

Escuela Técnica Superior de Ingeniería Universidad de Huelva

Grado en Ingeniería Informática

Trabajo Fin de Grado

Estudio de técnicas de *deep learning* con
aplicación a la identificación de fresas en
imágenes

Isaac Pérez Borrero
05/07/2017

*Dedicado a
mis padres*

Resumen

En este Trabajo de Fin de Grado se aplican técnicas de *deep learning* para la creación de un sistema de visión que identifique fresas y su estado de madurez en imágenes de plantaciones de fresa, con el objetivo de ser implementado en un sistema robótico para la recolección automática de fresas. Para ello, se realiza una primera introducción a esta tecnología, haciendo un recorrido por su historia y al estado del arte de la misma, abordando sus fundamentos y cómo se aplican al procesamiento de imágenes.

Para enfrentarnos a nuestro problema, dividimos nuestro sistema en tres etapas secuenciales, tratadas de forma individual: detección de fresas, segmentación de fresas y clasificación de fresas para determinar su madurez. En base a los resultados obtenidos, queda demostrada la viabilidad de este sistema para su implementación práctica.

Palabras clave: aprendizaje profundo, aprendizaje automático, inteligencia artificial, procesamiento de imágenes, redes neuronales convolucionales.

Abstract

In this Final Degree Project we apply deep learning techniques for the creation of a vision system which identifies strawberries and their state of maturity in pictures taken of strawberry plantations, with the objective of being implemented in a robotic system for automatic strawberry picking. To make this system, first we introduced this technology, making a journey through its history and its state of the art, tackling its fundaments and how they can be applied to image processing.

To deal with our problem, we divided our system into three sequential stages, dealt with individually: strawberry detection, strawberry segmentation and classification of strawberries to determine their maturity. Based on the results obtained, the feasibility of this system is demonstrated for its practical implementation.

Key words: *deep learning, machine learning, artificial intelligence, image processing, convolutional neural networks.*

Índice general

Índice general	v
Índice de figuras	vii
Índice de tablas	ix
<hr/>	
1 Propuesta del proyecto	1
1.1 Motivación	1
1.2 Objetivos	2
1.3 <i>Hardware y software</i>	2
2 Introducción al <i>deep learning</i>	5
2.1 Introducción	5
2.2 Antecedentes	10
2.3 Estado del arte	13
3 Fundamentos teóricos	19
3.1 Introducción a las redes neuronales convolucionales	19
3.2 Principales tipos de capas	22
3.2.1 Capa de convolución	22
3.2.2 Capa de activación	24
3.2.3 Capa de <i>pooling</i>	25
3.2.4 Capa de <i>upsampling o transposed convolution</i>	26
3.2.5 Capa de <i>fully connected</i>	27
3.2.6 Capa de <i>softmax</i>	28
3.2.7 Otros tipos de capas	28
3.3 Casos de estudio	29
3.3.1 LeNet	29
3.3.2 AlexNet	29
3.3.3 VGG	30
3.3.4 GoogleNet	31
3.3.5 ResNet	32
3.4 Desarrollo de un sistema basado en redes neuronales convolucionales	34
3.4.1 Datos necesarios para desarrollar el sistema	34
3.4.2 Inicialización de parámetros y transferencia de modelos	37
3.4.3 Métricas de evaluación	38
3.4.4 Cálculo del error	40
3.4.5 Entrenamiento de una red convolucional	41
3.4.6 Hiperparámetros y control del entrenamiento	47
3.4.7 Obtención del modelo para la etapa de inferencia	52
4 Marco de aplicación	53
4.1 Etiquetado en imágenes	53
4.2 Detección en imágenes	55
4.3 Segmentación en imágenes	60
5 Experimentación	67
5.1 Material	69

5.2 Metodología	74
5.2.1 Detección	74
5.2.2 Segmentación	83
5.2.3 Clasificación	89
5.3 Resultados	95
5.3.1 Detección	95
5.3.2 Segmentación	99
5.3.3 Clasificación	102
5.3.4 Análisis de los resultados	105
6 Conclusiones y trabajo futuro	107
Bibliografía	109

Apéndice

A Configuración del sistema	115
A.1 Especificaciones hardware	115
A.2 Especificaciones software	116

Índice de figuras

1.1 De izquierda a derecha: etiquetado, detección y segmentación	1
1.2 Principales frameworks para trabajar con <i>deep learning</i>	3
2.1 Conceptualización de un sistema basado en <i>machine learning</i>	6
2.2 Relación entre el <i>deep learning</i> , <i>machine learning</i> y la inteligencia artificial. .	7
2.3 Diferencias entre un sistema basado en <i>machine learning</i> y uno basado en <i>deep learning</i>	8
2.4 Principales arquitecturas para redes neuronales	9
2.5 Principales usos del <i>deep learning</i> en la industria	14
2.6 Algunos resultados de la red StackGAN	15
2.7 Resultado del entrenamiento de la red de Google para describir imágenes	15
2.8 Principales referentes en <i>deep learning</i>	17
2.9 Panorama general de la industria del <i>machine learning</i>	17
3.1 Estructura básica de una red neuronal.	19
3.2 Comparativa entre un sistema basado en red neuronal y sistema basado en red neuronal convolucional.	20
3.3 Ejemplos de características obtenidas en diferentes capas (de menos a más profundidad) de una red convolucional	21
3.4 Ejemplo de la operación de convolución aplicada sobre una imagen para obtener los bordes	22
3.5 Cálculo de la salida y el total de parámetros a aprender de una capa de convolución en función de la entrada y sus parámetros.	23
3.6 Módulo <i>inception</i>	24
3.7 Ejemplo de la operación <i>max-pooling</i>	26
3.8 Ejemplo de la operación de <i>transposed convolution</i>	27
3.9 Arquitectura de la red LeNet	29
3.10 Arquitectura de la red AlexNet	30
3.11 Arquitectura de la red VGG	31
3.12 Arquitectura de la red GoogleNet	31
3.13 Concepto <i>residual</i>	32
3.14 Arquitectura de la red ResNet	33
3.15 Problemas típicos en visión artificial	35
3.16 Ejemplos de situaciones problemáticas a las que nos enfrentamos	35
3.17 Resultado de la variación de umbrales	40
3.18 Representación gráfica de una función y su derivada	42
3.19 <i>Stochastic gradient descent</i> sin y con momento	44
3.20 Efecto del preprocesamiento en el entrenamiento	45
3.21 Efecto del preprocesamiento en los datos	46
3.22 Algoritmo <i>batch normalization</i>	46
3.23 Comparativa entre el número de parámetros, el número de capas y el ren- dimiento de la red	47
3.24 Comportamiento del error durante el entrenamiento según el <i>learning rate</i> elegido	48

3.25 Efecto sobre el error a largo plazo por el cambio del <i>learning rate</i>	49
3.26 Comparativa de modelo sin ajustar, ajustado correctamente y con sobreajuste	49
3.27 Detección del sobreajuste mediante el análisis de la función de error	50
3.28 Detección del sobreajuste mediante el análisis del <i>accuracy</i>	50
3.29 Técnica <i>early stopping</i>	51
3.30 Diferencias en la generalización según el mínimo elegido	51
 4.1 Red convolucional para la clasificación de dígitos	53
4.2 Ejemplo de funcionamiento de la capa <i>SPP</i>	54
4.3 Ejemplo de salida de un sistema de detección en imágenes	55
4.4 Ejemplo del algoritmo de la ventana deslizante	56
4.5 Distancias a comparar con δ	56
4.6 Condición para eliminar un recuadro contenido en uno mayor	57
4.7 Rectángulo englobado por otro mayor	57
4.8 Ejemplo de agrupamiento mediante el algoritmo <i>group rectangles</i>	57
4.9 Cálculo de la métrica <i>Intersection over Union (IOU)</i>	58
4.10 Esquema de la red DetectNet	59
4.11 Transformación de los datos para entrenar la red DetectNet	60
4.12 Salida de una red convolucional para la segmentación de imágenes	61
4.13 Ejemplo de red convolucional aplicada a la segmentación de imágenes	61
4.14 Transformación de una <i>convolutional neural network</i> a <i>fully convolutional network</i>	62
4.15 Cálculo del <i>offset</i> producido por las capas de una red	63
4.16 Aumento del detalle en la segmentación según la combinación de mapas de características	64
4.17 Ejemplo de imagen con su máscara para realizar el entrenamiento	64
 5.1 Sistema propuesto	68
5.2 Imágenes de muestra del conjunto original	70
5.3 Imágenes recortadas del conjunto original	71
5.4 Capturas de pantalla del programa utilizado para marcar las fresas	72
5.5 Ejemplo de creación del conjunto de datos para la detección	72
5.6 Ejemplo de creación del conjunto de datos para la segmentación	73
5.7 Ejemplo de creación del conjunto de datos para la clasificación	74
5.8 Gráfica de la evolución del entrenamiento de la red DetectNet	75
5.9 Estructura de la red DetectNet	78
5.10 Gráfica de la evolución del entrenamiento de la red propuesta	79
5.11 Estructura de la red propuesta para la detección	82
5.12 Gráfica de la evolución del entrenamiento de la red FCN-AlexNet	83
5.13 Estructura de la red FCN-AlexNet	85
5.14 Gráfica de la evolución del entrenamiento de la red propuesta para la segmentación	86
5.15 Estructura de la red propuesta para la segmentación	89
5.16 Gráfica de la evolución del entrenamiento de la red AlexNet	90
5.17 Estructura de la red propuesta para la clasificación	92
5.18 Gráfica de la evolución del entrenamiento de la red propuesta para la clasificación	93
5.19 Estructura de la red propuesta para la clasificación	94
5.20 Imágenes del conjunto de test para la detección	95
5.21 Resultados de la red DetectNet sobre el conjunto de test	96
5.22 Resultados de la red propuesta para la detección sobre el conjunto de test	97
5.23 Algunos filtros y características extraídas por la red	98
5.24 Imágenes del conjunto de test para la segmentación	99

5.25 Resultados de la red FCN-AlexNet sobre algunas imágenes	100
5.26 Resultados de la red propuesta para la segmentación sobre algunas imágenes	101
5.27 Algunos filtros y características extraídas por la red	102
5.28 Imágenes del conjunto de test para la clasificación	103
5.29 Clasificación realizada por la red sobre imágenes de test	103
5.30 Clasificación realizada por la red sobre imágenes de test	104
5.31 Algunos filtros y características extraídas por la red	105

Índice de tablas

3.1 Principales funciones de activación	25
3.2 Matriz de confusión	39
5.1 Matriz de confusión	96
5.2 Matriz de confusión.	97
5.3 Matriz de confusión.	103
5.4 Matriz de confusión.	104
5.5 Comparativa de las métricas para ambas redes en la detección.	105
5.6 Comparativa de las métricas para ambas redes en la segmentación.	105
5.7 Matriz de confusión de la red de referencia para la clasificación.	106
5.8 Matriz de confusión de la red propuesta para la clasificación.	106

CAPÍTULO 1

Propuesta del proyecto

A continuación se hace una presentación del proyecto, recogiendo la motivación detrás de este, los objetivos que lo impulsan y cómo se llevará a cabo.

1.1 Motivación

En los últimos años se ha vivido una nueva revolución en el campo de la informática, más concretamente en el campo de la inteligencia artificial, con el desarrollo del aprendizaje profundo, conocido por su nombre en inglés: *deep learning*. Esta disciplina se fundamenta principalmente en una técnica de inteligencia artificial conocida como redes neuronales, en base a esta técnica, se han desarrollado los algoritmos de *deep learning*.

Gracias a estos algoritmos y la ingente cantidad de datos que disponemos hoy día, junto con el avance de los sistemas hardware que hacen posible su uso más allá del estudio teórico, han permitido el comienzo de esta nueva disciplina, que promete traer consigo infinidad de mejoras en muchos campos relacionados con la inteligencia artificial, ofreciéndonos la posibilidad de abordar tareas que, hasta ahora, se habían reservado únicamente para los seres humanos.

Todo ello hace ver la potencialidad de este nuevo campo, que comienza a expandirse estos días y presenta una oportunidad única para adentrarse en él.

Con la intención de estudiar y conocer esta nueva rama de la inteligencia artificial nace este Trabajo de Fin de Grado. Debido a la multitud de campos de aplicación de esta tecnología, hemos decidido enfocar este trabajo en el análisis de imágenes y, más concretamente, el análisis de imágenes de plantaciones de fresa.

Dentro del análisis de imágenes hay diferentes tipos de problemas que se pueden abordar mediante inteligencia artificial, de forma genérica y de menor a mayor grado de complejidad, suele hablarse de etiquetado, detección y segmentación. En la figura 1.1 se ilustran los resultados de aplicar estas técnicas sobre una imagen de ejemplo.

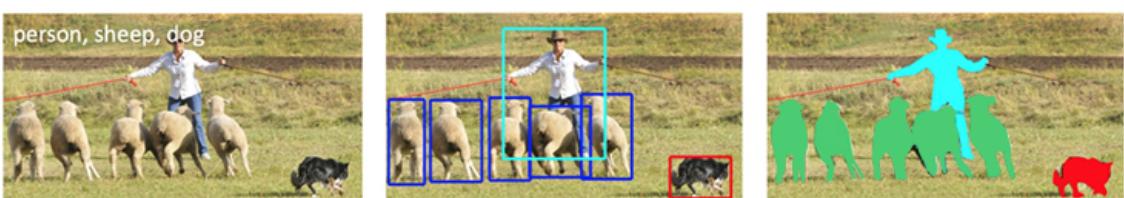


Figura 1.1: De izquierda a derecha: etiquetado, detección y segmentación¹.

¹Fuente: <https://code.facebook.com/posts/561187904071636>

1.2 Objetivos

Mediante la realización de este trabajo nos planteamos cumplir una serie de objetivos que, a continuación, se detallan.

- Objetivos generales:
 - **Introducción al *deep learning*:** realizaremos una primera introducción que nos permita comprender en qué consiste, estudiaremos cómo se ha llegado a su desarrollo y el estado actual de esta tecnología.
 - **Estudio teórico del *deep learning*:** para poder aplicarlo necesitaremos estudiar sus fundamentos teóricos además de las técnicas que podemos utilizar.
 - **Etiquetado en imágenes:** presentaremos una visión general de lo que consiste el etiquetado en imágenes y cómo se puede afrontar usando *deep learning*. Nuestro enfoque consistirá en construir una red que determine si las regiones de la imagen detectadas como fresa corresponden a una fresa madura, inmadura o no es una fresa.
 - **Detección en imágenes:** explicaremos en qué consiste la detección en imágenes y cómo se aplica el *deep learning* a este problema. Mediante el diseño de una red, detectaremos las zonas de la imagen donde hay fresas.
 - **Segmentación en imágenes:** se expondrá, de forma general, la segmentación en imágenes y cómo afrontarla aplicando *deep learning*. Con una red para la segmentación, pretendemos detectar los píxeles que corresponden a una fresa, de forma que pueda ser distinguida de todo aquello que no es fresa para poder determinar su madurez mediante la red que diseñemos para el etiquetado.
- Objetivos específicos:
 - Estudio de los conceptos básicos necesarios para trabajar con algoritmos de *deep learning*.
 - Diseño de un sistema hardware para trabajar con *deep learning*.
 - Obtención y preparación de los datos necesarios para los algoritmos a utilizar.
 - Estudio de los conceptos básicos necesarios para la creación de un sistema de etiquetado en imágenes basado en *deep learning*.
 - Diseño y evaluación del sistema de etiquetado de imágenes.
 - Estudio de los conceptos básicos necesarios para la creación de un sistema de etiquetado en imágenes basado en *deep learning*.
 - Diseño y evaluación del sistema de detección en imágenes.
 - Estudio de los conceptos básicos necesarios para la creación de un sistema de segmentación en imágenes basado en *deep learning*.
 - Diseño y evaluación de un sistema de segmentación en imágenes.

1.3 Hardware y software

Para poder abordar los objetivos descritos crearemos un equipo dotado con los componentes necesarios para satisfacer las necesidades de los algoritmos que vamos a utilizar.

Una de las mayores exigencias de estos algoritmos tiene que ver con la tarjeta gráfica, ya que requieren una gran cantidad de cálculos y, la mayoría de ellos, se pueden realizar de forma paralela; es por ello que decidimos diseñar el equipo en base a la tarjeta gráfica elegida, en nuestro caso dos NVIDIA GTX 1080. La descripción completa del sistema se puede encontrar en el apéndice A.1.

Una vez tenemos el equipo, necesitamos un software que nos ayude a desarrollar nuestros programas. A pesar de ser una tecnología reciente, existen multitud de *frameworks* de trabajo para el *deep learning* (ver figura 1.2).



Figura 1.2: Principales *frameworks* para trabajar con *deep learning*².

Estos *frameworks* tienen implementadas la mayoría de las funcionalidades necesarias para el desarrollo y evaluación de un sistema basado en *deep learning*.

A la hora de elegir la plataforma donde trabajaremos nos decantaremos por aquellas que tengan una mejor documentación y un buen respaldo por parte de la comunidad de usuarios, además de una buena velocidad de ejecución de los sistemas que desarrollemos. Un requisito adicional que buscamos es la posibilidad de obtener el sistema creado para llevarlo a la plataforma donde se realice el despliegue del mismo.

La opción que se ha elegido, Nvidia Digits [DIGITS], ofrece una capa más de abstracción sobre estos *frameworks*, lo que nos da mayor agilidad y nos facilita el trabajo. Nvidia Digits integra dos *frameworks* muy populares: Caffe [Caffe] y Torch [Torch]; en base a ellos diseñaremos nuestros sistemas.

Una vez diseñado un sistema con Nvidia Digits, este nos ofrece la posibilidad de exportarlo, lo que nos permite integrarlo en otras plataformas y *frameworks* para su uso. En nuestro caso, la opción más interesante es la posibilidad de exportarlo a TensorRT [TensorRT], que es una tecnología desarrollada por Nvidia para optimizar la ejecución de estos sistemas en la fase de despliegue.

Por tanto necesitaremos instalar en el equipo Nvidia Digits, Caffe, Torch y todas sus dependencias. En el apéndice A.2 se detalla el software que instalaremos en el sistema.

²Fuente: <https://developer.nvidia.com/deep-learning-software>

CAPÍTULO 2

Introducción al *deep learning*

Antes de comenzar a abordar la creación de sistemas basados en *deep learning*, en este capítulo se ofrece, en su primer punto, una introducción al *deep learning* que permite encuadrar esta disciplina como un campo específico del *machine learning*.

En el segundo y tercer punto de este capítulo se realiza un recorrido histórico y un repaso al estado actual de esta tecnología, así como a algunas de sus aplicaciones.

2.1 Introducción

Para que un ordenador realice una determinada tarea es necesario proporcionarle una serie de pasos ordenados que debe realizar para su resolución (algoritmo). Esto provoca que ante cualquier problema que nos planteemos afrontar mediante un ordenador tengamos que dar con el algoritmo que lo soluciona. Por lo tanto, necesitamos encontrar los pasos necesarios que debe realizar nuestro algoritmo. Sin embargo, no es algo trivial encontrar estos pasos para tareas complejas, como son, por ejemplo, identificar un objeto en una imagen, el reconocimiento de voz, conducción autónoma...

En consecuencia, para poder abordar determinados problemas es inviable describir a mano una serie de pasos que los solucionen, necesitamos de otro enfoque; es aquí cuando aparece el *machine learning* (aprendizaje automático). Mediante el aprendizaje automático pretendemos que sea el propio sistema el que encuentre los pasos que necesita nuestro algoritmo para resolver el problema.

Con el objetivo de comprender de qué trata el *machine learning*, necesitamos repasar algunos términos y conceptos previos, los cuales nos ayudaran a conocer mejor sus fundamentos.

Según el Diccionario de la Real Academia Española (DRAE), se define aprendizaje como «Adquisición por la práctica de una conducta duradera» [DRAE, Aprendizaje]. Los seres humanos somos capaces de realizar una determinada tarea gracias a un proceso de aprendizaje, mediante el cual interiorizamos cómo debemos resolverla. Esta metodología que usamos a la hora de enfrentarnos a determinados problemas es la base del *machine learning*.

Esta disciplina se compone de un conjunto de algoritmos que intentan abordar determinados problemas mediante un proceso de aprendizaje con un conjunto de datos. El objetivo será extraer el conocimiento que hay en el conjunto de datos, mediante la búsqueda de patrones en los mismos; esto nos permitirá generar un modelo que nos resuelva nuestro problema sin necesidad de indicarle explícitamente los pasos que debe seguir. Este modelo representa el algoritmo que estábamos buscando.

Esto hace que a la hora de hablar de *machine learning* se distingan diferentes etapas, de forma genérica se distinguen dos etapas: una primera de entrenamiento, donde se genera el modelo con el conjunto de datos de entrenamiento, y otra segunda etapa de inferencia, donde se usa el modelo obtenido en la etapa anterior, para someterlo a nuevos datos y obtener resultados. En la figura 2.1 se esquematiza estas etapas.

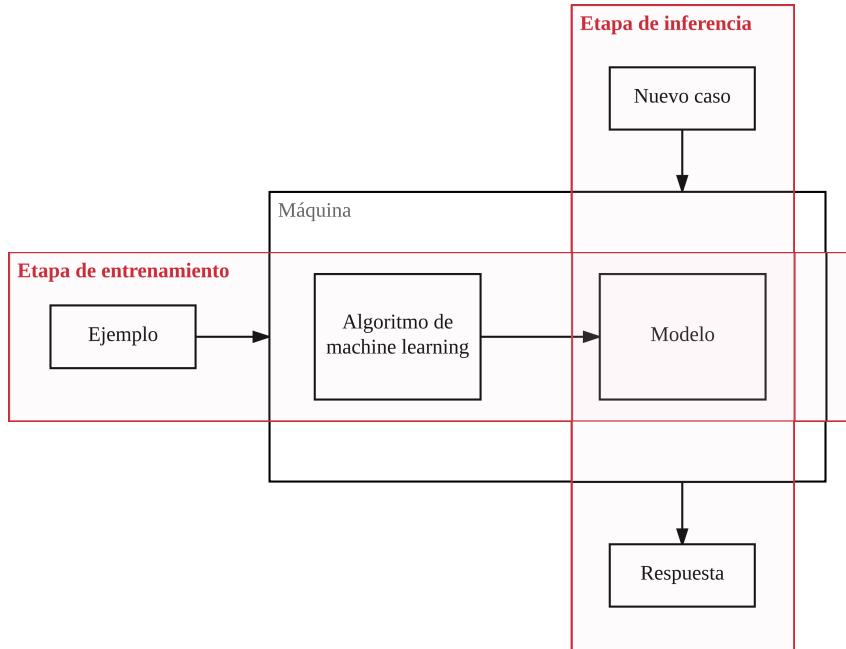


Figura 2.1: Conceptualización de un sistema basado en *machine learning*.

El aprendizaje automático en sí, es una disciplina que engloba a una gran variedad de algoritmos, muchos de ellos con diferentes propósitos. Según la forma de abordar el problema se distinguen tres tipos de aprendizaje:

- **Aprendizaje supervisado:** el conjunto de entrenamiento incluye la salida que debe dar el algoritmo.
- **Aprendizaje no supervisado:** el conjunto de entrenamiento no incluye la salida que debe dar el algoritmo.
- **Aprendizaje por refuerzo:** el conjunto de entrenamiento se convierte en estados en los que se encuentra el sistema y la forma de aprender se basa en analizar la bondad de las acciones tomadas en cada estado.

Otra clasificación de los algoritmos de *machine learning* es según el tipo de problema que se pretende abordar, lo que depende del tipo de la salida del algoritmo. Algunas de las tareas más comunes son las siguientes:

- **Clasificación:** tenemos dos o más clases y debemos decidir a cual de ellas pertenece el dato de entrada.
- **Análisis de la regresión:** pretende modelar la relación entre los datos de entrada.
- **Clustering (agrupamiento):** Consiste en dividir las entradas en agrupaciones que tengan características que les haga ser de un determinado grupo.

Como epílogo del *machine learning*, recogemos algunos de los algoritmos más relevantes:

- **Decision trees**: generan un modelo que representa mediante una estructura de árbol condiciones para los atributos de entradas que le permitan determinar la salida.
- **K nearest neighbors**: esta técnica permite clasificar un dato en base a la clase más frecuente a la que pertenecen sus K vecinos (datos) más cercanos.
- **Support Vector Machines**: el objetivo del *SVM* es generar un modelo que pueda mapear los datos de entrada a un espacio de características de una dimensión mayor, para luego encontrar el hiperplano que los separe y maximice el margen entre las clases, permitiendo así su clasificación.
- **Neural Networks**: generan el modelo mediante el uso de un grafo formado por el modelo matemático de las neuronas biológicas.

Todos estos algoritmos se engloban dentro del aprendizaje automático y, este, a su vez, se considera una rama de la inteligencia artificial. En la actualidad esta disciplina ha dado lugar a otras especialidades, como es el caso del *deep learning*. En la figura 2.2 se recoge la relación entre estas disciplinas.

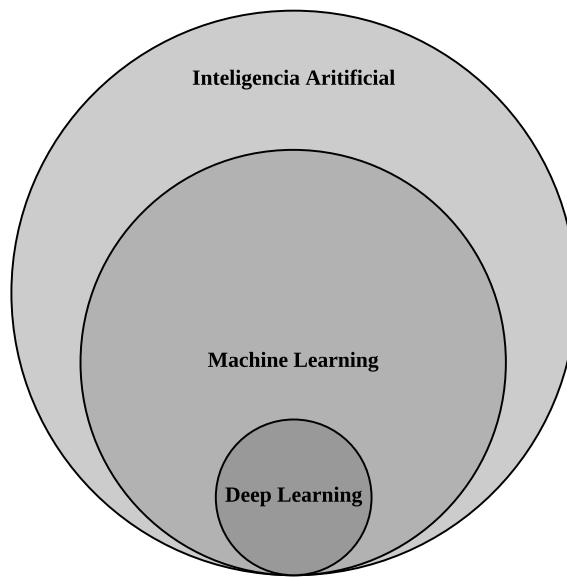


Figura 2.2: Relación entre el *deep learning*, *machine learning* y la inteligencia artificial.

El *deep learning* se está convirtiendo en toda una revolución por los resultados que está obteniendo en problemas donde otras técnicas de aprendizaje automático no conseguían buenos resultados.

Hasta ahora, los sistemas basados en *machine learning*, necesitaban unos datos de entrada para el sistema que no provenían directamente del problema, sino que se componían de una serie de características extraídas de los datos originales. Estas características se eligen en base a una decisión personal del experto que trabaja con el algoritmo y constituyen la base para la obtención del modelo; lo que obliga a elegir características lo suficientemente significativas para que ayuden, al algoritmo, a realizar su trabajo correctamente.

Debido a esto, los algoritmos de *machine learning* se veían limitados a trabajar sobre unas características prefijadas. Es aquí donde toma ventaja el *deep learning*, al pretender

que sea el propio algoritmo el que, suministrado con los datos originales del problema, sea capaz de encontrar las características más relevantes para resolverlo, sin requerir la intervención de un experto para que las elija. Para ello hace uso de una técnica de *machine learning*: las redes neuronales. Esta diferencia en el modo de funcionamiento entre el *machine learning* y el *deep learning* se esquematiza en la figura 2.3.

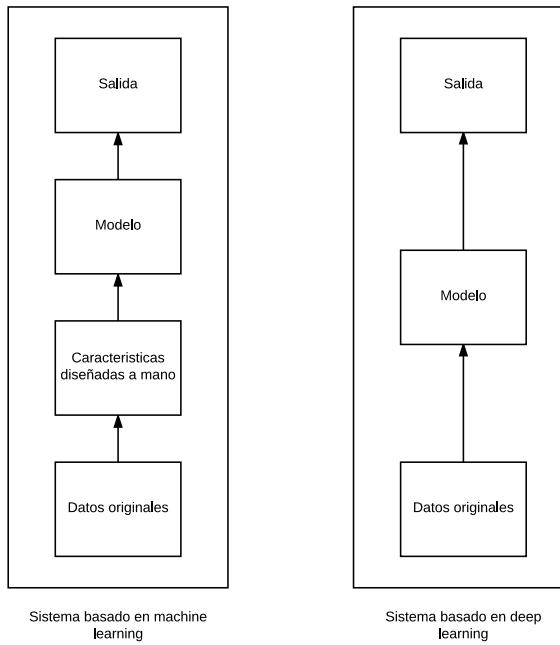


Figura 2.3: Diferencias entre un sistema basado en *machine learning* y uno basado en *deep learning*.

Los algoritmos basados en *deep learning* se caracterizan, dado la multitud de parámetros que tienen que ajustar, por necesitar una gran cantidad de datos de entrenamiento y un sistema con gran capacidad de procesamiento. Estas necesidades se están viendo resueltas, hoy día, gracias al abaratamiento que se está produciendo de los dispositivos de almacenamiento, lo que permite poder recoger más información durante más tiempo sin incrementar el coste. Este es el origen de otras disciplinas relacionadas con estos grandes volúmenes de datos: el *Big Data*. Y por otro lado, la mejora de la potencia de cómputo que están desarrollando las últimas *GPUs*¹ del mercado, sobre todo gracias al procesamiento paralelo que estas permiten.

Todo esto hace que el *deep learning* se esté explorando, hoy en día, para resolver infinidad de problemas inabordables o con resultados escasos hace apenas unos años.

Al igual que el *machine learning*, el *deep learning* agrupa a un conjunto de algoritmos, con diferentes enfoques y para diferentes problemas. Estos algoritmos se usan en campos muy variados, desde problemas relacionados con la visión artificial al procesamiento de audio, pasando por la medicina o la bolsa.

Como se mencionó anteriormente, estas técnicas se fundamentan en las redes neuronales, es por ello que a la hora de hablar de los algoritmos de *deep learning* también se habla de la arquitectura de los mismos, que es lo que permite diferenciarlos. La arquitectura se describe mediante las conexiones entre las diferentes capas de la red, el número de ellas y el tipo de neuronas que hay en cada capa. Esta arquitectura varía según el enfoque necesario para el problema a abordar; lo que provoca, debido a los múltiples problemas a los que se aplica, la aparición de una infinidad de arquitecturas (ver figura 2.4).

¹*Graphics Processor Unit*. Coprocesador dedicado al procesamiento de gráficos u operaciones de coma flotante.

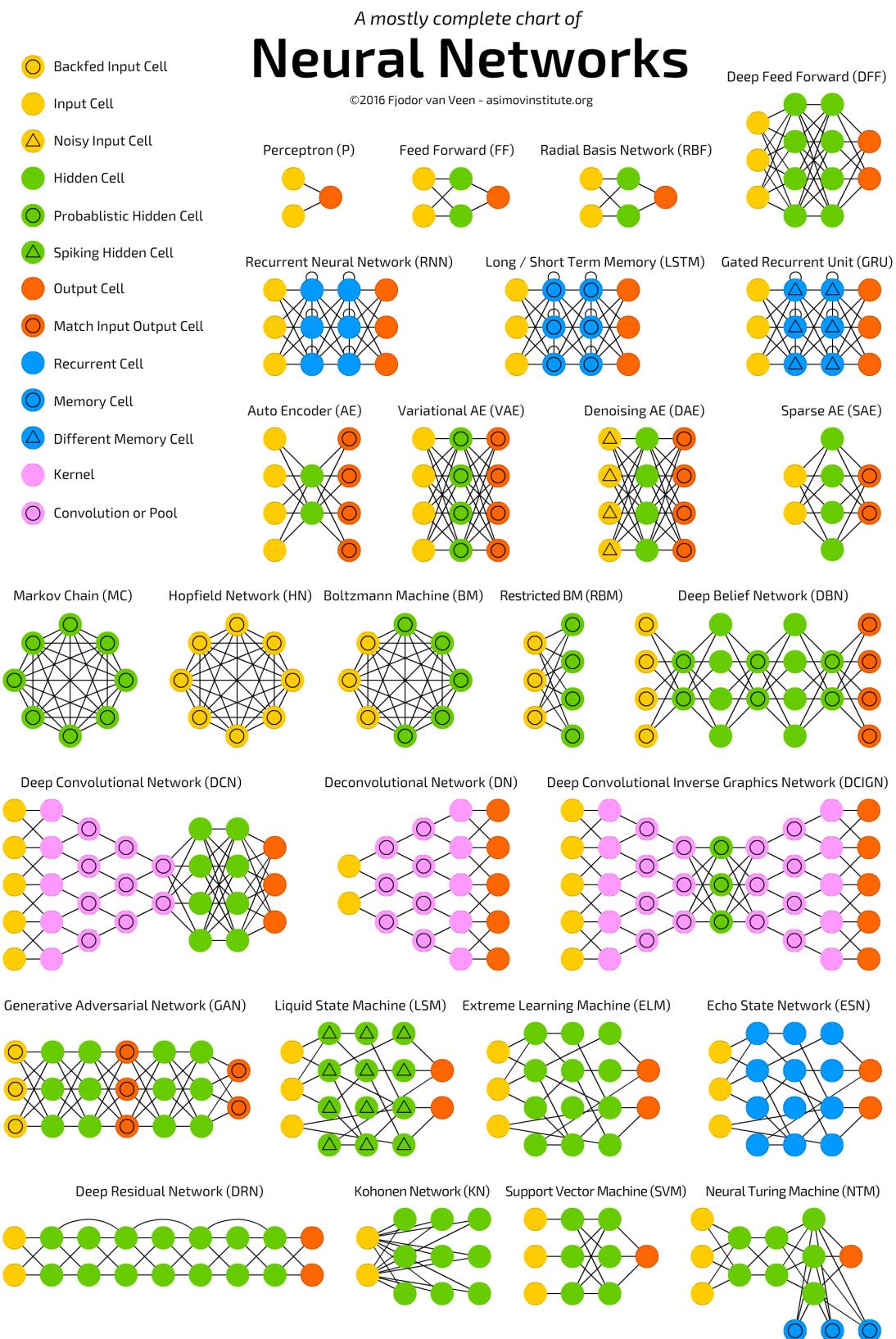


Figura 2.4: Principales arquitecturas para redes neuronales².

²Fuente: <http://www.asimovinstitute.org/wp-content/uploads/2016/09/neuralnetworks.png>

A continuación presentamos una clasificación más general de los diferentes algoritmos que hay en el *deep learning*:

- *Autoencoders*: son usados para problemas de aprendizaje no supervisado, se trata de una red entrenada con el objetivo de reproducir la entrada en la salida. Imponiendo restricciones a las capas internas de la red, como limitar el número de capas, podemos descubrir estructuras en los datos. Se suelen utilizar para la compresión de los datos o para el aprendizaje de características.
- *Restricted Boltzmann Machines* (RBM): consiste en una red neuronal con una probabilidad de activación en cada una de sus conexiones, se compone de una capa de unidades visibles y una capa de unidades ocultas. Son usadas en el aprendizaje no supervisado con el objetivo de encontrar relaciones ocultas entre diferentes elementos.
- *Sparse coding*: son utilizadas para aprender un conjunto de funciones básicas que describan los datos de entrada.
- *Convolutional Neural Networks* (CNN): estas redes usan las neuronas como campos receptivos, de manera similar a las neuronas biológicas de la corteza visual primaria. Son usadas sobretodo en problemas de visión artificial.

Dado que nuestro Trabajo de Fin de Grado esta enfocado a problemas relacionados con la visión, dentro de los diferentes algoritmos disponibles, utilizaremos las *convolutional neural networks* para afrontar los objetivos propuestos.

En la siguiente sección repasaremos la historia del *deep learning* y los avances que han permitido llegar hasta donde se encuentra en la actualidad.

2.2 Antecedentes

Hablar de la historia del *deep learning* implica hablar de la historia del *machine learning* y de la inteligencia artificial.

Uno de los mayores objetivos del ser humano ha sido siempre comprender el mundo y nuestra presencia en él. Desde la Antigua Grecia, multitud de pensadores afrontaron estas cuestiones, dando lugar a una nueva forma de pensamiento basada en la racionalidad, que dio origen a las matemáticas y la lógica.

Gracias al avance que producía la investigación en estos temas, se pudo plantear la construcción de máquinas que de una forma u otra mejoraban la vida de las personas, ya sea facilitando el trabajo o realizando alguna tarea por el ser humano.

Sin embargo, no es hasta la llegada de la máquina de vapor y, posteriormente, de la electricidad, junto con el avance en muchos otros campos, cuando se pudieron crear máquinas más sofisticadas que resolvían, cada vez, tareas más complejas.

Una de las tareas que requerían bastante complejidad para resolverlas con una máquina era el cálculo. Desde el ábaco se venía desarrollando máquinas para realizar los cálculos de forma más productiva, ya que era necesario hacerlos a mano. Fueron muchas las máquinas que se desarrollaron para mejorar el trabajo que realizaba el ábaco; cabe destacar la máquina analítica de Charles Babbage, desarrollada en 1816 con el objetivo de crear una máquina programable para hacer cualquier cálculo, y que es considerada como la primera computadora de uso general. Estas máquinas, aunque innovadoras, estaban siempre sujetas a la intervención de una persona y propensa a fallos.

Con la llegada del siglo XX, y gracias a los descubrimientos que se producen en matemáticas, lógica y los avances de la electrónica, que permiten diseñar componentes eléctricos que sustituyan a los mecánicos, asistimos a la creación de una de las máquinas más complejas diseñadas por el hombre: el ordenador. Por primera vez se podía realizar cálculos en base a una lista de instrucciones.

En 1936 Alan Turing publica un estudio [Turing, 1936] donde describe las máquinas de Turing, que formalizan el concepto de algoritmo, y que es considerado como el comienzo de la computación moderna.

Más tarde, en el año 1950, otra vez Alan Turing, publica un artículo en el que se hace una pregunta: *¿puede pensar una máquina?* [Turing, 1950]. En él presenta una prueba que permite determinar si una máquina es inteligente o no: el famoso test de Turing. Este artículo le valió para ser considerado como el padre de la inteligencia artificial.

Durante una conferencia en USA, en el año 1956, sobre informática teórica, es cuando surge, por primera vez, el término inteligencia artificial, referido a una disciplina que busca reproducir comportamiento inteligente con la ayuda de una máquina.

Dentro de este nuevo campo de estudio comenzaron a salir prometedoras investigaciones. En lo que afecta al *deep learning*, una de las investigaciones más trascendentales fue la presentación por parte de Warren McCulloch y Walter Pitts, en el año 1943, de un modelo matemático de las neuronas biológicas [McCulloch y Pitts, 1943], considerado como el primer trabajo en el campo de la inteligencia artificial, aun cuando no existía el término, y base de las redes neuronales artificiales.

Otro avance que permitió el desarrollo de las redes neuronales fue el trabajo realizado por Donald Hebb en 1949 que, por primera vez, lograba explicar los procesos de aprendizaje [Hebb, 1949] mediante una regla, conocida como regla de Hebb; esta regla describía cómo se actualizaban los pesos de las conexiones sinápticas entre neuronas.

Durante los años 50 se producen grandes avances en el campo de las redes neuronales. En el año 1959, Frank Rosenblatt desarrolla el *perceptron* [Rosenblatt, 1958], considerado como la primera red neuronal artificial con capacidad de aprender. Posteriormente, en el año 1960, se desarrolla el modelo Adaline [Widrow, 1960], siendo la primera red neuronal aplicada a un problema real, con la novedad de corregir los pesos durante el entrenamiento en base al error que cometía.

Igualmente, comienzan a aparecer algunos de los algoritmos más utilizados en *machine learning* y que permitieron trabajar con problemas de clasificación y reconocimiento de patrones. Cabe destacar el algoritmo *nearest neighbor* [Fix y Hodges, 1951] y el SVM [Vapnik y Lerner, 1963].

A pesar de que todos los avances que se producían hacían de las redes neuronales un campo muy prometedor, en el año 1969, Marvin Minsky y Seymour Papert presentan una investigación sobre redes neuronales en la que probaron que el *perceptron* no era capaz de resolver algunos tipos de problemas, como el aprendizaje de una función no-lineal [Minsky y Papert, 1969]. Esto supuso el desinterés por esta técnica, dado que las funciones no-lineales son extensamente empleadas en problemas de computación y del mundo real.

En la segunda mitad de la década de los 70 la inteligencia artificial sufrió su primer invierno. Diferentes agencias que financiaban la investigación en este campo cortan los fondos tras numerosos años de muchas expectativas y pocos avances.

Algunos investigadores siguieron interesados en el campo de las redes neuronales, como es el caso de Paul Werbos, que explicó en su tesis de 1974 el proceso de entrenamiento de una red neuronal artificial a través de la propagación hacia atrás de errores, conocido como el algoritmo de *backpropagation* [Werbos, 1974]. Este algoritmo es usado

hoy día para el entrenamiento de redes neuronales; sin embargo, pasó desapercibido para la mayoría de investigadores de la época, debido también al desinterés que originó el artículo de Minsky y Papert sobre el uso de estas técnicas.

Los años 80 estuvieron marcados por el nacimiento de los sistemas expertos, basados en reglas. Estos fueron rápidamente adoptados en el sector corporativo, lo que generó un nuevo interés en *machine learning*.

A su vez, los ordenadores siguen una evolución paralela, se pasa del uso del transistor a los circuitos integrados y se comienza a conseguir mejoras en la capacidad de cómputo. También comienzan a llegar nuevos lenguajes de programación, que hacen más productiva la programación y facilita el uso de los ordenadores. Con las mejoras en la capacidad de cómputo comienza a hablarse del procesamiento paralelo y, por supuesto, la llegada de internet facilita el intercambio de conocimiento.

