Universidad de Valladolid

ESCUELA DE INGENIERÍA INFORMÁTICA

TRABAJO FIN DE GRADO

GRADO EN INGENIERÍA INFORMÁTICA
MENCIÓN EN INGENIERÍA DE SOFTWARE

Caracterización de piezas mediante técnicas de Deep Learning

Autor:
Istaliyan Ivanov Manov

Tutor:
Dr.Benjamín Sahelices Fernández

A mi familia.

Resumen

El control de calidad es una parte del proceso de producción de suma relevancia. Identificar correctamente las piezas que sufran imperfecciones, antes de que sean usadas en el producto final, es una de las tareas clave en este proceso. Para poder automatizarla es posible usar técnicas de Deep Learning que han visto gran aplicación durante la última década. Este Trabajo de Fin de Grado está orientado a aprender los conceptos sobre los que se apoyan estas técnicas y aplicarlos, usando la librería *fastai*, para realizar clasificadores de imágenes mediante los que automatizar la detección de piezas defectuosas.

Conceptos clave:
Deep Learning, redes neuronales, descenso de gradiente, *backpropagation*, *overfitting*, convolución, *fastai*, *Transfer Learning*.

Abstract

Quality control is an extremely important part of the production process. Identifying correctly the components that suffer imperfections, before they are used in the final product, is one key part of this process. In order to automate it, is possible to use Deep Learning techniques that have seen great use during the last decade. This project is aimed at learning que concepts on which this techniques are based on and appliying them, using the *fastai* library to create classifiers with which to automate this detection.

Key concepts:
Deep Learning, neural networks, gradient descent, backpropagation, overfitting, convolution, *fastai*, transfer learning.

Índice general

Índice de figuras

Índice de cuadros

Capítulo 1

Introducción

En este capítulo, en primer lugar, se va a describir el origen de las cuestiones principales que han motivado la realización de este trabajo. A continuación se indicarán los objetivos que se plantean al iniciar el proyecto así como los requisitos técnicos necesarios para llevarlo a cabo. Por último presentará la estructura que va a seguir esta memoria.

1.1. Motivación

En los últimos años hemos sido testigos de grandes avances en el campo de la Inteligencia Artificial (IA).En concreto el auge del Aprendizaje Profundo (en inglés, *Deep Learning*) ha derivado del gran potencial que tiene para obtener una solución satisfactoria a cierto tipo de problemas. Algunos de estos problemas pueden ser:

- Clasificación de imágenes.
- Detección de objetos.
- Segmentación.
- Traducción automática.
- Detección de anomalías.

La clasificación de imágenes es uno de los problemas donde los avances en Deep Learning has supuesto una mejora significativa respecto a las soluciones obtenidas por métodos mas tradicionales de Visión Artificial. Pero estas mejoras en los resultados han venido acompañadas de una mayor complejidad a la hora de desarrollar aplicaciones basadas en Deep Learning que han vinculado esta disciplina casi por completo al mundo académico, sin poder integrarlas de manera rápida y efectiva en los procesos de desarrollo habituales en las empresas.

Esto está empezando a cambiar gracias a la introducción de librerías para Deep Learning como fast.ai [1] que ofrecen una forma sencilla de obtener resultados convincentes a problemas como los mencionados anteriormente.

La posibilidad de desarrollar aplicaciones usando fast.ai de forma rápida y sin la necesidad de muchos años de estudio del campo puede suponer un punto clave en la integración de la Inteligencia Artificial en la industria, de forma que se pueda obtener los beneficios derivados de los avances en Deep Learning sin una gran barrera de entrada.

El control de calidad en los procesos industriales se puede ver altamente beneficiado por este tipo de aplicaciones debido al estrecho vínculo que tiene con la clasificación de imágenes. Automatizar la identificación de piezas defectuosas puede mejorar la calidad y la velocidad a la que se producen los productos y por tanto este será el problema con el que trataremos en este Trabajo de Fin de Grado.

