *datasets* son una barrera difícil de superar hasta para los mejores modelos que se han conseguido.

6.4. Resumen de resultados

Configuración	***Dataset***	**Red**	**Mod. arqui. base**	***Data Augmen.***	**Descartes**	**Función de coste**	**Número de filtros**	***Learning rate***	***Batch size***	***Epochs***	**Media Jaccard test por trozo**	**Media Jaccard test por imagen**	**Media Jaccard por imagen + post-proc.**	**Media VOC por trozo**	**Media VOC por imagen**	**Media VOC por trozo + post-proc.**	**Media DET**	**Media DET + post-proc.**
Base	FIBSEM	U-Net	-	Keras	-	*Binary cross entropy*	16	0,001	1	230	0,8431 ± 0,0023	0,8403 ± 0,0023	-	0,9167 ± 0,0012	0,9153 ± 0,0012	-	0,8877 ± 0,0033	-
Base	FIBSEM	U-Net	-	Keras	-	*Binary cross entropy*	16	0,05	1	400	0,0053 ± 0,0167	-	-	0,4287 ± 0,1413	-	-	-	-
Base	FIBSEM	U-Net	-	Keras	-	*Binary cross entropy*	16	0,1	1	400	0,0053 ± 0,0167	-	-	0,4287 ± 0,1413	-	-	-	-
Troceado	FIBSEM	U-Net	-	Propio	-	*Jaccard loss*	16	0,001	6	360	0,3523 ± 0,3347	-	-	0,5791 ± 0,2292	-	-	-	-
Troceado	FIBSEM	U-Net	-	Keras	-	*Binary cross entropy*	16	0,0001	6	360	0,8130 ± 0,0002	-	-	0,9005 ± 0,0001	-	-	-	-
Troceado	FIBSEM	U-Net	-	Keras	-	*Binary cross entropy*	16	0,0005	6	360	0,8409 ± 0,0005	-	-	0,9154 ± 0,0002	-	-	-	-
Troceado	FIBSEM	U-Net	-	Keras	-	*Binary cross entropy*	16	0,01	6	360	0,8430 ± 0,0078	-		0,9167 ± 0,0041	-	-	-	-
Base + pesos en las clases (0,7;1,3)	FIBSEM	U-Net	-	Propio	-	*Binary cross entropy*	16	0,001	6	360	0,8448 ± 0,0029	-	-	0,9176 ± 0,0015	-	-	-	-
Troceado	FIBSEM	U-Net	-	Keras	-	*Binary cross entropy*	16	0,005	6	360	0,8473 ± 0,0023	-	-	0,9190 ± 0,0012	-	-	-	-
Troceado	FIBSEM	U-Net	-	Keras	-	*Binary cross entropy*	16	0,001	4	230	0,8487 ± 0,0031	-	-	0,9197 ± 0,0016	-	-	-	-
Troceado	FIBSEM	U-Net	ReLU	Propio	-	*Binary cross entropy*	16	0,001	6	360	0,8583 ± 0,0042	-	-	0,9248 ± 0,0022	-	-	-	-
Base	FIBSEM	U-Net	-	Keras	-	*Binary cross entropy*	16	0,01	1	360	0,8614 ± 0,0029	-	-	0,9265 ± 0,0015	-	-	-	-
Troceado	FIBSEM	U-Net	*Average pooling*	Propio	-	*Binary cross entropy*	16	0,001	6	360	0,8617 ± 0,0025	-	-	0,9266 ± 0,0013	-	-	-	-
Troceado	FIBSEM	U-Net	*Spatial dropout*	Propio	-	*Binary cross entropy*	16	0,001	6	360	0,8623 ± 0,0036	-	-	0,9269 ± 0,0019	-	-	-	-
Troceado	FIBSEM	U-Net	-	Propio	-	*Dice cross*	16	0,001	6	360	0,8624 ± 0,0023	-	-	0,9270 ± 0,0012	-	-	-	-