Además de los avances que llegaban desde las matemáticas o la informática, diferentes estudios en el campo de la neurociencia permitieron desarrollar los algoritmos de *deep learning* que utilizamos hoy día, como es el caso del trabajo realizado por Hubel y Wiesel, que lograron descifrar el funcionamiento principal del sistema de visión del cerebro gracias a un estudio realizado sobre la corteza visual de los gatos [Hubel y Wiesel, 1959] y que les valió su reconocimiento a través de un premio Nobel.

Parece que durante los años 80 comienza a recuperarse el interés por las redes neuronales y comienzan a llegar nuevos estudios en este campo. Kunihiko Fukushima introduce el *Neocognitron* [Fukushima, 1980], una red neuronal que servirá de inspiración a las redes neuronales convolucionales. También llegarían las *Hopfield Networks* [Hopfield, 1982], que son las primeras redes neuronales recurrentes de la mano de John Hopfield y, posteriormente, las *Boltzmann Machine* [Ackey, et al., 1985], gracias al trabajo de David Ackley, Geoffrey Hinton y Terry Sejnowski. Es en esta época cuando se presenta, por primera vez, el aprendizaje por refuerzo a través de la tesis de Christopher Watkins [Watkins, 1989].

Pero sin duda lo que propició un gran avance para el *deep learning* fue el redescubrimiento del algoritmo *backpropagation* gracias al artículo publicado por David Rumelhart, Ronald Williams y Geoffrey Hinton [Rumelhart, et al., 1986] en 1986, en el que mostraban cómo una red neuronal con muchas capas ocultas podía ser entrenadas de forma eficaz con este método y, a su vez, le permitía aprender funciones no lineales, superando la limitación que tenía el *perceptron*.

La primera aplicación práctica del *backpropagation* vino gracias al trabajo de Yann LeCun en 1989 [LeCun, et al., 1989], en el que presentó una red llamada *LeNet* para la clasificación de dígitos escritos a mano.

Con la llegada de los 90 el trabajo en *machine learning* se pasó de un enfoque orientado al conocimiento a uno orientado a los datos. Los investigadores comenzaron a crear programas para analizar grandes cantidades de datos y extraer conclusiones de los mismos.

A pesar de los buenos resultados de las redes neuronales, la aparición de otras técnicas como las máquinas de vector de soporte o los árboles de decisión, que conseguían resultados similares y con menor complejidad, provocó que fueran perdiendo popularidad.

No por ello se abandonó la investigación en redes neuronales; en 1997, Sepp Hochreiter y Jürgen Schmidhuber presentan las LSTM (*long-short term memory recurrent neural networks*) [Hochreiter y Schmidhuber, 1997], que supuso una mejora en la eficiencia y la factibilidad de las *recurrent neural networks*.

Para esta fecha la inteligencia artificial comenzaba a mostrar resultados esperanzadores. En 1997 el ordenador *Deep Blue* de IBM vence al campeón mundial de ajedrez Gary Kasparov.

El trabajo que Yann LeCun empezó en 1989 continuó mejorándose hasta dar, en 1998, con la primera red neuronal convolucional [LeCun, et al., 1998] de la historia. Este tipo de red fundó la base del *deep learning* aplicado al tratamiento de imágenes.

El continuo aumento de la potencia de cálculo junto con la gran abundancia de datos disponibles realimentan el interés por el *machine learning* y comienza a hacer viable el entrenamiento de redes neuronales más profundas.

En 2006, Geoffrey Hinton acuña, por primera vez, el término *deep learning* e introduce las *deep belief networks* [Hinton, et al., 2006], además de ofrecer una forma efectiva de entrenar redes mucho más profundas que las que se habían utilizado hasta ahora, dando comienzo a la era del *deep learning*.

Un hecho que acabó por popularizar el uso del *deep learning* fue la presentación, por parte de unos estudiantes de doctorado bajo la supervisión de Geoffrey Hinton, de una red neuronal convolucional llamada Alexnet [Krizhevsky, et al., 2012] para la competición Imagenet [Deng, et al., 2009] (consiste en el etiquetado de imágenes con mil clases y millones de imágenes de entrenamiento), y en la que lograron bajar el error que se producía usando técnicas tradicionales del 26 % al 16 %. Esto provocó un gran interés por esta tecnología por parte de investigadores y grandes compañías, que pronto comenzaron a aplicarlo a diferentes campos, además de la visión.

Actualmente se ha bajado del considerado error humano [Russakovsky, et al., 2015] (5 %) en ImageNet, gracias a la red Resnet, presentada por el equipo de investigación de Microsoft [He, et al., 2015].

En la siguiente sección repasaremos el estado actual del *deep learning* y cómo puede llegar a cambiar nuestras vidas.

2.3 Estado del arte

Con la irrupción de la red convolucional Alexnet, comenzó una carrera por la creación de nuevas redes que mejorasen los resultados de esta. La competición ImageNet sigue desarrollándose, en la actualidad, con un error situado por debajo del 3 %.

Los buenos resultados cosechados por este tipo de redes dieron lugar a nuevas competiciones que buscaban poner a prueba el desempeño de estas redes en tareas más exigentes, como es el caso de la competición Microsoft COCO [Lin, et al., 2014], en la que, a pesar de su novedad, se están consiguiendo resultados muy prometedores; prueba de ello es la red ganadora de la competición en 2016, diseñada por un equipo de investigadores de Microsoft Asia y la Universidad de Tsinghua [Li, et al., 2016].

El uso de las redes convolucionales están permitiendo realizar tareas relacionadas con la visión donde éstas obtienen mejores resultados que los expertos humanos; ejemplo de ello es la detección del tipo de cáncer que posee una persona en base al análisis de imágenes de células extraídas del paciente [Liu, et al., 2017].

En la actualidad, los algoritmos de *deep learning* están demostrando su utilidad en una gran variedad de situaciones. En el campo de la traducción cabe destacar el nuevo sistema de Google, *Google's Neural Machine Translation* [Wu, et al., 2016]. Según Google, con la llegada del *deep learning* han conseguido mejorar su traductor más en un año que en los últimos diez juntos. Otro ejemplo de la versatilidad del *deep learning* es la transformación de texto en habla [Arik, et al., 2017]. Podemos encontrar multitud de ejemplos de aplicación, en la figura 2.5 se recoge algunos de los usos más comunes del *deep learning* en la industria.

General use case	Industry
Sound	
Voice recognition	UX/UI, Automotive, Security, IoT
Voice search	Handset maker, Telecoms
Sentiment analysis	CRM
Flaw detection (engine noise)	Automotive, Aviation
Fraud detection (latent audio artifacts)	Finance, Credit Cards
Time Series	
Log analysis/Risk detection	Data centers, Security, Finance
Enterprise resource planning	Manufacturing, Auto., Supply chain
Predictive analysis using sensor data	IoT, Smart home, Hardware manufact.
Business and Economic analytics	Finance, Accounting, Government
Recommendation engine	E-commerce, Media, Social Networks
Text	
Sentiment Analysis	CRM, Social media, Reputation mgt.
Augmented search, Theme detection	Finance
Threat detection	Social media, Govt.
Fraud detection	Insurance, Finance
Image	
Facial recognition	
Image search	Social media
Machine vision	Automotive, aviation
Photo clustering	Telecom, Handset makers
Video	
Motion detection	Gaming, UX, UI
Real-time threat detection	Security, Airports

Figura 2.5: Principales usos del *deep learning* en la industria³.

Una de las novedades más interesantes con respecto al *deep learning* son las *generative adversarial networks* [Goodfellow et al., 2014]. Según Yann LeCun, se trata de la mayor innovación en diez años en el *machine learning*. Consiste en el uso de dos redes neuronales que compiten una contra otra; una de ellas, llamada la red discriminadora, es la encargada de averiguar si su entrada pertenece a un dato real del conjunto de datos o a la salida de la otra red, llamada generadora; esta red se encarga de, a partir de una entrada aleatoria, generar una salida de forma que a la red discriminadora le sea imposible distinguir si procede del conjunto de datos o de la red generadora. Con esta técnica se pueden conseguir resultados sorprendentes; a modo de ejemplo, en la figura 2.6 se muestran los obtenidos usando la red StackGAN [Zhang et al., 2016], que es capaz de generar imágenes en base a texto.

³Fuente: https://deeplearning4j.org/img/use_case_industries.png

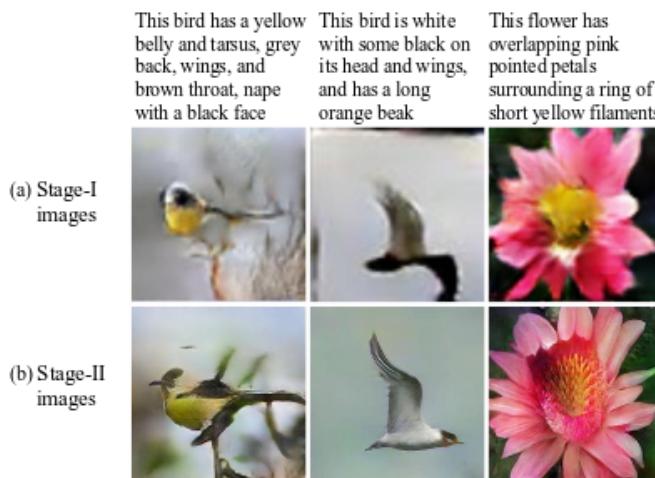


Figura 2.6: Algunos resultados de la red StackGAN⁴.

Otro ámbito de estudio dentro del *deep learning* tiene que ver con el uso de redes neuronales recurrentes; estas redes tienen un comportamiento dinámico, ya que pueden trabajar con secuencias de datos al disponer de una realimentación, la cual también las dota de una memoria interna. Esto posibilita su uso para el reconocimiento de patrones en el tiempo o incluso la predicción, algo aprovechado para el desarrollo de sistemas que trabajan con la bolsa de valores. Un ejemplo de aplicación de las redes recurrentes es la descripción textual de una imagen; en la figura 2.7 se muestran los resultados de la red desarrollada por el equipo de investigación de Google para esta tarea [Vinyals et al., 2016].

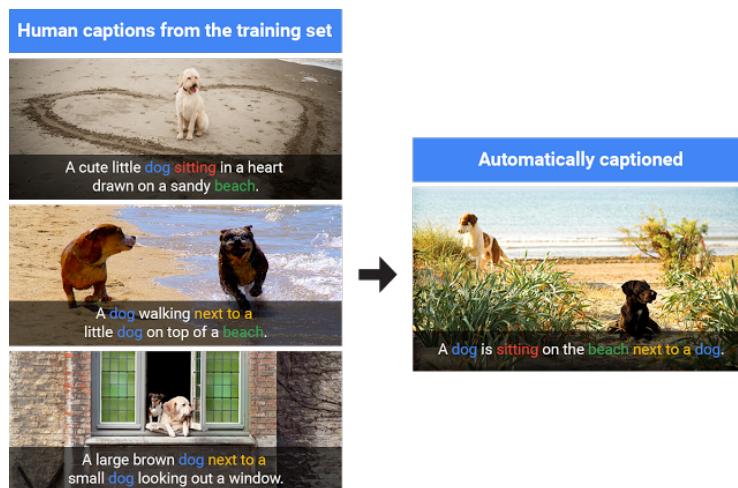


Figura 2.7: Resultado del entrenamiento de la red de Google para describir imágenes⁵.

El principal problema de las redes neuronales recurrentes es que solo pueden recordar cosas durante un periodo de tiempo limitado (se encuentra su equivalencia con la corteza cerebral donde reside la memoria de trabajo [Baddeley, 1974]. Con la que solo memorizamos cosas de forma temporal). Esto propició el interés por el desarrollo de redes que tuviesen un «hipocampo» como módulo separado de memoria que les permitiese almacenar información durante más tiempo. Un ejemplo de red bajo este enfoque son las *long short-term memory* (LSTM). La investigación en base a este planteamiento dio origen, en 2014, a las *neural turing machines* [Graves, et al., 2014], utilizando una red neuronal y un banco de memoria, basa su funcionamiento en usar a la red neuronal como controladora

⁴Fuente: <https://arxiv.org/pdf/1612.03242.pdf>

⁵Fuente: <https://research.googleblog.com/2016/09/show-and-tell-image-captioning-open.html>

de la memoria. Con el uso de este tipo de redes se pretende utilizar la memoria para almacenar detalles sutiles y utilizar la red neuronal que la controla para que se concentre en aprender las regularidades globales. Una investigación posterior en este campo dio lugar en 2016 a las *differentiable neural computer* (DNC) [Graves, et al., 2016], una extensión de las *neural turing machines*; según sus autores, se demuestra que son capaces de resolver tareas complejas y estructuradas, las cuales eran inaccesibles para las redes neuronales sin una memoria externa de lectura-escritura. Tomando el ejemplo de aplicación que muestran los investigadores en su artículo, la DNC se puede usar para que aprenda a navegar en una amplia variedad de sistemas de metros y después utilizar lo que ha aprendido para navegar en el metro de Londres, algo impensable para una red neuronal que no disponga de memoria, ya que tendría que aprender cada sistema de metro desde cero.

Quizás los algoritmos de *deep learning* más interesantes son los relacionados con el uso de aprendizaje por refuerzo y el aprendizaje no supervisado, debido a la utilidad práctica de los mismos. Geoffrey Hinton es uno de los muchos investigadores que son partidarios de este enfoque. Según Hinton: “*the brain has about 10^{14} synapses and we only live for about 10^9 seconds. So we have a lot more parameters than data. This motivates the idea that we must do a lot of unsupervised learning since the perceptual input (including proprioception) is the only place we can get 10^5 dimensions of constraint per second*”. También Alan Turing parecía ver útil aplicar estos tipos de aprendizaje: “*instead of trying to produce a programme to simulate the adult mind, why not rather try to produce one which simulates the child's? If this were then subjected to an appropriate course of education one would obtain the adult brain*”.

Mediante la combinación del *deep learning* con el aprendizaje por refuerzo, en lo que se conoce como *deep reinforcement learnig*, empresas como DeepMind, que ya demostraron ser capaces de entrenar un sistema con *deep reinforcement learnig* para jugar al Atari [Mnih et al., 2013], han desarrollado algoritmos más complejos capaces de ganar al mejor jugador de Go [Silver et al., 2016], enfocándose hoy día en entornos más exigentes como StarCraft [DeepMind, 2016]. Con esta metodología se facilita el entrenamiento de redes para desarrollar agentes que aprendan interactuando con su entorno sin intervención humana, a la vez que aprovechan la potencia del *deep learning*.

Lo que sin duda está suponiendo un beneficio para el desarrollo de esta tecnología es la mentalidad abierta que están teniendo las empresas y los investigadores, que ponen a disposición de la comunidad los avances que obtienen, de forma que otros investigadores o desarrolladores puedan beneficiarse de ello y seguir ayudando a mejorarla.

No solo se está liberando los avances teóricos, cada vez existen más *frameworks* de *deep learning* que permiten utilizar la mayoría de novedades que se producen de forma fácil y gratuita, permitiendo la participación de la comunidad de usuarios en la mejora de los mismos. Prueba de esta filosofía es la liberación, por parte de la empresa OpenAI [OpenAI, 2017] y DeepMind [Beattie, et al., 2016], de diferentes softwares utilizados por la industria para el entrenamiento de las redes en entornos simulados.

Dentro de la multitud de investigadores trabajando en este campo, podemos destacar algunos personajes reconocidos. En la figura 2.8, se encuentran de izquierda a derecha: Yann LeCun, creador de las redes neuronales convolucionales, fundador y director del New York University Center for Data Science y director de Facebook AI Research; Geoffrey Hinton, considerado como uno de los padres del *deep learning*, es profesor de la Universidad de Toronto y forma parte del equipo de investigación de Google, es miembro de la Royal Society y ha recibido numerosos reconocimientos internacionales por su trayectoria; Yoshua Bengio, profesor de la Universidad de Montréal y co-director del Canadian Institute for Advanced Research, cuenta con multitud de publicaciones y es una persona destacada dentro del *deep learning*; Andrew Ng, profesor en la Universidad de Stanford y

director de su laboratorio de inteligencia artificial, es co-fundador de Coursera, una plataforma de cursos online, y uno de los desarrolladores de ROS, un sistema operativo para robots, fue el fundador de Google Brain en Google, encargada de la investigación en *deep learning* en la compañía y, hasta marzo de 2017, era el científico jefe en Baidu Research.



Figura 2.8: Principales referentes en *deep learning*⁶.

A modo de resumen, en la figura 2.9 se recoge el panorama del *machine learning* en la actualidad, donde se aprecia cómo la implantación de esta tecnología está dando lugar a un gran ecosistema de empresas que, de una forma u otra, están participando en su desarrollo. Algunas de las empresas más destacadas en la actualidad por su implicación con el *deep learning* son: NVIDIA, la cual esta aprovechando su posición en el campo de las GPUs para aplicarla a esta tecnología; DeepMind, que fue adquirida por Google, donde realizan investigaciones pioneras; y cabe destacar OpenIA, una empresa sin ánimo de lucro fundada por Elon Musk y otros líderes de la industria con el objetivo de desarrollar una inteligencia artificial para beneficio de la humanidad.

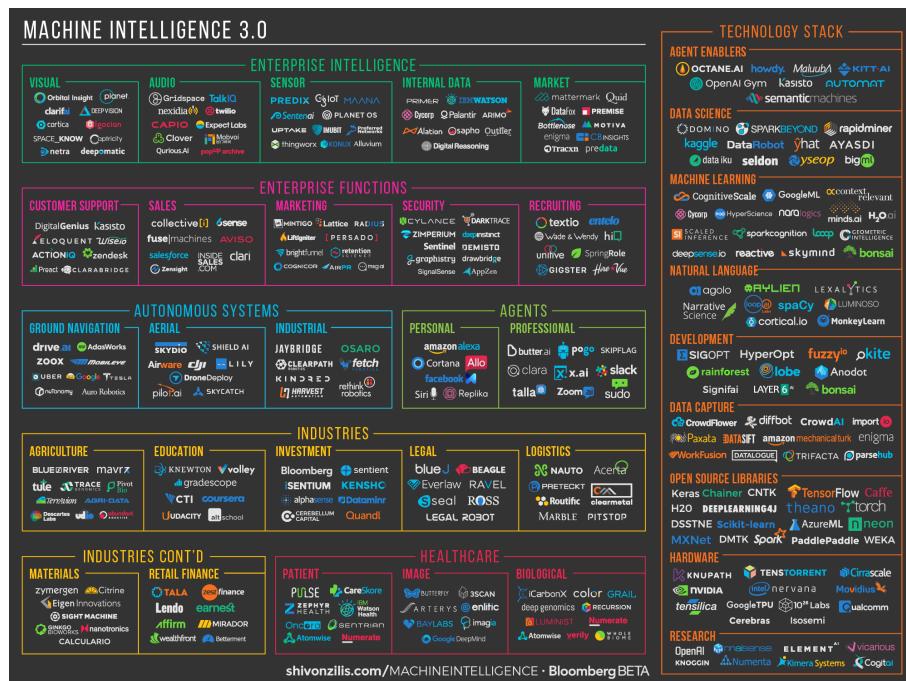


Figura 2.9: Panorama general de la industria del *machine learning*⁷.

⁶Fuente: <http://www.kdnuggets.com/wp-content/uploads/photo.jpg>

El desarrollo en la industria *hardware* está permitiendo el uso de estos algoritmos en casi cualquier escenario, gracias al bajo consumo de los componentes y la elevada potencia de cómputo. Empresas como NVIDIA diseñan sistemas embebidos idóneos para el *deep learning*, como la Jetson TX2 [NVIDIA Jetson TX2]; lo cual abre la puerta a la creación de robots y sistemas autónomos equipados con esta tecnología. De lo pequeño a lo grande, por parte también NVIDIA se están desarrollando superordenadores equipados con multitud de GPUs que les permiten realizar investigaciones complejas con grandes cantidades de datos utilizando estos algoritmos [NVIDIA DGX-1].

Tendremos que esperar en los próximos años para recoger los frutos de todos estos avances y ver a qué llegan otros como el *one-shot learning* [Santoro et al., 2016], que pretende realizar el aprendizaje en base a uno o unos pocos ejemplos; o las nuevas investigaciones para acelerar el entrenamiento de las redes, Google ha presentado una forma de realizar entrenamientos super rápidos y altamente precisos gracias a lo que llaman *Mixture of Experts Layers* [Shazeer et al., 2017]; para ello, proponen mantener múltiples «expertos» dentro de la red, siendo cada uno una red neuronal. Otra investigación interesante es la centrada en la transferencia de conocimiento, conocido como *transfer learning*. Busca utilizar lo aprendido en otras tareas para realizar una nueva sin partir desde cero; dentro de esta investigación, DeepMind ha presentado PathNet [Fernando et al., 2017], una red de redes con la que pretenden crear una inteligencia artificial genérica basada en este principio. La conducción autónoma o el trabajo en medicina son algunos de los campos donde hay puestas más esperanzas de que se puedan realizar avances sorprendentes.

El *deep learning* abre la puerta a la automatización de multitud de puestos de trabajo, lo que conlleva a una pérdida de los mismos. Existe un debate abierto sobre cómo legislar el uso de la inteligencia artificial y cómo puede afectar a la sociedad su implantación.

A día de hoy, sin conocer cuáles serán sus limitaciones, la investigación en *deep learning* sigue desarrollándose de forma vertiginosa, tomando como inspiración nuestro cerebro, a pesar de no saber aún cómo funciona.

⁷ Fuente: <https://www.oreilly.com/ideas/the-current-state-of-machine-intelligence-3-0>

CAPÍTULO 3

Fundamentos teóricos

En el presente capítulo realizaremos un estudio de los conceptos teóricos necesarios para poder utilizar las redes neuronales convolucionales.

3.1 Introducción a las redes neuronales convolucionales

Como se expuso en el capítulo anterior, las redes convolucionales basan su funcionamiento en las redes neuronales y en la corteza visual humana.

Típicamente, una red neuronal consiste en un conjunto de unidades neuronales con conexiones entre ellas. Cada una de estas neuronas consta de una serie de entradas $\{x_0, \dots, x_n\}$ ponderadas por una serie de pesos $\{w_0, \dots, w_n\}$ y da como salida, y , el resultado de aplicar una cierta función, llamada función de activación, g , a la suma ponderada de las entradas por los pesos $\sum_0^n x_i * w_i$.

Las conexiones entre estas unidades establecen las entradas de las mismas, que pueden ser la salida de una unidad de la capa anterior o los datos de entrada. También, en base a estas conexiones pueden ser agrupadas en capas, siendo la primera la que recibe los datos de entrada y la última la que da la salida de la red, llamándose capas ocultas al resto de capas.

El entrenamiento de la red consiste en ajustar los pesos de las conexiones entre unidades de forma que se minimice una cierta función de error E , calculada en base al error cometido por la red en la salida sobre el conjunto de entrenamiento. En la figura 3.1 se muestra la estructura básica de una red neuronal.

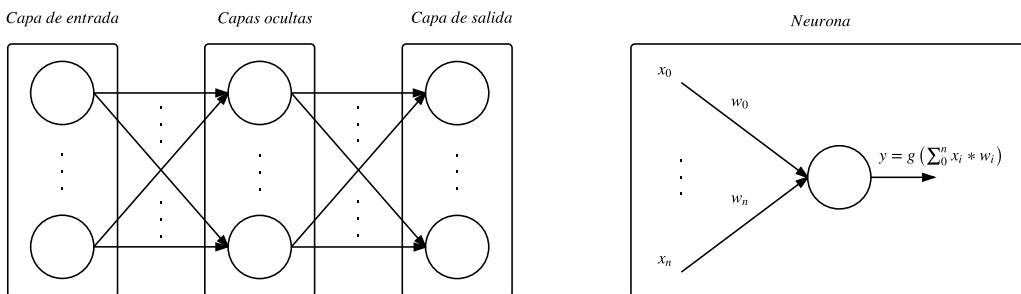


Figura 3.1: Estructura básica de una red neuronal.

Mientras que las redes neuronales están formadas únicamente por capas de neuronas artificiales, las redes convolucionales utilizan otros tipos de capas en su estructura para simular el comportamiento de la corteza visual, además de las redes neuronales. Todos los tipos de capa que podemos utilizar en una red convolucional se encuentran, de forma

detallada, en la sección 3.2. Principalmente, una red convolucional hace uso de capas de convolución, activación, *pooling* y *fully connected*.

Una de las principales diferencias entre ambas reside en la entrada de la red; mientras que en la red neuronal un experto debía elegir las características relevantes en el universo de discurso para alimentar la entrada de la red, la red convolucional pretende encontrar estas características a partir de los datos originales. En el ejemplo de la visión artificial, las redes neuronales tradicionales debían recibir, como entrada, algunas características extraídas a partir de las imágenes a procesar, mientras que la red convolucional puede trabajar directamente con la imagen de entrada y encontrar las características relevantes en la misma, sin necesidad de indicarlas a mano. Matemáticamente hablando, una red neuronal trabaja con un tensor de orden 1 en la entrada, cuyo tamaño viene determinado por el número de unidades neuronales presentes en la capa de entrada, mientras que una red convolucional recibe como entrada un tensor de orden n y, dependiendo de la tarea a realizar, incluso de cualquier tamaño.

Usando el ejemplo de clasificación de imágenes para identificar la presencia de coches en las mismas, si optamos por usar una red neuronal, debemos diseñar un extracto de características para crear la entrada de la red y obtener la salida mediante una unidad neuronal que nos indicará la presencia, o no, de un coche en la imagen de entrada. Si utilizamos una red convolucional, la imagen a clasificar es utilizada como entrada de la red donde las diferentes capas de la misma realizarán diferentes procesamientos sobre la imagen, de forma que al llegar a la última de las capas, donde se produce la clasificación de la imagen, se hayan obtenido las características más relevantes en la imagen que permite clasificarla correctamente. En la figura 3.2 encontramos la comparación de ambos sistemas en base al ejemplo descrito.

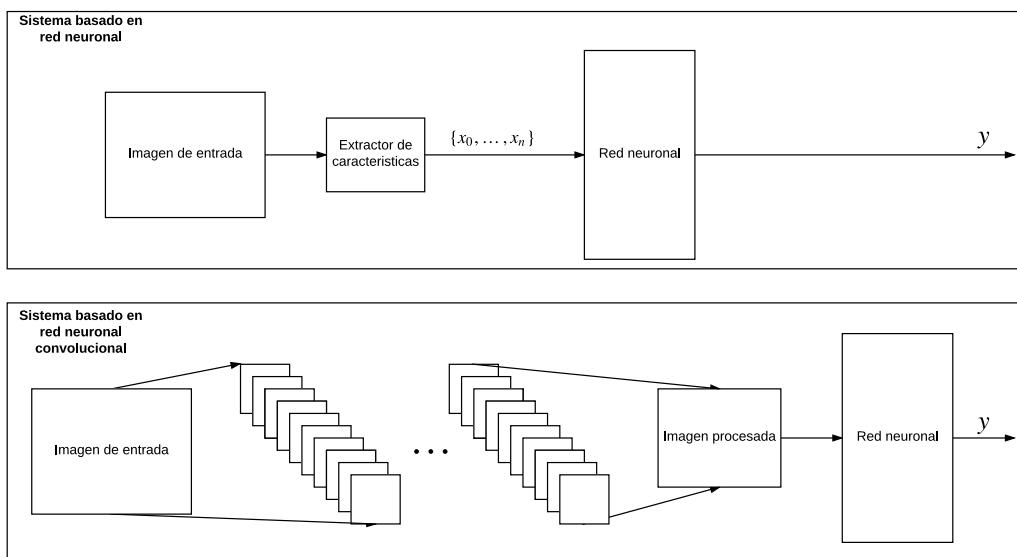


Figura 3.2: Comparativa entre un sistema basado en red neuronal y sistema basado en red neuronal convolucional.

La extracción de características se produce gracias a las capas de convolución; estas capas combinadas con otras operaciones, permiten detectar características complejas a más profundidad en la red; en lo que a nuestro ejemplo se refiere, observaríamos cómo las primeras capas buscan características muy básicas como color o líneas, mientras que si observamos capas más profundas en la red podríamos apreciar cómo es capaz de detectar la presencia de objetos más complejos, como las ruedas del coche.

Este hecho fue comprobado por Matthew D. Zeiler y Rob Fergus en un artículo que publicaron en 2014, donde mostraban una forma de entender el buen desempeño de estas redes, gracias a la capacidad de las mismas para reconocer características complejas a más profundidad [Zeiler et al., 2014]. En la figura 3.3 se muestran algunos ejemplos presentes en este artículo donde, de izquierda a derecha, se pueden observar las características detectadas por diferentes capas de una red convolucional.

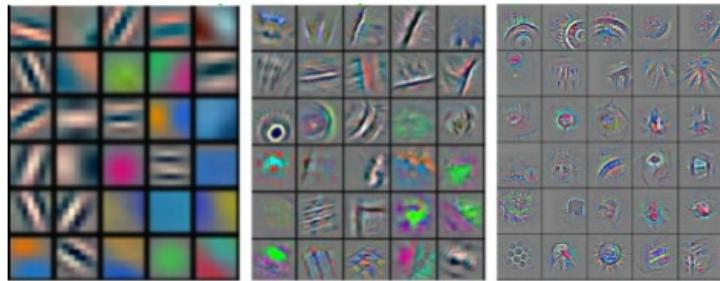


Figura 3.3: Ejemplos de características obtenidas en diferentes capas (de menos a más profundidad) de una red convolucional¹.

Por lo tanto, una red convolucional deberá encontrar qué valores son los adecuados para cada *kernel* de convolución que hay en la red, además de otros parámetros de diferentes capas, como las conexiones entre unidades en la capa de la red neuronal.

La búsqueda de estos valores parten de una estructura de la red previamente diseñada. La estructura que le demos a la red condicionará los resultados, ya que limitará las posibilidades de extraer más información de la imagen de entrada. Deberemos realizar pruebas de diferentes estructuras para la red, donde decidamos tanto el número de capas, como el tipo de las mismas y los parámetros de cada capa, como el tamaño del *kernel* de convolución. No solo la estructura condicionará las limitaciones de la red, sino también el número de parámetros que debe aprender, ya que, a mayor número de capas, más parámetros deberemos ajustar. Esto provoca la necesidad de encontrar un equilibrio entre el rendimiento y la efectividad; a mayor tamaño, mayor tiempo de entrenamiento y de ejecución.

A la hora de hablar del aprendizaje de una red, nos referimos al ajuste de los valores que hacen posible la disminución del error que comete la red a su salida; para ello deberemos decidir cómo calcularemos ese error y cómo se actualizarán los parámetros en base a este mediante algún algoritmo de entrenamiento.

El resultado que obtengamos de la red dependerá de multitud factores, entre ellos: la estructura elegida para la misma, la inicialización que hagamos de los parámetros a ajustar en la red, el algoritmo de entrenamiento, el tiempo de entrenamiento, el conjunto de entrenamiento...

Es por ello que será necesario realizar multitud de pruebas para encontrar la mejor alternativa de las diferentes opciones que disponemos, así como contar con un conjunto de entrenamiento que facilite el trabajo de la red y le permita aprender con la suficiente robustez. Todos estos detalles serán estudiados, más profundamente, en las posteriores secciones.

¹Fuente: <https://www.cs.nyu.edu/~fergus/papers/zeilerECCV2014.pdf>

3.2 Principales tipos de capas

Dentro de una red convolucional, puede requerirse realizar multitud de operaciones. Tanto para el entrenamiento como para la etapa de inferencia es necesario disponer de una arquitectura compuesta de diferentes tipos de capas que realicen el procesamiento a la imagen; sin embargo, algunas capas son utilizadas únicamente durante el entrenamiento, ya sea para que este funcione mejor o para extraer algunos cálculos sobre el mismo, como el error cometido.

En las siguientes secciones detallaremos el funcionamiento y el uso de algunas de las capas más usadas.

3.2.1. Capa de convolución

Una capa de convolución consiste en la aplicación de una serie de convoluciones sobre la entrada de la capa, pudiendo ser la imagen de entrada o la salida de otra capa de convolución.

Dado que la convolución busca extraer características, nos referiremos a la salida y la entrada como mapa de características de entrada o de salida según corresponda.

Uno de los aspectos clave de la convolución y que justifican su idoneidad para trabajar con imágenes, es su invariación a la translación, lo que permite obtener los mismos resultados sin importar la posición del objeto en la imagen.

La convolución es una operación matemática que consiste en la transformación de dos funciones o piezas de información (un mapa de características y un kernel de convolución) en un nuevo mapa de características. La convolución puede ser interpretada como un filtro, donde el kernel de convolución filtra al mapa de características para obtener cierta información, como pueden ser bordes (ver figura 3.4). Matemáticamente, y en el caso de las imágenes digitales, la convolución de una imagen bidimensional I y un kernel bidimensional de convolución $K \in \mathcal{M}_{m \times n}(\mathbb{R})$ se define mediante el operador $*$ como:

$$(I * K)(i, j) = \sum_m \sum_n I(i + m, j + n)K(m, n)$$

Normalmente, cuando utilizamos la convolución en las redes convolucionales, existe la posibilidad de añadir un término constante (*bias*), b , a la operación, de forma que la convolución queda definida como:

$$(I * K)(i, j) = \sum_m \sum_n I(i + m, j + n)K(m, n) + b$$

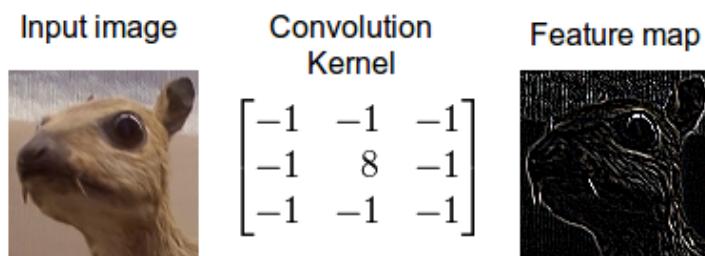


Figura 3.4: Ejemplo de la operación de convolución aplicada sobre una imagen para obtener los bordes².

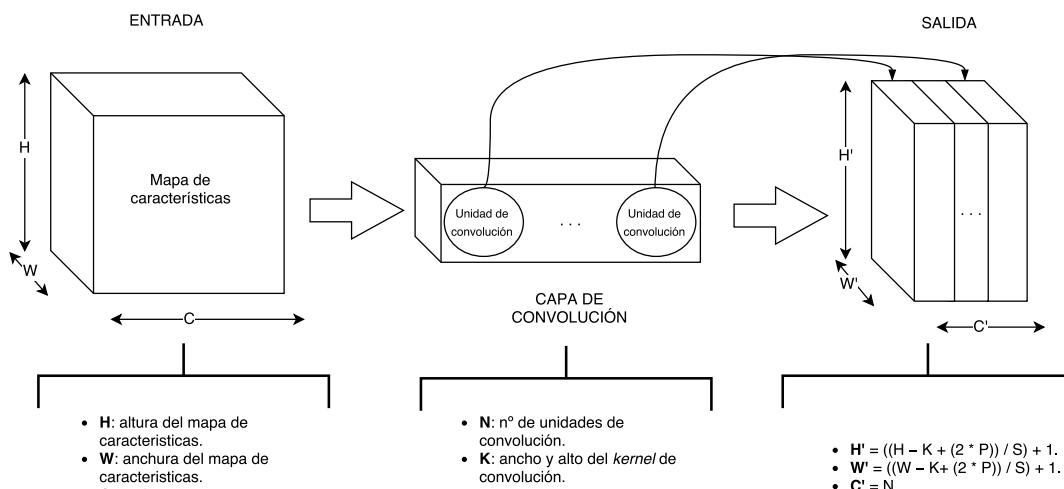
La salida de una capa de convolución se obtiene mediante la concatenación de las salidas de cada una de las operaciones de convolución que se realizan en dicha capa.

Por tanto, a la hora de utilizar una capa de convolución existen una serie de parámetros que deberemos definir:

- **Número de convoluciones:** cantidad de operaciones de convolución que se van a realizar en la capa.
- **Padding:** podemos añadirle una serie de elementos extras en los límites del mapa de características de entrada para que el *kernel* de convolución procese los límites de la entrada. Necesitaremos indicar la cantidad de *padding* a añadir y el tipo del mismo (duplicar el píxel más cercano, etc).
- **Stride:** mediante el *stride* indicamos sobre qué elementos de la entrada se realiza la operación de convolución, con un *stride* de 1 se realizaría sobre cada uno de los elementos (dependiendo del *padding*) mientras que con un *stride* de 2 por ejemplo, solo se realizaría la convolución cada dos elementos.
- **Tamaño del kernel:** debemos indicar el tamaño del *kernel* o matriz de convolución.

La elección de estos parámetros no solo condicionará la dimensión de la salida, sino también el número de parámetros que tendrá que aprender la red.

Algunos parámetros como el *padding* y el *stride* afectan únicamente al tamaño de la salida, mientras que el tamaño del *kernel* puede afectar tanto a la salida como al número de parámetros a aprender. Otro elemento clave que nos condiciona la dimensión de nuestro *kernel* y, por tanto, el número de parámetros, es la entrada de la capa. En la figura 3.5 podemos ver como calcular el número de parámetros que hay en una capa, así como el tamaño del mapa de características de salida en función de parámetros elegidos.



$$\text{Total de parámetros} = N \cdot K \cdot K \cdot C$$

Figura 3.5: Cálculo de la salida y el total de parámetros a aprender de una capa de convolución en función de la entrada y sus parámetros.

²Fuente: <https://devblogs.nvidia.com/parallelforall/deep-learning-nutshell-core-concepts/#convolutional-neural-network>

Inception

Como se ha visto, el uso de una capa de convolución implica realizar una serie de convoluciones, todas del mismo tipo, sobre la entrada, pero surge el problema de elegir los mejores parámetros con los que tratar el mapa de características de entrada.

En 2014, Christian Szegedy y un grupo de investigadores presentaron la red GoogleNet [Szegedy et al., 2014] y, junto a esta red, los módulos *inception*. Estos módulos introducían la idea de utilizar varias convoluciones a la vez sobre la entrada, en lugar de elegir solo una. Basándose en esta idea, estos módulos aplicaban varias convoluciones a la entrada pero con diferentes configuraciones, siendo concatenados los resultados de estas convoluciones para dar la salida.

Gracias al uso de estos módulos no es necesario elegir un único tipo de convolución para trabajar con la entrada; dejamos que la red convolucional extraiga características tanto globales (tamaño de los *kernels* mayores), como locales (tamaño de los *kernels* menores) y, mediante el ajuste de los parámetros, encuentre los más interesantes.

En la primera versión de los módulos *inception*, estos aplicaban únicamente tres tipos de convoluciones, tal y como se recoge en la figura 3.6. En los años siguientes se presentaron varias propuestas de mejora de estos módulos (ver [Szegedy et al., 2015] y [Szegedy et al., 2016]).

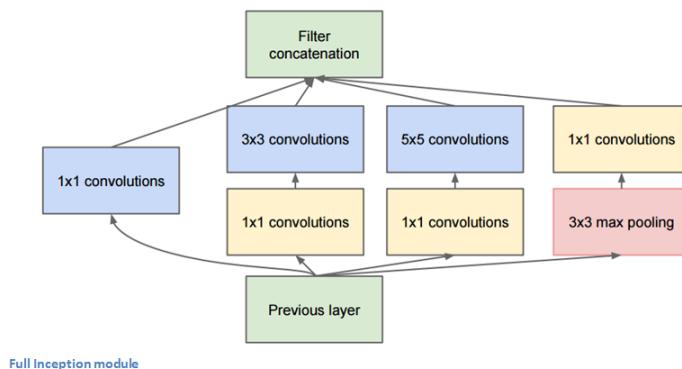


Figura 3.6: Módulo *inception*³.

3.2.2. Capa de activación

Al igual que en las unidades neuronales, que se aplicaba a la salida una función de activación, en las redes convolucionales, se aplican estas funciones de activación a la salida de las capas de convolución, como una forma de introducir no-linealidades.

La función elegida es aplicada a cada uno de los elementos del mapa de características que recibe a la entrada, por lo que únicamente modifica su contenido y no su tamaño.

Existen multitud de funciones en la literatura. En la tabla 3.1 destacamos las más utilizadas.

³Fuente: <https://adeshpande3.github.io/assets/GoogLeNet3.png>

Tabla 3.1: Principales funciones de activación⁴.

Nombre	Gráfica	Ecuación
Sigmoide		$f(x) = \frac{1}{1+e^{-x}}$
TanH		$f(x) = \frac{1}{1+e^{-2x}} - 1$
ReLU		$f(x) = \begin{cases} 0, & \text{si } x < 0 \\ x, & \text{si } x \geq 0 \end{cases}$
PReLU		$f(x) = \begin{cases} \alpha x, & \text{si } x < 0 \\ x, & \text{si } x \geq 0 \end{cases}$

3.2.3. Capa de *pooling*

El *pooling* o submuestreo es una técnica utilizada para reducir el tamaño del mapa de características e intentar condensar su información; típicamente se utilizan después de una capa de convolución.