1.2. Objetivos

El objetivo principal de este trabajo es desarrollar las competencias necesarias para automatizar el proceso de identificación de piezas industriales defectuosas usando técnicas de Deep Learning y la librería fast.ai.

Para conseguir este objetivo plantearemos varios objetivos intermedios.

1. **Aprendizaje teórico de los conceptos básicos usados en Aprendizaje Automático:** Antes de poder adentrarnos en el campo de Deep Learning primero debemos familiarizarnos con algunas de las herramientas matemáticas básicas que son comúnmente utilizadas en aprendizaje automático (en inglés, Machine Learning) ya que Deep Learning es un subcampo de este. Centraremos el estudio en los conceptos que estén mas estrechamente relacionados con la clasificación de imágenes.

2. **Aprendizaje teórico de conceptos usados en Deep Learning:** Una vez explorados los conceptos principales de Machine Learning pasaremos a entender como usarlos de manera efectiva en aplicaciones de Deep Learning así como los conceptos propios de este campo como las redes neuronales.

3. **Aprendizaje teórico y práctico de la librería fast.ai:** Para poder poner en práctica los conceptos teóricos aprendidos hasta este punto deberemos familiarizarnos con la librería fast.ai, entender cual es el proceso típico definido por los creadores para obtener buenos resultados y aplicarlo a ciertos ejemplos para comprobar lo aprendido.

4. **Puesta en práctica de los conocimientos adquiridos mediante un caso real:** Por último usaremos imágenes obtenidas de un proceso de fabricación de piezas real para clasificarlas en función de si son incorrectas o no. Se comprobará como se comportan distintas arquitecturas y se compararán los resultados para de esta forma obtener los modelos mas efectivos posibles.

1.3. Requisitos técnicos

En esta sección vamos a describir los materiales tanto elementos hardware como software, usados durante el desarrollo de este proyecto.

1.3.1. Hardware

El trabajo completo se ha llevado a cabo en un PC portátil MSI GF63 Thin 9SC con las siguientes características:

- **Procesador:** Intel Core i7-9750H
  Nucleos: 6
  Frecuencia: 2,60 GHz
  Cache: 12 MB Intel Smart Cache
- **Memoria:** 16GB DDR4 2666MHz
- **Almacenamiento:** 512GB NVMe PCIe SSD
- **Tarjeta gráfica:** GTX 1650 MAX Q GDDR5
  NVIDIA CUDA Cores: 1024
  Frecuencia básica: 930 - 1395 MHz
  Frecuencia modificada: 1125 - 1560 MHz
  Velocidad de memoria: Hasta 8Gbps
  Ancho de banda de memoria: 128 GB/s

Esta última parte referente a la tarjeta gráfica usada se debe tener en cuenta en caso de querer replicar los resultados obtenidos a lo largo de este proyecto debido a que las técnicas de Deep Learning usan una gran cantidad de datos. Para acelerar los cálculos necesarios se ha usado la GPU para la ejecución de todos los programas creados salvo que se indique lo contrario. Intentar ejecutar los programas con una tarjeta gráfica con características inferiores puede producir resultados dispares en cuanto a tiempo de ejecución o directamente la imposibilidad de ejecutar el código.

1.3.2. Software

- **Sistema operativo:** Ubuntu 20.04.2 LTS

  Frente a la elección entre desarrollar este proyecto en Windows o alguna distribución de Linux se ha tomado la decisión de usar Linux por varios motivos. En primer lugar es el sistema que se ha utilizado mayoritariamente durante el curso de la carrera y por tanto el factor comodidad se debe tener en cuenta. Debemos tener en cuenta la linea

de comandos y el sistema de gestión de paquetes que hacen trabajar con múltiples versiones de distintos paquetes software una mejor experiencia. También se debe tener en cuenta que al usar tarjetas gráficas de NVIDIA es posible que se creen conflictos con sistemas operativos que usen el kernel Linux debido a los drivers necesarios. Respecto a este punto la distribución usada ha minimizado estos errores que son mas comunes en otras distribuciones.