Configuración	***Dataset***	**Red**	**Mod. arqui. base**	***Data Augmen.***	**Descartes**	**Función de coste**	**Número de filtros**	***Learning rate***	***Batch size***	***Epochs***	**Media Jaccard test por trozo**	**Media Jaccard test por imagen**	**Media Jaccard por imagen + post-proc.**	**Media VOC por trozo**	**Media VOC por imagen**	**Media VOC por trozo + post-proc.**	**Media DET**	**Media DET + post-proc.**
Troceado	FIBSEM	U-Net	-	Propio	-	*Binary cross entropy*	16	0,001	6	360	0,8641 ± 0,0026	-	-	0,9279 ± 0,0013	-	-	-	-
Troceado	FIBSEM	U-Net	-	Propio	5 %	*Binary cross entropy*	16	0,001	1	360	0,8642 ± 0,0031	-	-	0,9280 ± 0,0016	-	-	-	-
Troceado	FIBSEM	U-Net	-	Propio (ela=0,4)	-	*Binary cross entropy*	16	0,001	6	360	0,8655 ± 0,0034	-	-	0,9287 ± 0,0017	-	-	-	-
Base	FIBSEM	U-Net	-	Propio (ela=0,7)	-	*Binary cross entropy*	16	0,01	1	360	0,8660 ± 0,0044	-	-	0,9289 ± 0,0022	-	-	-	-
Troceado	FIBSEM	ResUNet	-	Propio	-	*Binary cross entropy*	32	0,001	6	360	-	0,8577 ± 0,0014	0,8752 ± 0,0009	-	0,9246 ± 0,0007	0,9339 ± 0,0004	0,8980 ± 0,0045	0,9144 ± 0,0053
Troceado	FIBSEM	U-Net	-	Propio (ela=0,4)	-	*Binary cross entropy*	32	0,001	6	360	-	0,8661 ± 0,0038	0,8786 ± 0,0028	-	0,9291 ± 0,0020	0,9357 ± 0,0015	0,8955 ± 0,0064	0,9128 ± 0,0049
Troceado	FIBSEM	ResUNet	-	Keras	-	*Binary cross entropy*	32	0,001	6	360	0,8704 ± 0,0028	0,8679 ± 0,0020	0,8815 ± 0,0020	0,9313 ± 0,0014	0,9300 ± 0,0011	0,9372 ± 0,0011	-	-
Troceado	**FIBSEM**	**U-Net**	**-**	**Keras**	**-**	***Binary cross entropy***	**32**	**0,001**	**6**	**360**	**0,8735 ± 0,0016**	**0,8700 ± 0,0023**	**0,8826 ± 0,0013**	**0,9329 ± 0,0009**	**0,9312 ± 0,0012**	**0,9378 ± 0,0007**	**0,8980 ± 0,0037**	**0,9179 ± 0,0049**
Troceado	Lucchi++	U-Net	-	Keras	-	*Binary cross entropy*	32	0,001	6	360	0,8946 ± 0,0030	0,8927 ± 0,0037	0,9084 ± 0,0022	0,9424 ± 0,0020	0,9424 ± 0,0020	0,9508 ± 0,0011	0,8983 ± 0,0071	0,9351 ± 0,0036
Troceado	Kasthuri++	U-Net	-	Keras	-	*Binary cross entropy*	32	0,001	6	360	0,7764 ± 0,0032	0,9066 ± 0,0033	0,8926 ± 0,0022	0,9521 ± 0,0017	0,9521 ± 0,0017	0,9450 ± 0,0011	0,7242 ± 0,2414	0,7705 ± 0,0072
Troceado +Adam	Achucarro	U-Net	-	Keras	5 %	*Binary cross entropy*	16	0,0001	1	360	-	0,5998 ± 0,0182	0,6226 ± 0,0097	0,7892 ± 0,0099	0,7892 ± 0,0099	0,8018 ± 0,0052	0,5653 ± 0,0394	0,6936 ± 0,0252

6.5. Resultados en otros *datasets*

Configuración	***Dataset***	**Red**	**Mod. arqui. base**	***Data Augmen.***	**Descartes**	**Función de coste**	**Número de filtros**	***Learning rate***	***Batch size***	***Epochs***	**Media Jaccard test por trozo**	**Media Jaccard test por imagen**	**Media Jaccard por imagen + post-proc.**	**Media VOC por trozo**	**Media VOC por imagen**	**Media VOC por trozo + post-proc.**	**Media DET**	**Media DET + post-proc.**
Troceado	Lucchi++	U-Net	-	Keras	-	*Binary cross entropy*	32	0,001	6	360	0,8946 ± 0,0030	0,8927 ± 0,0037	0,9084 ± 0,0022	0,9424 ± 0,0020	0,9424 ± 0,0020	0,9508 ± 0,0011	0,8983 ± 0,0071	0,9351 ± 0,0036
Troceado	Kasthuri++	U-Net	-	Keras	-	*Binary cross entropy*	32	0,001	6	360	0,7764 ± 0,0032	0,9066 ± 0,0033	0,8926 ± 0,0022	0,9521 ± 0,0017	0,9521 ± 0,0017	0,9450 ± 0,0011	0,7242 ± 0,2414	0,7705 ± 0,0072
Troceado +Adam	Achucarro	U-Net	-	Keras	5 %	*Binary cross entropy*	16	0,0001	1	360	-	0,5998 ± 0,0182	0,6226 ± 0,0097	0,7892 ± 0,0099	0,7892 ± 0,0099	0,8018 ± 0,0052	0,5653 ± 0,0394	0,6936 ± 0,0252

Capítulo 7

Conclusiones

En este trabajo, se han obtenido resultados comparables al estado del arte, como se puede apreciar en las Tablas 7.1 y 7.2.