De forma que, introduciendo este tipo de capa en la red, se reduce el número de parámetros de la misma y el tiempo de ejecución y entrenamiento.

El *pooling* opera de forma parecida a la convolución, realizando una determinada operación de forma local sobre toda la entrada. Es por ello que cuenta con el mismo tipo de parámetros que la capa de convolución. Estos parámetros son:

- **Tamaño de la ventana de muestra:** zona del mapa de características de la entrada que analizamos en cada paso.
- **Stride:** cada cuántos elementos en la entrada se efectúa la operación.

Normalmente, para asignar el valor de salida se suele aplicar una operación para extraer algún valor representativo de la región que se analiza, siendo las más utilizadas las siguientes opciones:

- **Max pooling:** consiste en asignar el máximo de la región analizada como salida.
- **Average pooling:** obtiene como salida la media de los valores de la región analizada.
- **Learned p-norm pooling:** aplica una norma de tipo p . Donde p será un parámetro más a aprender por la red, cuando $p = 1$ corresponde a un *average-pooling* y cuando $p = \infty$ se aplica un *max-pooling*. L_p pooling se puede expresar como:

$$y_{i,j,k} = \left[\sum_{(m,n) \in \mathcal{R}_{ij}} (a_{m,n,k})^p \right]^{\frac{1}{p}}$$

Donde $y_{i,j,k}$ es la salida de la operación en la posición (i, j) en el k -ésimo mapa de características de la salida, y $a_{m,n,k}$ es el valor en la posición (m, n) dentro de la región analizada \mathcal{R}_{ij} en el k -ésimo mapa de características de la entrada.

⁴Fuente: https://en.wikipedia.org/wiki/Activation_function

En la figura 3.7 se muestra el resultado de aplicar una operación de *max-pooling* con tamaño de ventana 2×2 y un *stride* de 2.

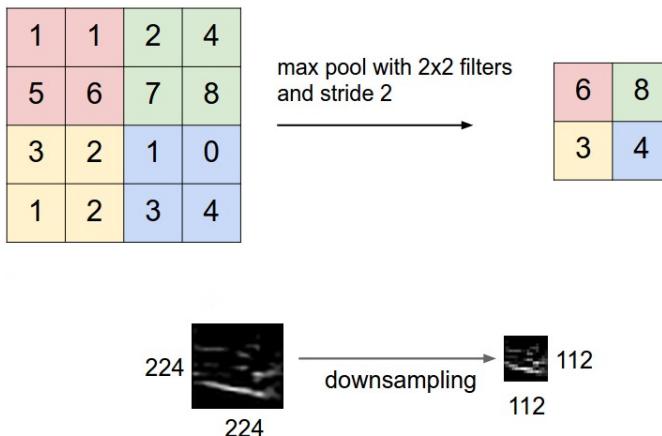


Figura 3.7: Ejemplo de la operación *max-pooling*⁵.

3.2.4. Capa de *upsampling* o *transposed convolution*

Esta capa suele utilizarse para devolver una imagen a su tamaño original. Normalmente, en una red convolucional, las capas de *pooling* y convolución pueden reducir el tamaño de la entrada a lo largo de la red; sin embargo, en algunos casos es necesario obtener en la salida las mismas dimensiones que en la entrada, como en el caso de la segmentación, donde debemos hacer corresponder cada píxel de la imagen de entrada con la clase asignada.

La operación de *upsampling* consiste en reescalar la entrada al tamaño deseado, calculando el valor de cada elemento usando un método de interpolación como la interpolación bilineal.

Por otro lado, la operación de *transposed convolution*, pretende revertir la operación de convolución; en la literatura puede encontrarse esta operación con el nombre de deconvolución, a pesar de no ser la operación inversa a la convolución matemática, es por ello que se intenta no utilizar este término. También podemos encontrar la operación *dilated convolution*, donde los huecos que inserta el *stride* como se vera más adelante, son introducidos en el *kernel* de convolución en lugar de en la entrada.

Dada una entrada, la operación de *transposed convolution* intenta aprender cómo devolverla al tamaño deseado. Esta operación se realiza mediante una convolución a la entrada, pero con la diferencia de que el parámetro *stride* se utiliza para añadir elementos entre los que hay en la entrada como lo hace el *padding* en los bordes. Esto permite que la salida sea más grande que la entrada y, por tanto, reescalarla al tamaño deseado.

En la figura 3.8 podemos ver el resultado de aplicar una operación de *transposed convolution* con *kernel* 3×3 sobre una entrada de tamaño 3×3 , con un *padding* de 1 y un *stride* de 2, dando como resultado una salida de tamaño 5×5 (normalmente los huecos se llenan con ceros).

⁵Fuente: <http://cs231n.github.io/convolutional-networks/>

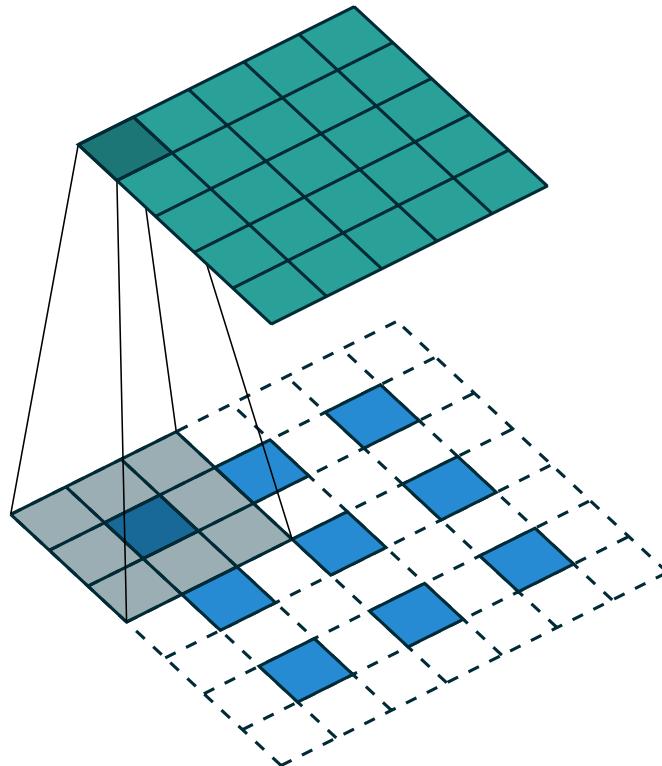


Figura 3.8: Ejemplo de la operación de *transposed convolution*⁶.

De forma que deberemos ajustar los parámetros *stride* y *padding*, para añadir elementos en el mapa de características de la entrada y conseguir el tamaño de salida deseado. Por tanto, en este tipo de capas contamos con los siguientes parámetros:

- **Tamaño del kernel:** zona de la entrada que se analizará para dar la salida. En el caso de la *transposed convolution*, sus valores serán ajustados para minimizar el error que comete, mientras que en el *upsampling* solo servirá para determinar la zona de la entrada que se debe tener en cuenta para calcular el valor de salida.
- **Stride:** en una capa de convolución, el *stride* indicaba en qué factor quedaba reducida la entrada a la salida, mientras que en una capa de *upsampling* o *transposed convolution* determinará el factor al que se reescalará la entrada.
- **Padding:** podemos añadirle una serie de elementos extras en los límites del mapa de características de entrada para que se procesen los límites de la entrada. Necesitaremos indicar la cantidad de *padding* a añadir y el tipo del mismo (duplicar el píxel más cercano, etc).

3.2.5. Capa de *fully connected*

La capa *fully connected* no es más que una red neuronal de una capa de entrada y una capa de salida, en la que todas las unidades de la capa de entrada están conectadas con todas las unidades de la capa de salida, de ahí su nombre.

Suele utilizarse para la clasificación de las características extraídas por las capas de convolución, es por ello que se sitúan al final de la red.

⁶Fuente: https://github.com/vdumoulin/conv_arithmetic

En este tipo de capas nos encontraremos con los siguientes parámetros:

- **Número de unidades de entrada:** debemos calcular el tamaño del mapa de características que recibe esta capa para determinar así cuántas unidades neuronales deberemos poner como entrada, tantas como elementos hay en la entrada.
- **Función de activación:** esta función suele elegirse de entre las que se usan en las capas de activación y que fueron recogidas en la sección 3.2.2.
- **Número de unidades de salida:** suele escogerse según el número de clases que tenemos para clasificar, aunque si esta red se sitúa antes que otra puede elegirse una cantidad diferente.

Cabe destacar que esta capa suele ser la que más parámetros aporta a la red, ya que si la entrada de esta capa tiene dimensión $H * W * C$, necesitaremos $H * W * C$ unidades de entrada y si disponemos de N unidades de salida, tendremos un total de $H * W * C * N$ parámetros en esta capa.

3.2.6. Capa de *softmax*

La capa *softmax* es usada cuando se desea transformar la salida de la red neuronal en probabilidades para dar la salida de la red como la probabilidad de pertenencia de la entrada a cada una de las clases. Para ello realiza una aplicación $S(v) : \mathbb{R}^N \rightarrow \mathbb{R}^N$:

$$S(v) : \begin{bmatrix} v_1 \\ \vdots \\ v_N \end{bmatrix} \rightarrow \begin{bmatrix} S_1 \\ \vdots \\ S_N \end{bmatrix}$$

Donde cada elemento se obtiene de la siguiente fórmula:

$$S_i = \frac{e^{v_i}}{\sum_j e^{v_j}} \quad \forall i \in 1, \dots, N$$

Esta capa no necesita definir ningún parámetro adicional, únicamente indicar el número de clases con la que trabajamos para establecer su salida.

3.2.7. Otros tipos de capas

Las capas mencionadas en los apartados anteriores son las más comunes en una red convolucional, pero según el problema que se aborde o el entrenamiento que se realice, puede ser necesario realizar algunos otros tipos de operaciones al inicio, al final o incluso en mitad de la red.

Durante el entrenamiento, por ejemplo, puede ser útil realizar algún tipo de normalización de la entrada antes de darle la misma a la red o controlar el sobreajuste mediante la utilización del *dropout* (ver Sección 3.4.6).

Normalmente se ha optado por incluir estas operaciones como parte de la estructura de la red, como es el caso del cálculo del error para realizar el entrenamiento (ver Sección 3.4.4) o el cálculo de métricas de la red (ver Sección 3.4.3). De este modo, recogemos en un único esquema todo los procesos que se realizan en la red, indicando en cada capa la operación realizada y según su disposición en el esquema, el momento en el que se aplican, ayudándonos a comprender mejor todo el proceso.

3.3 Casos de estudio

Con objetivo de estudiar cómo se utilizan las capas vistas con anterioridad, se hace un análisis de algunos casos de usos de redes neuronales convolucionales que tienen bastante popularidad, ya sea por los resultados conseguidos o por las novedades que presentaban.

3.3.1. LeNet

Se trata de la primera red convolucional de la historia. Fue introducida por Yann LeCunn en 1998 [LeCun, et al., 1998], fue entrenada para el reconocimiento de dígitos escritos a manos. Para ello, se utilizó la base de datos MNIST [LeCun et al., 1998], que consta de 10 clases, dígitos del 0 al 9, con 60000 imágenes de entrenamiento, todas ellas en escala de grises y un tamaño de 32×32 .

Esta red se compone de un total de 7 capas de cuatro tipos: convolución, *pooling*, activación y *fully connected*. Utiliza convoluciones 5×5 , un *average pooling* y como funciones de activación utiliza la sigmoide y la tanh, la clasificación se realiza mediante una red neuronal con tres capas.

En la figura 3.9 se recoge la estructura de la red, que fue entrenada usando solo la CPU.

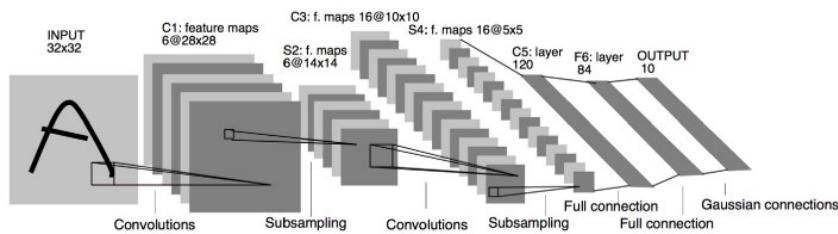


Figura 3.9: Arquitectura de la red LeNet⁷.

3.3.2. AlexNet

En 2012, Alex Krizhevsky, Ilya Sutskever y Geoffrey Hinton presentaron la red AlexNet [Krizhevsky, et al., 2012]. Esta red participó en la competición ImageNet en 2012 donde logró mejorar los resultados obtenidos hasta la fecha de forma significativa, lo que sirvió para popularizar el uso de las redes convolucionales en temas de visión. Este artículo ha sido citado un total de 6184 veces y es considerado como una de las publicaciones más influyentes en *deep learning*.

Esta red, con una estructura similar a la LeNet, cuenta con 11 capas con diferentes tipos de convoluciones, además del *average pooling* y la función ReLU, introduce una serie de novedades con respecto al entrenamiento, mediante una nueva forma de controlar el sobreajuste (ver Sección 3.4.6) con el uso del *dropout*. En la figura 3.10 se muestra la estructura de la red.

A pesar de tener más parámetros para ajustar que la LeNet, dispuso de un conjunto de datos mucho mayor con 1,2 millones de imágenes para el entrenamiento y mil clases para clasificar. En este caso, las imágenes eran a color y con un tamaño de 224×224 .

⁷ Fuente: <http://img2016.itdadao.com/d/file/tech/2016/08/18/qr2uYzz.png>

Fue implementada en GPU, lo que abrió la puerta al uso de las mismas para el *deep learning*, se logró incrementar la velocidad de entrenamiento 50 veces con respecto a la CPU. La red fue entrenada durante una semana en dos GPUs.

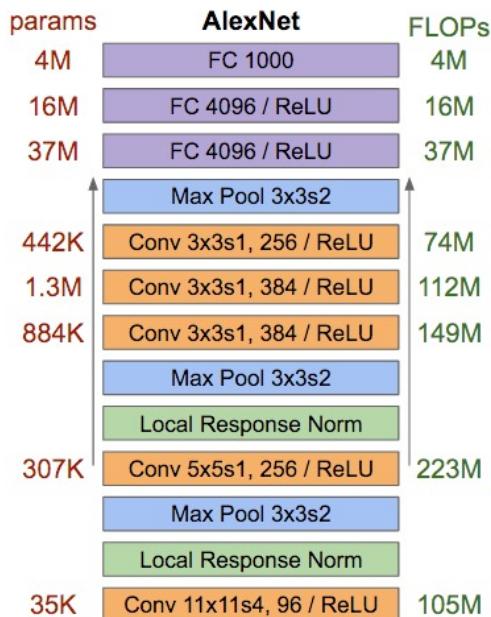


Figura 3.10: Arquitectura de la red AlexNet⁸.

3.3.3. VGG

Presentada por Karen Simonyan y Andrew Zisserman [Simonyan et al., 2014], quedó segunda en la competición ImageNet en 2014, donde logró bajar el error a un 7.3 %, se diseñó de forma simple pero con un incremento notable de la profundidad de la red, pasando a un total de 19 capas.

Esta formada únicamente por convoluciones 3×3 con función de activación ReLU entre ellas y algunos *pooling*, en concreto *max pooling*. En la figura 3.11 se detalla su estructura.

Esta red demostró que incluso con una estructura poco compleja y con la suficiente profundidad, se pueden conseguir buenos resultados. Fue en este año cuando se comenzó a optar por aumentar la profundidad de las redes para conseguir mejores resultados.

⁸Fuente: <https://groups.google.com/forum/#topic/caffe-users/cUD3IF5NMOk>

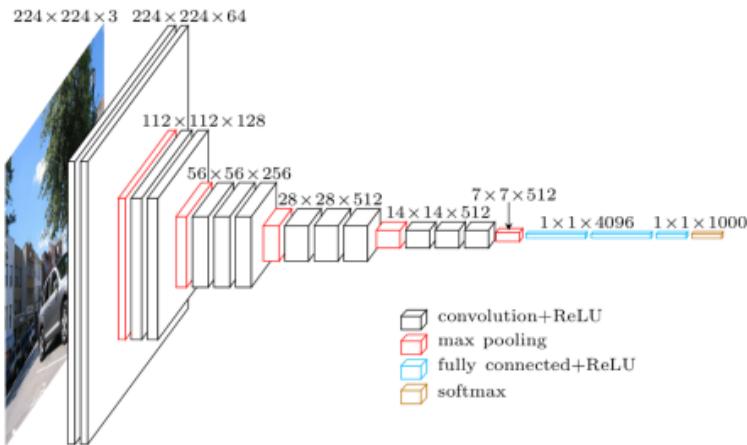


Figura 3.11: Arquitectura de la red VGG⁹.

3.3.4. GoogleNet

GoogleNet [Szegedy et al., 2014] fue la red ganadora de la competición ImageNet en 2014, donde lograron un 6.67 % de error e introdujeron, por primera vez, el concepto de los módulos *inception*, gracias a los cuales lograron reducir drásticamente el número de parámetros que tenía que aprender la red.

Otra estrategia que utilizaron para reducir el número de parámetros de la red fue usar el *average pooling* en lugar de una capa *fully connected* al final de la red, lo que redujo el número de parámetros.

Esta red contaba con un total de 100 capas y, gracias a sus buenos resultados, dio inicio a un desarrollo de redes más profundas. En la figura 3.12 se muestra su estructura. Se entrenó con varias GPUs en una semana.

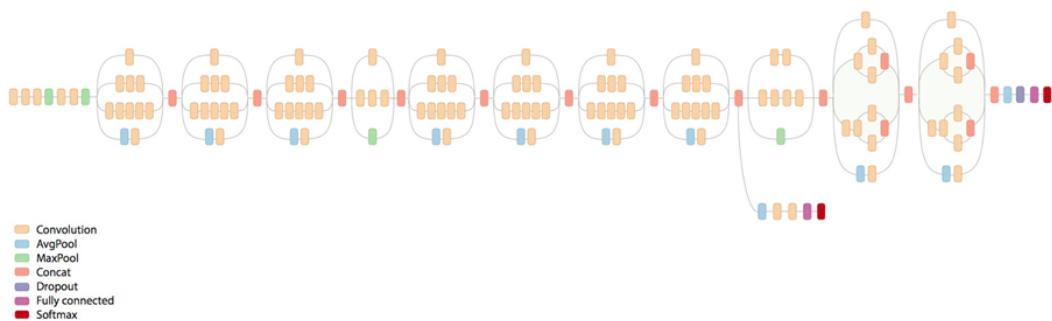


Figura 3.12: Arquitectura de la red GoogleNet¹⁰.

Fue la primera en introducir la idea de no seguir siempre con una estructura secuencial en la red gracias a los módulos *inception*. En los siguientes años se presentaron nuevos modelos de esta red y de los módulos *inception* (ver Sección 3.2.1).

⁹ Fuente: <https://blog.heuritech.com/2016/02/29/a-brief-report-of-the-heuritech-deep-learning-meetup-5/>

¹⁰ Fuente: <https://adshpande3.github.io/assets/GoogLeNet.png>

3.3.5. ResNet

Esta red fue la ganadora de la competición ImageNet en 2015 y fue presentada por el equipo de investigación de Microsoft [He, et al., 2015]. Presentó un nuevo concepto conocido como *residual*, que consistía en dar como entrada a una capa, la salida de la anterior combinada con una entrada anterior a esta. En la figura 3.13 se muestra de forma gráfica este concepto.

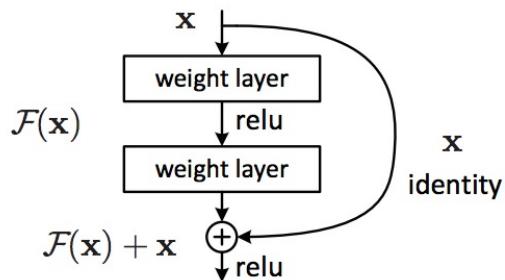


Figura 3.13: Concepto *residual*¹¹.

Haciendo uso de este concepto y sobre todo de un gran número de capas (152) fue capaz de conseguir un 3,2 % de error. En la figura 3.14 se muestra una comparación de la red VGG y la construcción de una red residual en base a una red básica. La red ganadora de la competición es una versión aumentada estos bloques de convolución.

¹¹Fuente: <https://culurciello.github.io/assets/nets/resnetb.jpg>

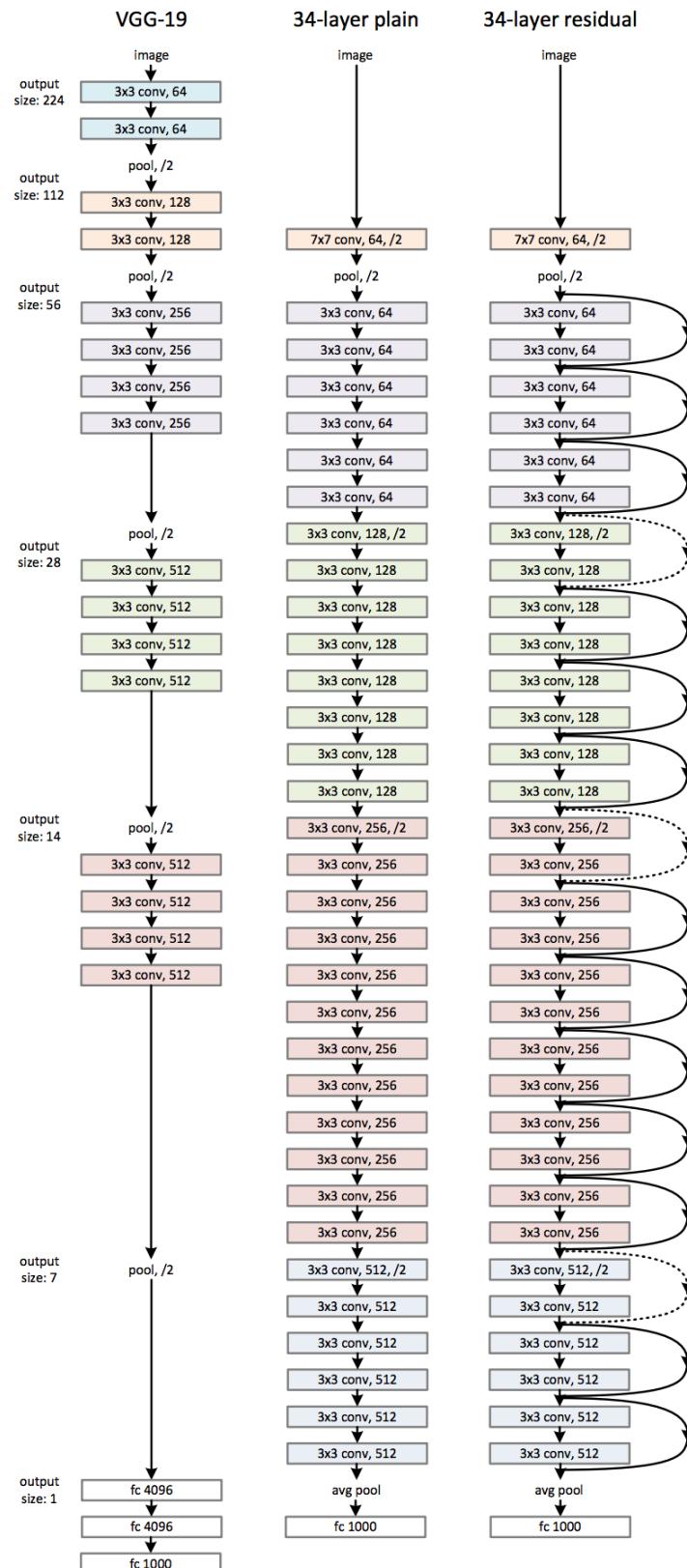


Figura 3.14: Arquitectura de la red ResNet¹².

¹²Fuente: <http://felixlaumon.github.io/assets/kaggle-right-whale/resnet.png>

3.4 Desarrollo de un sistema basado en redes neuronales convolucionales

Hasta ahora, hemos explicado en qué consisten las redes convolucionales y las capas que las componen, además de ver algunos ejemplos de redes populares. Sin embargo, aún no hemos hablado de cómo se diseñan estas redes.

Cuando se diseña una red convolucional, deben elegirse tanto el tipo como el número de capas y su disposición en la red; además de definir los parámetros que sean necesarios en cada capa. No obstante, esto se hace sin seguir ningún criterio especial, puesto que no se ha encontrado una metodología a seguir para su diseño, únicamente nos podemos ayudar de nuestra experiencia. Debido a esto, realizaremos nuestro diseño teniendo en cuenta las pautas indicadas por algunos expertos en la materia y tomando como referencias algunas de las redes convolucionales expuestas en el capítulo anterior.

Una vez que tengamos un primer diseño para nuestra red, tendremos que ir ajustando los parámetros de la misma mediante un proceso de aprendizaje. Para ello, necesitaremos un conjunto de datos y un algoritmo con los que realizar el entrenamiento; además de una forma de evaluar tanto el entrenamiento como el desempeño de la red. Esto nos permitirá poder entrenar la red correctamente y refinar su diseño según los resultados que vayamos obteniendo.

En esta sección abordaremos los conceptos necesarios para realizar, correctamente, todo este proceso.

3.4.1. Datos necesarios para desarrollar el sistema

Una de las partes más importantes para el desarrollo de una red convolucional son los datos que disponemos; puesto que nuestra red será entrenada y evaluada en base a ellos. Por esta razón, nuestros datos deberán recoger toda la casuística del problema que estamos tratando si deseamos obtener buenos resultados.

La cantidad de datos que necesitaremos dependerá del problema que abordemos, ya que no es lo mismo desarrollar un sistema para participar en Imagenet, donde hay mil clases, que un sistema que involucre trabajar con muchas menos clases y en un entorno más controlado.

A pesar de no existir una forma de conocer, a priori, la cantidad de datos necesarios para conseguir un buen rendimiento, parece haber consenso en que, a mayor cantidad de datos, se obtienen mejores resultados (ver [Cho et al., 2015] y [Soekhoe et al., 2016]). A veces, no es posible obtener todos los datos necesarios, ya sea por inviabilidad práctica o por un coste excesivo, como en el aprendizaje supervisado, donde se necesita un etiquetado de los datos para realizar el entrenamiento. Es en estos casos cuando se hace uso de técnicas para aumentar el conjunto de datos. Cuando trabajamos con imágenes, típicamente se suelen aplicar cambios de luminosidad, cambios de contraste, rotaciones y traslaciones. Mediante esta técnica podemos conseguir más datos a partir de los que ya tenemos y dotar de más robustez al sistema, al exigirle los mismos resultados a pesar de las variaciones que hagamos en los datos.

Otro factor a tener en cuenta es la calidad de los datos que disponemos. La presencia de ruido o datos mal etiquetados suponen una dificultad añadida para la red durante el aprendizaje; es por ello que debemos tener cuidado de que no haya errores en el conjunto de datos y aplicar técnicas para paliar los efectos del ruido si fuese necesario.

En el campo de la visión, nos enfrentamos a multitud de problemas al trabajar con imágenes. Dependiendo de la casuística de nuestro problema, puede que tengamos que

hacer frente a algunos de los escenarios planteados en la figura 3.15. Estas situaciones pueden dificultar el entrenamiento, puesto que puede requerir de un diseño más complejo de la red o la recolección de más datos para lograr un buen resultado.

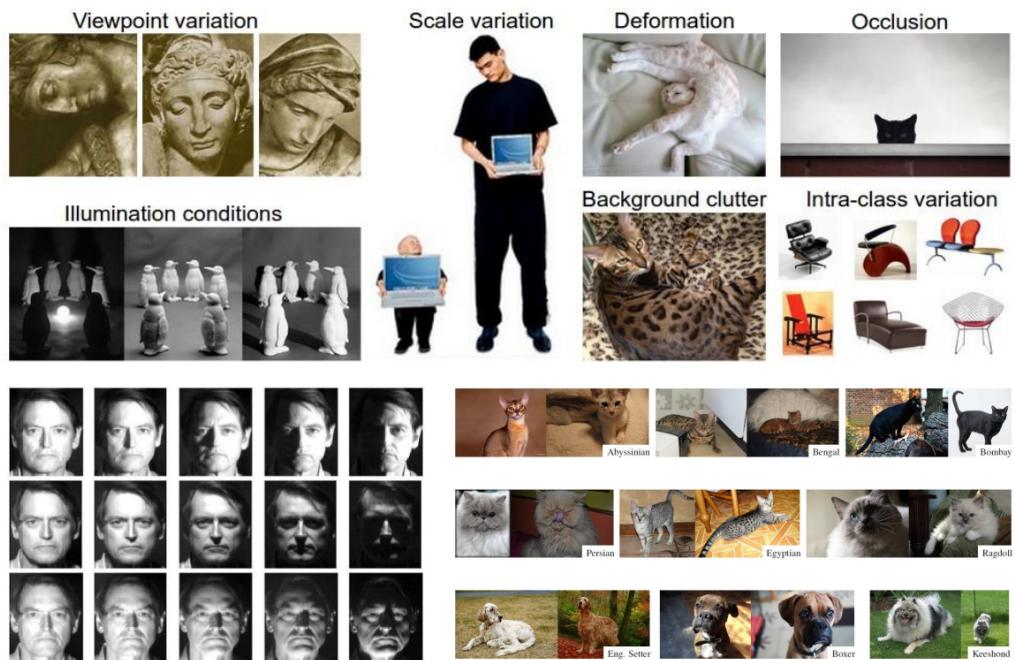


Figura 3.15: Problemas típicos en visión artificial¹³.

En nuestro caso, en concreto, encontramos los problemas expuestos en la figura 3.16.



Figura 3.16: Ejemplos de situaciones problemáticas a las que nos enfrentamos.

Tanto la cantidad como la calidad de los datos que disponemos determinará los resultados que obtengamos. No obstante, necesitaremos una forma de medir el desempeño de

¹³Fuente: <http://selfdrivingcars.mit.edu/>

nuestra red, para saber si es el deseado o no y poder corregir la situación. Estas métricas de evaluación se analizan en profundidad en la Sección 3.4.3.

Surge un problema a la hora de calcular estas métricas con la red, dado que solo disponemos de un conjunto de datos. Si este conjunto de datos es utilizado por completo para entrenar, las métricas que obtengamos solo nos hablarán de cómo de buena es la red con los datos que ha utilizado durante su entrenamiento, pero no sabemos cómo se comportará cuando le demos nuevos datos que no estén en este conjunto. Esto es debido a que el entrenamiento busca reducir el error que comete la red sobre los datos con los que realiza el entrenamiento. Sin embargo, la red puede dar muy buenos resultados sobre el conjunto de entrenamiento, pero funcionar bastante mal con datos nuevos; es lo que se conoce como sobreajuste (ver Sección 3.4.6). Nuestro objetivo será evitar este sobreajuste y asegurarnos de que la red puede aprender de estos datos a dar un buen rendimiento ante cualquier dato de entrada; es decir, buscamos que sea capaz de generalizar lo aprendido del conjunto de datos.

Una de las estrategias que se siguen para evaluar correctamente el comportamiento de la red, consiste en dividir el conjunto de datos en tres conjuntos:

- **Conjunto de entrenamiento:** es utilizado para construir el modelo y realizar el entrenamiento; con él se calcula el error que comete la red y se ajustan los parámetros en base a este.
- **Conjunto de validación:** mediante este conjunto se evalúa la red durante el entrenamiento para obtener su rendimiento. En base a los resultados que obtengamos se realizan ajustes en los hiperparámetros (ver Sección 3.4.6) para mejorar los resultados.
- **Conjunto de test:** este conjunto de datos no es utilizado por la red durante el entrenamiento, únicamente se usa para evaluar el rendimiento de la red una vez tengamos el modelo final.

Con estos tres conjuntos se pretende obtener una medida más real del desempeño de la red, al no verse influido ni el diseño de la red ni el entrenamiento por los datos que hay en el conjunto de test, con el que obtendremos las métricas de nuestro modelo.

Es difícil determinar la proporción de cada conjunto cuando tenemos que realizar la partición. Típicamente, se suele optar por utilizar el 50 % para el conjunto de entrenamiento y un 25 % para el conjunto de validación y test, respectivamente.

Existen multitud de técnicas para realizar la división del conjunto de datos; una de las más populares es el *cross validation*, que permite evaluar al sistema mediante la media aritmética obtenida de las medidas de evaluación sobre diferentes particiones del conjunto de datos, haciendo una designación del conjunto de evaluación y entrenamiento distinta en cada partición.

Otro problema al que nos enfrentamos es cuando no tenemos el mismo número de ejemplos para cada clase. Suele ser algo común en situaciones donde los datos de una cierta clase se dan con poca probabilidad. En estos casos existen dificultades cuando se analiza el rendimiento de la red. No podemos saber si los resultados reflejan un buen rendimiento por igual con todas las clases o benefician a las clases mayoritarias, en detrimento de las minoritarias. Normalmente se intenta paliar estos efectos recogiendo más datos, cambiando las métricas de evaluación para tener en cuenta esta situación o modificando el conjunto de datos para hacerlo más equitativo.

3.4.2. Inicialización de parámetros y transferencia de modelos

Una vez tenemos una estructura definida para la red, y antes de empezar el entrenamiento, necesitamos darle unos valores iniciales a los parámetros que se deberán ajustar durante el aprendizaje; un punto de partida para comenzar el entrenamiento.

Normalmente, los parámetros que debemos ajustar son los valores de los *kernels* de las capas de convolución, los pesos de las capas *fully connected* o algún otro parámetro de otra capa que deba aprender la red.

Las capas que cuentan con una unidad de *bias* (término constante que se aplica al realizar la operación), como es el caso de las capas de convolución o las capas *fully connected*, se pueden inicializar a cero, mientras que los demás parámetros deben inicializarse de forma que se rompa la simetría. Se considera que existe una simetría cuando todos los parámetros tienen el mismo valor, algo que intentaremos evitar. Como veremos en la Sección 3.4.5, el ajuste de los parámetros se produce propagando el error que se comete en la salida hacia las capas anteriores; de forma que, cada parámetro, se ajusta en proporción al error que ha cometido. Si todos los parámetros se inicializan a un mismo valor, siempre variarán de la misma forma, por lo que mantendrán los mismos valores, provocando una simetría. Esto puede ocasionar que el sistema nunca aprenda correctamente si se requiere que existan parámetros con valores diferentes. Es por ello que se intenta hacer una inicialización aleatoria que rompa esta simetría.

No es algo trivial la inicialización de estos parámetros, ya que según se haga, determinará la solución que podremos alcanzar, como se analizará en secciones posteriores. Si los parámetros de la red se inicializan a unos valores muy pequeños, la señal de entrada se irá reduciendo a lo largo de la red hasta llegar a ser demasiado insignificante para ser útil. Por el contrario, si se inicializan a unos valores muy grandes, la señal de entrada irá creciendo a través de la red de forma que será demasiado grande como para ser de utilidad. A continuación, presentamos las formas más comunes que existen de realizar esta inicialización:

- **Inicialización a partir de una distribución gaussiana:** fue utilizada por los creadores de la red Alexnet [Krizhevsky, et al., 2012]. Sin embargo, como indicaron los creadores de la red VGG [Simonyan et al., 2014], esta inicialización para modelos profundos (con muchas capas) presentan dificultades para converger, algo que fue corroborado también por los autores de la red Resnet [He, et al., 2015]. Esta inicialización consiste en asignar los valores a los parámetros de forma aleatoria mediante una distribución normal:

$$w_{ji} \sim N(\mu, \sigma^2)$$

Normalmente se suele elegir un $\mu = 0$ y un σ pequeño (0,001 p. ej.).

- **Inicialización Xavier:** esta técnica fue presentada por Xavier Glorot y Yoshua Bengio [Glorot et al., 2009]. Surge como solución al problema que aparecía cuando se hacía una inicialización puramente aleatoria. Según la función de activación que utilicemos, se considera que se satura al llegar al valor máximo o mínimo que da como salida. Si otorgamos los pesos de forma aleatoria, puede que provoquemos esta situación y tengamos muchas salidas saturadas; lo que exige más iteraciones al algoritmo de entrenamiento para ajustar los pesos. Sus autores sugieren inicializar los pesos de forma que la varianza de los pesos de una neurona sea igual a la unidad. De esta forma, reducimos la probabilidad de estar situándonos sobre las zonas que provocan la saturación de la función de activación. Si consideramos una neurona con n pesos, la varianza de la neurona será:

$$\text{Var}(\text{neurona}_j) = \text{Var}(w_{1j}x_1 + \cdots + w_{nj}x_n) = n\text{Var}(w_{ji}) \quad \forall i \in 1, \dots, n.$$

Por tanto, esta inicialización consiste en dar un valor a los pesos a partir de una distribución normal:

$$w_{ji} \sim N(0, \frac{1}{n})$$

Cabe recordar que en el artículo original los autores toman la media de unidades de entrada (N_{in}) y de salida (N_{out}) para calcular la varianza:

$$\begin{aligned} \text{Var}(w_{ij}) &= \frac{1}{N_{avg}} \\ N_{avg} &= \frac{N_{in}+N_{out}}{2} \end{aligned}$$

Sin embargo, esta forma de calcular la varianza es más exigente computacionalmente y se suele utilizar la implementación anterior.

- **Inicialización He:** fue introducida por los creadores de la red Resnet [He et al., 2015] como la mejor alternativa para inicializar una red que utilice funciones de activación ReLU. Según sus creadores, permite la convergencia de modelos extremadamente profundos, para los cuales, la inicialización *Xavier* no era eficaz. Consiste en asignar un valor aleatorio tomado de una distribución uniforme:

$$w_{ji} \sim U(0, \sqrt{\frac{2}{n}})$$

donde n es el número de entradas respectivas de la unidad j .

Otra estrategia frecuente para entrenar redes convolucionales consiste en utilizar una red ya entrenada como punto de partida, lo que se conoce como transferencia de modelos. En una red convolucional las primeras capas captan las características presentes en la entrada, mientras que las últimas capas neuronales son las encargadas de indicar a qué clase pertenece nuestra entrada. Diferentes autores han demostrado que estas capas de extracción de características, una vez entrenada, pueden ser retiradas de la red para aplicarlas en otros problemas, consiguiendo un buen rendimiento (ver [Razavian et al., 2014] y [Oquab et al., 2014]). Si partimos de una red ya entrenada en un problema diferente, solo tendremos que modificar las últimas capas que realizan la clasificación para adaptarla a nuestro nuevo problema. Por tanto, no necesitamos realizar el entrenamiento desde cero y podemos aprovechar las características aprendidas por la red para el problema anterior.

3.4.3. Métricas de evaluación

Las métricas de evaluación son las medidas que nos permiten estudiar el rendimiento de nuestro sistema. Nos dan una evaluación objetiva de nuestro modelo. Estas métricas nos permiten comparar diferentes redes.

La evaluación de un sistema dependerá del problema que esté resolviendo. En nuestro caso, al trabajar con problemas de clasificación, es importante tener en cuenta que en la salida de la red dispondremos de una vector con probabilidades de pertenencia a cada clase. Es necesario aplicar una operación (normalmente el índice del máximo) para elegir la clase que corresponde a la entrada. También suele aplicarse un umbral para exigir un mínimo de probabilidad a cada clase en la salida para tenerla en cuenta a la hora de asignar la clase final.

A continuación, presentaremos las principales métricas que se usan para este tipo de problemas y cómo nos pueden ayudar a desarrollar nuestro sistema.

- **Matriz de confusión:** mediante esta matriz almacenamos para cada una de las clases, el resultado de la clasificación hecha por el modelo para cada ejemplo de entrenamiento, de forma que representamos la clase a la que pertenece el ejemplo en las columnas y la clase asignada por el sistema en las filas. A partir de esta matriz podemos obtener las siguientes métricas:

- **Verdaderos positivos (TP):** son los ejemplos de la clase que son clasificados como tal. Se corresponden, para la clase i al elemento i -ésimo de la diagonal: a_{ii} .
- **Falsos positivos (FP):** son los ejemplos que no son de la clase y se han clasificado como si lo fueran. Se corresponden, para la clase i al valor calculado como:

$$\left(\sum_{j=1}^n a_{ij} \right) - a_{ii}$$

- **Falsos negativos (FN):** son los ejemplos de la clase que no son clasificados como tal. Se corresponden, para la clase i al valor calculado como:

$$\left(\sum_{j=1}^n a_{ji} \right) - a_{ii}$$

- **Verdaderos negativos (TN):** son los ejemplos que no son de la clase y se han clasificados como tal. Se corresponden, para la clase i al valor calculado como:

$$\sum_{\substack{j,k=1 \\ j \neq i \\ k \neq i}}^n a_{jk}$$

- **Precision:** representa, de todas las muestras clasificadas como de la clase, cuántas se han acertado. Se calcula como:

$$\text{Precision} = \frac{TP}{TP+FP}$$

- **Recall:** representa el número de muestras de la clase clasificadas correctamente respecto del total. Se calcula como:

$$\text{Recall} = \frac{TP}{TP+FN}$$

- **Accuracy:** representa el número de aciertos totales, de la clase y de lo que no es de la clase. Se calcula como:

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

En la tabla 3.2 se muestra esta matriz y donde hallar los valores para obtener las métricas mencionadas.