- **Lenguaje de programación:** Python 3.8.5

  En este caso la elección no ha sido tanto personal como influenciada por las elecciones de los profesionales especializados en Aprendizaje Automático. Esto es debido a varios factores. Python es un lenguaje de programación multiparadigma que fue creado en 1991 [2]. Una de las grandes ventajas de utilizar este lenguaje es la legibilidad y la sencillez de la sintaxis. A la hora de resolver problemas de Aprendizaje Automático usar un lenguaje fácil de entender supone disminuir la complejidad total del trabajo.

  Otra de las grandes ventajas que tiene Python frente a otros lenguajes es el gran número de librerías para Machine Learning que dispone. Algunos ejemplos de estas librerías son Tensorflow, Pytorch, scikit-learn y la que usaremos en este proyecto: fastai . Además de estas librerías, cuyo uso específico es el Machine Learning, Python dispone de una gran cantidad de librerías estrechamente relacionadas con este campo pero que pueden ser usadas para propósitos diferentes. Algunos ejemplos son Matplotlib, una librería dedicada a la generación de gráficas a partir de listas o arrays, u OpenCV, dedicada a la visión artificial en tiempo real.

- **Entorno de Deep Learning:** Anaconda

  Para simplificar el trabajo a la hora de lidiar con versiones específicas de múltiples librerías se ha decidido usar Anaconda que es una distribución libre de Python ampliamente usada en problemas como el Aprendizaje Automático u otros tipos de cómputo científico. La ventaja que ofrece esta distribución es que incluye Conda, que es un sistema de gestión de paquetes y entornos, además de numerosos paquetes usados habitualmente en Aprendizaje Automático.

  A continuación se listan las librerías y aplicaciones necesarias para el desarrollo del proyecto. Todas ellas vienen instaladas en la distribución base de Anaconda a excepción de fastai, Tensorboard y Torch. Puede que las versiones que vengan por defecto en la distribución base no se correspondan con las listadas a continuación y por tanto habría que modificarlas de forma manual.

  Conda 4.9.2

  Conda nos brinda la posibilidad de crear entornos virtuales fácilmente donde dispongamos de los paquetes que necesitemos con las versiones adecuadas para no crear conflictos ya que hay ocasiones en las que algunos paquetes requieren de una versión específica de otro. El manejo de estas dependencias hace de Conda una herramienta esencial.

  fastai 2.2.5

  La librería fastai, como ya se ha mencionado, tendrá un papel central en el desarrollo de este proyecto. Hay otras muchas librerías de Deep Learning que se podrían usar. Algunas de ellas son Tensorflow, Pytorch, Keras o Theano. La principal ventaja de fastai es que proporciona componentes de alto nivel, con los cuales es sencillo obtener

resultado satisfactorios a problemas típicos de Deep Learning, así como componentes de bajo nivel para usuarios experimentados que requieren modificaciones mas sutiles en los modelos. Para esto fastai usa PyTorch, que es una librería de bajo nivel, como base sobre la que añadir nuevas funcionalidades.

Jupyter Notebook 6.2.0

Jupyter Notebook [3] es una aplicación web comunicada con un kernel de cálculo. Esto permite la ejecución de múltiples lenguajes de programación, entre ellos Python, Julia, C/C++, R y otros. La estructura en la que se presenta esta aplicación es una secuencia de celdas. Estas sirven como entrada de texto en la que se incluye el código a ejecutar. Una vez ejecutado el código se muestra su salida a continuación o se pasa a la próxima celda en caso de que no haya nada que mostrar por pantalla. Las celdas pueden ser editadas y ejecutadas de nuevo sin tener que ejecutar el código completo del Notebook, solo las celdas que lo requieran. Esto permite experimentar de una manera rápida con distintas modificaciones.