Trabajo	**Descripción**	**Jaccard**	**VOC**	**DET**
Lucchi 2013 [62]	*Working sets + Inference*	0,734*	0,867	-
Liu 2018 [68]	Mask R-CNN	0,849	-	-
Este trabajo	2D U-Net	0,870 ± 0,002	0,932 ± 0,000	0,898 ± 0,003
Este trabajo (max.)	2D U-Net	0,875	0,934	0,902
Casser [9]	2D U-Net	0,878	0,935	-
Este trabajo	2D U-Net+*post-processing*	0,882 ± 0,001	0,937 ± 0,000	0,917 ± 0,004
Este trabajo (max.)	2D U-Net+*post-processing*	0,885	0,939	0,920
Casser [9]	2D U-Net+*Z-filtering*	0,890	0,942	-
Lucchi 2015 [65]	Kernelized SSVM	0,895*	0,948	-
Xiao [70]	Residual 3D U-Net (16 DA)	0,900	-	-
Oztel [66]	**CNN + *post-processing***	**0,907**	**-**	**-**

Tabla 7.1: Resultados obtenidos junto con otros trabajos del estado del arte. Los marcados con un '*' quieren decir que no se especifica en esos trabajos el valor exacto obtenido, por ello se deducido una cota inferior de ese valor. Los marcados como "max" son los mejores resultados entre todos los *runs* realizados. Se ha resaltado en negrita el mejor de los valores.

A pesar de que los resultados no superen los mejores valores del estado del arte, todos los valores que se han mostrado por parte de este proyecto son totalmente veraces. Es decir, en los otros trabajos del estado del arte no se habla nunca de la cantidad de veces que se han repetido los experimentos (*runs*), mientras que, en el caso de este proyecto, se ha presentado el problema real que había desde un principio. Por ello, varios de los datos mostrados aquí han sido las medias de todos los valores obtenidos en todos los *runs* de cada experimento, y no únicamente el valor más alto. Sin embargo, para poder hacer

la comparación más fácil también se han añadido los valores más altos obtenidos entre todos los *runs* (los marcados como "max").

Además de la incertidumbre en los valores presentados en los trabajos del estado del arte, existe también la duda de como han realizado dicho cálculo, ya que rara vez se presenta el código con el que se han realizado las pruebas. En este proyecto se comentó en la sección 6.3.11 que el índice de Jaccard podía ser calculado como una medía de los valores obtenidos por cada trozo de la imagen, o bien con las imágenes enteras. En los trabajos del estado del arte no se trata este tema, por lo que no se sabe si los resultados que muestran se han realizado de una manera u otra. Esto es un detalle importante ya que, como se puede observar en la Tabla 6.12, hay bastante diferencia en los resultados obtenidos.

Trabajo	**Dataset**	**Método**	**Jaccard**	**VOC**	**DET**
Casser [9]	**FIBSEM_EPFL**				
		2D U-Net	0,878	0,935	-
		2D U-Net+*Z-filtering*	**0,890**	**0,942**	-
Este trabajo	**FIBSEM_EPFL**				
		2D U-Net	0,870 ± 0,002	0,932 ± 0,000	0,898 ± 0,003
		2D U-Net+*post-processing*	0,882 ± 0,001	0,937 ± 0,000	0,917 ± 0,004
Este trabajo (max.)	**FIBSEM_EPFL**				
		2D U-Net	0,875	0,934	0,902
		2D U-Net+*post-processing*	0,885	0,939	0,920
Casser [9]	**Lucchi++**				
		2D U-Net	0,888	0,940	-
		2D U-Net+*Z-filtering*	0,900	0,946	-
Este trabajo	**Lucchi++**				
		2D U-Net	0,893 ± 0,003	0,942 ± 0,002	0,898 ± 0,007
		2D U-Net+*post-processing*	0,908 ± 0,002	0,951 ± 0,001	0,935 ± 0,003
Este trabajo (max.)	**Lucchi++**				
		2D U-Net	0,899	0,946	0,908
		2D U-Net+*post-processing*	**0,912**	**0,953**	**0,943**
Casser [9]	**Kasthuri++**				
		2D U-Net	0,845	0,920	-
		2D U-Net+*Z-filtering*	0,846	0,920	-
Este trabajo	**Kasthuri++**				
		2D U-Net	0,907 ± 0,003	0,952 ± 0,001	0,724 ± 0,241
		2D U-Net+*post-processing*	0,893 ± 0,002	0,945 ± 0,001	0,771 ± 0,007
Este trabajo (max.)	**Kasthuri++**				
		2D U-Net	**0,910**	**0,954**	**0,805**
		2D U-Net+*post-processing*	0,894	0,946	0,770

Tabla 7.2: Resultados obtenidos en otros *datasets* junto con los resultados presentados por Casser *et al.* en [9]. Los marcados como "max" son los mejores resultados entre todos los *runs* realizados. Se han resaltado en negrita el mejor de los valores para cada *dataset*.