	C_1	\dots	C_n
C_1	a_{11}	\dots	a_{1n}
\vdots	\vdots	\ddots	\vdots
C_n	a_{n1}	\dots	a_{nn}

Tabla 3.2: Matriz de confusión

Según deseemos que se comporte el sistema, puede que nos interese ajustar los umbrales de clasificación para evitar falsos positivos o relajarlos para aumentar los falsos negativos. En la figura 3.17 se muestra un ejemplo de un clasificador con diferentes umbrales.



Figura 3.17: Resultado de la variación de umbrales¹⁴.

Además de servirnos para elegir los mejores umbrales, estas métricas nos ayudan a controlar el aprendizaje, ya que nos permiten detectar cuándo se produce sobreajuste o en qué momento podemos parar el entrenamiento (ver Sección 3.4.6).

Dependiendo del problema al que nos enfrentemos (etiquetado, detección o segmentación), podemos necesitar obtener otras métricas adicionales para evaluar al sistema. Estas métricas serán estudiadas en las respectivas secciones en el Capítulo 4.

3.4.4. Cálculo del error

Nuestro objetivo durante el entrenamiento será minimizar el error que comete nuestro modelo al realizar la clasificación. Por esta razón, necesitaremos definir cómo calcularemos el error.

El cálculo del error, al igual que el cálculo de las métricas, dependerá del problema que estemos resolviendo. Al utilizar para nuestro trabajo aprendizaje supervisado, nuestros datos del conjunto de entrenamiento vendrán etiquetados con la clase correspondiente. El resultado de la clasificación realizada por la red es típicamente un vector de probabilidades, \vec{v} , con un tamaño, N , igual al número de clases, resultado de la última capa de la red:

$$[v_1, \dots, v_N]$$

Nuestra salida deseada corresponde con el vector que tiene un valor uno en la posición de la clase a la que pertenece el ejemplo y un cero en las demás posiciones. De esta forma, podemos medir el error en base a la salida obtenida del clasificador y la salida esperada.

Se dispone de varias funciones para medir este error:

- **Mean square error (MSE).** Se calcula como:

$$MSE = \frac{1}{2m} \sum_m \sum_n (SalidaClasificador_{m,n} - SalidaEsperada_{m,n})^2$$

Donde:

- m es el número de ejemplos en el conjunto de entrenamiento.

¹⁴Fuente: <https://web.stanford.edu/class/cs231a/>

- n es el número de clases en el conjunto de entrenamiento.
 - $SalidaClasificador_{m,n}$ es la probabilidad que obtenemos de nuestra red para la clase n -ésima del ejemplo m -ésimo.
 - $SalidaEsperada_{m,n}$ es la probabilidad esperada para la clase n -ésima del ejemplo m -ésimo.
- **Cross entropy error (CEE).** Se calcula como:

$$CEE = - \sum_m \sum_n \ln(SalidaClasificador_{m,n}) * SalidaEsperada_{m,n}$$

Donde:

- m es el número de ejemplos en el conjunto de entrenamiento.
- n es el número de clases en el conjunto de entrenamiento.
- $SalidaClasificador_{m,n}$ es la probabilidad que obtenemos de nuestra red para la clase n -ésima del ejemplo m -ésimo.
- $SalidaEsperada_{m,n}$ es la probabilidad esperada para la clase n -ésima del ejemplo m -ésimo.

Normalmente, el *cross entropy* es utilizado para problemas de clasificación, mientras que el *mean square error* es una de las mejores opciones para problemas de regresión.

La elección de la función con la que calcularemos el error afecta al desarrollo del entrenamiento, ya que este emplea técnicas para minimizar la función de error que utilicemos.

3.4.5. Entrenamiento de una red convolucional

Una vez tenemos una red diseñada, una técnica para inicializarla, un conjunto de datos, una forma de medir el error y evaluar nuestra red, podemos realizar el entrenamiento.

Nuestra función de error toma como entrada los parámetros de la red ($\theta \in \mathbb{R}^n$) y nos da a la salida un escalar que representa el error cometido:

$$J(\theta) : \mathbb{R}^n \rightarrow \mathbb{R}$$

Nuestro entrenamiento consiste en un problema de optimización. Debemos encontrar los valores para estos parámetros que minimicen nuestra función de error, también llamada *cost function* o *loss function*:

$$\theta^* = \arg \min J(\theta)$$

La forma de encontrar este mínimo implica utilizar el concepto de derivada de una función. Para una función $y = f(x)$, el valor de su derivada, $f'(x)$, nos proporciona la información para encontrar el mínimo (ver figura 3.18):

- $f'(x) > 0 \Rightarrow$ significa que debemos reducir el valor de nuestros parámetros para acercarnos al mínimo.
- $f'(x) < 0 \Rightarrow$ significa que debemos aumentar el valor de nuestros parámetros para acercarnos al mínimo

- $f'(x) = 0 \Rightarrow$ no nos proporciona información de cómo debemos modificar los parámetros, se trata de un óptimo local.

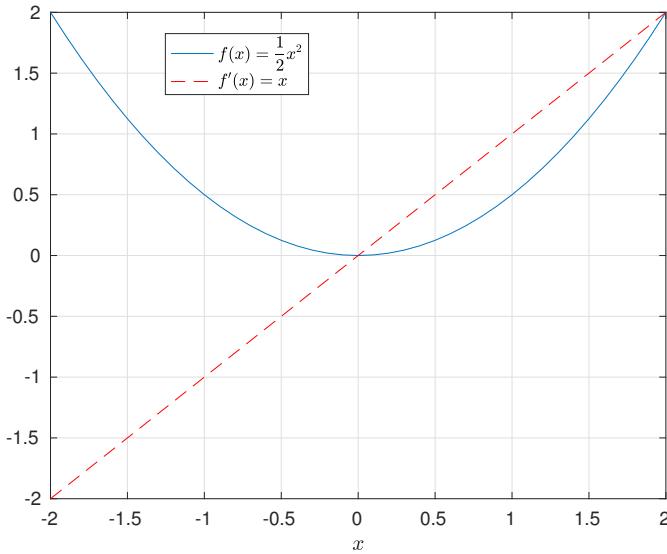


Figura 3.18: Representación gráfica de una función y su derivada.

En nuestro caso, al disponer de más de un parámetro para nuestra función de error, indicamos mediante la derivada parcial, $\frac{\partial}{\partial \theta_i} J(\theta)$, la derivada de la función con respecto al parámetro i -ésimo. Gracias al uso del gradiente, se agrupan todas las derivadas parciales de una función en un vector:

$$\nabla J(\theta) = \left[\frac{\partial}{\partial \theta_1} J(\theta), \dots, \frac{\partial}{\partial \theta_n} J(\theta) \right]$$

A la hora de buscar este mínimo se utiliza una técnica conocida como descenso por gradiente [Cauchy, 1847]. Esta técnica actualiza el valor de los parámetros un cierto factor en el sentido contrario al signo de la derivada:

$$\theta = \theta - \alpha \nabla_{\theta} J(\theta)$$

Mediante el parámetro α controlamos el tamaño del paso que se da en la actualización de los parámetros, se conoce como *learning rate*.

Cuando trabajamos con una red neuronal o convolucional, al recibir una entrada, esta se transmite desde el inicio hasta el final de la red realizando diferentes procesamientos hasta obtener la salida, es lo que se conoce como propagación hacia delante. Cuando realizamos el entrenamiento, al procesar una entrada en la red, en la salida obtenemos un error $J(\theta)$. Para ajustar los parámetros de la red, es necesario calcular el gradiente de dicho error; sin embargo, el cálculo de este es posible, únicamente, en las unidades de salida, puesto que sabemos la salida obtenida y la salida esperada y la derivada de nuestra función de error. Mientras que en las unidades ocultas de la red no sabemos cuál es el error que cometan. Como solución a este problema surge el algoritmo *backpropagation* [Werbos, 1974]. Mediante este algoritmo, el gradiente del error en la salida se propaga hacia atrás hasta las primeras unidades, de forma que podamos calcular el gradiente en las unidades ocultas de la red.

El algoritmo *backpropagation* va hacia atrás en la red, calculando el gradiente del error, δ_i^{l-1} , de la unidad i de la capa $l - 1$, a partir del error de las unidades de la capa l (con las que está conectada la unidad i). Este error se calcula como:

$$\delta_i^{l-1} = \sum_j \delta_j^l \theta_j^l$$

Esto es posible gracias a que podemos calcular el gradiente del error de la capa de salida, siendo este el punto de partida para calcular los demás. La idea intuitiva es que cada unidad es «responsable» del error que tienen cada una de las unidades a las que envía su salida y lo es en la medida que marca el peso de la conexión entre ellas.

Gracias al algoritmo *backpropagation* es posible entrenar redes neuronales profundas. Esta técnica es utilizada por el descenso por gradiente para realizar la actualización de los parámetros de la red.

El descenso por gradiente es utilizado por los algoritmos de entrenamiento como *batch gradient descent*, *stochastic gradient descent*, *mini-batch gradient descent* o *Adam*.

Batch gradient descent

El *batch gradient descent* calcula el gradiente del error utilizando todo el conjunto de entrenamiento. Debido a que necesitamos calcular el gradiente de todo el conjunto de entrenamiento para realizar una única actualización de los parámetros, el *batch gradient descent* puede ser muy lento en converger y es ineficiente para conjuntos de datos que no puedan guardarse en memoria:

```
for i in range(numIteracionesEntrenamiento):
    θ = θ - α ∇_θ J(θ)
end
```

Stochastic gradient descent

El *stochastic gradient descent* por contra, actualiza los parámetros por cada ejemplo de entrenamiento $(x^{(i)}, y^{(i)})$:

```
for i in range(numIteracionesEntrenamiento):
    θ = θ - α ∇_θ J(θ; x^{(i)}; y^{(i)})
end
```

Normalmente, se suele modificar aleatoriamente el orden de los ejemplos en cada entrenamiento. Este algoritmo es útil cuando tenemos un conjunto de datos demasiado grande, con el contra de que las continuas actualizaciones por cada ejemplo provocan una varianza alta de los valores de los parámetros, debido a no estar calculando el gradiente real, sino el de un único ejemplo.

Mini-batch gradient descent

El *mini-batch gradient descent* busca un compromiso entre lo mejor de los dos algoritmos anteriores, realizando la actualización de los parámetros cada n ejemplos de entrenamiento:

```
for i in range(numIteracionesEntrenamiento):
    θ = θ - α ∇_θ J(θ; x^{(i:i+n)}; y^{(i:i+n)})
end
```

Momento

Una novedad interesante es el uso del momento [Qian, 1999] durante el entrenamiento. El *stochastic gradient descent* tiene problemas para descender en zonas donde la curva de la superficie es mucho más abrupta en una dimensión que en otra [Sutton, 1986]. En estos casos, el *stochastic gradient descent* oscila a través de la zona estrecha, ya que el gradiente apuntará hacia uno de los lados más que al otro a lo largo de la superficie (ver figura 3.19).

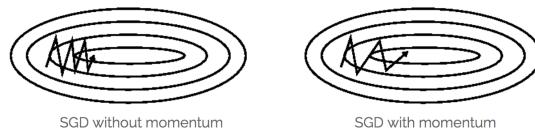


Figura 3.19: *Stochastic gradient descent* sin y con momento¹⁵.

El momento es un método que ayuda a acelerar al *stochastic gradient descent* en la dirección relevante y modera las oscilaciones. Esto ocurre gracias al añadir un factor $\gamma \in (0, 1]$ al gradiente del paso anterior:

$$\begin{aligned} v_t &= \gamma v_{t-1} + \alpha \nabla_{\theta} J(\theta) \\ \theta &= \theta - v_t \end{aligned}$$

Normalmente, se suele usar $\gamma = 0,9$ o un valor similar. Cuando usamos esta técnica, el gradiente aumenta en las dimensiones cuyos gradientes apuntan en la misma dirección, produciendo como resultado una convergencia más rápida y una reducción de la oscilación.

Además del *gradient descent* y sus variantes, existen multitud de algoritmos que se utilizan para realizar el aprendizaje. Uno de los algoritmos más populares para realizar el entrenamiento es el algoritmo *Adam*.

Adam

Adaptive Moment Estimation (Adam) [Kingma et al., 2014] es un algoritmo que calcula un *learning rate* adaptativo para cada parámetro. Se basa en el uso de los momentos de primer y segundo orden del gradiente.

Este algoritmo requiere la elección de tres parámetros: el *learning rate* (ϵ), el *decay rate* (β_1) del momento de primer orden y el *decay rate* (β_2) del momento de segundo orden.

Utiliza unos estimadores para calcular los momentos:

$$\begin{aligned} m_t &= \beta_1 m_{t-1} + (1 - \beta_1) \nabla_{\theta} J(\theta)_t \\ v_t &= \beta_2 v_{t-1} + (1 - \beta_2) \nabla_{\theta} J(\theta)_t \end{aligned}$$

Debido a que estos estimadores son inicializados a un vector de ceros, los autores de este algoritmo observaron que estaban sesgados hacia el cero, especialmente durante las primeras iteraciones. Como medida para contrarrestar este efecto calcularon los estimadores de forma que se corrigiera este sesgo:

¹⁵Fuente: http://www.datakit.cn/images/machinelearning/sgd_momentum.png

$$\hat{m}_t = \frac{m_t}{1 - \beta_1^t}$$

$$\hat{v}_t = \frac{v_t}{1 - \beta_2^t}$$

La regla de actualización de parámetros es la siguiente:

$$\theta_{t+1} = \theta_t - \frac{\alpha}{\sqrt{\hat{v}_t + \epsilon}} \hat{m}_t$$

Por lo que el pseudocódigo de este algoritmo queda como:

```
for i in range(numIteracionesEntrenamiento):
     $\theta_{t+1} = \theta_t - \frac{\alpha}{\sqrt{\hat{v}_t + \epsilon}} \hat{m}_t$ 
end
```

Los autores proponen como valores por defecto: 0,9 para β_1 , 0,999 para β_2 y 10^{-8} para ϵ . En el artículo original se muestra, de forma empírica, cómo este algoritmo es capaz de dar mejores resultados que otros métodos.

Además de los algoritmos vistos hasta ahora, existen otras técnicas que permiten mejorar los resultados y acelerar el aprendizaje, como el preprocesamiento de los datos o el *batch normalization*.

El preprocesamiento de los datos suele consistir en hacer que estos presenten una media igual a cero y una misma escala. Una forma de conseguirlo es restarle a cada dato la media y dividirlo por la desviación típica de dichos datos:

$$\hat{x}_j^{(i)} = \frac{x_j^{(i)} - E[X_j]}{\sqrt{Var[X_j]}}, \quad x_j^{(i)} \in X_j$$

Estas condiciones ayudan al entrenamiento tal y como refleja la figura 3.20, donde el preprocesamiento permite encontrar el mínimo de una forma más rápida. Durante el entrenamiento, el gradiente del error es usado para actualizar los parámetros, si trabajamos con diferentes escalas, la corrección para algún parámetro en una dimensión puede diferir (proporcionalmente hablando) a la de otra, provocando, como se vio anteriormente, la aparición de oscilaciones.

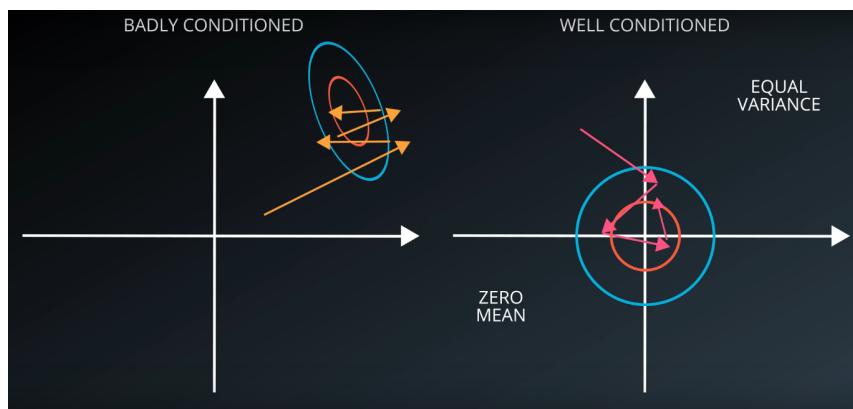


Figura 3.20: Efecto del preprocesamiento en el entrenamiento¹⁶.

¹⁶Fuente: <https://www.udacity.com/course/deep-learning--ud730>

Es por ello que a la hora de trabajar con nuestras imágenes aplicaremos este preprocesamiento para facilitar el entrenamiento. En la figura 3.21 podemos ver el efecto del preprocesamiento sobre los datos por etapas.

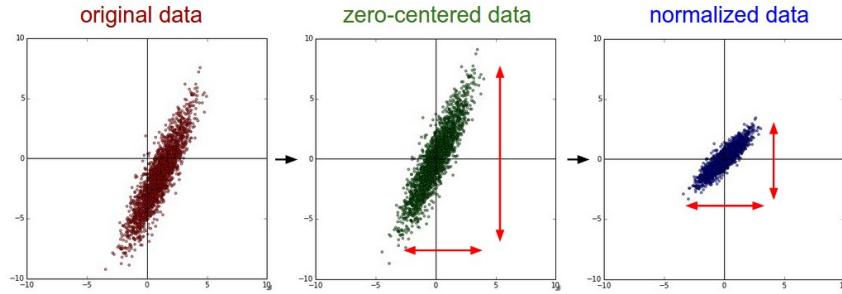


Figura 3.21: Efecto del preprocesamiento en los datos¹⁷.

Otra de las técnicas más usuales para facilitar el entrenamiento es el *batch normalization* [Ioffe et al., 2015]. Surge como solución al problema conocido como *internal covariate shift* (descrito en [Ioffe et al., 2015]) que aparece durante el entrenamiento como efecto de los diferentes procesamientos que hacen las capas de la red a los datos; provocando que estos sean demasiado grandes o demasiado pequeños, además de posibles traslaciones.

Esta técnica se utiliza para realizar el preprocesamiento de los datos en la cualquier capa de la red, en lugar de una única vez al principio. Esto provoca que el *batch normalization* se establezca como una capa más de la red.

El pseudocódigo de este algoritmo se encuentra en la figura 3.22. Básicamente, realiza la normalización de los datos de entrada de la capa actual y a continuación aplica una función lineal con los parámetros γ y β . Esta técnica trabaja solo con un conjunto de datos del conjunto original al igual que el *mini-batch gradient descent*. Al tener que aprender los valores para los parámetros γ y β , si la capa actual prefiere usar los datos originales sin preprocesar, solo tiene que asignar $\gamma = \sqrt{\text{Var}[X]}$ y $\beta = E[X]$.

Input: Values of x over a mini-batch: $\mathcal{B} = \{x_1 \dots m\}$; Parameters to be learned: γ, β Output: $\{y_i = \text{BN}_{\gamma, \beta}(x_i)\}$
$\mu_{\mathcal{B}} \leftarrow \frac{1}{m} \sum_{i=1}^m x_i \quad // \text{mini-batch mean}$ $\sigma_{\mathcal{B}}^2 \leftarrow \frac{1}{m} \sum_{i=1}^m (x_i - \mu_{\mathcal{B}})^2 \quad // \text{mini-batch variance}$ $\hat{x}_i \leftarrow \frac{x_i - \mu_{\mathcal{B}}}{\sqrt{\sigma_{\mathcal{B}}^2 + \epsilon}} \quad // \text{normalize}$ $y_i \leftarrow \gamma \hat{x}_i + \beta \equiv \text{BN}_{\gamma, \beta}(x_i) \quad // \text{scale and shift}$

Figura 3.22: Algoritmo *batch normalization*¹⁸.

En la siguiente sección hablaremos de las diferentes técnicas que existen para poder controlar nuestro entrenamiento, además de cómo se suele abordar todas las decisiones que debemos tomar para el desarrollo de nuestro sistema.

¹⁷Fuente: <http://cs231n.github.io/neural-networks-2/>

¹⁸Fuente: <https://arxiv.org/pdf/1502.03167.pdf>

3.4.6. Hiperparámetros y control del entrenamiento

Tanto para el diseño de la red como para su entrenamiento hay multitud de hiperparámetros que debemos asignar. Desde el tipo, número, disposición y parámetros de cada capa, a los parámetros del algoritmo de entrenamiento que estemos utilizando.

No se trata de una elección trivial, ya que condicionará los resultados que obtengamos. El diseño de la red, como se mencionó anteriormente, depende de nuestra experiencia, al no existir una metodología que podamos seguir. Sin embargo, una vez elijamos un algoritmo de entrenamiento, podremos analizar la evolución de diferentes valores de referencia durante el entrenamiento para poder detectar si este se realiza correctamente.

Lo primero que necesitaremos es establecer una estructura para nuestra red. Normalmente, la elección de la estructura se hace partiendo de alguna red conocida que presente buenos resultados en tareas similares. A partir de una estructura de referencia se pueden añadir nuevas capas o modificar las existentes. Estos cambios pueden venir motivados por el deseo de mejorar los resultados o conseguir resultados similares a los de la red original pero con una estructura más reducida y, por tanto, con menos exigencias de procesamiento. Una de las decisiones más habituales consiste en diseñar redes con la mayor profundidad (número de capas) posible, aunque se reduzca el número de operaciones en cada capa. Experimentalmente, se ha comprobado que se obtienen mejores resultados siguiendo esta estrategia (ver [Goodfellow et al., 2014]), como se puede observar en la figura 3.23.

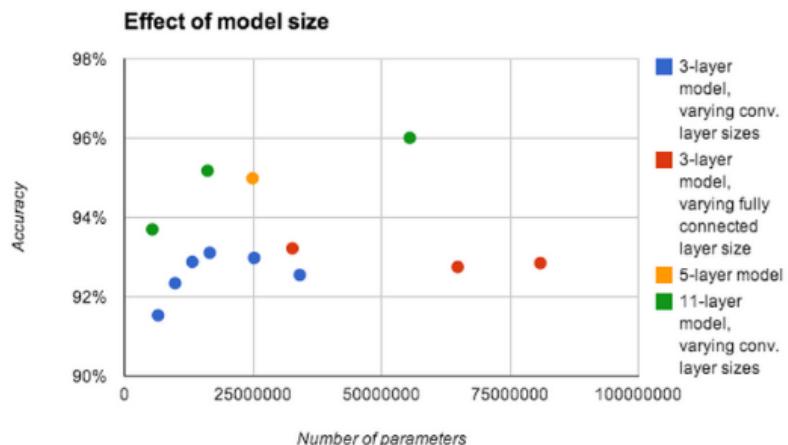


Figura 3.23: Comparativa entre el número de parámetros, el número de capas y el rendimiento de la red¹⁹.

Una vez elegida una estructura para la red deberemos elegir el algoritmo que utilizaremos para el entrenamiento. Es recomendable probar diferentes algoritmos sobre la red para ver cuál puede dar mejores resultados. La elección de los valores para los parámetros del algoritmo dependerán del problema que estemos tratando. Sin embargo, siempre podemos partir de los valores usados por otros investigadores e ir analizando la evolución del entrenamiento, pudiendo así modificarlos en caso de que no se obtengan buenos resultados.

En definitiva, se trata de un proceso de ensayo y error donde deberemos realizar una experimentación para ver si hemos mejorado lo que ya teníamos o debemos probar otra alternativa.

¹⁹ Fuente: <https://arxiv.org/pdf/1312.6082.pdf>

La forma que tenemos de detectar si el entrenamiento se ha desarrollado correctamente es analizando la evolución del error durante el mismo. Si visualizamos el error en función de las iteraciones del entrenamiento, deberíamos observar cómo este va disminuyendo a medida que avanza el entrenamiento. Sin embargo, según el valor del *learning rate* elegido (además de los otros parámetros que use el algoritmo), puede que nuestro error no presente el comportamiento que hemos descrito. En la figura 3.24 se muestra las formas más comunes en las que podemos encontrarnos la evolución del error durante el entrenamiento.

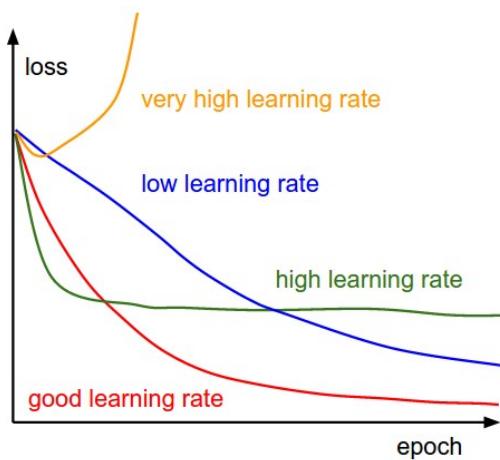


Figura 3.24: Comportamiento del error durante el entrenamiento según el *learning rate* elegido²⁰.

Debemos encontrar los valores del *learning rate* que hacen que nuestro error se vaya reduciendo a medida que avanza el entrenamiento. Una alternativa existente para controlar la disminución del error consiste en la reducción del *learning rate* durante el entrenamiento, de forma que, cuando se acerque al final del entrenamiento, este tenga un valor más pequeño que al principio. Con esta técnica se pretende que la actualización de los pesos de la red no sea tan «violenta» como al principio, de forma que mantenga lo aprendido durante las primeras iteraciones del entrenamiento. Una de las ventajas de usar *batch normalization* es que, al realizar normalizaciones entre capas, evita que se termine saturando la señal de entrada, lo que nos permite utilizar valores más altos para el *learning rates* (ver [Ioffe et al., 2015]). Hay que tener en cuenta que no siempre la modificación del *learning rate*, cuando no se aprecia una reducción de error significativa, tiene que suponer una buena decisión, aunque suponga una reducción más pronunciada del error; como se muestra en la figura 3.25, al reducir el *learning rate* conseguimos un mejor resultado momentáneo; pero a la larga, el valor anterior nos hubiera reportado mejores resultados.

²⁰Fuente: <http://cs231n.github.io/neural-networks-3/>

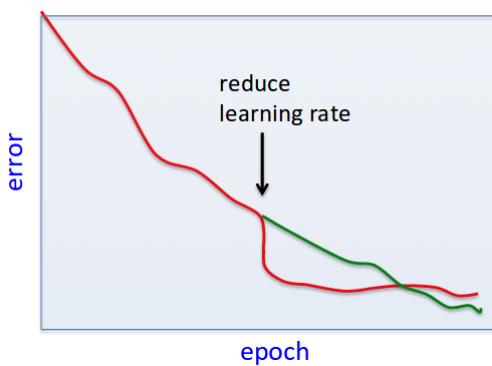


Figura 3.25: Efecto sobre el error a largo plazo por el cambio del *learning rate*²¹.

Cuando encontramos un valor adecuado para el *learning rate* debemos asegurarnos de que, a pesar de que el error durante el entrenamiento se va reduciendo, esto implique una mejora en los resultados que obtenemos del sistema. Como se expuso en capítulos anteriores, el conjunto de validación y test nos permitirán saber si nuestro sistema se está comportando correctamente. Comparando el error obtenido en el conjunto de entrenamiento con el error en el conjunto de validación o test podemos analizar cuándo nuestro sistema está generalizando correctamente o cuándo se produce sobreajuste. Se dice que nuestra red está sobreajustada si da muy buenos resultados con el conjunto de entrenamiento y unos resultados demasiado distantes a los del entrenamiento en el conjunto de evaluación. El sobreajuste implica que el sistema ha «sobreaprendido» los datos del conjunto de entrenamiento, siendo incapaz de generalizar correctamente y produciendo malos resultados cuando introducimos nuevos datos. En la figura 3.26 podemos ver un ejemplo de tres modelos: el primero, a la izquierda, está sin ajustar; el de en medio, es un ejemplo de una buena generalización a partir del conjunto de entrenamiento; el de la derecha, presenta sobreajuste, al haber intentado reducir demasiado el error del conjunto de entrenamiento, ha provocado que no sea capaz de generalizar correctamente.

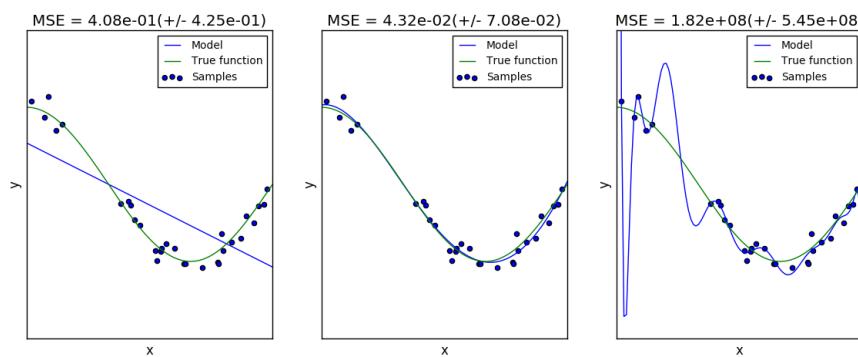


Figura 3.26: Comparativa de modelo sin ajustar, ajustado correctamente y con sobreajuste²².

Como se mencionó anteriormente, debemos analizar la evolución del error en ambos conjuntos para poder detectar un posible sobreajuste de nuestra red. Este es fácil de detectar si analizamos cómo evoluciona el error en el tiempo, comparando el conjunto de entrenamiento y de evaluación. En la figura 3.27 se muestra un ejemplo de esta comparación. El error en el conjunto de entrenamiento, con el paso de las iteraciones, comienza a presentar valores muy dispares a los del conjunto de validación, lo que suele suponer un síntoma de que nuestra red está sobreajustada.

²¹Fuente: http://www.cs.toronto.edu/~tijmen/csc321/slides/lecture_slides_lec6.pdf

²²Fuente: http://scikit-learn.org/stable/_images/sphx_glr_plot_underfitting_overfitting_001.png

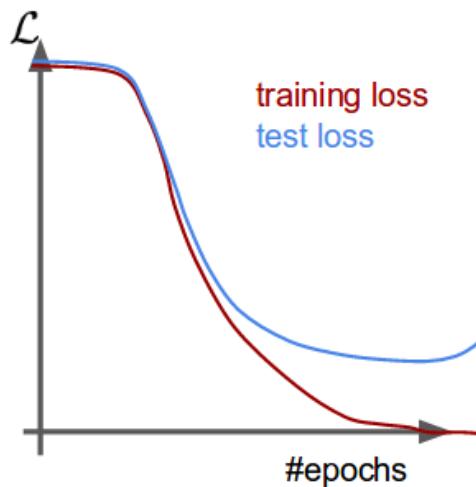


Figura 3.27: Detección del sobreajuste mediante el análisis de la función de error²³.

Las métricas de evaluación sobre ambos conjuntos también nos puede ayudar a detectar sobreajuste (ver figura 3.28).

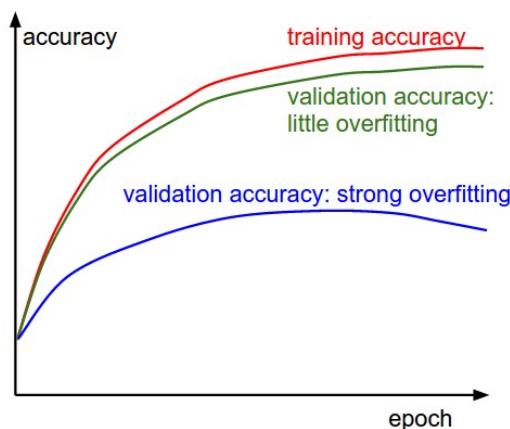


Figura 3.28: Detección del sobreajuste mediante el análisis del *accuracy*²⁴.

Para combatir el sobreajuste se suele hacer uso de una técnica muy popular conocida como *dropout* [Srivastava et al., 2014]. Esta técnica se basa en la idea de promediar los resultados de varias redes. Para lograrlo, aleatoriamente, «desactiva» de la red algunas de las unidades durante la propagación hacia delante. Esto implica que, si tenemos n unidades ocultas, al desactivar algunas aleatoriamente, podemos obtener hasta 2^n redes más pequeñas. Para cada iteración del entrenamiento, se escoge una de estas posibles redes y se entrena. De forma que, entrenar una red con esta técnica puede verse como entrenar una colección de 2^n redes más pequeñas. En la etapa de inferencia, no se utiliza el *dropout*, utilizamos la red completa, lo que provoca que obtengamos el resultado de nuestra red como una ponderación de las diferentes subredes que fueron entrenadas durante el entrenamiento (ver [Srivastava et al., 2014]). Gracias a esto, podemos controlar mejor el sobreajuste, al provocar que exista una redundancia en la red de características aprendidas y que se obtenga la salida como una ponderación de lo aprendido por las subredes, las cuales han sido entrenadas de forma independiente.

²³Fuente: <http://www.psi.toronto.edu/~jimmy/ece521/Lec10-nn3.pdf>

²⁴Fuente: <http://cs231n.github.io/neural-networks-3/>

Una vez que observamos que el entrenamiento transcurre correctamente, debemos decidir el tiempo que dejaremos el sistema entrenando. Mediante el análisis de diferentes métricas, podremos detectar cuándo tenemos unos resultados aceptables y cuándo nos conviene parar para evitar el sobreajuste. Si nuestra red lleva un tiempo sin apenas mejorar es mejor detener el entrenamiento, puede que para conseguir mejores resultados tengamos que modificar la estructura de la red, los parámetros del algoritmo o incluso los umbrales del sistema para dar la salida. Una estrategia que se sigue, con objetivo también de evitar el sobreajuste, es detener el entrenamiento cuando las métricas del conjunto de evaluación no mejoran; se conoce como *early stopping* (ver figura 3.29).

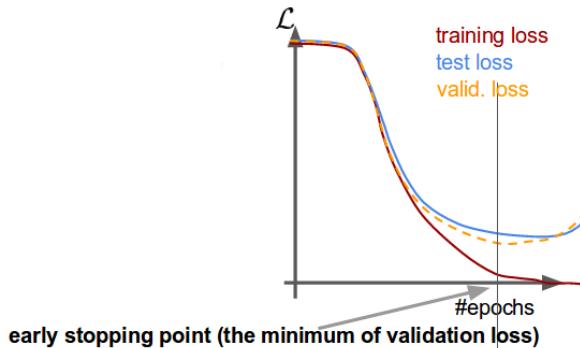


Figura 3.29: Técnica *early stopping*²⁵.

El hecho de elegir nuestro modelo cuando se produce el mínimo global de la función de error, no implica que este sea el más idóneo. Generalmente, preferiremos los mínimos que se encuentren en una zona con resultados similares y no mínimos aislados, ya que tienen una menor capacidad de generalizar (ver figura 3.30).

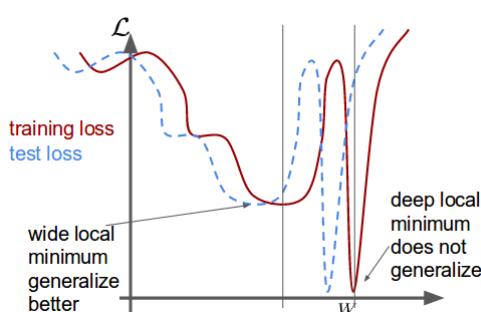


Figura 3.30: Diferencias en la generalización según el mínimo elegido²⁶.

Debemos tener presente todas las recomendaciones anteriores para lograr entrenar nuestra red correctamente.

El hecho de que existan redes que hayan obtenido buenos resultados en algunas tareas, como las vistas en el apartado 3.3, puede invitarnos a usarlas por delante de la que diseñemos, más incluso si la red que diseñemos nos reporta peores resultados que la que usamos como referencia en otros problemas. No obstante, nuestro modelo es una representación simplificada de la realidad, lo que supone descartar detalles innecesarios. Estas simplificaciones se basan en suposiciones; estas suposiciones pueden mantenerse en algunas situaciones, pero no pueden mantenerse en otras. Esto implica que un modelo que explica una cierta situación bien puede fallar en otra distinta. Debemos comprobar nues-

²⁵Fuente: <http://www.psi.toronto.edu/~jimmy/ece521/Lec10-nn3.pdf>

²⁶Fuente: <http://www.psi.toronto.edu/~jimmy/ece521/Lec10-nn3.pdf>

etros supuestos antes de confiar en un modelo. Cabe destacar el teorema conocido como *no free lunch* [Wolpert et al., 1997]. Este teorema dice que no existe un modelo que sea el mejor para todos los problemas. Por tanto, esto nos puede servir de motivación para encontrar una red que nos proporcione mejores resultados que alguna otra ya existente, en nuestro problema concreto.

3.4.7. Obtención del modelo para la etapa de inferencia

Tal y como se vio en la Sección 2.1, se distinguen dos etapas cuando estamos trabajando con *machine learning*. Hasta ahora hemos hablado de la etapa de entrenamiento, cuyo objetivo es proporcionarnos un modelo para usarlo en la etapa de inferencia.

Nuestro modelo entrenado consiste en una estructura de red predefinida con unos valores asignados durante el entrenamiento en las capas correspondientes. Para la fase de inferencia debemos eliminar de la red aquellas capas que no intervengan en la obtención de la salida (capas que calculen métricas p. ej), ya que, en esta fase, no se realizan actualizaciones de los parámetros como en la fase de entrenamiento.

El objetivo de esta fase es utilizar el modelo para obtener la salida ante un dato de entrada lo más rápido posible. Es común suministrar las entradas por lotes, para mejorar la velocidad de ejecución.

Podemos ver ambas fases como un proceso iterativo, donde puede que tengamos que volver a la fase de entrenamiento cuando no obtengamos los resultados deseados para mejorar el modelo.

CAPÍTULO 4

Marco de aplicación

Con el objetivo de llevar a la práctica todos los conocimientos teóricos estudiados, presentamos, en este capítulo, las nociones necesarias para la creación de estos sistemas; enfocándonos en el etiquetado, detección y segmentación.

4.1 Etiquetado en imágenes

El etiquetado de imágenes es una de las tareas más comunes que podemos encontrar a la hora de trabajar con inteligencia artificial aplicada al tratamiento de imágenes. Esta consiste en un problema de clasificación, donde debemos asignar una o más etiquetas a la imagen de entrada; de forma que, en base a estas etiquetas, podamos saber el contenido de la imagen. Normalmente, estas etiquetas pueden representar objetos que deseamos detectar en la imagen, aunque también puede representar otros tipos de conceptos, como una descripción de la misma (naturaleza, invierno, playa...).

Su interés puede verse motivado por multitud de aplicaciones de estos sistemas. Como se mencionó en la Sección 2.2, la primera red convolucional que se diseñó fue utilizada para el reconocimiento de dígitos escritos a mano, con el objetivo de clasificar cartas con direcciones escritas a mano. En la figura 4.1 se muestra la estructura principal de una red convolucional para la clasificación de dígitos.

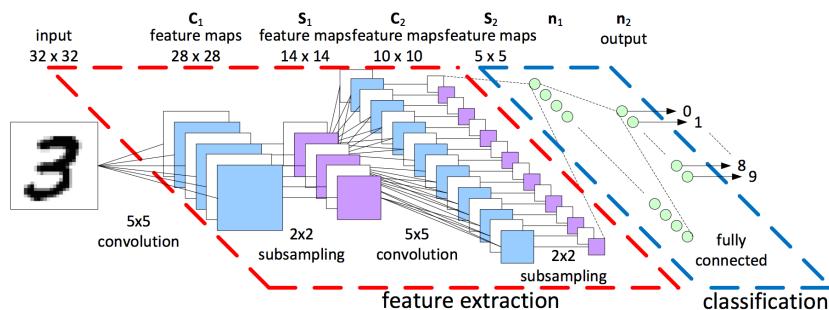


Figura 4.1: Red convolucional para la clasificación de dígitos¹.

Como se observa en la figura anterior, el sistema recibe como entrada una imagen y nos da a la salida la probabilidad de que la imagen de entrada sea alguna de las etiquetas que hemos creado en el sistema (en este caso los dígitos del 0 al 9).

Una vez que se definen las etiquetas que dispondrá nuestro sistema a la salida, deberemos recoger imágenes para su entrenamiento; asignando, según el formato exigido

¹Fuente: <https://www.kernix.com/doc/data/cnn.png>

por el entorno donde se desarrolle el sistema, las etiquetas correspondientes a cada una de las imágenes que disponemos.

Las imágenes de entrada al sistema suelen ser de tamaño fijo. Esto es debido al uso en las últimas capas de la red de unidades neuronales. Las unidades neuronales reciben como entrada un mapa de características, resultado de aplicar todas las operaciones anteriores a dicha capa. A diferencia de la convolución, que puede aplicarse a diferentes tamaños de imagen, la red neuronal necesita establecer el número de conexiones con la entrada, y esto se establece en función del tamaño de la entrada, ya que cada elemento en el mapa de características está conectado con la entrada de la red neuronal por lo que se necesita un tamaño fijo para establecer estas conexiones. Como consecuencia, se suele ajustar el tamaño de la imagen de entrada a un tamaño predefinido. Para solucionar este inconveniente, unos investigadores de Microsoft propusieron el *spatial pyramid pooling* (*SPP*) [He et al., 2014]. Esta técnica permite a una red convolucional recibir una entrada de cualquier tamaño. Para ello, añade una capa de *SPP* justo antes de la red neuronal. Esta capa realiza una serie de operaciones de pooling (típicamente *max-pooling*) sobre una entrada de cualquier tamaño, resultado de la última capa de convolución (compuesta de k filtros). En la capa *SPP* no se define los parámetros para la operación de *pooling* sino el tamaño deseado de salida, que será calculado por el número de valores en los que quedará reducida cualquiera de los canales del mapa de características de entrada, M valores de salida para cada canal; y del número de filtros aplicados en la capa anterior, o lo que es lo mismo, el número de canales, k , de la entrada de esta capa; es decir, el tamaño de salida será $k * M$. En la figura 4.2 se muestra una capa de *SPP* que recibe como entrada un mapa de características de 256 canales, que será reducido a 21 valores: 16 de dividir la entrada en 16 regiones, 4 de dividir la entrada en 4 regiones y un valor de la imagen completa. A estas divisiones de la imagen se les aplican la operación de *pooling* para obtener estos valores, dando como resultado $21 * 256$ valores a la salida, independientemente de la entrada dada. Esto permite establecer el número de conexiones necesarias para la red neuronal y a su vez trabajar a la entrada con cualquier tamaño.