Tensorboard 2.4.0

Tensorboard es un conjunto de aplicaciones web para visualización de gráficas y modelos. Originalmente fue creado para Tensorflow y posteriormente integrado en Pytorch. Esta herramienta puede ser sustituida por otras similares ya que ofrecen una funcionalidad parecida y durante este proyecto no se usarán todas las opciones de Tensorboard. Nos serviremos de ella como herramienta de apoyo a la hora de visualizar ciertos datos.

scikit-image 0.17.2

Esta librería [4] contiene algoritmos de procesamiento de imágenes. Al igual que el caso anterior puede ser sustituida por otras, como OpenCV, que ofrecen herramientas similares. Algunos de estos algoritmos pueden ser usados para aplicar modificaciones a las imágenes, algo que es muy útil a la hora de aumentar sintéticamente los datos empleados para los modelos.

1.4. Estructura del documento

A continuación se describirá el contenido de los capítulos y las secciones presentes en este Trabajo de Fin de Grado excluyendo el Capitulo 1 ya que es el actual.

Capítulo 2: En este capítulo se detallará las principales características de la metodología usada para determinar el plan que se seguirá durante este proyecto. Siguiendo esta metodología se elaborará un plan que tiene como objetivo cumplir los objetivos propuestos en un tiempo limitado. Se indicarán las tareas que se deberán realizar, sus duraciones y cuales son los principales problemas que pueden surgir y como afrontarlos.

Capítulo 3: En este capítulo se hará una descripción de las tecnologías que se usarán para conseguir los objetivos. Se darán ejemplos de su uso así como una visión global del tipo de problemas que pueden ser resueltos y cómo encajan en este proyecto.

Capítulo 4 : Este capítulo se centrará en primer lugar sobre los conceptos teóricos que conforman la base del Aprendizaje Automático. Se realizará la distinción entre las diferentes

categorías en las que se divide así como los principales algoritmos usados en este campo. Sobre esta base se describirá el principal modelo de computación usado en Aprendizaje Profundo, las redes neuronales convolucionales y sus principales componentes.

Capitulo 5 : En este capítulo se realizará una descripción de los datos con los que se va a trabajar. A continuación se mostrarán los resultados de los distintos experimentos realizados usando distintos enfoques así como una comparación entre ellos para determinar cual ha sido la mejor estrategia. Se

Capítulo 6 : Por último, en este capítulo, se presentarán las conclusiones que se han obtenido después de llevar a cabo el proceso. En esta sección se incluirán las principales dificultades que se han presentado así como cuales han sido las partes mas relevantes del trabajo. También se revisarán los objetivos presentados en el Capítulo 1 y se comprobará si han llegado a cumplirse o no.

Capítulo 2

Planificación

2.1. Introducción

En este capítulo se va a describir el enfoque usado para planificar este proyecto. En primer lugar se va a describir en que consiste este enfoque, identificando y explicando cada uno de los pasos necesarios y después se va a indicar como se ha llevado a la práctica cada uno de los pasos mencionados.

2.2. Step wise

El enfoque usado en la planificación de este proyecto tiene el nombre de *Step wise*. El nombre mismo ya nos indica la estructura que se va a seguir ya que la traducción es "paso a paso". Este no es el único enfoque que podemos usar ya que existen otras metodologías, como PRINCE2 [5] o SSADM [6], para gestionar un proyecto.

Antes de indicar cuales serán los pasos a seguir veamos en que se diferencia de otros métodos y cual es su origen.El enfoque *Step wise* [7] es creado por Robert T.Hughes en 1996. Debido al incremento en la producción de software surgió la necesidad de usar marcos de trabajo efectivos ya que el desarrollo era cada vez más complejo y los proyectos cada vez mas grandes. *Step wise* se centra en la gestión de este tipo de proyectos y toma algunos conceptos de PRINCE2 como la realización de un plan más general que después es refinado mediante un proceso iterativo o el uso de "productos" que son los resultados de las actividades realizadas.