A pesar de que los resultados en *FIBSEM_EPFL* no superen los mejores obtenidos

en el estado del arte, sí que los supera cuando se trata de otros *datasets* como *Lucchi++* y *Kasthuri++*, haciendo uso de la misma red, como en el trabajo presentando en [9].

Por otro lado, una de las razones por las que introdujo la métrica DET fue para hacer ciertas comprobaciones en los experimentos con otros *datasets* (Sección 6.3.14) e *inter-dataset* (Sección 6.3.15). Se quería comprobar que a pesar de que la segmentación en ciertos casos no fuera buena, como lo fueron sobretodo en ciertos experimentos *inter-dataset*, quizás la detección de las mitocondrias que mide esta métrica DET no fuera tan mala. Se ha concluido finalmente que esta métrica parece seguir una correlación con el índice de Jaccard, dando buenos resultados únicamente cuando el índice de Jaccard es bueno también.

Por todo ello, resulta interesante el estudio completo realizado en este proyecto, donde tanto los resultados de los experimentos como el código desarrollado está plenamente disponible y accesible. De esta manera, se pueden reproducir los resultados aquí obtenidos de los casi 300 experimentos que se han realizado (sin olvidar los *runs* de cada uno de ellos).

7.1. Trabajo futuro

Hay varias posibilidades que estudiar en el trabajo futuro:

1. Con el DA se consiguió la mayor mejora en los resultados, por ello, suena interesante el poder estudiar más a fondo este campo, implementando quizás otras técnicas que mejoren la segmentación para este tipo de imágenes biomédicas. Además, ahora mismo el DA reemplaza lo datos de entrenamiento por imágenes transformadas, pero resulta interesante la idea de poder añadir estas nuevas imágenes en vez de reemplazarlas, de manera que se pueda aumentar el conjunto de datos y con ello la segmentación.

   Por otro lado, puede llegar a ser interesante el estudio de las *Generative Adversarial Networks* (GANs), presentadas en [78], para poder generar nuevas imágenes de DA, como se ha realizado en otros trabajos [79][80].

2. Otras pruebas que merecen estudiarse son las relacionadas con la inicialización de los pesos de la red. Hasta ahora la inicialización de los pesos han sido los de por defecto de Keras, que está basada en el *Xavier uniform initializer* presentado en [81], pero quizás haya otras posibles inicializaciones que ayuden a realizar una mejor segmentación.

3. Otro de los campos a explotar puede ser el cambiar el *threshold* con el que se binariza la imagen antes de utilizar el Jaccard. En este proyecto se ha establecido ese valor a 0,5, ya que es un valor muy usado en otros trabajos del estado del arte.

4. El post-procesado a mejorado la segmentación realizada menos para el caso de *Kasthuri++*, donde los valores han sido algo peores. Por ello, en un futuro se pretende estudiar más a fondo las razones por la que ha pasado esto y el estudio de otras técnicas de post-procesado.

5. Otra prueba que resulta interesante es el combinar los datos de diferentes *datasets* para realizar el entrenamiento de la red, de manera que haya más variedad de mitocondrias y ver el impacto que tiene esto en nuevos datos.

6. Los resultados obtenidos con el *dataset* de *Achucarro* no han sido muy buenos, ya que el número de imágenes disponibles hasta el momento no eran suficientes para poder entrenar la red. No obstante se seguirá colaborando con ellos para poder obtener mejores resultados e impulsar la obtención de nuevos datos de entrenamiento.

Bibliografía

[1] I. Goodfellow, Y. Bengio, and A. Courville, *Deep Learning*. MIT Press, 2016, http://www.deeplearningbook.org.

[2] S. B. Kotsiantis, "Supervised machine learning: A review of classification techniques," *Emerging Artificial Intelligence Applications in Computer Engineering: Real Word Ai Systems with Applications in Ehealth, Hci, Information Retrieval and Pervasive Technologies*, vol. 160, pp. 3–24, 2007, pT: S; NR: 91; TC: 213; J9: FRONT ARTIF INTEL AP; PG: 22; GA: BMA76; UT: WOS:000271690300002.