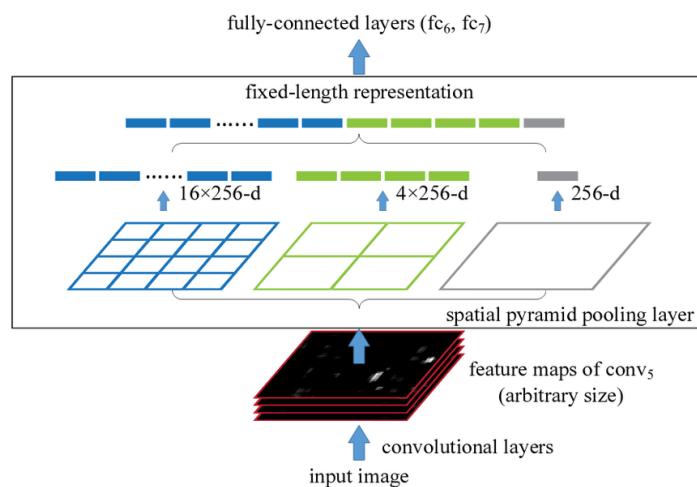


Figura 4.2: Ejemplo de funcionamiento de la capa *SPP*².

Para este tipo de sistemas, como se describió en la Sección 3.4.4, suele utilizarse como función de error el *cross entropy error*. El desarrollo y control de entrenamiento se realiza siguiendo las directrices indicadas en la Sección 3.4.6.

²Fuente: <https://arxiv.org/pdf/1406.4729.pdf>

Las métricas más comunes que se suelen dar para este tipo de sistemas suelen ser el *accuracy*, *recall* y *precision*.

Esta tecnología es útil en aquellos escenarios donde es necesario reconocer la presencia o no de determinados objetos en la imagen. Algunas compañías de redes sociales usan esta tecnología para sugerir a los usuarios posibles etiquetas para acompañar a las imágenes que comparten en dichas plataformas.

4.2 Detección en imágenes

El siguiente nivel de dificultad en el procesamiento de imágenes consiste en determinar, además de la presencia de determinados objetos en una imagen, la localización de los mismos. Este es el objetivo de un sistema para la detección en imágenes.

El sistema deberá, dada una imagen de entrada, dar a la salida la posición de los objetos que hay en la imagen y a qué clase corresponde. El formato exacto de la salida de la red dependerá del tipo de red implementada. La posición de los objetos se suele dar mediante 4 valores que permiten describir el recuadro que los engloba: x_{min} , y_{min} , x_{max} , y_{max} , referidos a la imagen de entrada. En la figura 4.3 se muestra el resultado que se espera obtener de este tipo de sistemas.

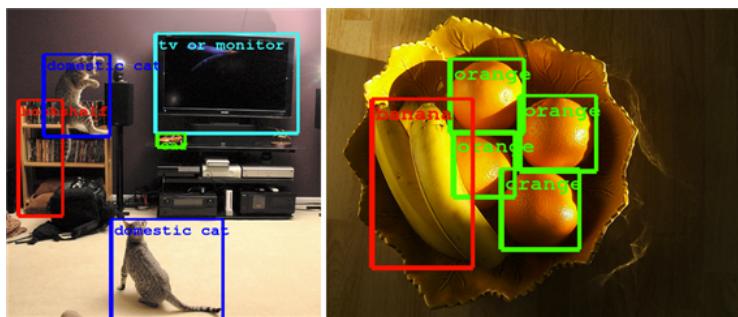


Figura 4.3: Ejemplo de salida de un sistema de detección en imágenes³.

El entrenamiento de estas redes exige, además de la clase a la que pertenece cada objeto presente en la imagen, el rectángulo que engloba a cada uno de ellos. La forma en la que se deban almacenar estos valores dependerá del entorno de trabajo donde se esté desarrollando el sistema.

La detección de objetos es un problema que se ha abordado en inteligencia artificial desde hace mucho tiempo. El enfoque tradicional consistía en emplear una técnica conocida como ventana deslizante. Esta técnica escala la imagen a diferentes tamaños y sobre cada uno extrae un recorte o ventana con un tamaño prefijado, desplazándola por cada imagen según un desplazamiento prefijado. Esto provoca que se obtengan una serie de ventanas, todas del mismo tamaño, de la imagen original (ver figura 4.4). Cada una de estas ventanas serán los posibles candidatos a albergar un objeto en su interior. A continuación, a cada una de las ventanas se les extraen una serie de características, normalmente empleando algún algoritmo para ello (HOG [Dalal et al., 2005] p. ej.). Estas características son suministradas a un algoritmo de clasificación como redes neuronales o SVM.

³Fuente: http://ncia.snu.ac.kr/xe/research_H



Figura 4.4: Ejemplo del algoritmo de la ventana deslizante⁴.

Con este proceso obtendríamos una serie de recuadros alrededor de los objetos de la imagen (correspondiente a las ventanas que han sido clasificadas como contenedoras de algún objeto a detectar). Para agrupar las detecciones de un mismo objeto en una única detección se emplean algoritmos de agrupamiento. Uno de los algoritmos más populares para la agrupación de rectángulos es el *group rectangles*. Mediante este algoritmo se agrupan rectángulos según un criterio de equivalencia que combina recuadros con tamaño y localización similar. Esta similitud se asigna mediante un parámetro, δ , que se calcula como:

$$\delta = \epsilon * \frac{\min(w_1, w_2) + \min(h_1, h_2)}{2}$$

Donde:

- w_i : es la anchura del rectángulo i .
- h_i : es la altura del rectángulo i .
- ϵ : es un parámetro para ajustar el valor de δ (normalmente se establece a 0,2).

Dos rectángulos se fusionan (calculando la media de ambos) si alguna de las 4 distancias de la figura 4.5 son menores o iguales que δ .

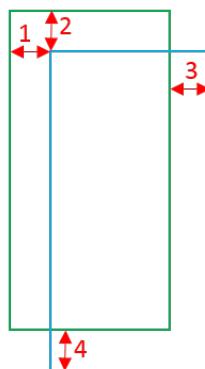


Figura 4.5: Distancias a comparar con δ ⁵.

Una vez se han fusionado los rectángulos similares, se eliminan aquellos que estén contenidos en otros (ver figura 4.7). Para ello, se definen dos valores:

⁴Fuente: https://courses.engr.illinois.edu/cs543/sp2011/lectures/Lecture%20-%20Sliding%20Window%20Detection%20-%20Vision_Spring2011.pdf

⁵Fuente: <http://mccormickml.com/2013/11/07/opencv-hog-detector-result-clustering/>

$$\begin{aligned} dx &= \epsilon * w_2 \\ dy &= \epsilon * h_2 \end{aligned}$$

Si se cumple la condición expresada en la figura 4.6, el recuadro pequeño se elimina (n_1 y n_2 son pesos que se pueden asignar a cada rectángulo para darle mayor o menor importancia).

```
x1 >= (x2 - dx) &&
y1 >= (y2 - dy) &&
(x1 + w1) <= (x2 + w2 + dx) &&
(y1 + h1) <= (y2 + h2 + dy) &&
(n2 > max(3, n1) || n1 < 3)
```

Figura 4.6: Condición para eliminar un recuadro contenido en uno mayor⁶.

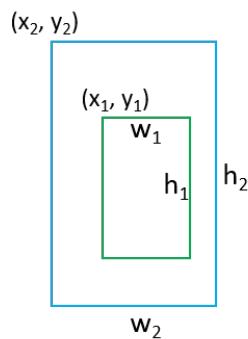


Figura 4.7: Rectángulo englobado por otro mayor⁷.

También es posible establecer un tamaño mínimo y máximo para las detecciones, y un mínimo de recuadros en una detección para considerarla válida, lo que dota de robustez al sistema. En la figura 4.8 se muestra un ejemplo del comportamiento de este algoritmo.

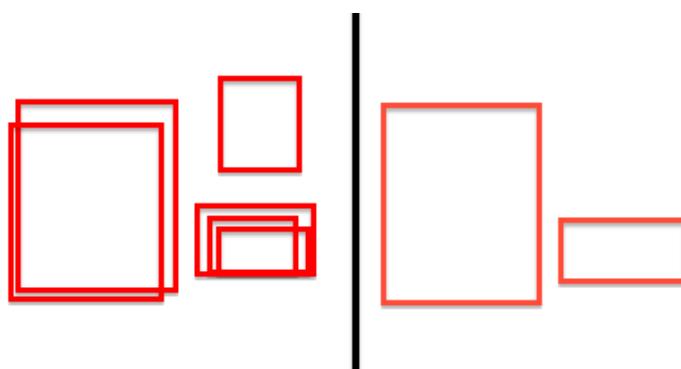


Figura 4.8: Ejemplo de agrupamiento mediante el algoritmo *group rectangles*⁸.

⁶Fuente: <http://mccormickml.com/2013/11/07/opencv-hog-detector-result-clustering/>

⁷Fuente: <http://mccormickml.com/2013/11/07/opencv-hog-detector-result-clustering/>

⁸Fuente: <https://stackoverflow.com/questions/21421070/opencv-grouprectangles-getting-grouped-and-ungrouped-rectangles>

La aplicación de redes convolucionales a este tipo de problemas supone la integración de todo este proceso en el funcionamiento de la red. Una de las primeras aplicaciones de las redes convolucionales para la detección en imágenes fue la red *R-CNN* [Girshick et al., 2013]. Esta red define una serie de regiones en base a recuadros, que deben ser analizados por una red convolucional que haga la clasificación de estas regiones. Una vez se obtiene todas las posibles regiones con objetos, estas pueden no contener al objeto correctamente delimitado, debido a que son regiones predefinidas. Para corregir este problema se hace uso de una técnica llamada *bounding-box regression*, explicada en detalle en [Girshick et al., 2013]; busca aprender la transformación necesaria a aplicar a los recuadros clasificados de alguna clase para que contengan al objeto completamente, ya que, en un principio, estos recuadros están prefijados en la red y puede que no engloben al objeto correctamente; esta técnica realiza un aprendizaje para dar el recuadro correctamente, usando, para ello, el mapa de características de la última capa de convolución, el recuadro obtenido por la red y el recuadro esperado o verdadero. Esta red se ha ido optimizando con el paso del tiempo gracias a nuevas ideas que han permitido mejorar los resultados y hacerla más eficiente (ver [Ren et al., 2015]).

Recientemente, otras redes como YOLO [Redmon et al., 2015] han enfocado el problema desde otra perspectiva, tratando, la detección de objetos en imágenes, como un problema de regresión, donde existen, de forma separada, rectángulos y las probabilidades asociadas a cada clase. Una red convolucional es encargada de dar los rectángulos y las probabilidades de las clases para estos rectángulos directamente de la imagen y con una sola evaluación. Esta red divide la salida de la red en una serie de recuadros de tamaño $S \times S$. Si el centro de un objeto cae dentro de una de estas celdas, es responsabilidad de esta celda detectar dicho objeto. Cada celda predice B rectángulos y una puntuación de la confianza de dicha detección. Buscamos que esta confianza sea cero si no hay un objeto en la celda o, en caso contrario, sea igual a la IOU (métrica utilizada para calcular la similitud entre rectángulos [ver figura 4.9]) entre el rectángulo predicho y el rectángulo verdadero.

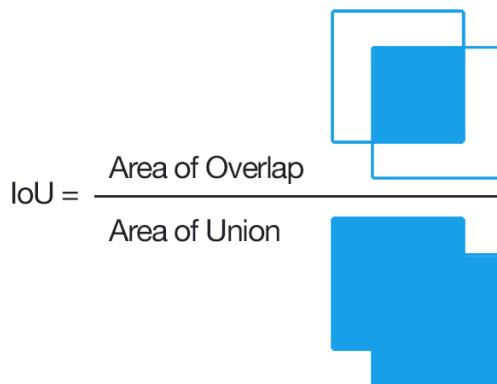


Figura 4.9: Cálculo de la métrica *Intersection over Union* (IOU)⁹.

Cada predicción del rectángulo consiste en 5 valores: x, y, w, h , y una confianza. La coordenada (x, y) representa el centro del rectángulo relativo a los límites de la celda. El alto y el ancho se dan respecto a la imagen completa. Cada celda predice también C probabilidades condicionales, $Pr(\text{Clase}_i | \text{Objeto})$, tantas como clases haya. Esta probabilidad está condicionada al objeto contenido en la celda. Solo se predice un conjunto de probabilidades de clases por celda, independientemente del número de rectángulos B . Estas predicciones son codificadas como un tensor $S \times S \times (B * 5 + C)$. En la etapa de

⁹ Fuente: <http://www.pyimagesearch.com/2016/11/07/intersection-over-union-iou-for-object-detection/>

inferencia se multiplica la probabilidad condicionada de cada clase y la confianza en la predicción de cada rectángulo:

$$Pr(Clase_i|Objeto) * Pr(Objeto) * IOU_{prediccion} = Pr(Clase_i) * IOU_{prediccion}$$

De esta forma, ya que cada celda puede dar incluso más de un rectángulo, obtendremos una serie de rectángulos sobre la imagen. Para la etapa de inferencia queremos quedarnos solo con un rectángulo para cada objeto, para ello se asigna un «predictor» responsable de cada objeto. Este predictor para cada objeto se asigna en base a cuál de ellos tiene, durante el entrenamiento, mayor IOU con la predicción verdadera. En base a esta red, los desarrolladores de Nvidia Digits presentaron una red llamada DetectNet. Usando la GoogleNet, esta red aplica la misma técnica que la red YOLO, aunque el agrupamiento de las detecciones se realiza con el algoritmo *group rectangles*. El objetivo de esta red es, obtener, para cada celda a la salida, si un objeto está presente en la celda y donde está las esquinas del rectángulo de este objeto en relación al centro de la celda. En la figura 4.10 se puede observar de forma esquemática la estructura de la red.

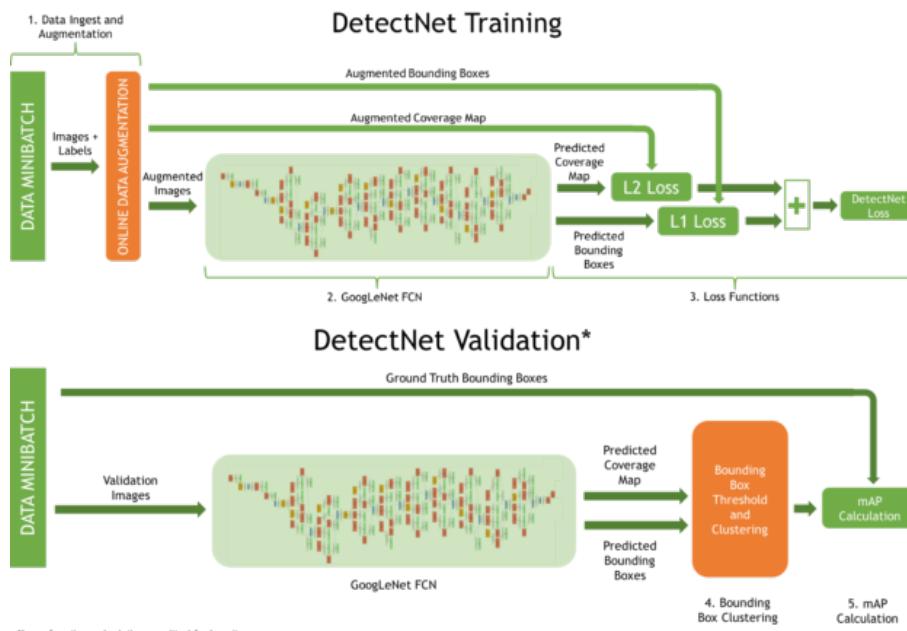


Figura 4.10: Esquema de la red DetectNet¹⁰.

Para el entrenamiento de estas redes necesitamos una forma diferente de calcular el error, ya que ni a la salida ni en el conjunto de entrenamiento tenemos vectores de probabilidades, sino celdas con las coordenadas del rectángulo que engloban y la clase a la que pertenece dicho rectángulo. Para la red DetectNet, el error es calculado como la suma de dos errores:

- *coverage loss*: calculado como la suma de las diferencias al cuadrado entre la salida obtenida donde se encuentra la probabilidad asociada a cada clase para cada celda, $celda_i^p$ y la probabilidad esperada, $celda_i^t$.

$$\frac{1}{2N} \sum_{i=1}^N |celda_i^t - celda_i^p|^2$$

¹⁰Fuente: <https://devblogs.nvidia.com/parallelforall/detectnet-deep-neural-network-object-detection-digits/>

- *bbox loss*: es la diferencia absoluta de la media para las esquinas del rectángulo predicho ($x_1^p, y_1^p, x_2^p, y_2^p$), y el verdadero ($x_1^t, y_1^t, x_2^t, y_2^t$) para los objetos detectado por cada celda.

$$\frac{1}{2N} \sum_{i=1}^N [|x_1^p - x_1^t| + |y_1^p - y_1^t| + |x_2^p - x_2^t| + |y_2^p - y_2^t|]$$

El entrenamiento consistirá en minimizar la suma de estos dos errores.

Para poder calcular este error, es necesario transformar el conjunto de entrenamiento, de forma que podamos obtener la información esperada en cada celda. En la figura 4.11 podemos ver como se hace esta transformación.

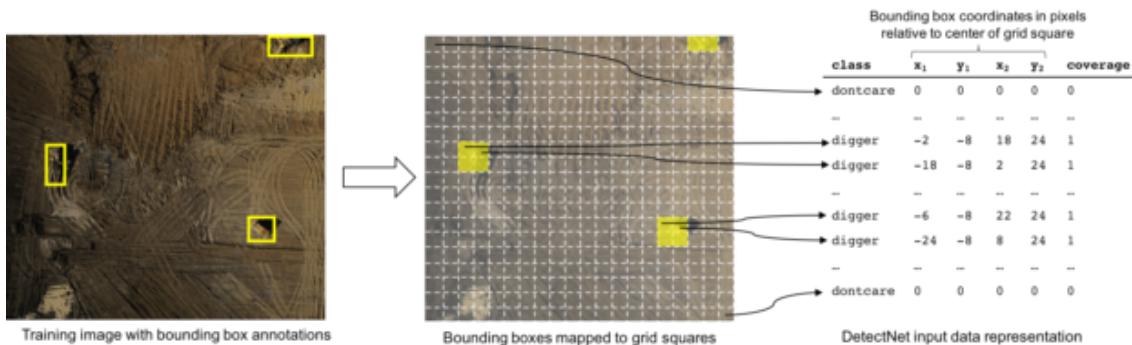


Figura 4.11: Transformación de los datos para entrenar la red DetectNet¹¹.

A diferencia de las redes convolucionales aplicadas al etiquetado, en este tipo de sistemas obtendremos los recuadros de los objetos junto a la clase del mismo. Debido a esto, para considerar un verdadero positivo como tal, se exige un valor mínimo para el IOU con el rectángulo del conjunto de entrenamiento (0,7 p. ej.). Para este tipo de problema no se suele hablar de verdaderos negativos, ya que, serían irrelevantes, al ser demasiados. Esto hace que se usen, como principales métricas, el *recall* y el *precision*.

La detección en imágenes permite el diseño de sistemas que puedan interactuar con el entorno a través de actuadores. Con el uso de estas redes es posible estudiar el comportamiento de los objetos que detecta e interaccionar con ellos. Cabe destacar todo el interés que suscitan estos sistemas para ser implementados en fábricas o robots que realicen tareas de manipulación, aprovechando el potencial que otorga el *deep learning*.

4.3 Segmentación en imágenes

El siguiente problema que podemos abordar usando redes convolucionales aplicadas al tratamiento de imágenes es la segmentación. Este problema supone un paso más allá dado por la detección, donde debemos encontrar qué píxeles hay en la imagen pertenecientes a las clases definidas en nuestro problema.

Tradicionalmente, este problema se ha abordado haciendo uso de técnicas de *clustering* para agrupar por color o aplicando técnicas de umbralización. En la figura 4.12 podemos ver la salida esperada de una red convolucional para la segmentación de imágenes.

¹¹Fuente: <https://devblogs.nvidia.com/parallelforall/detectnet-deep-neural-network-object-detection-digits/>



Figura 4.12: Salida de una red convolucional para la segmentación de imágenes¹².

La estrategia seguida para aplicar redes convolucionales a la segmentación de imágenes se basa en realizar una red para clasificar, en lugar de imágenes, píxeles. Para ello, se prescinde de las capas de redes neuronales, haciendo uso únicamente de capas de convolución, activación, *upsampling* o *transposed convolution* y *pooling*. El objetivo de esta red será aplicar estas operaciones a la imagen de entrada para, a la salida, dar una distribución de probabilidad a nivel de píxel como salida de la última capa de convolución, dando tantos canales como clases y un alto y ancho iguales a los de la imagen original. Para lograr esto, al verse reducida la imagen al aplicar diferentes operaciones de convolución y *pooling* en la red, se suele aplicar técnicas para devolver la imagen a su tamaño original, aplicando para ello capas de *upsampling* o *transposed convolution*. En la figura 4.13 encontramos cómo es la estructura de una red para la segmentación.

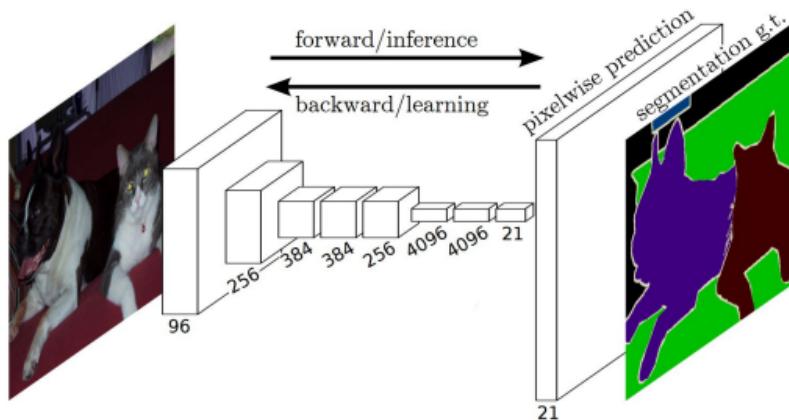


Figura 4.13: Ejemplo de red convolucional aplicada a la segmentación de imágenes¹³.

Una red convolucional para la clasificación puede ser utilizada para la segmentación. Un grupo de investigadores presentaron una técnica para realizar esta transformación [Long et al., 2015]. Proponen eliminar la capa *fully connected* de la red y, en su lugar, poner una capa de convolución, con tamaño de *kernel* igual a 1×1 para extraer la clase del píxel concreto, siendo después reescalado el mapa de características de esta capa al tamaño original para dar la salida de la red, aplicando una operación de *transposed convolution* con tantas salidas como clases existan. Mediante esta técnica se busca reemplazar la red neuronal, encargada de reconocer cada clase, por una capa de convolución de 1×1 con tantas salidas como conexiones existían en la capa *fully connected* para que aprenda a reconocer dichas clases, pero con la ventaja de poder reconocerla en cualquier lugar de

¹²Fuente: <https://arxiv.org/pdf/1503.01640.pdf>

¹³Fuente: https://people.eecs.berkeley.edu/~jonlong/long_shelhamer_fcn.pdf

la imagen. Esto es posible gracias a que la convolución es robusta a traslaciones. En la figura 4.14 se muestra la salida obtenida si eliminamos la ultima capa de *fully connected*.

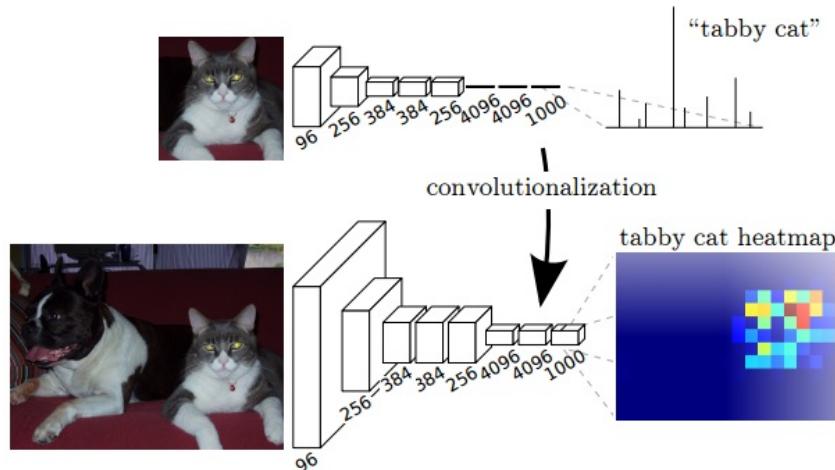


Figura 4.14: Transformación de una *convolutional neural network* a *fully convolutional network*¹⁴.

Para utilizar este tipo de red debemos calcular el factor de reducción que aplicamos a la imagen, como consecuencia de las diferentes operaciones que se realiza en la red, para después asignarlo como *stride* en la capa de *transposed convolution*, consiguiendo así devolver una salida con el mismo tamaño que la imagen de entrada.

Además del factor de reducción al que se ve sometida la imagen, debemos calcular el desplazamiento que se le hace a la misma. En cada capa de convolución o *pooling*, se produce, a la salida, un desplazamiento o *offset* con respecto a la entrada de $(P - (K - 1)/2)/S$, siendo S el valor de *stride*, P el valor de *padding* y K el tamaño del *kernel* ($((K - 1)/2 - P)$ para el *transposed convolution*). Conociendo esto, debemos calcular el desplazamiento producido a la imagen por las diferentes capas para ajustar el *padding* inicial a la imagen, con el objetivo de que, gracias a este, las diferentes operaciones en la red no provoquen que se elimine parte de la imagen, debido al desplazamiento que se produce. Aplicando este *padding* inicial permitimos que la zona de la imagen que se van eliminando en la red afecten solo a la parte añadida por el *padding*.

El *offset* acumulado por las diferentes capas puede calcularse a partir del *offset* que produce cada capa. Debemos calcularlo desde la última capa hacia atrás, de forma que, la unión de una capa L_1 con una capa L_2 , con *offset* O_1 y O_2 respectivamente, producen un *offset* acumulado de O_1/F , siendo F el factor de escala acumulado hasta la capa L_2 . Este *offset* se suma al de la capa anterior, partiendo desde la última capa e inicializándolo al *offset* que hay en esta. En la figura 4.15 se puede ver un ejemplo del cálculo de estos valores para una red de ejemplo. Una vez obtenido el *offset* acumulado, sabemos el desplazamiento al que se ve sometida la imagen de entrada, por lo que, a la salida, debemos eliminar de esta tantos píxeles como *offset* acumulado haya, para que se pueda emparejar correctamente cada píxel de la imagen original con la salida de la red.

¹⁴Fuente: https://people.eecs.berkeley.edu/~jonlong/long_shelhamer_fcn.pdf

Layer	Stride	Pad	Kernel size	Scaling Factor (1/S)	Cumulative Scaling Factor	Offset (P-(K-1)/2)/S	Cumulative Offset (calculated backward from last layer)
conv1	4	100	11	1/4	1/4	23.75	18
pool1	2	0	3	1/2	1/8	-0.5	-77
conv2	1	2	5	1	1/8	0	-73
pool2	2	0	3	1/2	1/16	-0.5	-73
conv3	1	1	3	1	1/16	0	-65
conv4	1	1	3	1	1/16	0	-65
conv5	1	1	3	1	1/16	0	-65
pool5	2	0	3	1/2	1/32	-0.5	-65
conv6	1	0	6	1	1/32	-2.5	-49
conv7	1	0	1	1	1/32	0	31
upscore	32	0	63	32*	1	31*	31

Figura 4.15: Cálculo del *offset* producido por las capas de una red¹⁵.

Siguiendo esta técnica, no podremos obtener resultados detallados, ya que la segmentación se obtiene en base al último mapa de características antes de la capa de *transposed convolution* de la red, lo que significa que la capa de *transposed convolution* asigna la distribución de probabilidad de cada píxel de la imagen original basándose en este mapa de características que, normalmente, debido a las operaciones de la red, es de un tamaño mucho más reducido que el original. Al calcular el factor de reducción al que se ve sometida la imagen de entrada para asignarlo como *stride* de la capa de *transposed convolution* estamos indicando la región a analizar en este último mapa de características para obtener la distribución de probabilidad de cada píxel original, lo que supone que, si calculamos un *stride* de 32, estamos asignando, en base a cada elemento de este último mapa de características, la distribución de probabilidad de una región de 32×32 correspondiente a la entrada, en lugar de a un único píxel.

Con el objetivo de conseguir resultados más precisos, los diseñadores de las *fully convolutional networks*[Long et al., 2015] proponen utilizar, además del último mapa de características, otras salidas de capas anteriores a la última, donde la entrada no estaba tan reducida y, mediante la combinación de todas poder detectar detalles más precisos. La capa encargada de sustituir a la red neuronal cuenta con convoluciones 1×1 para determinar la clase de cada píxel, que después es reescalada con una capa de *transposed convolution* a su tamaño original.

Si aplicamos el *transposed convolution* a un mapa de características de la red donde este tenía un tamaño mayor que en la última capa, podemos combinar la salida de la última capa con esta, con la diferencia de que, solo necesitamos aplicar como *stride* en la capa *transposed convolution* el factor de escala entre estos dos mapas de características y no el de la imagen completa. Antes de combinar ambos mapas, puede ser necesario realizar algún recorte al mapa de características para que ambos tengan el mismo tamaño. Una vez hemos reescalado el mapa más pequeño y se ha recortado para que tenga el mismo tamaño que el mapa grande, podemos fusionarlos en un único mapa de características. Esta operación se puede realizar concatenando ambos mapas componente a componente.

¹⁵Fuente: <https://devblogs.nvidia.com/parallelforall/image-segmentation-using-digits-5/>

En la figura 4.16 podemos ver diferentes combinaciones de mapas de características que permiten obtener resultados más detallados a más cercanía con el inicio de la red.

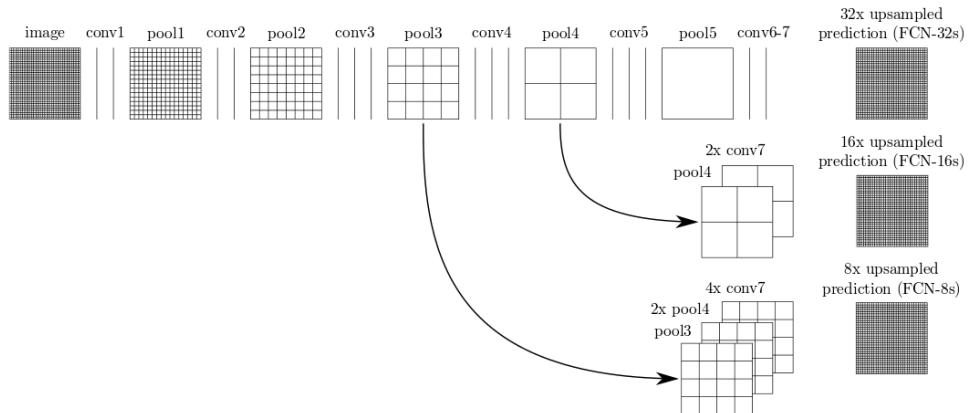


Figura 4.16: Aumento del detalle en la segmentación según la combinación de mapas de características¹⁶.

Con esta misma filosofía la red U-NET [Ronneberger et al., 2015] consigue resultados realmente precisos al aplicarla a la segmentación de imágenes biomédicas.

Para poder entrenar este tipo de redes necesitaremos, por cada imagen del conjunto de entrenamiento, una máscara que nos permita enseñar a la red la salida esperada. Típicamente, en esta máscara se asigna un color único para cada clase existente en nuestro problema.

Al no disponer de capas *fully connected*, podemos tener imágenes de diferentes tamaños en el conjunto de entrenamiento. En la figura 4.17 podemos ver un ejemplo de la máscara correspondiente a una imagen para el entrenamiento de una red de segmentación.



Figura 4.17: Ejemplo de imagen con su máscara para realizar el entrenamiento¹⁷.

El entrenamiento de estas redes se realiza como en las redes dedicadas a la clasificación, solo que, en lugar de imágenes, clasificamos píxeles y el error se obtiene en base a ellos. Suele usarse como función a optimizar el *cross entropy error* como ocurre en los problemas de clasificación.

Las métricas de evaluación son obtenidas a nivel de píxel y suele emplearse las mismas que para los problemas de clasificación: *accuracy*, *recall* y *precision*.

La segmentación supone un nivel más de detalle que la detección, lo que da lugar a que sea utilizada en contextos que requieren este nivel de detalle, como en las imágenes médicas, ya que permiten identificar determinado tipo de células o tejidos de entre todos

¹⁶Fuente: https://people.eecs.berkeley.edu/~jonlong/long_shelhamer_fcn.pdf

¹⁷Fuente: <https://devblogs.nvidia.com/parallelforall/image-segmentation-using-digits-5/>

los presentes. Al proporcionar un mayor nivel de detalle de la escena que observamos, nos permite comprenderla mejor y poder interactuar con el entorno de una manera más precisa y con los objetos o personas que hay en él.

CAPÍTULO 5

Experimentación

Como clausura de este Trabajo de Fin de Grado, presentamos, en este capítulo de experimentación, los resultados obtenidos, al aplicar las técnicas de *deep learning* estudiadas, en un problema real.

Buscando la aplicación de las técnicas de detección, segmentación y clasificación visuales y estando estas (debido al tratamiento que se les ha dado en este trabajo) relacionadas con la visión, encontramos, en aquellos escenarios que implican la interacción de robots con su entorno, uno de los ambientes más idóneos para su aplicación.

En el sector de la agricultura existe un especial interés por el desarrollo de sistemas que permitan automatizar las tareas que se realizan, con el objetivo de optimizar los costes derivados de esta actividad. Hoy día, se cuentan con sistemas capaces de automatizar el riego e incluso de la recolección de la cosecha, como en el caso de las plantaciones de trigo. Sin embargo, siguen existiendo determinados cultivos que requieren de técnicas mucho más sofisticadas que las utilizadas hasta ahora (a causa de la complejidad práctica de su recolección). Un ejemplo de ello, son los cultivos de fresa, donde se precisa de una elevada mano de obra para su recolección, la cual debe realizarse examinando con detenimiento la planta, en busca de todos sus frutos, siendo estos recolectados uno a uno.

Actualmente, el sector fresero supone el 50 % de los cultivos de *berries* de la provincia de Huelva¹, donde se encuentra el 95 % de la producción en España², disponiendo de 5.400 hectáreas de cultivo dedicada a la fresa³, una producción de 294.650 toneladas y una facturación de 395 millones de euros⁴, siendo considerado el mayor productor de Europa y el segundo del mundo, solo por detrás de California, con 15.000 hectáreas⁵ donde esta actividad genera 2.600 millones de dólares⁶. El consumo de este fruto en la Unión Europea ronda los 1,2 millones de toneladas⁷. Durante la temporada de la fresa, solo en Huelva, se requiere de 90.000 personas para la campaña⁷. Hoy en día, los agricultores están teniendo problemas para encontrar suficiente mano de obra para su recolección⁸.

Esta situación ha propiciado la aparición de compañías que buscan crear máquinas capaces de recolectar estos frutos. Prueba de ello son empresas como Agrobot⁹ o Harvest Croo¹⁰.

¹ <http://www.20minutos.es/noticia/2896836/0/desciende-casi-7-superficie-plantada-fresa-aumenta-22-otros-berries/>

² <http://www.elmundo.es/economia/2016/03/09/56d8868d22601dc4368b461b.html>

³ http://www.euroganaderia.eu/ganaderia/noviembre/la-superficie-plantada-de-fresa-desciende-un-7_1802_100_2770_0_1_in.html

⁴ <http://www.fepex.es/noticias/detalle/sector-fresero-Huelva-culmina-campa%C3%B1a-aumento-facturacion-8>

⁵ https://www.nass.usda.gov/Data_and_Statistics/index.php

⁶ http://www.bbc.com/mundo/noticias/2015/01/150105_eeuu_california_cultivo_fresas_fumigantes_pesticidas_jg

⁷ http://cincodias.elpais.com/cincodias/2017/03/13/empresas/1489430603_640231.html

⁸ http://cadenaser.com/emisora/2017/03/22/radio_huelva/1490188403_398039.html

⁹ <http://www.agrobot.es/>

¹⁰ <http://harvestcroorobotics.com/>

El uso de este tipo de máquinas podría suponer un ahorro significativo del coste de la recolección de este fruto, que puede hacer rentable su recolección por debajo del precio actual y cubrir la demanda de mano de obra exigida por los agricultores. Además, gracias a estas máquinas, se puede acercar la informática al campo, con todas las ventajas que pueda traer consigo.

El estado actual de la tecnología facilita la creación de máquinas con este propósito. Debido al interés que suscita, decidimos probar la viabilidad de un sistema que permita, a una máquina equipada con una cámara, la detección de fresas listas para su recolección. Como forma de abordar este problema, diseñamos un sistema dividido en tres subsistemas: un primer sistema se dedica a la detección de las fresas presentes en la imagen, para saber donde se encuentran; estas detecciones son segmentadas para eliminar todos los píxeles que no pertenecen a la fresa; gracias a esto, el tercer sistema, el de clasificación, podrá determinar más fácilmente si la detección segmentada corresponde a una fresa y si está madura o no. En la figura 5.1 se muestra de forma esquemática el sistema que queremos diseñar.

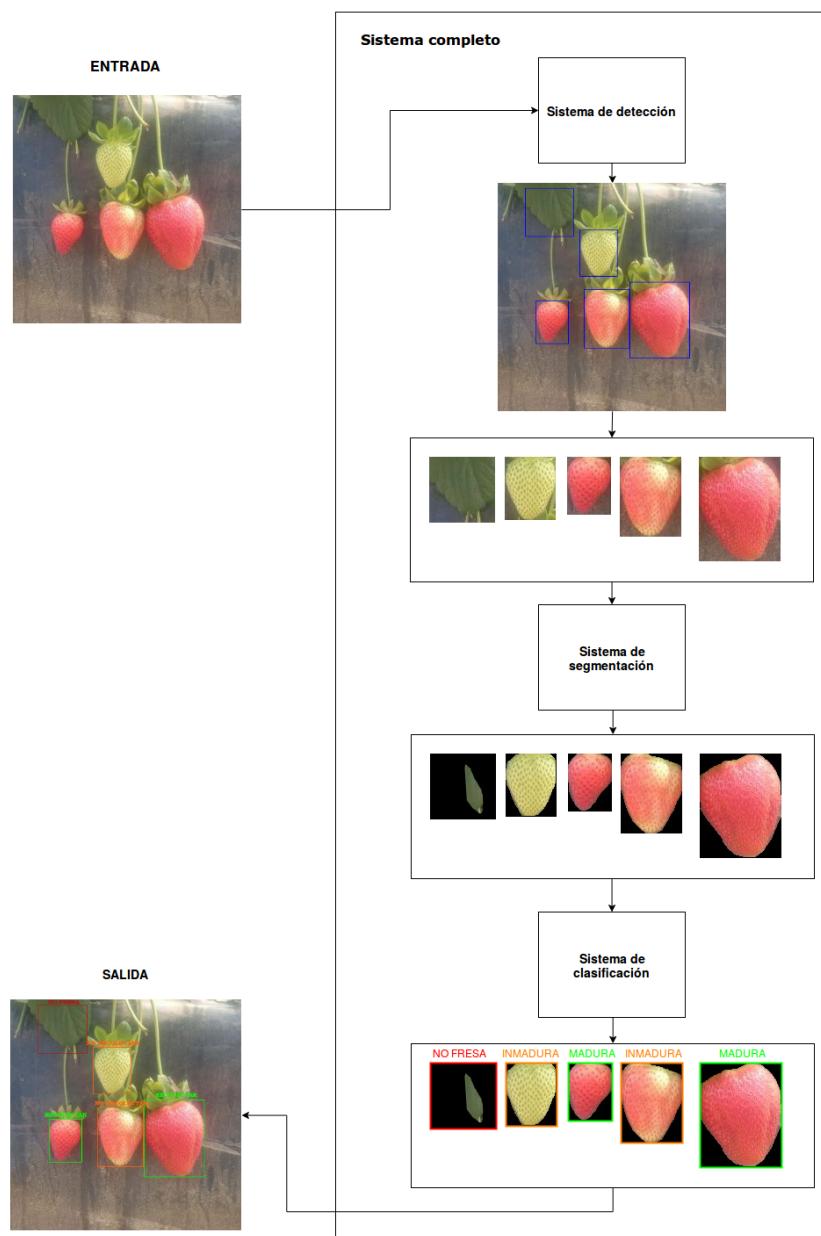


Figura 5.1: Sistema propuesto.