Step wise por su parte es una metodología escalable, resulta adecuada tanto para proyectos pequeños como grandes, y centrada en la planificación a diferencia de PRINCE2 donde se incluye la monitorización y el control en pasos posteriores.

Para realizar la planificación del proyecto *Step wise* propone los siguientes pasos:

- **Paso 0**: Escoger el proyecto
- **Paso 1**: Identificar límites del proyecto
- **Paso 2**: Identificar infraestructura del proyecto
- **Paso 3**: Analizar las características del proyecto
- **Paso 4**: Identificar los productos y las actividades
- **Paso 5**: Estimar el esfuerzo de cada actividad
- **Paso 6**: Identificar los riesgos
- **Paso 7**: Asignar recursos
- **Paso 8**: Revisión y publicación del plan
- **Paso 9**: Ejecución del plan
- **Paso 10**: Modificaciones de bajo nivel del plan

Al ser una metodología con gran flexibilidad hay pasos de la elaboración del plan que no serán aplicables a este proyecto o que deberán ser reformulados para poder ser usados de forma efectiva.

2.2.1. Escoger el proyecto

A este paso se le atribuye el número 0 ya que muchas veces el proyecto es asignado de forma externa a la planificación y por tanto no hay realmente una elección. Esto es común en muchas organizaciones y es habitual que las personas encargadas de seleccionar los proyectos que mejor se ajusten a la organización no sean las mismas que las encargadas de realizar la planificación. En este caso en cambio sí que hubo una elección y el proyecto es "Caracterización de piezas industriales mediante técnicas de Deep Learning".

2.2.2. Limites del proyecto

Una vez indicados los objetivos del proyecto en el primer capítulo debemos poder evaluar en que medida se han cumplido. Para ello se usarán los siguientes indicadores:

- **Cumplimiento de las fechas planificadas**: Se puede evaluar fácilmente e indica si se ha seguido el plan elaborado inicialmente.
- **Calidad del proyecto**: Más difícil de evaluar que el indicador anterior y por tanto se requerirá de evaluación externa.
- **Coste del proyecto**: Un coste muy superior al previsto indicaría una mala previsión en el momento de creación del plan.

También se debe saber quién está involucrado en el proyecto. Se debe designar a la autoridad que en este caso es el tutor académico del proyecto, Dr.Benjamín Sahelices Fernández y por último indicar quienes son los *stakeholders*.

Los stakeholders son las personas que tienen algún interés en que el proyecto sea exitoso. En este caso el alumno realizando el proyecto, Istaliyan Ivanov Manov.

Al designar estos dos roles se deben indicar los canales de comunicación que se van a utilizar entre ellos. En este caso son el correo electrónico para cuestiones mas generales que no precisen de mucho detalle y reuniones virtuales realizadas mediante la plataforma *Jitsi*.

2.2.3. Infraestructura del proyecto

Los proyectos de desarrollo de software habitualmente forman parte de una estrategia más general dentro de una organización que queda reflejada en el portfolio. Algunos proyectos pueden estar relacionados con otros y por tanto se deben tener estas dependencias en consideración. Este proyecto es completamente independiente en este sentido aunque sí que se debe tener en cuenta que se realiza dentro de la Universidad de Valladolid y por tanto debe seguir los estándares especificados por esta.

Estos estándares se encuentran recogidos en el BOCYL [8] y tal y como especifica el Artículo 5 se debe respetar el formato y estructura designados por la Facultad de Ingeniería Informática [9].

2.2.4. Características del proyecto

Una de las características más importantes de un proyecto es si este está orientado a un objetivo, como es el caso de este trabajo, o a un producto. Como se indica en el capítulo de introducción el objetivo principal es obtener las competencias necesarias en el campo de Deep Learning para desarrollar aplicaciones que permitan resolver problemas asociados tradicionalmente a este campo.