[3] "La célula. estructura y función." [Online]. Available: https://www.hiru.eus/es/biologia/la-celula-estructura-y-funcion

[4] A. Lucchi, K. Smith, R. Achanta, V. Lepetit, and P. Fua, "A fully automated approach to segmentation of irregularly shaped cellular structures in em images," in *Medical Image Computing and Computer-Assisted Intervention – MICCAI 2010*, T. Jiang, N. Navab, J. P. W. Pluim, and M. A. Viergever, Eds. Berlin, Heidelberg: Springer Berlin Heidelberg, 2010, pp. 463–471, iD: 10.1007/9783-642157455_57.

[5] A. C. Poole, R. E. Thomas, L. A. Andrews, H. M. McBride, A. J. Whitworth, and L. J. Pallanck, "The PINK1/Parkin pathway regulates mitochondrial morphology," *Proceedings of the National Academy of Sciences*, vol. 105, no. 5, pp. 1638–1643, 2008, pmid:18230723.

[6] S. Fulda, L. Galluzzi, and G. Kroemer, "Targeting mitochondria for cancer therapy," *Nature Reviews Drug Discovery*, vol. 9, no. 6, pp. 447–464, -06 2010. [Online]. Available: https://www.nature.com/articles/nrd3137

[7] M. B. de Moura, L. S. dos Santos, and B. V. Houten, "Mitochondrial dysfunction in neurodegenerative diseases and cancer," *Environmental and molecular mutagenesis*, vol. 51, no. 5, p. NA, Jun 2010. [Online]. Available: https://www.ncbi.nlm.nih.gov/pubmed/20544881

[8] D. C. Wallace, "Mitochondria and cancer," *Nature Reviews Cancer*, vol. 12, no. 10, p. 685, 2012.

[9] V. Casser, K. Kang, H. Pfister, and D. Haehn, "Fast mitochondria segmentation for connectomics," *arXiv preprint arXiv:1812.06024*, 2018.

[10] timvandevall, "Animal cell anatomy," Última visita: 22/02/2019. [Online]. Available: https://i0.wp.com/www.timvandevall.com/wp-content/uploads/blank-animal-cell-diagram.png?ssl=1

[11] Justin Johnson and A. Karpathy, "Stanford university cs231n: Convolutional neural networks for visual recognition," spring 2018. Última visita: 17/02/2019. [Online]. Available: http://cs231n.stanford.edu/

[12] J. Torres, *DEEP LEARNING Introducción práctica con Keras*, 3rd ed., Junio, 2019. [Online]. Available: https://github.com/jorditorresBCN/Deep-Learning-Introduccion-practica-con-Keras

[13] F. Rosenblatt, "The perceptron: A probabilistic model for information storage and organization in the brain," *Psychological Review*, vol. 65, no. 6, pp. 386–408, 1958.

[14] W. S. McCulloch and W. Pitts, "A logical calculus of the ideas immanent in nervous activity," *The Bulletin of mathematical biophysics*, vol. 5, no. 4, pp. 115–133, 1943.

[15] D.-A. Clevert, T. Unterthiner, and S. Hochreiter, "Fast and accurate deep network learning by exponential linear units (elus)," *arXiv preprint arXiv:1511.07289*, 2015.

[16] B. Fortuner, "Machine learning cheatsheet. loss functions." Última visita: 17/02/2019. [Online]. Available: https://ml-cheatsheet.readthedocs.io/en/latest/loss_functions.html

[17] P. Veličković, "Multilayer perceptron (mlp)," Última visita: 20/02/ 2019, Última visita: 18/02/2019. [Online]. Available: https://github.com/PetarV-/TikZ/tree/master/Multilayer%20perceptron

[18] G. Cybenko, "Approximation by superpositions of a sigmoidal function," *Mathematics of control, signals and systems*, vol. 2, no. 4, pp. 303–314, 1989.

[19] Q. Chen and N. Wong, "New simulation methodology of 3D surface roughness loss for interconnects modeling," in *Proceedings of the Conference on Design, Automation and Test in Europe*. European Design and Automation Association, 2009, pp. 1184–1189.

[20] S. V. Das and Sanjiv, *Deep Learning*. [Online]. Available: https://srdas.github.io/DLBook/

[21] M. A. Nielsen, *Neural networks and deep learning*. Determination press San Francisco, CA, USA:, 2015, vol. 25.

[22] B. T. Polyak, "Some methods of speeding up the convergence of iteration methods," *USSR Computational Mathematics and Mathematical Physics*, vol. 4, no. 5, pp. 1–17, 1964.

[23] I. Sutskever, J. Martens, G. Dahl, and G. Hinton, "On the importance of initialization and momentum in deep learning," in *International conference on machine learning*, 2013, pp. 1139–1147.