Para la realización de estos sistemas emplearemos redes convolucionales adaptadas al problema que se este abordando en cada sistema.

En cada una de las tres redes que debemos diseñar realizaremos dos propuestas: una primera propuesta, en la que utilizaremos una red conocida, que ha sido empleada para otros problemas con buenos resultados, y una segunda propuesta, donde realizaremos el diseño de una red lo más reducida posible, de forma que obtenga resultados cercanos a los de la primera red propuesta, pero que necesite menos capacidad de cómputo y esto haga que sea más susceptible de ser implantada en un sistema que trabaje en tiempo real. Esto nos permitirá estudiar el efecto que tiene el tamaño de la red en los resultados y en qué sistemas podemos reducir el tamaño de la red sin repercutir en el desempeño de forma excesiva.

5.1 Material

Para el entrenamiento y evaluación de nuestras redes disponemos de un total de 700 imágenes, obtenidas en una plantación de fresas en la provincia de Huelva. Esta captura se realizó con una cámara de 20.7 megapíxeles de un móvil Sony Xperia Z2, en diferentes días y a diferentes horas. Las imágenes son guardadas con una resolución de 3840×2160 en formato «.JPG».

Bajo la suposición de que nuestra cámara irá a la altura del lomo de fresa, viendo a este desde un lateral y únicamente al lomo que tenga de frente, tomamos todas las imágenes simulando esta posición. En la figura 5.2 podemos ver algunas de estas capturas.



Figura 5.2: Imágenes de muestra del conjunto original.

Estas imágenes poseen un tamaño demasiado grande que ralentizaría la ejecución de nuestras redes. Por lo tanto, optamos por reescalarlas a un tamaño de 640×360 . Debido a que algunas de las imágenes contienen en sus límites partes de fresas que no aparecen enteras, para evitar condicionar los resultados y aumentar la cantidad de píxeles en la imagen que son de fresas, decidimos recortar las imágenes en anchura a un tamaño final de 360×360 , buscando dejar dentro de la imagen la zona donde haya más fresas, evitando siempre que se recorte alguna. En la figura 5.3 se muestra algunas de estas imágenes recortadas.



Figura 5.3: Imágenes recortadas del conjunto original.

Para cada una de las redes, necesitaremos preparar y dividir el conjunto de datos que utilizará.

Nos valemos de un *script* realizado en Matlab para marcar los contornos de las fresas que haya en la imagen de forma individual cada una, almacenando la máscara obtenida con este marcado para cada imagen. Posteriormente, utilizaremos esta información para crear los conjuntos de datos para cada red. En la figura 5.4 se encuentran algunas capturas del programa que se ha utilizado para crear estas máscaras.

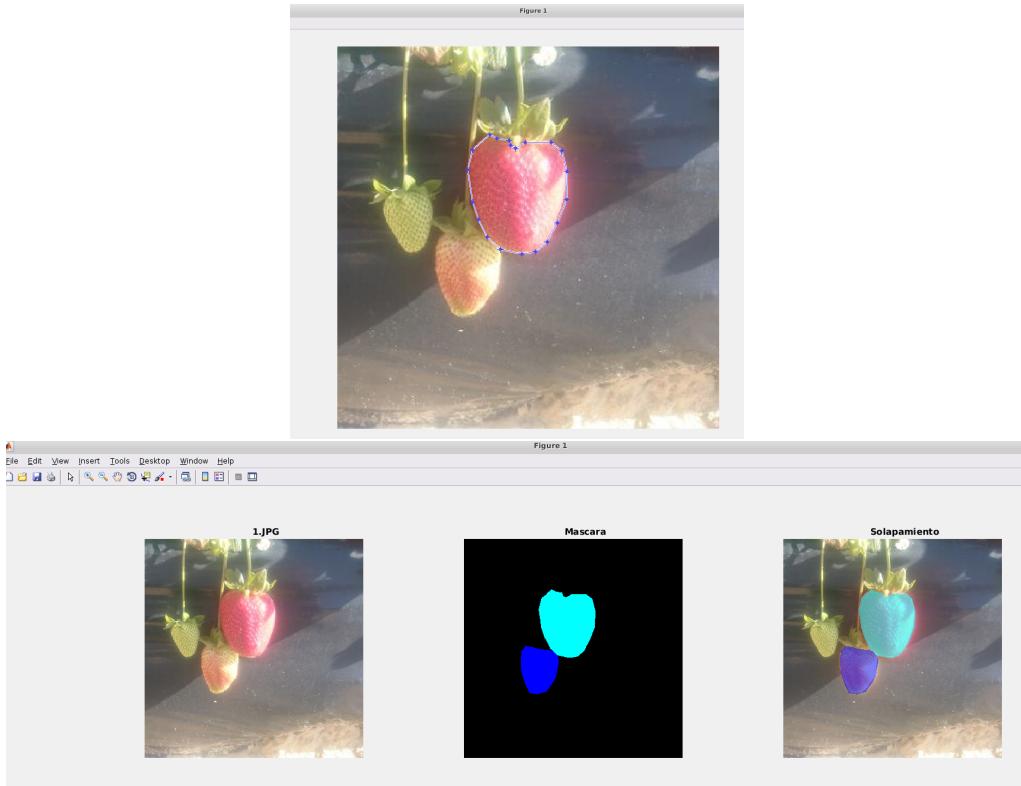


Figura 5.4: Capturas de pantalla del programa utilizado para marcar las fresas.

La división en subconjuntos sigue la distribución indicada en la Sección 3.4.1. Usamos para el entrenamiento el 50 % de los datos y el 25 % para validación y test respectivamente. El reparto se realizará de forma aleatoria, mediante un *script* que diseñemos en Matlab.

Para desarrollar la red de detección, hacemos uso de las imágenes recortadas de tamaño 360×360 . Creando un *script* en Matlab, extraemos de las máscaras anteriores los recuadros que engloban a cada fresa por cada imagen.

Debido a que el entrenamiento se realiza en Digits, debemos guardar estos recuadros como un fichero con formato «.txt», con el mismo nombre que la imagen de la que se han obtenido los recuadros. El formato concreto que utiliza Digits se compone de 8 campos por cada rectángulo, que se almacenan por línea y de los que solo tenemos que rellenar, el primer campo para indicar el nombre de la clase del rectángulo y los campos que van del 5 al 8 para indicar las coordenadas en píxeles, con respecto a la imagen, de las esquinas superior izquierda e inferior derecha. Los demás campos son puestos a 0.

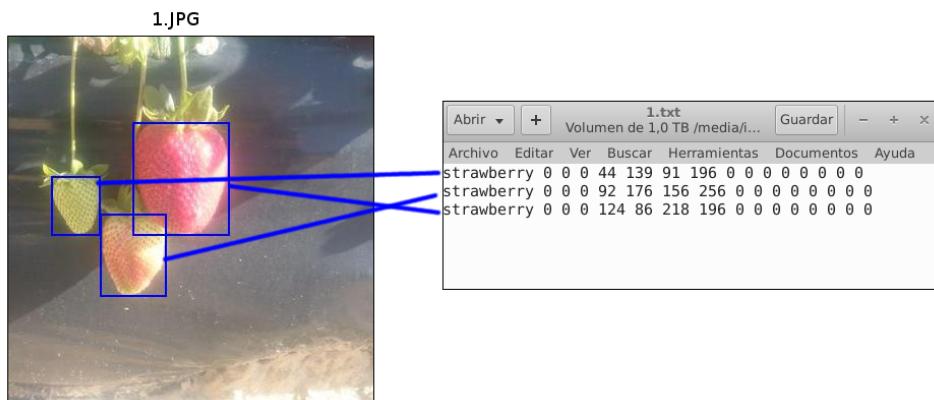


Figura 5.5: Ejemplo de creación del conjunto de datos para la detección.

En total disponemos de 350 imágenes para entrenar y 175 para evaluación y test respectivamente.

Para crear el conjunto de segmentación, debemos tener presente que las fresas suponen una ínfima cantidad de píxeles respecto al total de la imagen, es por ello que para la creación del conjunto de datos, desarrollamos un *script* en Matlab que nos genere las máscaras, marcando los píxeles todas las fresas que tenemos en la imagen, quedándonos, en lugar de con toda la imagen, con el recuadro que englobe al máximo número de píxeles marcados como fresa y que supongan más de un 30 % del total del recuadro. Esto es posible gracias a que la segmentación no necesita que todas las imágenes tengan el mismo tamaño.

En Digits, será necesario indicar la imagen y su máscara correspondiente, con el mismo nombre y en formato «.png», guardando los píxeles marcados como un color con formato indexado. En la figura 5.6 vemos un ejemplo de como queda el conjunto de datos para la segmentación.



Figura 5.6: Ejemplo de creación del conjunto de datos para la segmentación.

En este conjunto de datos, en lugar de dividirlo por imágenes, lo dividimos por cantidad de píxeles de fresas en cada conjunto, buscando dejar entorno al 50 % de estos píxeles para entrenar y el 25 % para evaluación y test respectivamente. Creando un *script* en Matlab, elegimos de forma aleatoria imágenes hasta que se completen los porcentajes. Esto da lugar a que utilicemos, 506 imágenes para entrenar, 253 imágenes para evaluar y 221 para test.

Para la red de clasificación, las imágenes provendrán del sistema de segmentación, que nos permitirá eliminar aquellos píxeles que no sean de la fresa. Por tanto, para crear este sistema, obtenemos, de las fresas marcadas de forma individual, solo aquellos píxeles que estén marcados como fresa, asignando el color negro a todo lo demás. Cada fresa es extraída de forma individual y almacenada en una carpeta según el conjunto al que pertenezca.

Después de un análisis del tamaño medio de las imágenes, se determina utilizar 80×80 como tamaño de las imágenes de la red. Las fresas que sean más grande que este tamaño, serán reescaladas al mismo y las que sean muy pequeñas, se llenarán las zonas sobrantes con color negro. Al dividir el conjunto de fresas en fresa madura e inmadura, obtenemos, de forma aleatoria, de aquellas zonas donde no existe fresa en la imagen, recortes de tamaño 80×80 para crear los ejemplos negativos. En la figura 5.7 se recoge una muestra del conjunto creado.

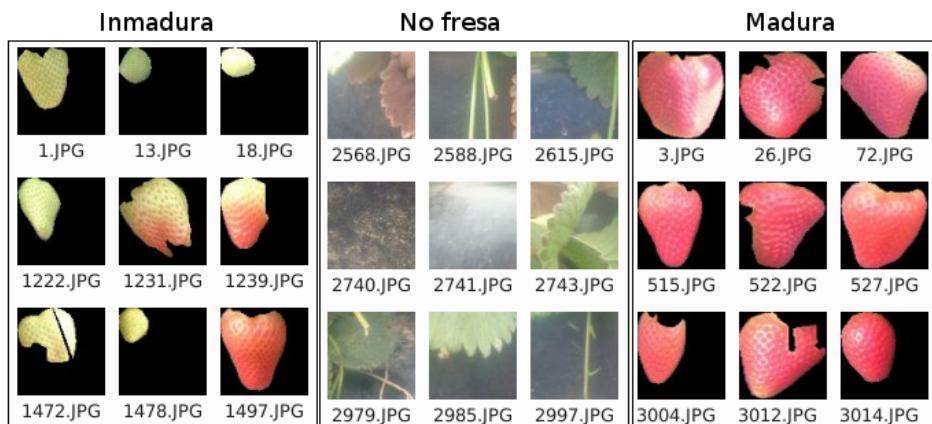


Figura 5.7: Ejemplo de creación del conjunto de datos para la clasificación.

Nos quedamos con un total de 300 imágenes de cada clase (900) para entrenar y 150 de cada clase (450) para validación y test respectivamente.

5.2 Metodología

A continuación, para cada uno de los sistemas que tenemos que crear, diseñamos dos redes, una primera red que servirá como referencia y que será elegida de entre alguna de las redes conocidas, como las de la Sección 3.3. Y una segunda red, de diseño propio, inspirándonos en otras redes conocidas, pero con la premisa de diseñarla con el mínimo número de capas posibles y de operaciones en cada capa. Con el objetivo final de estudiar como repercute en el desempeño del sistema utilizar una red más reducida.

Se especifica también cómo se realiza el entrenamiento de cada red, los resultados obtenidos en el entrenamiento y sobre el conjunto de validación y la elección de los parámetros que fue realizada.

5.2.1. Detección

Para el sistema de detección, decidimos utilizar como red de referencia la red DetectNet¹¹, propuesta por Nvidia para realizar la detección en Digits.

Para el sistema de detección, extraemos las métricas de *recall* y *accuracy*, en nuestro caso, estamos más interesados en que no se pierda ninguna fresa, aunque se detecte algo que no es una fresa como positivo, ya que puede ser descartado posteriormente por el sistema de clasificación. Por lo tanto, elegiremos a la mejor red del entrenamiento a aquella que nos de mayor valor de *recall*.

DetectNet

Esta red fue descrita en la Sección 4.2. Basa su diseño en la red GoogleNet, descrita en la Sección 3.3.4. El esquema de esta red para Caffe lo podemos encontrar en la figura 5.9.

La red hace uso de una capa conocida como *local response normalization* descrita en [Szegedy et al., 2014], es utilizada para realizar una normalización después de las primeras capas de activación.

¹¹ <https://devblogs.nvidia.com/parallelforall/detectnet-deep-neural-network-object-detection-digits/>

Partimos de un modelo de red ya entrenado en la competición ImageNet, en lugar de entrenar la red de cero.

Durante el entrenamiento, se normaliza la imagen y se realizan aleatoriamente transformaciones a las imágenes con las que se entrena la red: volteos, giros, cambios de tonalidad, contraste o escalado. Debido a que la entrada se reduce 16 veces su tamaño, es asignado 16 como valor para el *stride*.

El entrenamiento se realiza haciendo uso del algoritmo Adam, con los valores para los parámetros de $\beta_1 = 0,9$, $\beta_2 = 0,999$, $\epsilon = 10^{-8}$ y $\alpha = 0,0001$. Asignamos un *batch size* de 14 imágenes para el entrenamiento y 7 para la evaluación. El entrenamiento tiene una duración de 40 minutos con 500 iteraciones. En la figura 5.8 podemos ver la evolución del entrenamiento.

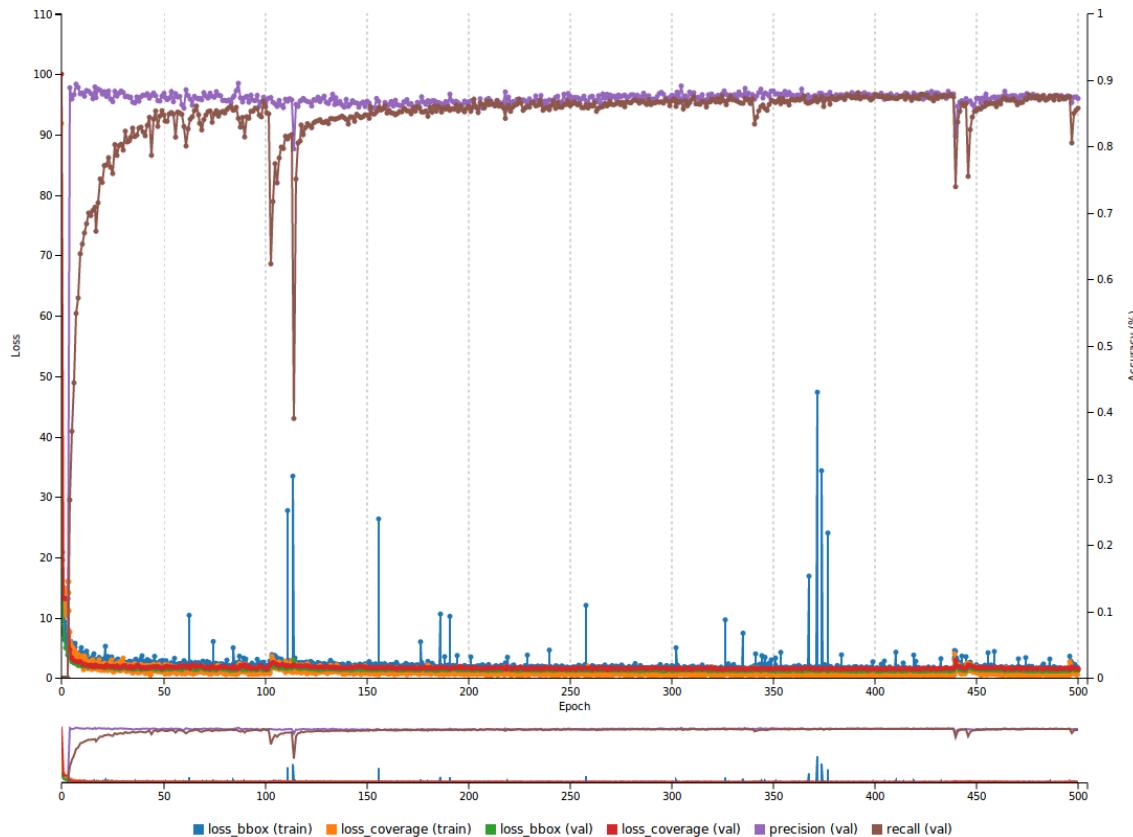
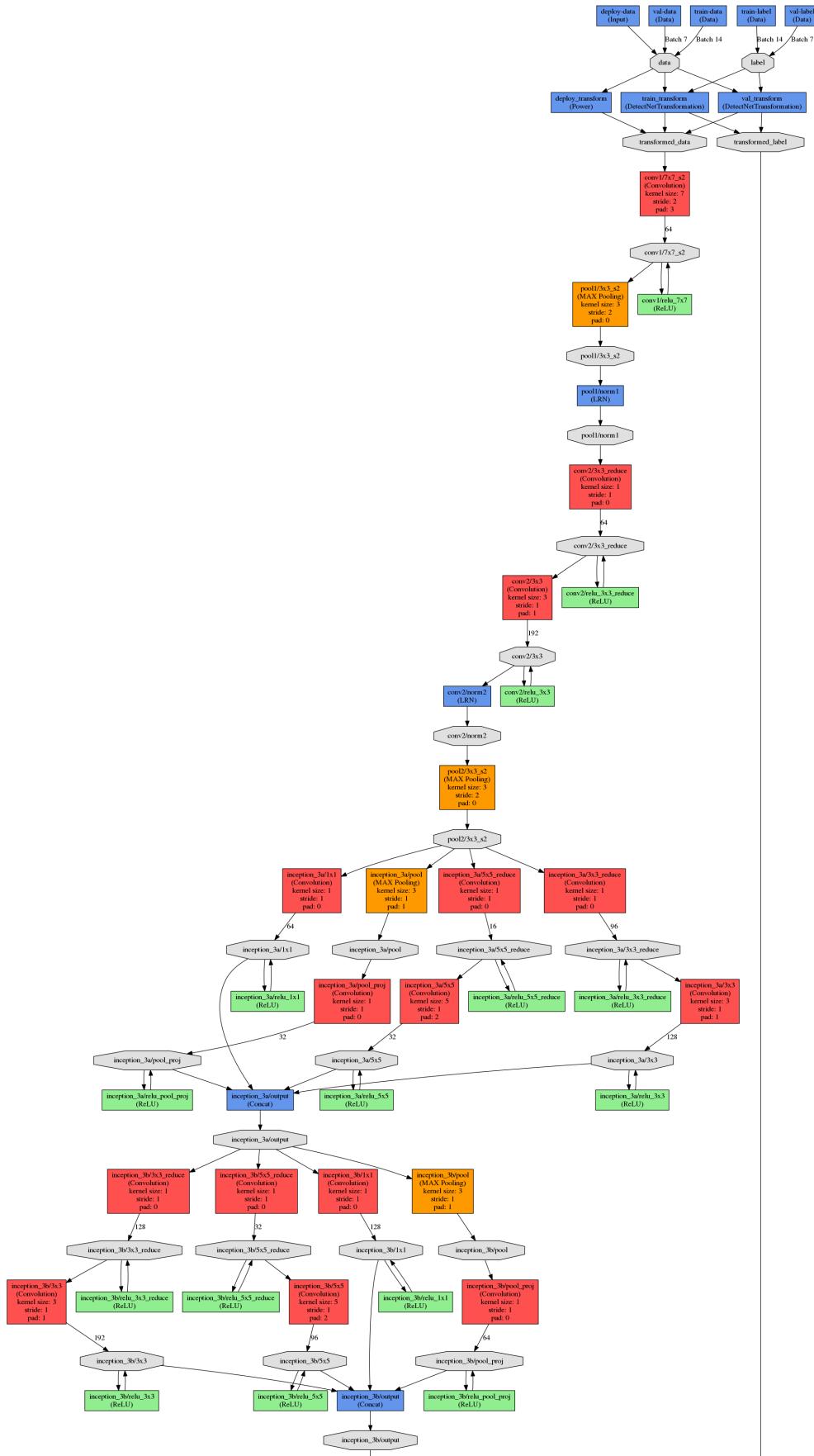
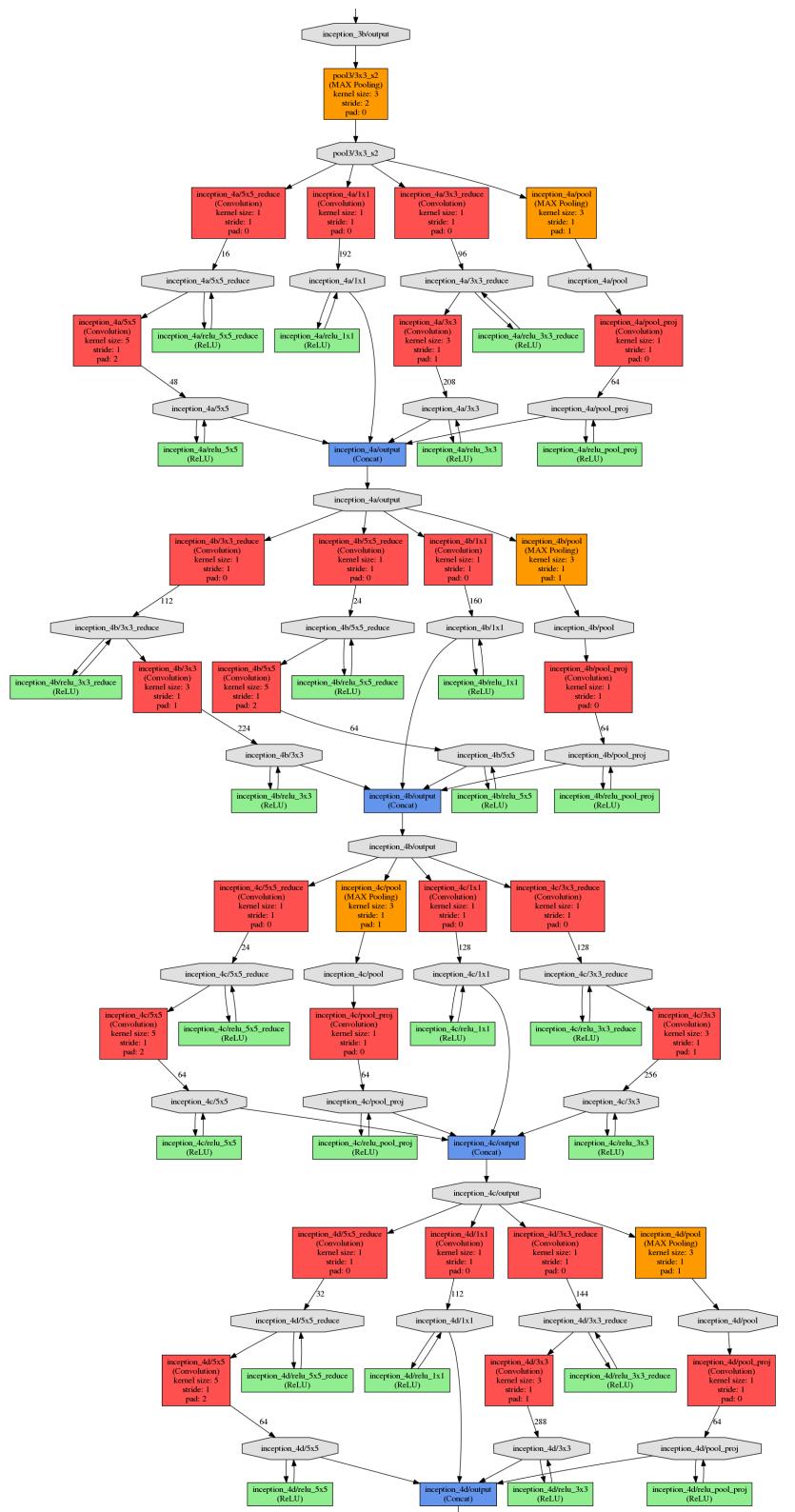


Figura 5.8: Gráfica de la evolución del entrenamiento de la red DetectNet.

Después de 500 iteraciones de entrenamiento, los mejores resultados, sobre el conjunto de evaluación, se producen en la iteración 475, siendo los siguientes:

- $loss_bbox = 1,26937$
- $loss_coverage = 1,52803$
- $precision = 96,8153$
- $recall = 95,8139$





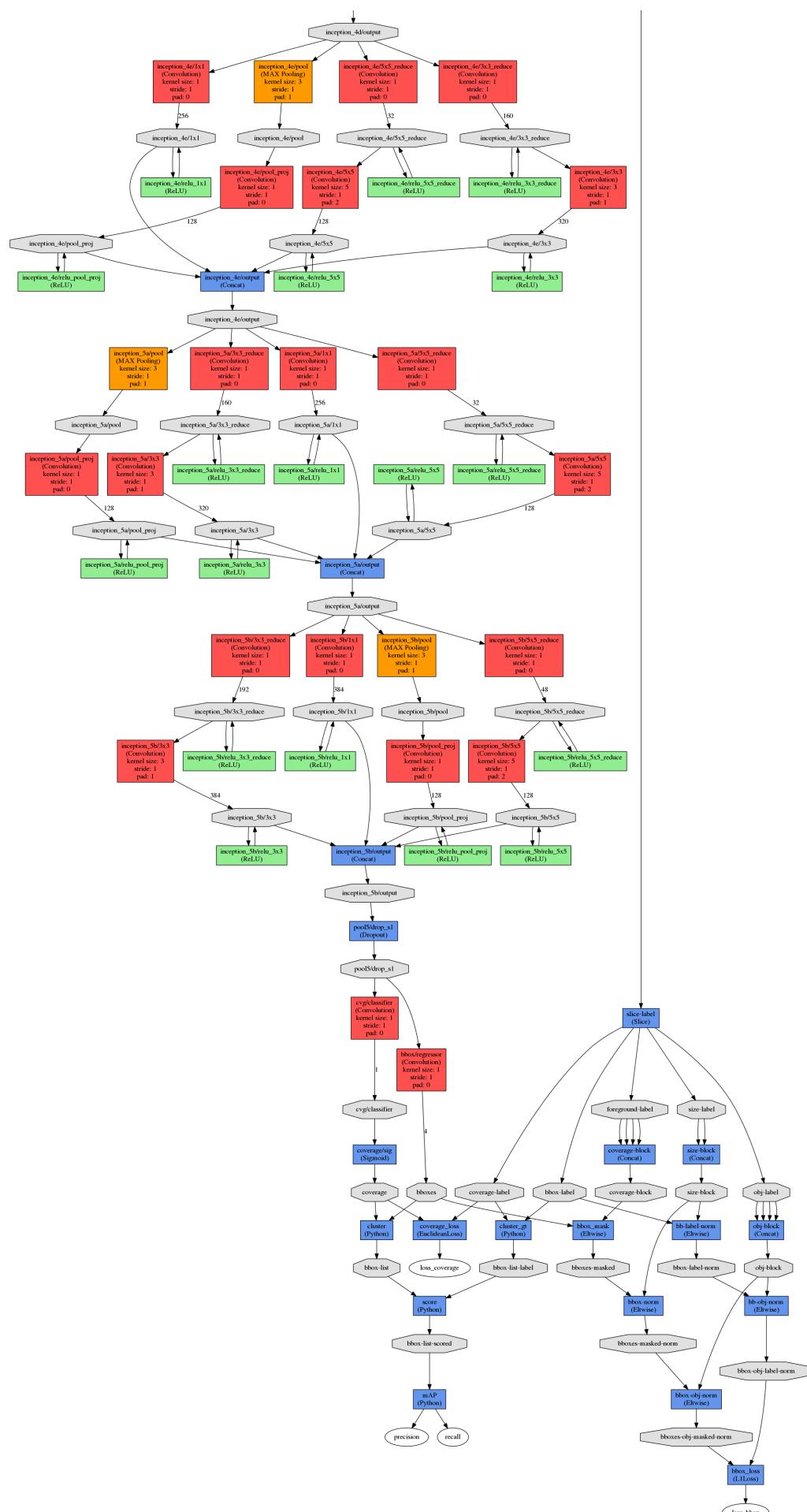


Figura 5.9: Estructura de la red DetectNet.

Red propuesta

Esta red intenta alcanzar resultados similares a los de la red DetectNet, pero con menos operaciones y un tamaño más pequeño. Centramos el mayor número de operaciones en las capas más profundas de la red, donde la imagen se encuentra más reducida que al principio. En la figura 5.11 se encuentra el esquema de la red.

Para esta red también normalizamos la imagen y realizamos transformaciones aleatorias a las imágenes con las que se entrena: volteos, giros, cambios de tonalidad, contraste o escalado. Al verse reducida la entrada 8 veces su tamaño, asignamos 8 como valor para el *stride*.

El entrenamiento se realiza haciendo uso del algoritmo Adam, con los valores para los parámetros de $\beta_1 = 0,9$, $\beta_2 = 0,999$, $\epsilon = 10^{-8}$ y $\alpha = 0,005$. Asignamos un *batch size* de 50 imágenes para el entrenamiento y 25 para la evaluación. Los parámetros de la red son inicializados siguiendo la inicialización *He*, descrita en la Sección 3.4.2. El entrenamiento tiene una duración de 1 hora y 47 minutos, con un total de 7100 iteraciones. En la figura 5.10 podemos ver cómo transcurrió el mismo.

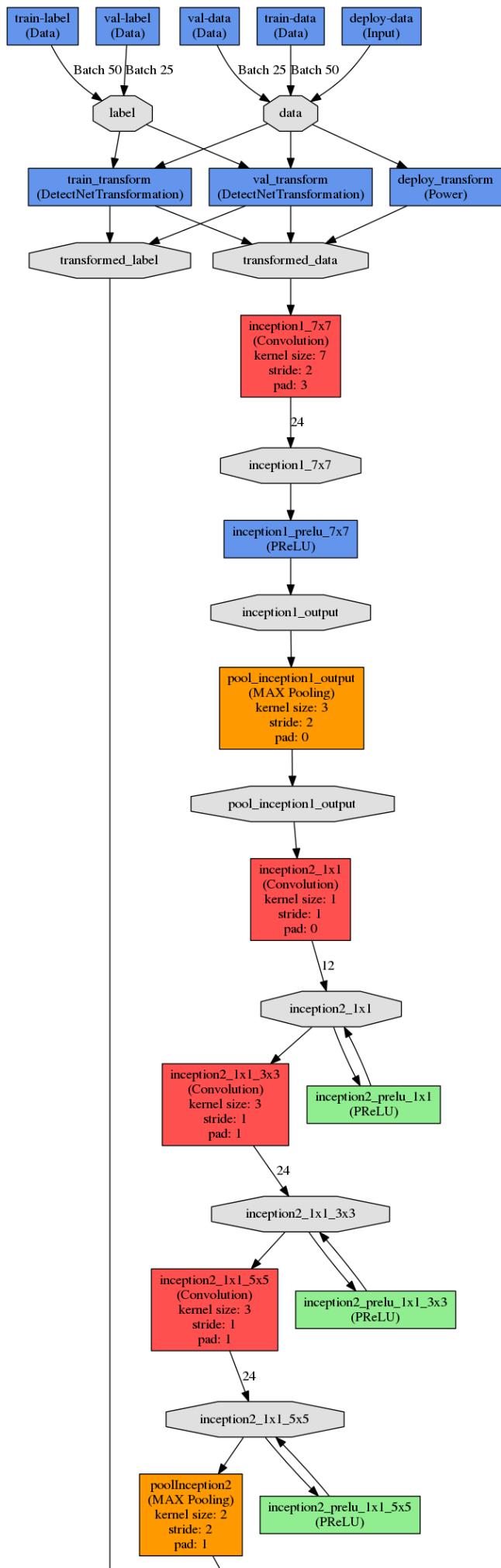


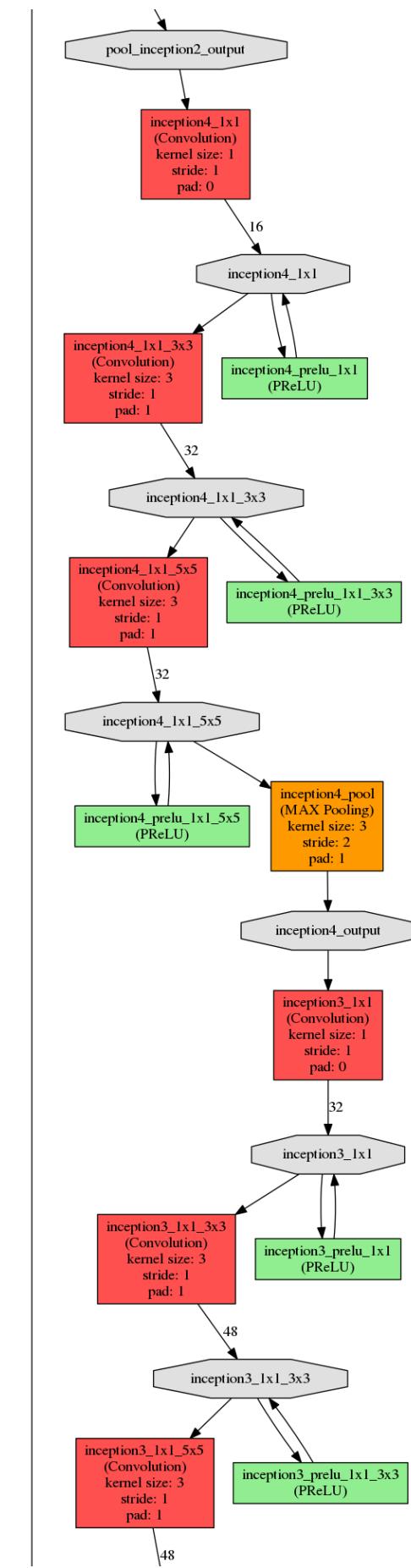
Figura 5.10: Gráfica de la evolución del entrenamiento de la red propuesta¹².

En la iteración 3053 obtenemos los mejores resultados sobre el conjunto de evaluación:

- $loss_bbox = 1,82281$
- $loss_coverage = 2,07643$
- $precision = 95,1821$
- $recall = 85,1329$

¹²Para esta gráfica se produjo un error al generarla, por lo que se muestran todos los resultados en base al eje vertical izquierdo, siendo la franja de 0 a 100 donde debe mirarse el % para el *precision* y el *recall*.





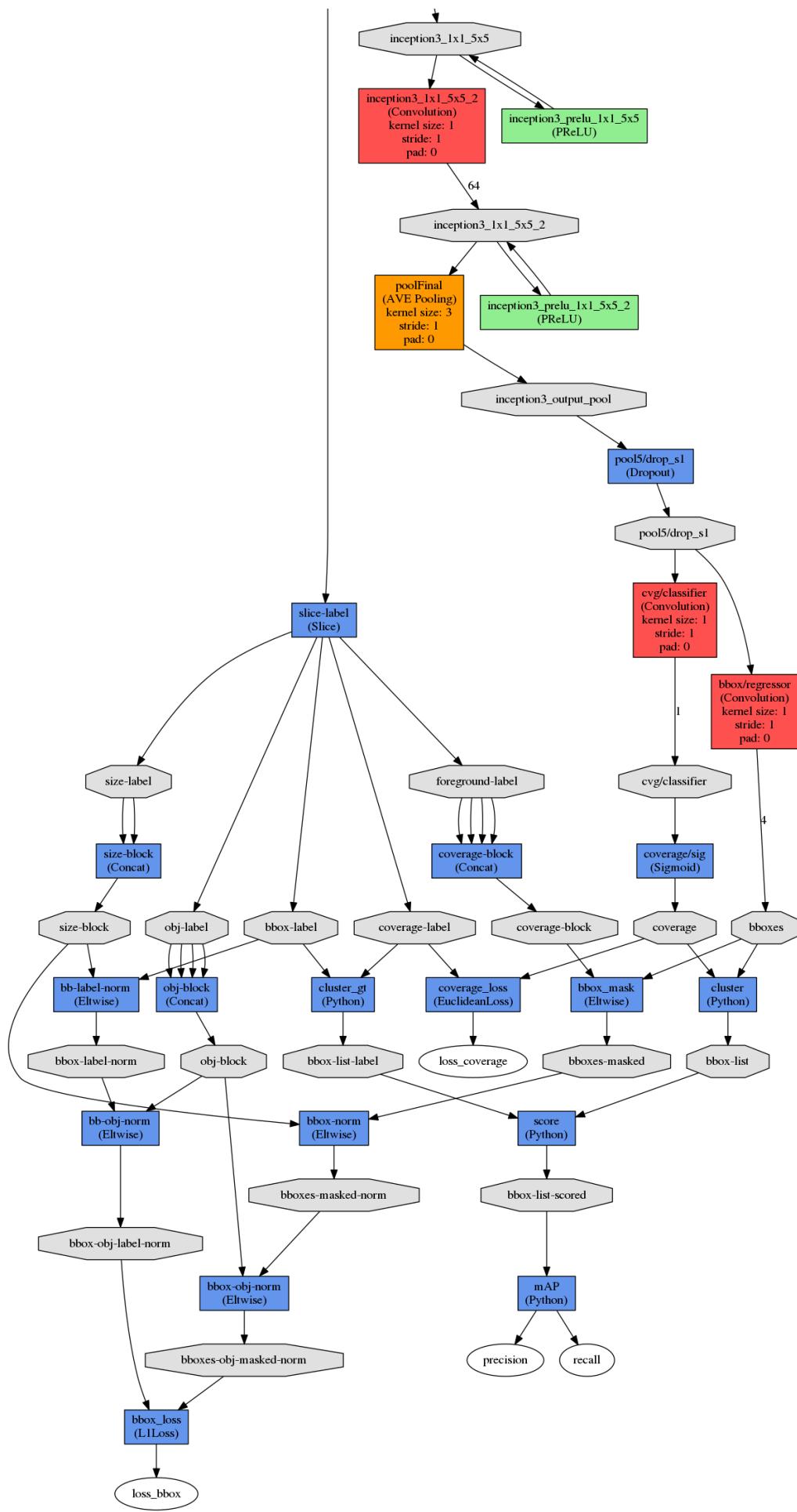


Figura 5.11: Estructura de la red propuesta para la detección.

5.2.2. Segmentación

La segmentación puede suponer un problema al disponer de un % de píxeles mucho menor para las fresas que para el resto de la imagen, considerado como negativo.

Para este problema, decidimos utilizar la red FCN-Alexnet como red de referencia. Esta red no es más que la red AlexNet descrita en la Sección 3.10 pero adaptada a realizar la segmentación, como se indicó en la Sección 4.3.

Como red propuesta, diseñamos una basada en la red U-NET, con muchas menos convoluciones y menos capas.

Eligiéremos del entrenamiento la red obtenida en la iteración que dio menos error y mayor *accuracy*.

FCN-AlexNet

Esta red es entrenada partiendo del modelo obtenido de la red AlexNet para la competición Imagenet. Para obtener todos los pesos de la red entrenada y asignarlos a nuestra red de segmentación, hacemos uso de un *script* de Python que se encarga de la transferencia de parámetros entre estos modelos. En la figura 5.13 podemos ver su estructura.

Durante el entrenamiento, únicamente se normaliza la imagen sin aplicar ningún tipo de transformación a la misma.

El entrenamiento se realiza haciendo uso del algoritmo Adam, con los valores para los parámetros de $\beta_1 = 0,9$, $\beta_2 = 0,999$, $\epsilon = 10^{-8}$ y $\alpha = 0,0001$. Asignamos un *batch size* de 1 imagen para el entrenamiento y 1 para la evaluación. El entrenamiento tiene una duración de 5 horas y 21 minutos con 200 iteraciones. En la figura 5.12 podemos ver el progreso del entrenamiento.