Para completar este objetivo se deberá escoger, basándonos en la información obtenida, la metodología que seguirá el proyecto. Dependiendo de la organización en la que se desarrolle o los requisitos de los usuarios finales puede ser mas adecuada una metodología u otra. Si los clientes desean probar el producto a medida que se está elaborando se pueden usar entregas incrementales. En caso de que haya incertidumbre sobre la parte técnica del proyecto se pueden usar prototipos mediante los que se puede adquirir conocimiento y evitar errores futuros. Si el proyecto requiere de una metodología mas ligeras se puede aplicar SCRUM o *Extreme Programming*.

Para este proyecto se ha seleccionado una metodología mas tradicional, el desarrollo en espiral, que deriva del desarrollo clásico en cascada. El motivo por el que se ha escogido es debido al desconocimiento en el campo del Deep Learning. Esta forma de afrontar el proyecto permite refinar cada uno de los pasos que se han dado de forma iterativa, de modo que se pueda realizar un aprendizaje teórico, implementar los conceptos aprendidos y volver a la teoría ya que probablemente haya partes de la implementación incorrectas o que puedan mejorarse.

2.2.5. Productos y actividades

Los productos son los resultados tangibles de las actividades. Como regla general en *Step wise* debemos asegurarnos de que una actividad siempre produce un producto y que todos los productos provengan de una actividad. En el siguiente diagrama se distinguen las distintas categorías y productos individuales que van a ser creados.

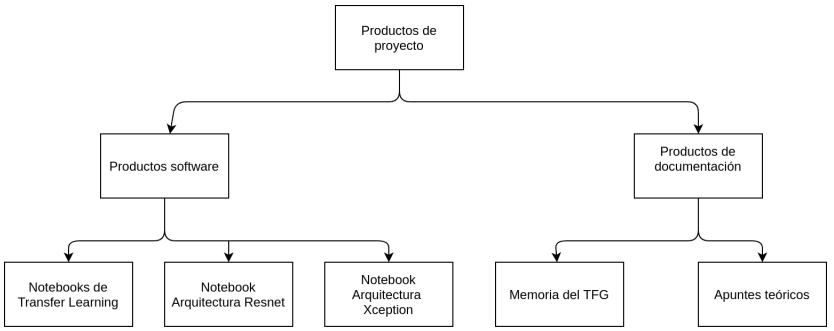


Figura 2.1: Productos que van a ser creados a lo largo del proyecto.

Las hojas de este árbol constituyen productos individuales a excepción de "Notebooks de Transfer Learning" y "Apuntes Teóricos" ya que en caso de introducir todos los productos individuales en estas categorías se obtendría un árbol demasiado grande como para ser introducido de forma adecuada en esta memoria.

La categoría "Apuntes Teóricos" aunque no vayan a ser introducidos en la entrega final de este proyecto suponen un paso esencial para el correcto aprendizaje de la parte teórica y son el resultado de su actividad correspondiente.

Una vez determinados los productos que se van a crear se deben establecer las actividades que va a ser necesario realizar para obtener estos productos.

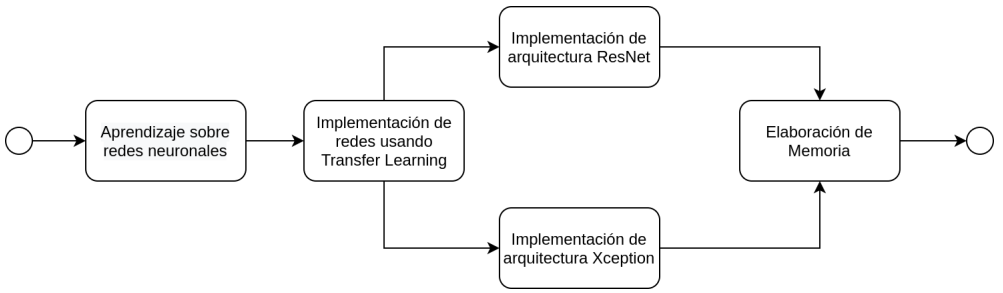


Figura 2.2: Actividades que se van a realizar para obtener los productos.