[24] R. A. Jacobs, "Increased rates of convergence through learning rate adaptation," *Neural Networks*, vol. 1, no. 4, pp. 295–307, 1988.

[25] J. Duchi, E. Hazan, and Y. Singer, "Adaptive subgradient methods for online learning and stochastic optimization," *Journal of Machine Learning Research*, vol. 12, no. Jul, pp. 2121–2159, 2011.

[26] T. Tieleman and G. Hinton, "Lecture 6.5-rmsprop: Divide the gradient by a running average of its recent magnitude," *COURSERA: Neural networks for machine learning*, vol. 4, no. 2, pp. 26–31, 2012.

[27] D. P. Kingma and J. Ba, "Adam: A method for stochastic optimization," *arXiv preprint arXiv:1412.6980*, 2014.

[28] V. Bushaev, "Adam — latest trends in deep learning optimization." -10-24T05:37:00.020Z 2018. [Online]. Available: https://towardsdatascience.com/adam-latest-trends-in-deep-learning-optimization-6be9a291375c

[29] D. E. Rumelhart, G. E. Hinton, and R. J. Williams, "Learning representations by back-propagating errors," *Cognitive modeling*, vol. 5, no. 3, p. 1, 1988.

[30] "Dogs vs. cats," Última visita: 26/02/2019. [Online]. Available: https://kaggle.com/c/dogs-vs-cats

[31] N. Srivastava, G. Hinton, A. Krizhevsky, I. Sutskever, and R. Salakhutdinov, "Dropout: a simple way to prevent neural networks from overfitting," *The journal of machine learning research*, vol. 15, no. 1, pp. 1929–1958, 2014.

[32] L. Breiman, "Bagging predictors," *Machine Learning*, vol. 24, no. 2, pp. 123–140, 1996.

[33] Y. LeCun, B. Boser, J. S. Denker, D. Henderson, R. E. Howard, W. Hubbard, and L. D. Jackel, "Backpropagation applied to handwritten zip code recognition," *Neural computation*, vol. 1, no. 4, pp. 541–551, 1989.

[34] A. Krizhevsky, G. Hinton *et al.*, "Learning multiple layers of features from tiny images," Citeseer, Tech. Rep., 2009.

[35] Y.-T. Zhou and R. Chellappa, "Computation of optical flow using a neural network," in *IEEE International Conference on Neural Networks*, vol. 1998, 1988, pp. 71–78.

[36] Y. Lecun, L. Bottou, Y. Bengio, and P. Haffner, "Gradient-based learning applied to document recognition," *Proceedings of the IEEE*, vol. 86, no. 11, pp. 2278–2324, NOV 1998, pT: J; NR: 121; TC: 6306; J9: P IEEE; PG: 47; GA: 131CC; UT: WOS:000076557300010.

[37] A. Krizhevsky, I. Sutskever, and G. E. Hinton, "Imagenet classification with deep convolutional neural networks," in *Advances in neural information processing systems*, 2012, pp. 1097–1105.

[38] O. Russakovsky, J. Deng, H. Su, J. Krause, S. Satheesh, S. Ma, Z. Huang, A. Karpathy, A. Khosla, M. Bernstein, A. C. Berg, and L. Fei-Fei, "ImageNet Large Scale Visual Recognition Challenge," *International Journal of Computer Vision (IJCV)*, vol. 115, no. 3, pp. 211–252, 2015.

[39] S. Herath, M. Harandi, and F. Porikli, "Going deeper into action recognition: A survey," *Image and Vision Computing*, vol. 60, pp. 4–21, 2017.

[40] S. Lowry, N. Sünderhauf, P. Newman, J. J. Leonard, D. Cox, P. Corke, and M. J. Milford, "Visual place recognition: A survey," *IEEE Transactions on Robotics*, vol. 32, no. 1, pp. 1–19, 2016.

[41] J. Deng, W. Dong, R. Socher, L.-J. Li, K. Li, and L. Fei-Fei, "Imagenet: A large-scale hierarchical image database," in *2009 IEEE International Conference on Computer Vision and pattern recognition*. Ieee, 2009, pp. 248–255.

[42] K. He, X. Zhang, S. Ren, and J. Sun, "Deep residual learning for image recognition," in *Proceedings of the IEEE Conference on Computer Vision and pattern recognition*, 2016, pp. 770–778.

[43] G. Litjens, T. Kooi, B. E. Bejnordi, A. A. A. Setio, F. Ciompi, M. Ghafoorian, J. A. Van Der Laak, B. V. Ginneken, and C. I. Sánchez, "A survey on deep learning in medical image analysis," *Medical image analysis*, vol. 42, pp. 60–88, 2017.