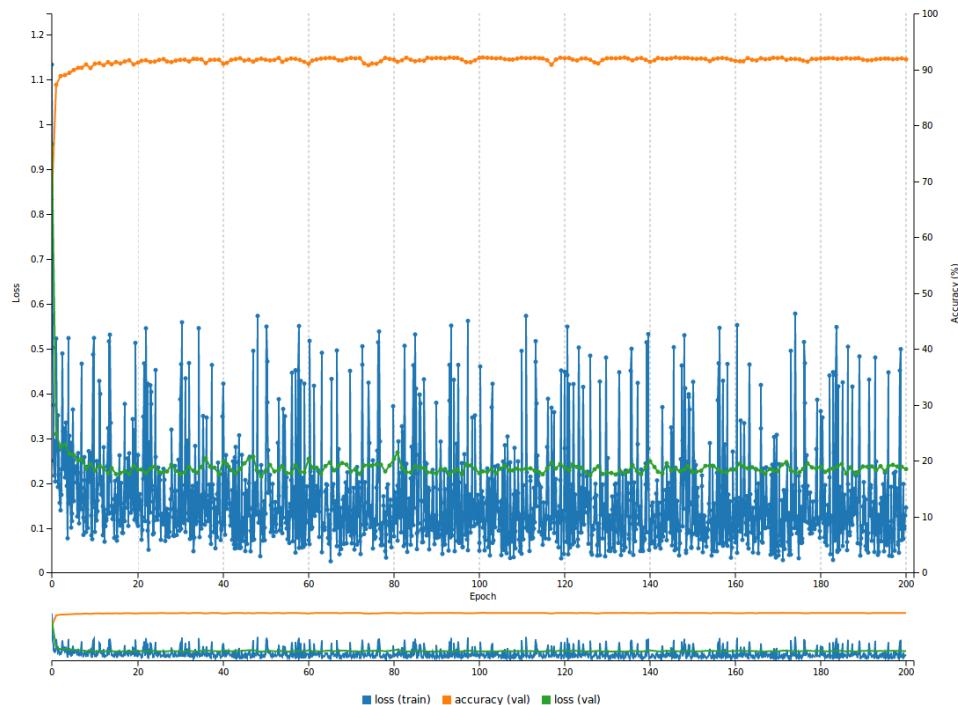
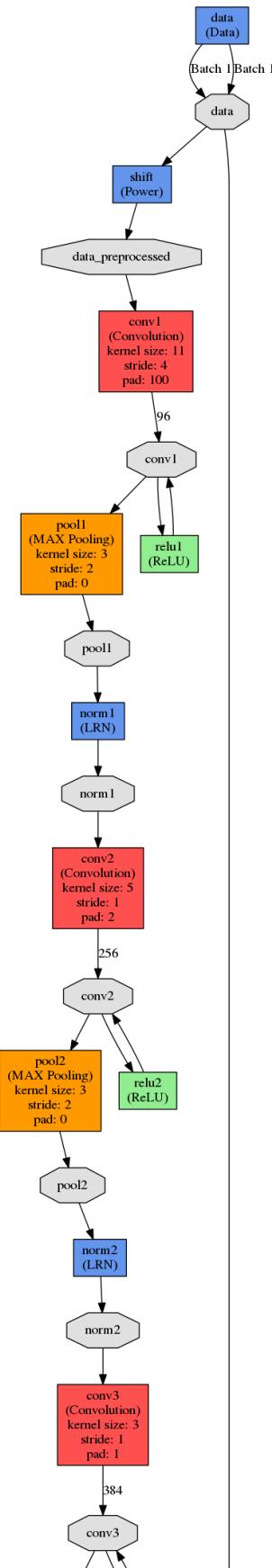


Figura 5.12: Gráfica de la evolución del entrenamiento de la red FCN-AlexNet.

En la iteración 119 se dan los mejores resultados sobre el conjunto de evaluación, siendo los siguientes:

■ $accuracy = 92,1494$

■ $loss = 0,242652$



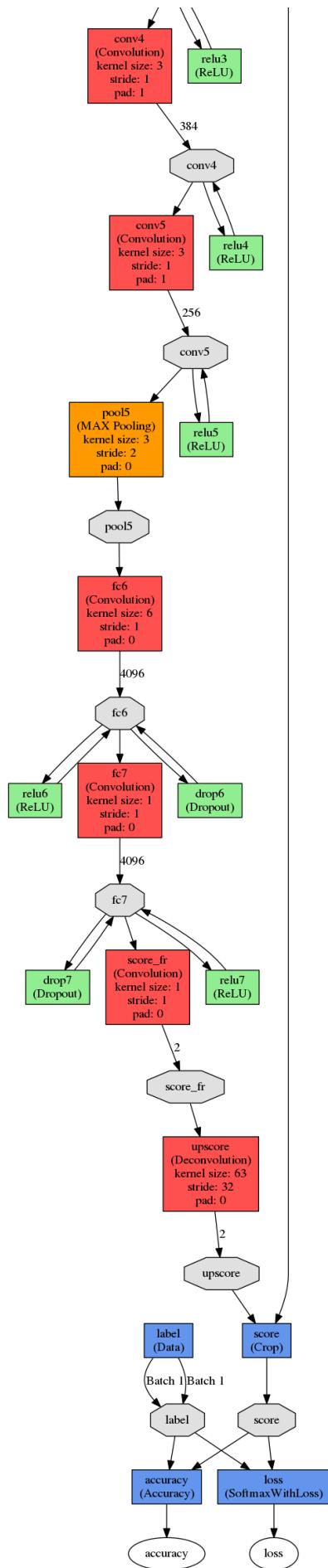


Figura 5.13: Estructura de la red FCN-AlexNet.

Red propuesta

Como red propuesta decidimos utilizar una variante de la red U-NET, debido a los resultados precisos que esta es capaz de conseguir. Sin embargo, decidimos reducir el número de convoluciones y el número de capas para no repercutir en el tiempo de ejecución de la red. En la figura 5.15 se observa su estructura.

Al igual que la red de referencia, solo normalizamos la imagen sin aplicar ningún procesamiento. A diferencia de la red anterior, para esta red realizamos el entrenamiento desde cero, con una inicialización *He*.

El entrenamiento se realiza haciendo uso del algoritmo Adam, con los valores para los parámetros de $\beta_1 = 0,9$, $\beta_2 = 0,999$, $\epsilon = 10^{-8}$ y $\alpha = 0,001$. Asignamos un *batch size* de 1 imagen para el entrenamiento y 1 para la evaluación. El entrenamiento tiene una duración de 10 horas y 28 minutos con 200 iteraciones. En la figura 5.14 podemos ver el progreso del entrenamiento.

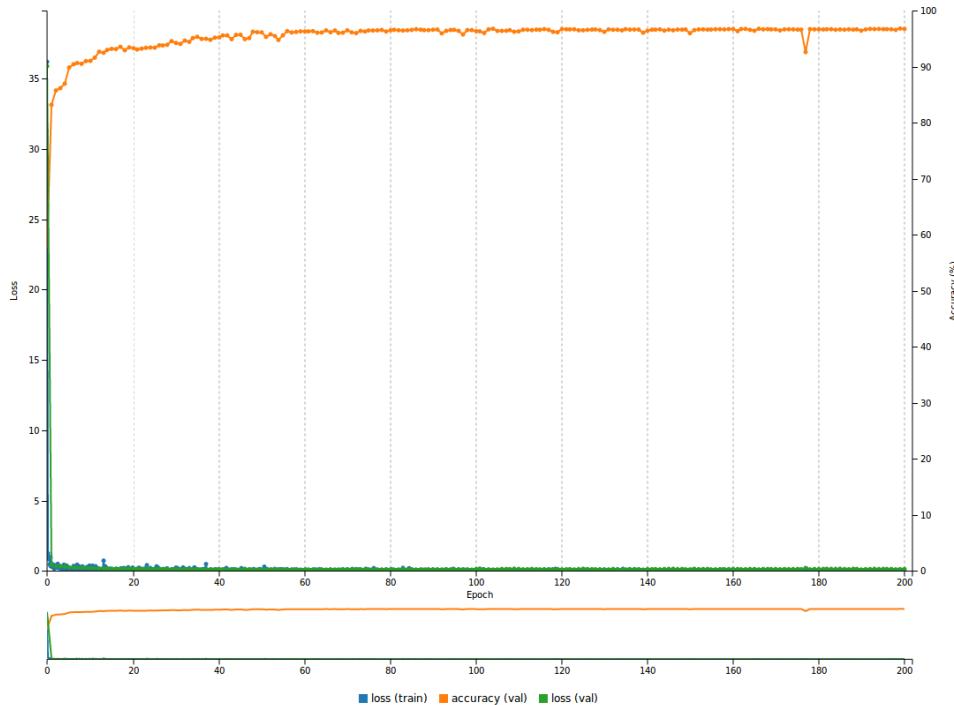
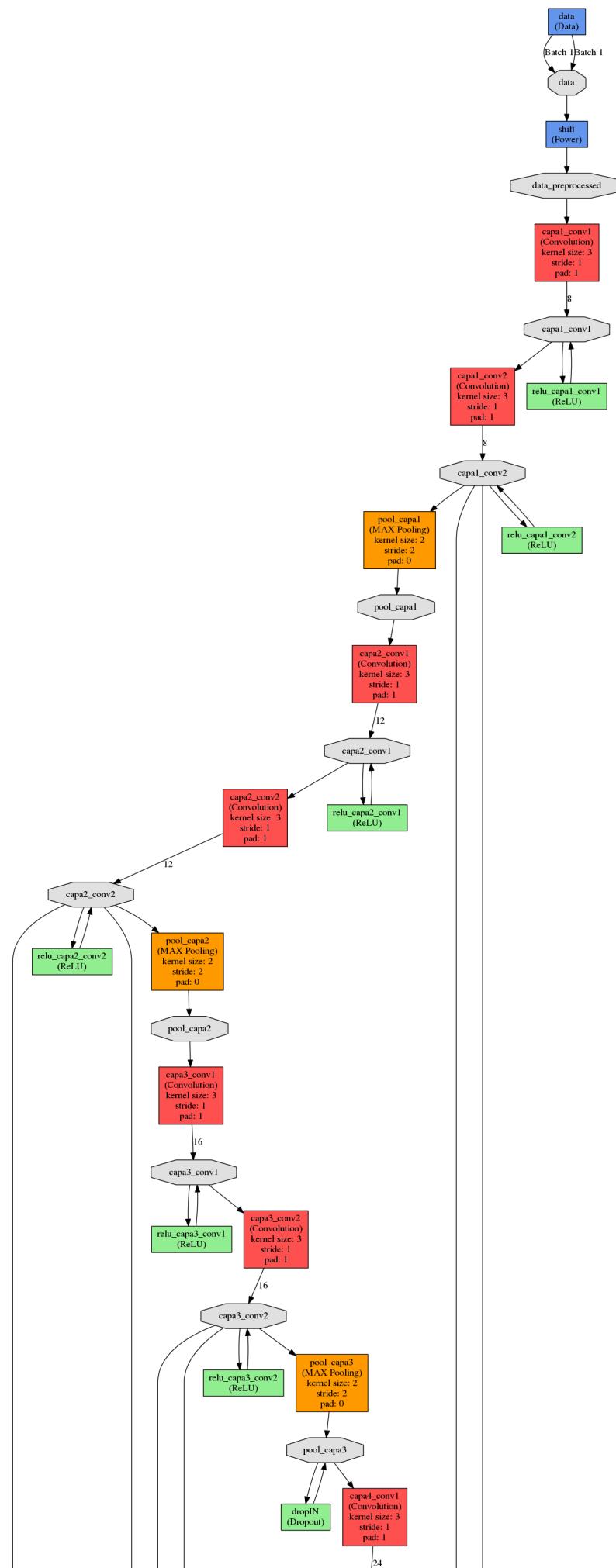
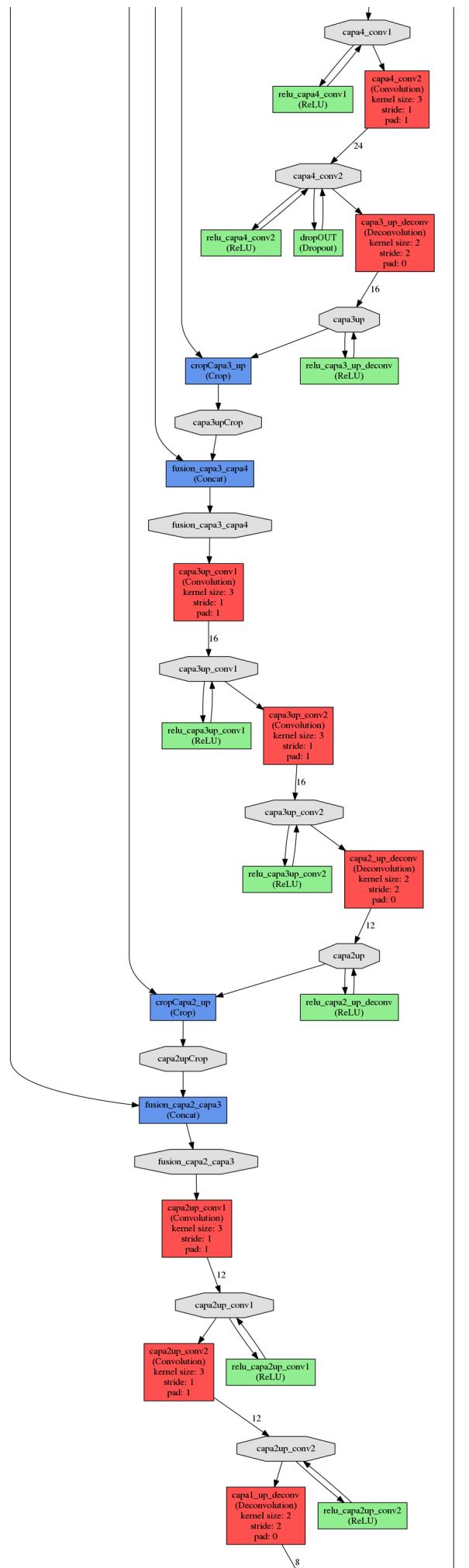


Figura 5.14: Gráfica de la evolución del entrenamiento de la red propuesta para la segmentación.

En la iteración 199 se dan los mejores resultados sobre el conjunto de evaluación, siendo los siguientes:

- $accuracy = 96,87639$
- $loss = 0,132236$





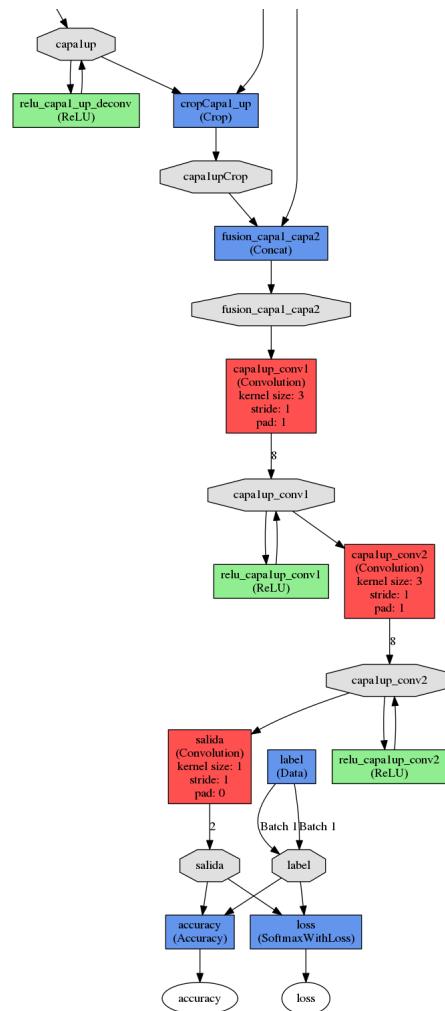


Figura 5.15: Estructura de la red propuesta para la segmentación.

5.2.3. Clasificación

Dado que el problema de la clasificación fue el primero que se abordó con redes neuronales convolucionales, disponemos de multitud de redes para utilizarlas como referencias.

En nuestro caso decidimos de utilizar la red AlexNet. Para nuestro problema solo necesitamos modificar el diseño original para asignar correctamente el número de clases que tenemos.

Para compararnos con esta red, proponemos una con un número de capas mucho más reducida e inspirada en los módulos *inception* (ver Sección 3.6).

Elegiremos en el entrenamiento la red de la iteración con menos error y mayor valor de *accuracy*.

AlexNet

Esta red fue descrita en la Sección 3.10. En la figura 5.17 podemos ver su estructura.

Para la clasificación, únicamente normalizamos la imagen, sin aplicarle ninguna transformación.

El entrenamiento se realiza haciendo uso del algoritmo Adam, con los valores para los parámetros de $\beta_1 = 0,9$, $\beta_2 = 0,999$, $\epsilon = 10^{-8}$ y $\alpha = 0,000001$. Asignamos un *batch size* de 128 imágenes para el entrenamiento y 32 para la evaluación. La red es entrenada desde cero con una inicialización gaussiana (ver Sección 3.4.2). El entrenamiento solo toma 7 minutos en realizar 500 iteraciones. En la figura 5.16 podemos ver el progreso del entrenamiento.

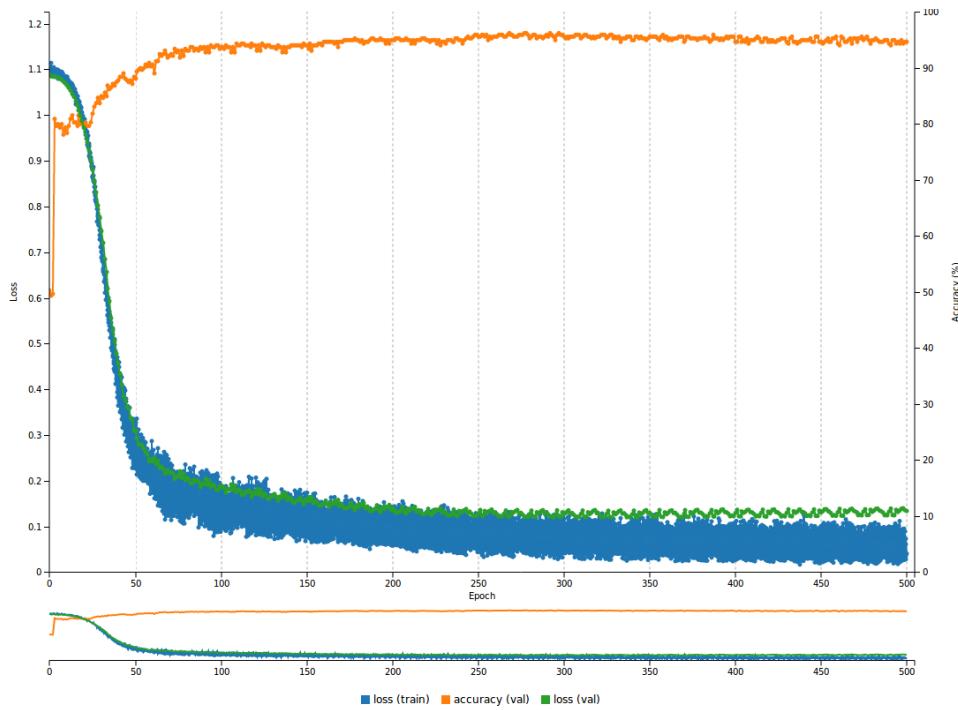
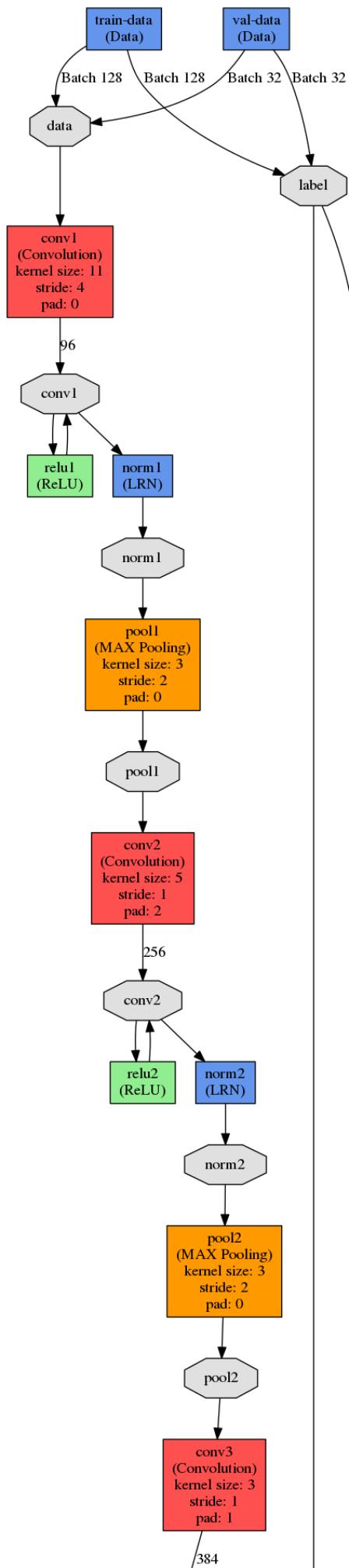


Figura 5.16: Gráfica de la evolución del entrenamiento de la red AlexNet.

En la iteración 297 se dan los mejores resultados sobre el conjunto de evaluación, siendo los siguientes:

- $accuracy = 96,25$
- $loss = 0,121661$



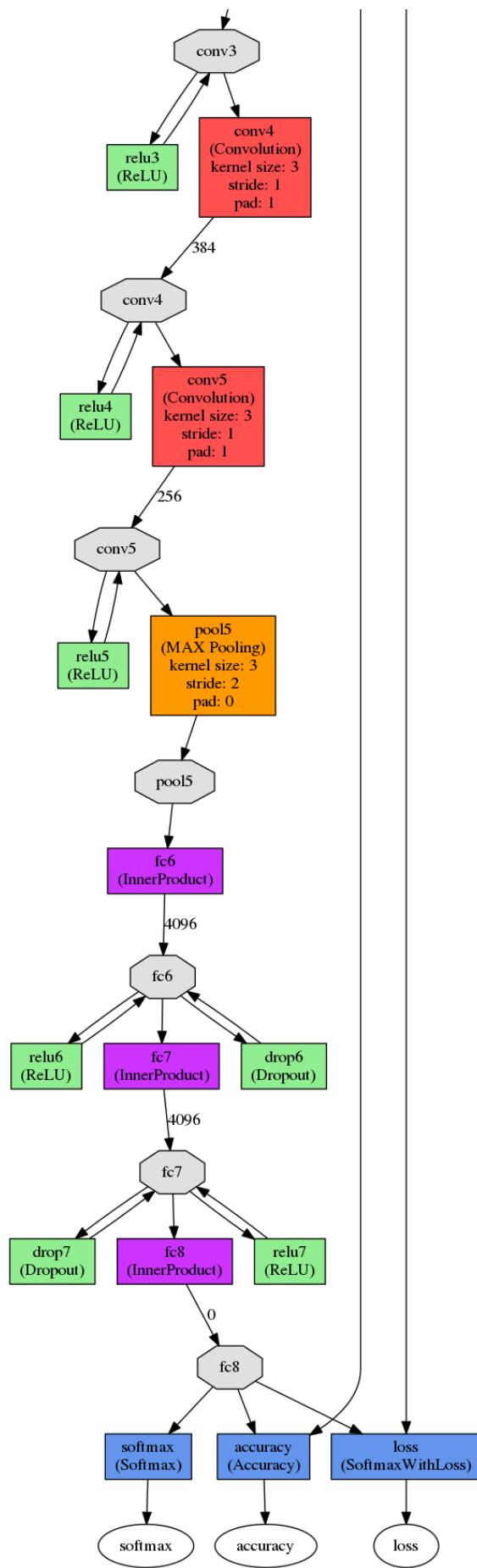


Figura 5.17: Estructura de la red propuesta para la clasificación.

Red propuesta

Tomando como referencia los módulos *inception*, diseñamos una red que use la misma filosofía para extraer características de la imagen. En la figura 5.19 se muestra su estructura.

Como en la red anterior, solo normalizamos la imagen y no aplicamos ninguna transformación.

El entrenamiento se realiza haciendo uso del algoritmo Adam, con los valores para los parámetros de $\beta_1 = 0,9$, $\beta_2 = 0,999$, $\epsilon = 10^{-8}$ y $\alpha = 0,00001$. Asignamos un *batch size* de 30 imagen para el entrenamiento y 15 para la evaluación. La red es entrenada desde cero con una inicialización *Xavier*. El entrenamiento solo dura 1 minuto y realiza 500 iteraciones. En la figura 5.18 podemos ver el progreso del entrenamiento.

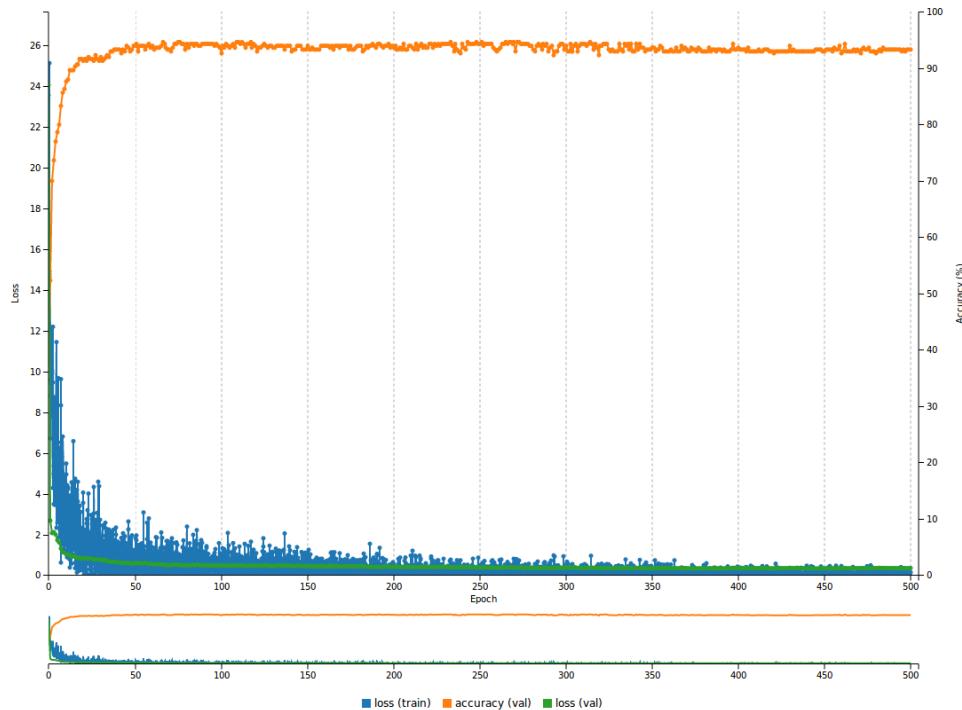
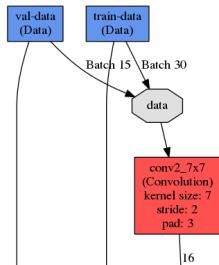


Figura 5.18: Gráfica de la evolución del entrenamiento de la red propuesta para la clasificación

En la iteración 314 se dan los mejores resultados sobre el conjunto de evaluación, siendo los siguientes:

- $accuracy = 94,6667$
- $loss = 0,354492$



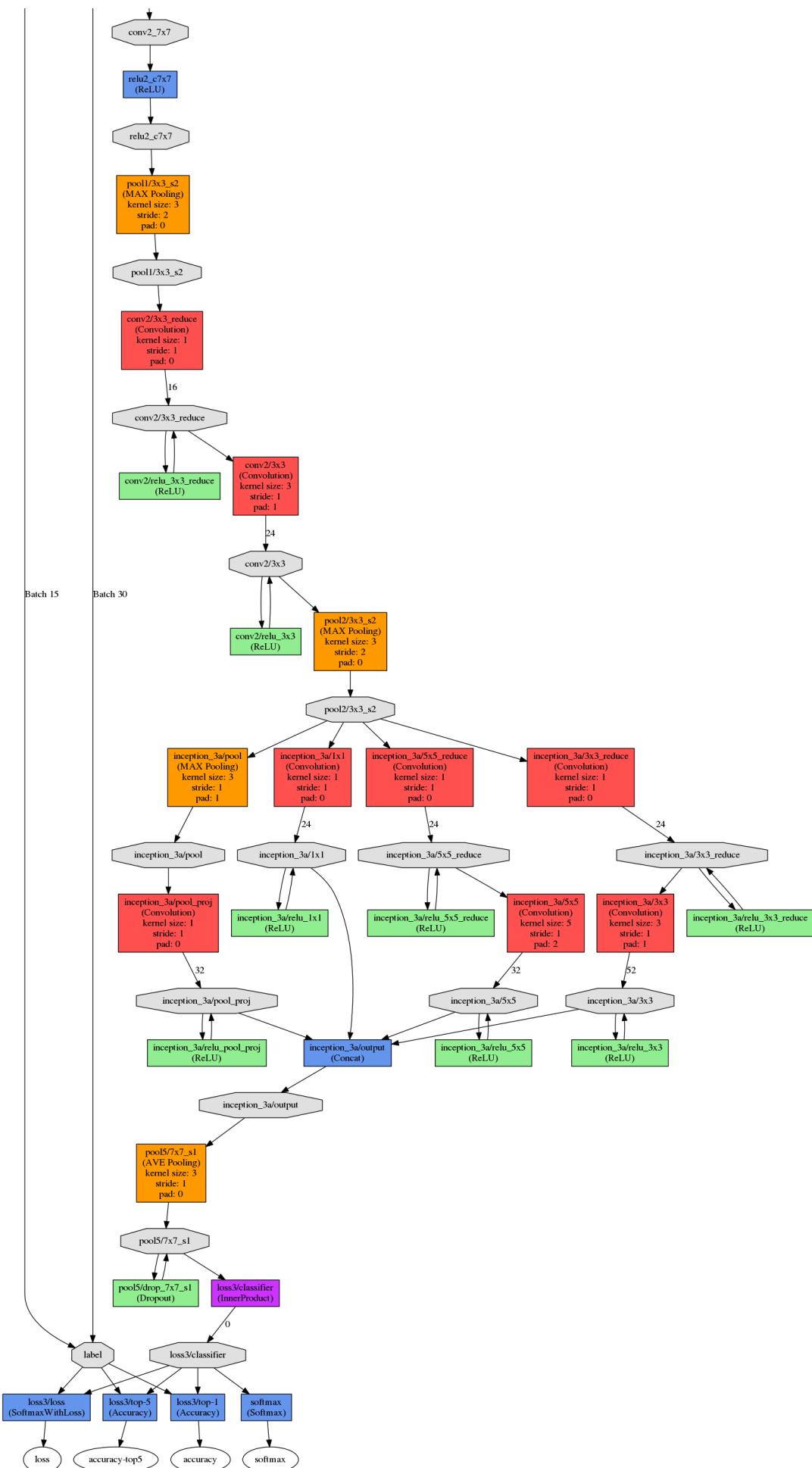


Figura 5.19: Estructura de la red propuesta para la clasificación.

5.3 Resultados

Para presentar los resultados, hacemos usos de los conjuntos de test. Utilizaremos las redes que con las que obtuvimos mejores resultados en el entrenamiento, volviéndolas a comparar sobre estos conjuntos.

Para cada problema y para cada una de nuestras redes, mostraremos las métricas para el conjunto de test, algunos resultados sobre estas imágenes de test y algunas características aprendidas por los filtros de la red propuesta, como ejemplo visual de lo que provoca el entrenamiento de estos sistemas.

5.3.1. Detección

A continuación, presentamos los resultados obtenidos con las dos redes que entramos, utilizando el mejor valor de sus parámetros durante el entrenamiento.

En la figura 5.20 se muestran algunas de las imágenes de test utilizada.



Figura 5.20: Imágenes del conjunto de test para la detección.

Presentamos la matriz de confusión de ambas redes para las imágenes del conjunto de test y algunos ejemplos de la salida de ambas.

DetectNet

Utilizando la red obtenida en la iteración 475 obtenemos la siguiente matriz de confusión:

	Fresas	Negativos
Fresas (predicción)	572	25
Negativos (predicción)	32	-

Tabla 5.1: Matriz de confusión

Lo que supone un 95.81 % de *accuracy* y un 94.70 % de *recall*.

En la figura 5.21 se muestran algunos resultados obtenido de esta red sobre el conjunto de test.



Figura 5.21: Resultados de la red DetectNet sobre el conjunto de test.

Red propuesta

Utilizando la red obtenida en la iteración 3053 obtenemos la siguiente matriz de confusión:

	Fresas	Negativos
Fresas (predicción)	515	37
Negativos (predicción)	89	-

Tabla 5.2: Matriz de confusión.

Lo que supone un 93.92 % de *accuracy* y un 86.54 % de *recall*.

En la figura 5.22 se muestran algunos de los resultados obtenido de esta red sobre el conjunto de test y en la figura 5.23 se muestran que filtros ha aprendido la red y las características que extraen.



Figura 5.22: Resultados de la red propuesta para la detección sobre el conjunto de test.

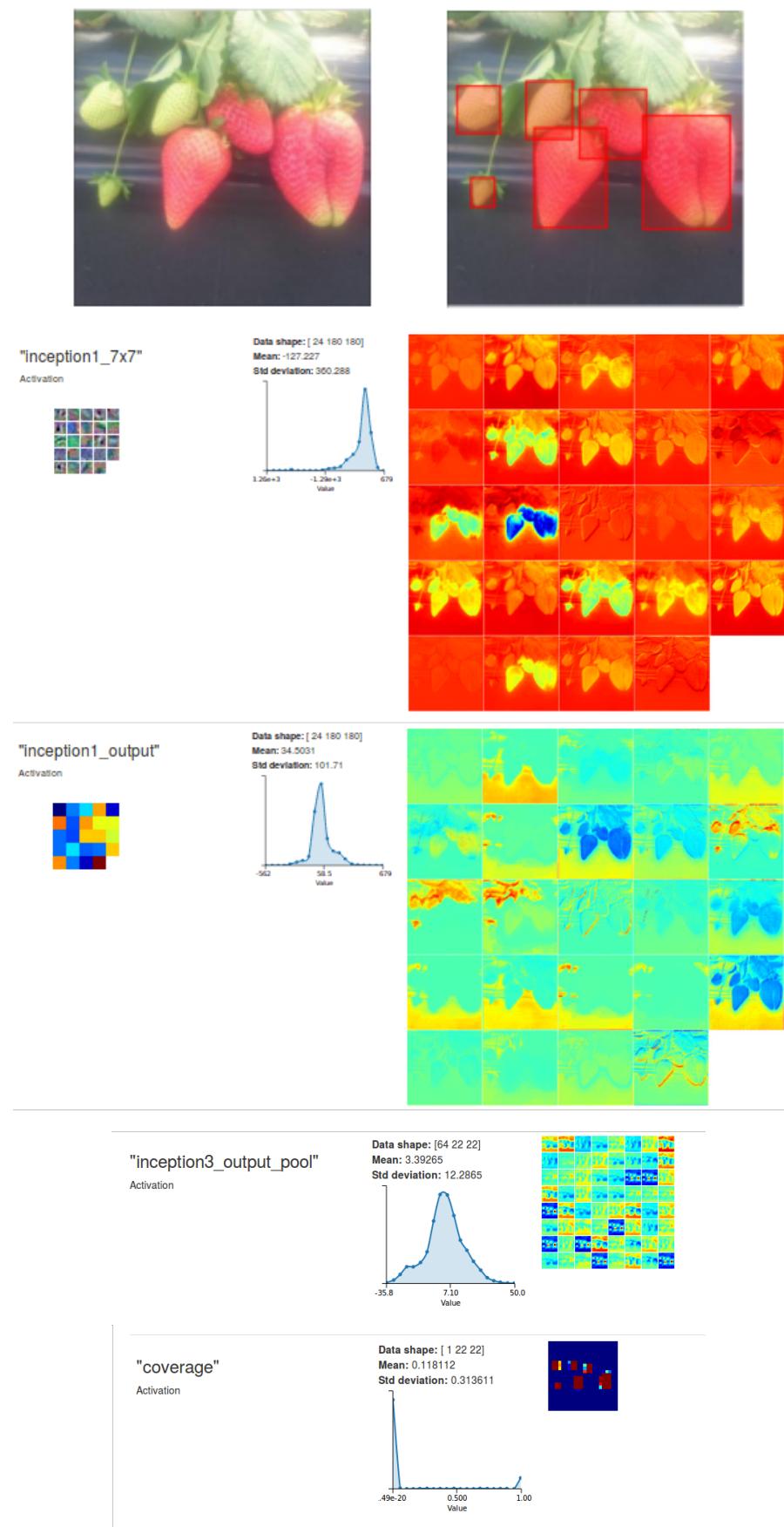


Figura 5.23: Algunos filtros y características extraídas por la red.

5.3.2. Segmentación

En las siguientes secciones presentaremos los resultados obtenidos por las redes utilizando la mejor red que obtuvimos de cada entrenamiento.

Aportamos las métricas de ambas redes para las imágenes del conjunto de test y algunos ejemplos de la salida de ambas. También mostraremos, para la red propuesta, algunos filtros aprendidos y que características busca en la imagen.

En la figura 5.24 se muestran algunas de las imágenes de test utilizada.



Figura 5.24: Imágenes del conjunto de test para la segmentación

FCN-AlexNet

Utilizando la red obtenida en la iteración 119 obtenemos un 92.7249 % de *accuracy* con un error de 0.222691 sobre el conjunto de test.

En la figura 5.25 se muestran algunos resultados obtenido de esta red sobre imágenes de 360×360 .



Figura 5.25: Resultados de la red FCN-AlexNet sobre algunas imágenes.

Red propuesta

Con la red de la iteración 199 conseguimos un 97.2791 % de *accuracy* con un error de 0.0948996 sobre el conjunto de test.

En la figura 5.26 se muestran algunos resultados obtenido de esta red sobre imágenes de 360×360 . Y en la figura 5.27 podemos observar algunos de los filtros aprendidos por la red y que características extraen.

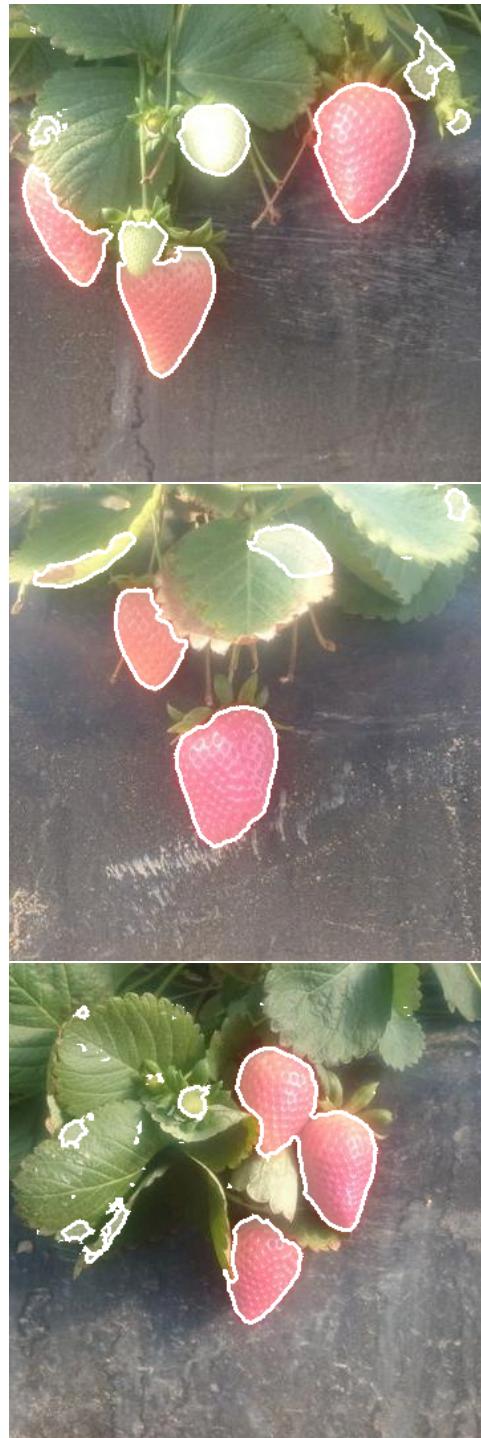


Figura 5.26: Resultados de la red propuesta para la segmentación sobre algunas imágenes.