[44] A. Qayyum, S. M. Anwar, M. Majid, M. Awais, and M. Alnowami, "Medical image analysis using convolutional neural networks: A review," *arXiv preprint arXiv:1709.02250*, 2017.

[45] O. Ronneberger, P. Fischer, and T. Brox, "U-net: Convolutional networks for biomedical image segmentation," in *International Conference on Medical image computing and computer-assisted intervention*. Springer, 2015, pp. 234–241.

[46] I. Arganda-Carreras, S. C. Turaga, D. R. Berger, D. Cireşan, A. Giusti, L. M. Gambardella, J. Schmidhuber, D. Laptev, S. Dwivedi, J. M. Buhmann *et al.*, "Crowdsourcing the creation of image segmentation algorithms for connectomics," *Frontiers in neuroanatomy*, vol. 9, p. 142, 2015.

[47] Z. Zhou, M. M. R. Siddiquee, N. Tajbakhsh, and J. Liang, *Unet++ : A nested u-net architecture for medical image segmentation*, ser. Deep Learning in Medical Image Analysis and Multimodal Learning for Clinical Decision Support. Springer, 2018, pp. 3–11.

[48] W. Chen, Y. Zhang, J. He, Y. Qiao, Y. Chen, H. Shi, and X. Tang, "W-net: Bridged u-net for 2d medical image segmentation," *arXiv preprint arXiv:1807.04459*, 2018.

[49] G. Litjens, R. Toth, W. van de Ven, C. Hoeks, S. Kerkstra, B. van Ginneken, G. Vincent, G. Guillard, N. Birbeck, and J. Zhang, "Evaluation of prostate segmentation algorithms for mri: the promise12 challenge," *Medical image analysis*, vol. 18, no. 2, pp. 359–373, 2014.

[50] F. Milletari, N. Navab, and S.-A. Ahmadi, "V-net: Fully convolutional neural networks for volumetric medical image segmentation," in *2016 Fourth International Conference on 3D Vision (3DV)*. IEEE, 2016, pp. 565–571.

[51] Özgün Çiçek, A. Abdulkadir, S. S. Lienkamp, T. Brox, and O. Ronneberger, "3D U-Net: learning dense volumetric segmentation from sparse annotation," in *International conference on medical image computing and computer-assisted intervention*. Springer, 2016, pp. 424–432.

[52] S. Jégou, M. Drozdzal, D. Vazquez, A. Romero, and Y. Bengio, "The one hundred layers tiramisu: Fully convolutional densenets for semantic segmentation," in *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition Workshops*, 2017, pp. 11–19.

[53] G. Huang, Z. Liu, L. V. D. Maaten, and K. Q. Weinberger, "Densely connected convolutional networks," in *Proceedings of the IEEE Conference on Computer Vision and pattern recognition*, 2017, pp. 4700–4708.

[54] R. LaLonde and U. Bagci, "Capsules for object segmentation," *arXiv preprint arXiv:1804.04241*, 2018.

[55] S. Sabour, N. Frosst, and G. E. Hinton, "Dynamic routing between capsules," in *Advances in neural information processing systems*, 2017, pp. 3856–3866.

[56] R. Girshick, J. Donahue, T. Darrell, and J. Malik, "Rich feature hierarchies for accurate object detection and semantic segmentation," in *Proceedings of the IEEE Conference on Computer Vision and pattern recognition*, 2014, pp. 580–587.

[57] R. Girshick, "Fast r-cnn," in *Proceedings of the IEEE International Conference on Computer Vision*, 2015, pp. 1440–1448.

[58] S. Ren, K. He, R. Girshick, and J. Sun, "Faster r-cnn: Towards real-time object detection with region proposal networks," in *Advances in neural information processing systems*, 2015, pp. 91–99.

[59] K. He, G. Gkioxari, P. Dollár, and R. Girshick, "Mask r-cnn," in *Proceedings of the IEEE International Conference on Computer Vision*, 2017, pp. 2961–2969.

[60] A. Lucchi, K. Smith, R. Achanta, G. Knott, and P. Fua, "Supervoxel-based segmentation of mitochondria in em image stacks with learned shape features," *IEEE Transactions on Medical Imaging*, vol. 31, no. 2, pp. 474–486, 2012.

[61] A. Lucchi, Y. Li, K. Smith, and P. Fua, *Structured image segmentation using kernelized features*, ser. Computer Vision–ECCV 2012. Springer, 2012, pp. 400–413.

[62] A. Lucchi, Y. Li, and P. Fua, "Learning for structured prediction using approximate subgradient descent with working sets," in *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*, 2013, pp. 1987–1994.

[63] I. Tsochantaridis, T. Joachims, T. Hofmann, and Y. Altun, "Large margin methods for structured and interdependent output variables," *Journal of machine learning research*, vol. 6, no. Sep, pp. 1453–1484, 2005.

[64] A. Lucchi, C. Becker, P. M. Neila, and P. Fua, "Exploiting enclosing membranes and contextual cues for mitochondria segmentation," in *International Conference on Medical Image Computing and Computer-Assisted Intervention*. Springer, 2014, pp. 65–72.

[65] A. Lucchi, P. Márquez-Neila, C. Becker, Y. Li, K. Smith, G. Knott, and P. Fua, "Learning structured models for segmentation of 2-d and 3-d imagery," *IEEE Transactions on Medical Imaging*, vol. 34, no. 5, pp. 1096–1110, 2015.

[66] I. Oztel, G. Yolcu, I. Ersoy, T. White, and F. Bunyak, "Mitochondria segmentation in electron microscopy volumes using deep convolutional neural network," in *2017 IEEE International Conference on Bioinformatics and Biomedicine (BIBM)*. IEEE, 2017, pp. 1195–1200.

[67] S. Beucher, "Use of watersheds in contour detection," in *Proceedings of the International Workshop on Image Processing*. CCETT, 1979.

[68] J. Liu, W. Li, C. Xiao, B. Hong, Q. Xie, and H. Han, "Automatic detection and segmentation of mitochondria from sem images using deep neural network," in *2018 40th Annual International Conference of the IEEE Engineering in Medicine and Biology Society (EMBC)*. IEEE, 2018, pp. 628–631.

[69] T.-Y. Lin, P. Dollár, R. Girshick, K. He, B. Hariharan, and S. Belongie, "Feature pyramid networks for object detection," in *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*, 2017, pp. 2117–2125.

[70] C. Xiao, X. Chen, W. Li, L. Li, L. Wang, Q. Xie, and H. Han, "Automatic mitochondria segmentation for em data using a 3d supervised convolutional network," *Frontiers in Neuroanatomy*, vol. 12, 2018. [Online]. Available: https://www.frontiersin.org/articles/10.3389/fnana.2018.00092/full

[71] Y. LeCun, C. Cortes, and C. J. Burges, "Mnist handwritten digit database," 1998, Última visita: 24/02/2019. [Online]. Available: http://yann.lecun.com/exdb/mnist/

[72] N. Kasthuri, K. J. Hayworth, D. R. Berger, R. L. Schalek, J. A. Conchello, S. Knowles-Barley, D. Lee, A. Vázquez-Reina, V. Kaynig, and T. R. Jones, "Saturated reconstruction of a volume of neocortex," *Cell*, vol. 162, no. 3, pp. 648–661, 2015.

[73] P. Jaccard, "Étude comparative de la distribution florale dans une portion des alpes et des jura," *Bull Soc Vaudoise Sci Nat*, vol. 37, pp. 547–579, 1901.

[74] M. Everingham, L. VanGool, C. K. I. Williams, J. Winn, and A. Zisserman, "The pascal visual object classes (voc) challenge," *International Journal of Computer Vision*, vol. 88, no. 2, pp. 338„ jun 2010.

[75] P. Matula, M. Maška, D. V. Sorokin, P. Matula, C. O. de Solórzano, and M. Kozubek, "Cell tracking accuracy measurement based on comparison of acyclic oriented graphs," *PloS one*, vol. 10, no. 12, p. e0144959, 2015.

[76] J. Tompson, R. Goroshin, A. Jain, Y. LeCun, and C. Bregler, "Efficient object localization using convolutional networks," in *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*, 2015, pp. 648–656.

[77] "Satellite image segmentation: a workflow with u-net," -06-05T14:43:17.580Z 2018. [Online]. Available: https://medium.com/vooban-ai/satellite-image-segmentation-a-workflow-with-u-net-7ff992b2a56e

[78] I. Goodfellow, J. Pouget-Abadie, M. Mirza, B. Xu, D. Warde-Farley, S. Ozair, A. Courville, and Y. Bengio, "Generative adversarial nets," in *Advances in neural information processing systems*, 2014, pp. 2672–2680.

[79] A. Antoniou, A. Storkey, and H. Edwards, "Data augmentation generative adversarial networks," *arXiv preprint arXiv:1711.04340*, 2017.

[80] M. Frid-Adar, I. Diamant, E. Klang, M. Amitai, J. Goldberger, and H. Greenspan, "Gan-based synthetic medical image augmentation for increased cnn performance in liver lesion classification," *Neurocomputing*, vol. 321, pp. 321–331, 2018.

[81] X. Glorot and Y. Bengio, "Understanding the difficulty of training deep feedforward neural networks," in *Proceedings of the thirteenth international conference on artificial intelligence and statistics*, 2010, pp. 249–256.