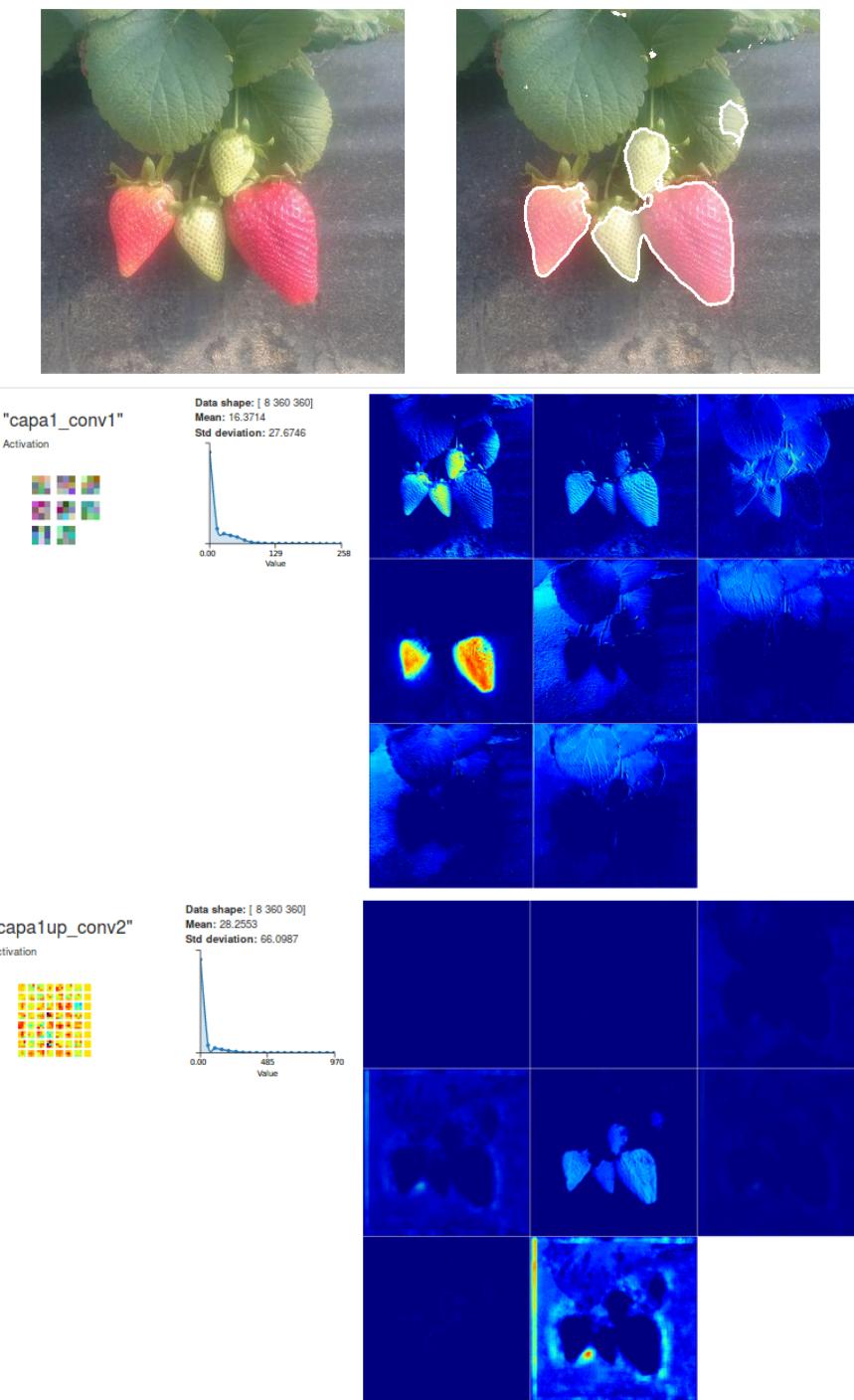


Figura 5.27: Algunos filtros y características extraídas por la red.

5.3.3. Clasificación

En estas sección se recoge los resultados obtenidos por las mejores redes del entrenamiento.

Se muestra, para cada red, la matriz de confusión y las métricas obtenidas sobre el conjunto de test y algunos ejemplos de la salida de ambas redes para estas imágenes, además de algunos ejemplos de filtros aprendidos.

En la figura 5.28 se muestran algunas de las imágenes de test utilizadas.

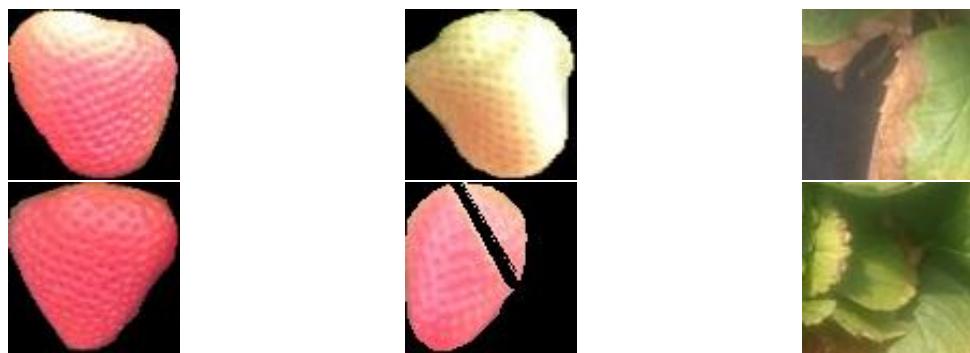


Figura 5.28: Imágenes del conjunto de test para la clasificación.

AlexNet

Utilizando la red obtenida en la iteración 297 obtenemos la siguiente matriz de confusión:

	Madura	Inmadura	Negativo	TOTAL
Madura (predicción)	146	9	0	155
Inmadura (predicción)	4	141	0	145
Negativo (predicción)	0	0	150	150
TOTAL	150	150	150	450

Tabla 5.3: Matriz de confusión.

En la figura 5.29 se muestran algunos ejemplos de imágenes clasificadas por la red.

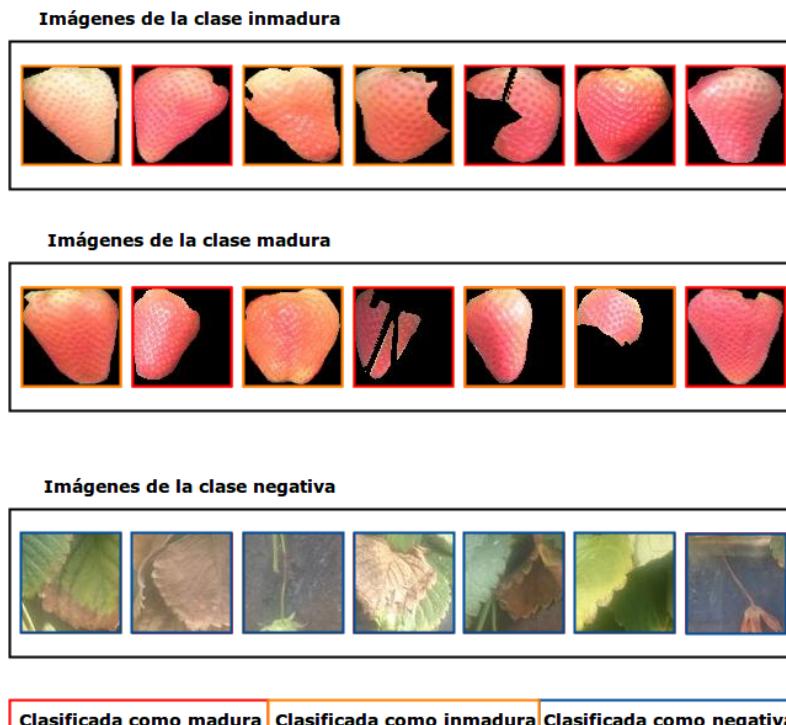


Figura 5.29: Clasificación realizada por la red sobre imágenes de test.

Red propuesta

Utilizando la red obtenida en la iteración 314 obtenemos la siguiente matriz de confusión:

	Madura	Inmadura	Negativo	TOTAL
Madura (predicción)	142	9	0	151
Inmadura (predicción)	8	141	0	149
Negativo (predicción)	0	0	150	150
TOTAL	150	150	150	450

Tabla 5.4: Matriz de confusión.

En la figura 5.30 se muestran algunos ejemplos de imágenes clasificadas por la red. En la figura 5.31 podemos ver alguno de los filtros aprendidos y que características buscan.

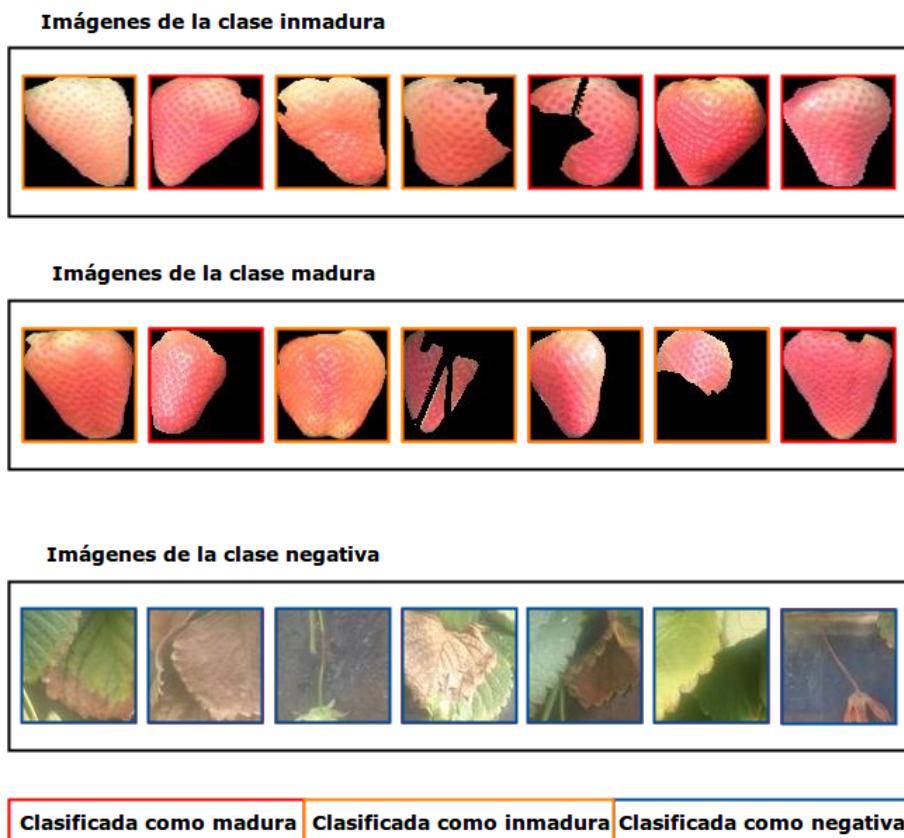


Figura 5.30: Clasificación realizada por la red sobre imágenes de test.

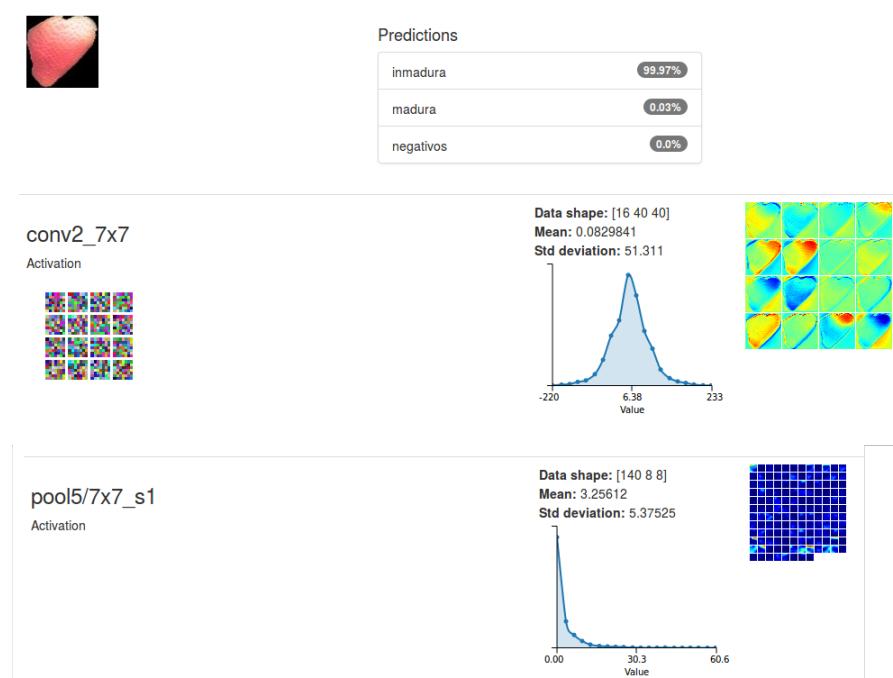


Figura 5.31: Algunos filtros y características extraídas por la red.

5.3.4. Análisis de los resultados

Para el subsistema de detección, tal y como observamos en la tabla 5.5, ambas redes consiguen unas métricas de *accuracy* parecidas, aunque existe una notoria diferencia en el *recall*, viéndose reflejado en la pérdida de algunas fresas por parte de la red propuesta. Siendo la detección una etapa crítica, sería más conveniente optar por la red de referencia, en lugar de la red propuesta, para el subsistema de detección.

	<i>Accuracy</i>	<i>Recall</i>
Red propuesta	93.92 %	86.54 %
Red de referencia	95.81 %	94.70 %

Tabla 5.5: Comparativa de las métricas para ambas redes en la detección.

En el subsistema de segmentación, se obtienen mejores resultados (ver tabla 5.6) y con detalles más precisos en la red propuesta, siendo esta, a su vez, más ligera que la red de referencia. Dado que nos interesa segmentar de la forma más precisa posible la fresa, nos decantaríamos por esta red.

	<i>Accuracy</i>	Error (conjunto de test)
Red propuesta	97.2791 %	0.0948996
Red de referencia	92.7249 %	0.222691

Tabla 5.6: Comparativa de las métricas para ambas redes en la segmentación.

En el caso de la clasificación, encontramos resultados similares en ambas redes (ver tabla 5.7 y 5.8), produciéndose los errores en fresas que sería discutible su pertenencia a la clase que tienen asignadas; por lo que podemos decir que ambas redes dan buenos resultados. Dado que la red propuesta es más reducida que la de referencia, parece mejor opción utilizar la red propuesta.

	Madura	Inmadura	Negativo	TOTAL
Madura (predicción)	146	9	0	155
Inmadura (predicción)	4	141	0	145
Negativo (predicción)	0	0	150	150
TOTAL	150	150	150	450

Tabla 5.7: Matriz de confusión de la red de referencia para la clasificación.

	Madura	Inmadura	Negativo	TOTAL
Madura (predicción)	142	9	0	151
Inmadura (predicción)	8	141	0	149
Negativo (predicción)	0	0	150	150
TOTAL	150	150	150	450

Tabla 5.8: Matriz de confusión de la red propuesta para la clasificación.

CAPÍTULO 6

Conclusiones y trabajo futuro

En este Trabajo de Fin de Grado se ha creado un sistema basado en técnicas de *deep learning*, con el fin de identificar, en imágenes, las fresas y cuáles se deben recolectar. La creación de este sistema ha supuesto la culminación de todos los objetivos planteados y que se resumen de la siguiente forma:

- **Introducción al *deep learning*:** se ha realizado una introducción al *deep learning* que nos ha permitido situar a este como un nuevo campo dentro del *machine learning*, analizando las ventajas que supone con respecto al *machine learning* y algunas de las técnicas que aporta, entre ellas, las redes neuronales convolucionales, que fueron las elegidas para abordar nuestro problema.
- **Estudio teórico del *deep learning*:** se han abordado todos los fundamentos teóricos necesarios para comprender y utilizar las redes neuronales convolucionales. Esto nos ha permitido presentar casos de uso de estas redes e ir familiarizándonos con ellas, para, a continuación, afrontar el desarrollo de estas, desde su diseño hasta su entrenamiento.
- **Deep learning aplicado al procesamiento de imágenes:** se ha profundizado en el conocimiento de las redes neuronales convolucionales aplicadas al procesamiento de imágenes y a sus tres principales aplicaciones: clasificación, detección y segmentación.
- **Aplicación del *deep learning* a un problema real:** se ha aplicado las redes neuronales convolucionales a la identificación de fresas en imágenes.
 - **Obtención y preparación de los datos necesarios para los algoritmos utilizados:** se han adquirido un total de 700 imágenes para poder entrenar y evaluar nuestras redes. Estas imágenes se han marcado a mano una a una para posteriormente elaborar los diferentes conjuntos de datos.
 - **Diseño y evaluación de los sistemas creados para detección, segmentación y clasificación de fresas en imágenes:** se ha realizado, para cada sistema, una comparativa entre una red de referencia y una red propuesta en el marco de este trabajo, junto con los resultados obtenidos por estas redes y un análisis de los mismos. Estos resultados demuestran la viabilidad de su implementación práctica.
- **Diseño de un sistema *hardware* para trabajar con *deep learning*:** toda la implementación de nuestro sistema ha requerido del diseño de un equipo informático equipado con los componentes necesarios para poder crear el sistema junto con el correspondiente *software*.

A pesar de ser una tecnología reciente y con escasa documentación, todo nuestro esfuerzo se ha visto recompensado, al ser capaces de aplicar todo lo estudiado a un problema real y comprender su funcionamiento con bastante profundidad.

Utilizando únicamente las imágenes captadas por la cámara de un móvil, en un entorno abierto, sin ningún control de las condiciones ambientales y sin ningún tipo de preprocesamiento, hemos sido capaces de detectar las fresas presentes en la imagen y determinar cuáles de ellas estaban listas para su recolección.

Como trabajo futuro queda estudiar cómo afecta a los resultados los cambios en la estructura y el entrenamiento de la red y los cambios en los datos que disponemos. Los cambios en la estructura pueden ser realizados a mano, ya sea estableciendo más capas o más operaciones en cada una, o a través de un algoritmo genético que nos mejore la estructura que tenemos; además de estudiar el efecto que tiene el partir de una red ya entrenada en otro conjunto de datos más grandes. En lo referente a los cambios en los datos, queda ver cómo afecta a los resultados el utilizar algún preprocesamiento sobre las imágenes u obtener más imágenes para el entrenamiento.

Bibliografía

- [Turing, 1936] A. M. Turing. (1936). *On computable numbers, with an application to the Entscheidungsproblem.* https://www.cs.virginia.edu/~robins/Turing_Paper_1936.pdf.
- [Turing, 1950] A. M. Turing. (1950). *Computing Machinery and Intelligence.* <https://www.csee.umbc.edu/courses/471/papers/turing.pdf>.
- [McCulloch y Pitts, 1943] W. S. McCulloch y W. H. Pitts. (1943). *A logical calculus of the ideas immanent in nervous activity.* <https://pdfs.semanticscholar.org/5272/8a99829792c3272043842455f3a110e841b1.pdf>.
- [Hebb, 1949] D. O. Hebb. (1949). *The Organization of Behavior.* http://s-f-walker.org.uk/pubsebooks/pdfs/The_Organization_of_Behavior-Donald_O._Hebb.pdf.
- [Rosenblatt, 1958] F. Rosenblatt. (1958). *The perceptron: a probabilistic model for information storage and organization in the brain.* <http://www.ling.upenn.edu/courses/cogs501/Rosenblatt1958.pdf>.
- [Widrow, 1960] B. Widrow. (1960). *An adaptive "ADALINE" neuron using chemical "memistors".* <http://www-isl.stanford.edu/~widrow/papers/t1960anadaptive.pdf>.
- [Fix y Hodges, 1951] E. Fix y J.L. Hodges (1951). *An Important Contribution to Nonparametric Discriminant Analysis and Density Estimation.* <http://www-isl.stanford.edu/~widrow/papers/t1960anadaptive.pdf>.
- [Vapnik y Lerner, 1963] V. N. Vapnik y A. Y. Lerner. (1963). *Pattern recognition using generalized portrait method.* <http://web.cs.iastate.edu/~cs573x/vapnik-portraits1963.pdf>.
- [Minsky y Papert, 1969] M. Minsky y S. Papert. (1969). *Perceptrons. An Introduction to Computational Geometry.* <http://science.sciencemag.org/content/165/3895/780>.
- [Werbos, 1974] P. Werbos. (1974). *Beyond regression : new tools for prediction and analysis in the behavioral sciences.* https://www.researchgate.net/publication/35657389_Beyond_regression_new_tools_for_prediction_and_analysis_in_the_behavioral_sciences.
- [Fukushima, 1980] K. Fukushima. (1980). *Neocognitron: A Self-organizing Neural Network Model for a Mechanism of Pattern Recognition. Unaffected by Shift in Position.* <http://www.cs.princeton.edu/courses/archive/spr08/cos598B/Readings/Fukushima1980.pdf>.
- [Hubel y Wiesel, 1959] D. H. Hubel y T. N. Wiesel. (1959). *Receptive fields of single neurones in the cat's striate cortex.* <https://www.ncbi.nlm.nih.gov/pmc/articles/PMC1363130/pdf/jphysiol01298-0128.pdf>.

- [Hopfield, 1982] J. J. Hopfield. (1982). *Neural networks and physical systems with emergent collective computational abilities.* <http://www.pnas.org/content/79/8/2554.full.pdf>.
- [Ackey, et al., 1985] D. H. Ackey, G. E. Hinton y T. J. Sejnowski. (1985). *A learning algorithm for Boltzmann Machines.* <http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.935.4615&rep=rep1&type=pdf>.
- [Watkins, 1989] C. J. C. H. Watkins. (1989). *Learning from Delayed Rewards.* http://www.cs.rhul.ac.uk/~chrisw/new_thesis.pdf.
- [Rumelhart, et al., 1986] D. E. Rumelhart, G. E. Hinton y R. J. Williams. (1986) *Learning representations by back-propagating errors.* https://www.iro.umontreal.ca/~vincentp/ift3395/lectures/backprop_old.pdf.
- [LeCun, et al., 1989] Y. LeCun, L. Bottou, Y. Bengio y P. Haffner. (1998) *Gradient-based learning applied to document recognition.* <http://yann.lecun.com/exdb/publis/pdf/lecun-89e.pdf>.
- [Hochreiter y Schmidhuber, 1997] S. Hochreiter y J. Schmidhuber. (1997) *Long short-term memory.* https://www.researchgate.net/profile/Sepp_Hochreiter/publication/13853244_Long_Short-term_Memory/links/5700e75608aea6b7746a0624/Long-Short-term-Memory.pdf.
- [LeCun, et al., 1998] Y. LeCun, B. Boser, J. S. Denker, D. Henderson, R. E. Howard, W. Hubbard, L. D. Jackel. (1998) *Backpropagation Applied to Handwritten Zip Code Recognition.* <http://yann.lecun.com/exdb/publis/pdf/lecun-98.pdf>.
- [Hinton, et al., 2006] G. E. Hinton, S. Osindero y Y. Teh. (2006) *A fast learning algorithm for deep belief nets.* <http://www.cs.toronto.edu/~hinton/absps/fastnc.pdf>.
- [Krizhevsky, et al., 2012] A. Krizhevsky, I. Sutskever y G. E. Hinton. (2012) *ImageNet Classification with Deep Convolutional Neural Networks.* <https://papers.nips.cc/paper/4824-imagenet-classification-with-deep-convolutional-neural-networks.pdf>.
- [Russakovsky, et al., 2015] O. Russakovsky, J. Deng, H. Su, J. Krause, S. Satheesh, S. Ma, Z. Huang, A. Karpathy, A. Khosla, M. Bernstein, A. C. Berg y L. Fei-Fei. (2015) *ImageNet: A Large-Scale Hierarchical Image Database.* <https://arxiv.org/pdf/1409.0575.pdf>.
- [Deng, et al., 2009] J. Deng, W. Dong, R. Socher, L. Li, K. Li y L. Fei-Fei. (2009) *ImageNet: A Large-Scale Hierarchical Image Database.* http://www.image-net.org/papers/imagenet_cvpr09.pdf.
- [He, et al., 2015] K. He, X. Zhang, S. Ren y J. Sun. (2015) *Deep Residual Learning for Image Recognition.* <https://arxiv.org/pdf/1512.03385v1.pdf>.
- [Lin, et al., 2014] T. Lin, M. Maire, S. Belongie, L. Bourdev, R. Girshick, J. Hays, P. Perona, D. Ramanan, C. L. Zitnick y P. Dollár. (2014) *Microsoft COCO: Common Objects in Context.* <https://arxiv.org/pdf/1405.0312.pdf>.
- [Li, et al., 2016] Y. Li, H. Qi, J. Dai, X. Ji y Y. Wei. (2016) *Fully Convolutional Instance-aware Semantic Segmentation.* <https://arxiv.org/pdf/1611.07709.pdf>.

- [Liu, et al., 2017] Y. Liu, K. Gadepalli, M. Norouzi, G. Dahl, T. Kohlberger, A. Boyko, S. Venugopalan, A. Timofeev, P. Nelson, G. Corrado, J. Hipp, L. Peng y M. Stumpe. (2017) *Detecting Cancer Metastases on Gigapixel Pathology Images*. <https://arxiv.org/pdf/1703.02442.pdf>.
- [Wu, et al., 2016] Y. Wu, M. Schuster, Z. Chen, Q. Le, M. Norouzi, W. Macherey, M. Kruskun, Y. Cao, Q. Gao, K. Macherey, J. Klingner, A. Shah, M. Johnson, X. Liu, L. Laiser, S. Gouws, Y. Kato, T. Kudo, H. Kazawa, K. Stevens, G. Kurian, N. Patil, W. Wang, C. Young, J. Smith, J. Riesa, A. Rudnick, O. Vinyals, G. Corrado, M. Hughes y J. Dean. (2016) *Google's Neural Machine Translation System: Bridging the Gap between Human and Machine Translation*. <https://arxiv.org/pdf/1609.08144.pdf>.
- [Arik, et al., 2017] S. Arik, M. Chrzanowski, A. Coates, G. Diamos, A. Gibiansky, Y. Kang, X. Li, J. Miller, A. Ng, J. Raiman, S. Sengupta y M. Shoeybi. (2017) *Deep Voice: Real-time Neural Text-to-Speech*. <https://arxiv.org/pdf/1702.07825.pdf>.
- [Goodfellow et al., 2014] I. Goodfellow, J. Pouget-Abadie, M. Mirza, B. Xu, D. Warde-Farley, S. Ozair, A. Courville y Y. Bengio. (2014) *Generative Adversarial Nets*. <https://arxiv.org/pdf/1406.2661.pdf>.
- [Zhang et al., 2016] H. Zhang, T. Xu, H. Li, S. Zhang, X. Huang, X. Wang y D. Metaxas. (2016) *StackGAN: Text to Photo-realistic Image Synthesis with Stacked Generative Adversarial Networks*. <https://arxiv.org/pdf/1612.03242v1.pdf>.
- [Vinyals et al., 2016] O. Vinyals, A. Toshev, S. Bengio, y D. Erhan. (2016) *Show and Tell: Lessons learned from the 2015 MSCOCO Image Captioning Challenge*. <https://arxiv.org/pdf/1609.06647.pdf>.
- [Baddeley, 1974] A. Baddeley. (1974) *Working memory*. <http://www.cs.indiana.edu/~port/HDphonol/Baddeley.wkg.mem.Science.pdf>.
- [Graves, et al., 2014] A. Graves, G. Wayne e I. Danihelka. (2014) *Neural Turing Machines*. <https://arxiv.org/pdf/1410.5401.pdf>.
- [Graves, et al., 2016] A. Graves, G. Wayne, M. Reynolds, T. Harley, I. Danihelka, A. Grabska-Barwińska, S. Colmenarejo, E. Grefenstette, T. Ramalho, J. Agapiou, A. Badia, K. Hermann, Y. Zwols, G. Ostrovski, A. Cain, H. King, C. Summerfield, P. Blunsom, K. Kavukcuoglu y D. Hassabis. (2016) *Hybrid computing using a neural network with dynamic external memory*. <http://www.nature.com/nature/journal/v538/n7626/full/nature20101.html>.
- [Beattie, et al., 2016] C. Beattie, J. Z. Leibo, D. Teplyashin, T. Ward, M. Wainwright, H. Küttler, A. Lefrancq, S. Green, V. Valdés, A. Sadik, J. Schrittwieser, K. Anderson, S. York, M. Cant, A. Cain, A. Bolton, S. Gaffney, H. King, D. Hassabis, S. Legg y S. Petersen. (2016) *DeepMind Lab*. <https://arxiv.org/pdf/1612.03801.pdf>.
- [OpenAI, 2017] OpenAI. (2017) *OpenAI Universe*. <https://universe.openai.com/>.
- [Mnih et al., 2013] V. Mnih, K. Kavukcuoglu D. Silver, A. Graves, I. Antonoglou, D. Wierstra y M. Riedmiller. (2013) *Playing Atari with Deep Reinforcement Learning*. <https://arxiv.org/pdf/1312.5602v1.pdf>.
- [Silver et al., 2016] D. Silver, A. Huang, C. J. Maddison, A. Guez, L. Sifre, G. van den Driessche, J. Schrittwieser, I. Antonoglou, V. Panneershelvam, M. Lanctot, S. Dieleman, D. Grewe, J. Nham, N. Kalchbrenner, I. Sutskever, T. Lillicrap, M. Leach, K. Kavukcuoglu, T. Graepel y D. Hassabis. (2016) *Mastering the game of Go with deep*

- neural networks and tree search.* <https://storage.googleapis.com/deepmind-media/alphago/AlphaGoNaturePaper.pdf>.
- [DeepMind, 2016] DeepMind. (2016) *DeepMind and Blizzard to release StarCraft II as an AI research environment.* <https://deepmind.com/blog/deepmind-and-blizzard-release-starcraft-ii-ai-research-environment/>.
- [Santoro et al., 2016] A. Santoro, S. Bartunov, M. Botvinick, D. Wierstra y T. Lillicrap. (2016) *One-shot Learning with Memory-Augmented Neural Networks.* <https://arxiv.org/pdf/1605.06065.pdf>.
- [Shazeer et al., 2017] N. Shazeer, A. Mirhoseini, K. Maziarz, A. Davis, Q. Le, G. Hinton y J. Dean. (2017) *Outrageously Large Neural Networks: The Sparsely-Gated Mixture-of-Experts Layer.* <https://arxiv.org/pdf/1701.06538.pdf>.
- [Fernando et al., 2017] C. Fernando, D. Banarse, C. Blundell, Y. Zwols, D. Ha, A. A. Rusu, A. Pritzel y D. Wierstra. (2017) *PathNet: Evolution Channels Gradient Descent in Super Neural Networks.* <https://arxiv.org/pdf/1701.08734.pdf>.
- [Zeiler et al., 2014] M. D. Zeiler y R. Fergus. (2014) *Visualizing and Understanding Convolutional Networks.* <https://www.cs.nyu.edu/~fergus/papers/zeilerECCV2014.pdf>.
- [Szegedy et al., 2014] C. Szegedy, W. Liu, Y. Jia, P. Sermanet, S. Reed, D. Anguelov, D. Erhan, V. Vanhoucke y A. Rabinovich. (2014) *Going Deeper with Convolutions.* <https://arxiv.org/pdf/1409.4842.pdf>.
- [Szegedy et al., 2015] C. Szegedy, V. Vanhoucke, S. Ioffe, J. Shlens y Z. Wojna. (2015) *Rethinking the Inception Architecture for Computer Vision.* <https://arxiv.org/pdf/1512.00567.pdf>.
- [Szegedy et al., 2016] C. Szegedy, S. Ioffe, V. Vanhoucke y A. Alemi. (2016) *Inception-v4, Inception-ResNet and the Impact of Residual Connections on Learning.* <https://arxiv.org/pdf/1602.07261.pdf>.
- [LeCun et al., 1998] Yann LeCun, Corinna Cortes y C. J.C. Burges. (1998) *THE MNIST DATABASE of handwritten digits.* <http://yann.lecun.com/exdb/mnist/>.
- [Simonyan et al., 2014] K. Simonyan y A. Zisserman. (2014) *Very Deep Convolutional Networks for Large-Scale Visual Recognition.* <https://arxiv.org/pdf/1409.1556.pdf>.
- [Cho et al., 2015] J. Cho, K. Lee, E. Shin, G. Choy y S. Do. (2015) *How much data is needed to train a medical image deep learning system to achieve necessary high accuracy?.* <https://arxiv.org/pdf/1511.06348.pdf>.
- [Soekhoe et al., 2016] D. Soekhoe, P. van der Putten y A. Plaat. (2016) *On the Impact of data set Size in Transfer Learning using Deep Neural Networks.* http://liacs.leidenuniv.nl/~plaata1/papers/ida2016_camera_ready.pdf.
- [He et al., 2015] K. He, X. Zhang, S. Ren y J. Sun. (2015) *Delving Deep into Rectifiers: Surpassing Human-Level Performance on ImageNet Classification.* <https://arxiv.org/pdf/1502.01852v1.pdf>.
- [Glorot et al., 2009] X. Glorot y Y. Bengio. (2009) *Understanding the difficulty of training deep feedforward neural networks.* http://machinelearning.wustl.edu/mlpapers/paper_files/AISTATS2010_GlorotB10.pdf.

- [Razavian et al., 2014] A. S. Razavian, H. Azizpour, J. Sullivan y S. Carlsson. (2014) *CNN Features off-the-shelf: an Astounding Baseline for Recognition.* http://www.cv-foundation.org/openaccess/content_cvpr_workshops_2014/W15/papers/Razavian_CNN_Features_Off-the-Shelf_2014_CVPR_paper.pdf.
- [Oquab et al., 2014] M. Oquab, L. Bottou, I. Laptev y J. Sivic. (2014) *Learning and Transferring Mid-Level Image Representations using Convolutional Neural Networks.* http://www.cv-foundation.org/openaccess/content_cvpr_2014/papers/Oquab_Learning_and_Transferring_2014_CVPR_paper.pdf.
- [Cauchy, 1847] A. Cauchy. (1847) *Méthode générale pour la résolution des systèmes d'équations simultanées.* <http://gallica.bnf.fr/ark:/12148/bpt6k90190w/f406>.
- [Qian, 1999] N. Qian. (1999) *On the momentum term in gradient descent learning algorithms.* <http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.57.5612&rep=rep1&type=pdf>.
- [Sutton, 1986] R. Sutton. (1986) *Two Problems with Backpropagation and Other Steepest-Descent Learning Procedures for Networks.* https://wiki.eecs.yorku.ca/course_archive/2013-14/F/4403/_media/sutton-86.pdf.
- [Kingma et al., 2014] D. P. Kingma y J. Ba. (2014) *Adam: a Method for Stochastic Optimization..* <https://arxiv.org/pdf/1412.6980.pdf>.
- [Ioffe et al., 2015] S. Ioffe, C. Szegedy (2015) *Batch Normalization: Accelerating Deep Network Training by Reducing Internal Covariate Shift.* <https://arxiv.org/pdf/1502.03167.pdf>.
- [Goodfellow et al., 2014] I. J. Goodfellow, Y. Bulatov, J. Ibarz, S. Arnoud y V. Shet. (2014) *Multi-digit Number Recognition from Street View Imagery using Deep Convolutional Neural Networks.* <https://arxiv.org/pdf/1312.6082.pdf>.
- [Srivastava et al., 2014] N. Srivastava, G. Hinton, A. Krizhevsky, I. Sutskever y R. Salakhutdinov. (2014) *Dropout: A Simple Way to Prevent Neural Networks from Overfitting.* <http://jmlr.org/papers/volume15/srivastava14a/srivastava14a.pdf>.
- [Wolpert et al., 1997] D. H. Wolpert y W. G. Macready. (1997) *No Free Lunch Theorems for Optimization.* <https://ti.arc.nasa.gov/m/profile/dhw/papers/78.pdf>.
- [He et al., 2014] K. He, X. Zhang, S. Ren, J. Sun. (2014) *Spatial Pyramid Pooling in Deep Convolutional Networks for Visual Recognition.* <https://arxiv.org/pdf/1406.4729.pdf>.
- [Dalal et al., 2005] N. Dalal y B. Triggs. (2005) *Histogram of oriented gradients for human detection.* <https://ieeexplore.ieee.org/document/1467360>.
- [Girshick et al., 2013] R. Girshick, J. Donahue, T. Darrell y J. Malik. (2013) *Rich feature hierarchies for accurate object detection and semantic segmentation.* <https://arxiv.org/pdf/1311.2524.pdf>.
- [Ren et al., 2015] S. Ren, K. He, R. Girshick y J. Sun. (2015) *Faster R-CNN: Towards Real-Time Object Detection with Region Proposal Networks.* <https://arxiv.org/pdf/1506.01497.pdf>.
- [Redmon et al., 2015] J. Redmon, S. Divvala, R. Girshick Y A. Farhadi. (2015) *You Only Look Once: Unified, Real-Time Object Detection.* <http://arxiv.org/abs/1506.02640>.

- [Long et al., 2015] J. Long, E. Shelhamer y T. Darrell. (2015) *Fully Convolutional Networks for Semantic Segmentation*. https://people.eecs.berkeley.edu/~jonlong/long_shelhamer_fcn.pdf.
- [Ronneberger et al., 2015] O. Ronneberger, P. Fischer y T. Brox. (2015) *U-Net: Convolutional Networks for Biomedical Image Segmentation*. https://people.eecs.berkeley.edu/~jonlong/long_shelhamer_fcn.pdf.
- [DRAE, Aprendizaje] Real Academia Española. Aprendizaje. En Diccionario de la lengua española (23.a ed.). Recuperado de <http://dle.rae.es/?id=3IacRHm>.
- [DIGITS] NVIDIA DIGITS. Interactive Deep Learning GPU Training System. Recuperado de <https://developer.nvidia.com/digits>.
- [Caffe] Caffe. Deep learning framework by the BVLC. Recuperado de <http://caffe.berkeleyvision.org>.
- [Torch] Torch. A scientific computing framework for LuaJIT. Recuperado de <http://torch.ch>.
- [TensorRT] NVIDIA TensorRT. High performance deep learning inference for production deployment. Recuperado de <https://developer.nvidia.com/tensorrt>.
- [NVIDIA Jetson TX2] NVIDIA Jetson TX2 Module. Recuperado de <https://developer.nvidia.com/embedded/buy/jetson-tx2>.
- [NVIDIA DGX-1] NVIDIA DGX-1. AI Supercomputer. Recuperado de <http://www.nvidia.com/object/deep-learning-system.html>.

APÉNDICE A

Configuración del sistema

En el siguiente apéndice describimos el sistema que hemos preparado para poder realizar el trabajo. En las siguientes secciones detallamos los componentes *hardware* y *software* utilizados.

A.1 Especificaciones hardware

Como se indicó con anterioridad, el punto clave de un sistema que se utiliza para trabajar con *deep learning* es su tarjeta gráfica, es por ello que se debe priorizar en ella a la hora de elegir los componentes. A continuación detallamos los componentes *hardware* de nuestro sistema:

- **[CPU] Intel i7-5820K 3.3 Ghz:** ya que tenemos una tarjeta gráfica potente, debemos elegir un procesador que trabaje bien con ella y no provoque cuellos de botella. Al disponer de dos tarjetas encontramos este procesador una buena opción ya que además tiene hasta 12 hilos de ejecución.
- **[Memoria RAM] Kingston HX421C13SBK2/16 16GB 2133MHZ DDR4:** el procesador trabaja con DDR4 hasta 2133 Mhz, por lo que encontramos esta memoria una buena opción. Ante la posibilidad de utilizar conjuntos de datos grandes y tener que trabajar con ellos instalamos 64 Gb de memoria RAM en el equipo para tener holgura.
- **[Placa Base] Gigabyte PB/EATX/2011V3/X99/X99GAM:** nos interesa que sea compatible con el procesador, la memoria RAM y que tenga capacidad para al menos dos tarjeta gráfica, este modelo cumplía estos requisitos y fue elegido.
- **[Tarjeta gráfica] Gigabyte GV-N1080G1 GAMING-8GD:** disponemos de dos de estas tarjetas, son de la última generación de Nvidia y su uso para *deep learning* se ha popularizado, por lo que las encontramos como una buena elección.
- **[Disco duro] Samsung MZ-75E1T0B/EU SSD EVO 850:** normalmente trabajaremos con grandes conjuntos de datos y las pruebas que se realicen generarán mucha información, por ello, decidimos dotar al equipo con dos de estos discos, siendo la elección de SSD para agilizar el sistema y que no se produzcan cuellos de botellas.

A.2 Especificaciones software

Una vez decidido que usaremos Nvidia Digits nos centramos en instalar todo lo necesario para su uso.

En cuanto al sistema operativo, Nvidia Digits puede instalarse en Windows o Linux. Nosotros nos decantamos por Linux al ofrecer mejor documentación y soporte de la comunidad tanto para Nvidia Digits como para sus dependencias.

Dentro de las posibles distribuciones de Linux encontramos a Ubuntu como la mejor opción, al facilitar la instalación de programas y disponer de las últimas versiones de la mayoría de librerías que necesitaremos.

Para poder trabajar con Nvidia Digits necesitaremos instalar una serie de programas que este utiliza. Principalmente tienen que ver con librerías para el uso y optimización de la tarjeta gráfica y la instalación de Caffe y Torch, que son los *frameworks* disponibles en Nvidia Digits y que por tanto nos exige instalar. A continuación detallamos los programas y librerías instalados:

- **CUDA 8.0:** instalamos el *driver* para la tarjeta gráfica y el *toolkit*, todo ello viene integrado en la instalación de CUDA.
- **NVIDIA cuDNN 5.1:** se trata de una librería que contiene una implementación altamente optimizada de la mayoría de rutinas más comunes que usan las redes neuronales para las tarjetas gráficas de Nvidia.
- **NCCL:** es una colección de primitivas que optimizan la comunicación entre múltiples tarjetas gráficas como es nuestro caso.
- **Caffe:** instalamos NVcaffe, que se trata de una modificación de la librería original de Caffe hecha por Nvidia para dar soporte a determinadas características no encontradas en la versión oficial. La instalación de Caffe requerirá a su vez de la instalación de otras librerías y programas. Todos estos requisitos se detallan en la página de descarga del mismo, además de como instalarlo.
- **Torch7:** al instalar Torch seguimos las instrucciones proporcionadas por Nvidia Digits, ya que requiere de alguna que otra configuración posterior para solucionar algunos requisitos exigidos por este.
- **DIGITS 5.1.** Instalamos Nvidia Digits descargándolo desde el repositorio oficial en GitHub. Lo instalamos siguiendo las instrucciones de dicho repositorio.

Para la instalación de la mayoría de estas librerías requiere de algunas configuraciones extras, como añadir rutas donde se instalan algunas librerías y la creación y modificación de algunos archivos. Todo ello se puede encontrar detallado en la mayoría de sitios oficiales de estos programas, donde se explica cómo instalarlos correctamente.

Una vez tenemos todo instalado podemos usar Nvidia Digits, lanzándolo desde una consola con la orden `digits-devserver`.

Esta orden lanzará el servidor de Nvidia Digits, para acceder al programa solo necesitamos un navegador y acceder a la dirección `http://localhost:5000/`.