La actividad "Implementación de redes mediante Transfer Learning" es la responsable de crear los Notebooks de Tansfer Learning y por tanto de nuevo puede ser desglosada en múltiples actividades individuales encargadas de un solo Netebook pero esto haría el diagrama poco legible.

2.2.6. Estimar el esfuerzo de cada actividad

Cada una de las actividades previamente descrita supondrá un esfuerzo que debe ser cuantificado. Existen múltiples métricas que pueden ser usadas como las líneas de código (en inglés, *Source Lines Of Code* (SLOC)) o el número de ficheros a crear por cada actividad . De la misma forma se puede usar un enfoque "*top-down*" donde se estima el esfuerzo total y se divide entre las actividades o un enfoque "*bottom-up*" donde se hace una estimación de cada actividad y se suman para obtener una estimación total del proyecto.

En este caso se usará el enfoque "*top-down*" y se estimará el esfuerzo en horas necesarias para realizar la actividad.

Se realizará en primer lugar una estimación del esfuerzo más optimista teniendo en cuenta las horas indicadas para la realización de proyecto en la Guía Docente, donde se estiman 300 horas. A continuación se hará una estimación que permita un mayor margen de error a la hora de la planificación.

Actividad	Porcentaje	Horas (opt.)	Horas(real.)
Aprendizaje sobre redes neuronales	25 %	75	100
Implementación mediante Transfer Learning	15 %	45	60
Implementación de arquitectura Resnet	15 %	45	70
Implementación de arquitectura Xception	20 %	60	80
Elaboración de la memoria	25 %	75	110

Cuadro 2.1: Tabla de estimación *opt.* optimista y *real.* realista del esfuerzo para cada actividad

2.2.7. Identificar los riesgos

Un riesgo se define según la guia PM-BOK [10] (Project Management Body of Knowledge) como "un evento incierto que en caso de ocurrir puede tener consecuencias positivas o negativas". En primer lugar se van a identificar los riesgos detectados, para cada uno de estos riegos se va a indicar el impacto que puede tener en la realización del proyecto en una escala del 1 al 10 y la probabilidad de que ocurra este riesgo, siendo 10 certeza absoluta de ocurrir y 0 de no ocurrir.

Por último se determina el *Risk Exposure* que viene determinado por la siguiente fórmula: *Risk exposure* = Impacto $\times$ Probabilidad.

Habitualmente la probabilidad de que un evento ocurra se encuentra en el intervalo $[0, 1]$ pero en este caso el cálculo es solo orientativo.

Los riegos enumerados en la Tabla 2.2 pueden ser representados en un de forma más visual mediante un gráfico. En esta figura debemos prestar especial atención a los valores con altas probabilidades y altos impactos. Estos se van a encontrar en la parte superior derecha del gráfico y son los riesgos mas preocupantes para el correcto desarrollo del proyecto y para los que tenemos que crear planes con especial cuidado.

ID	Descripción	Impacto	Probabilidad	RE
R01	Contagio de Covid	4	1	4
R02	Equipo de trabajo se estropea	6	4	24
R03	La planificación ha sido optimista en vez de realista	5	3	15
R04	Los objetivos planteados están fuera de las posibilidades	7	2	14
R05	Desconocimiento de Deep Learning	7	8	56
R06	Desconocimiento de las librerías	5	5	25
R07	Bugs que alargan el tiempo de desarrollo	4	6	24
R08	Bugs de las librerías	8	0.5	4
R09	No se detecta que el proyecto está atrasado hasta una fase tardía	7	2	14
R10	Implementar conceptos exclusivamente vistos en artículos académicos/ experimentales	8	6	36

Cuadro 2.2: Tabla de riesgos y sus características

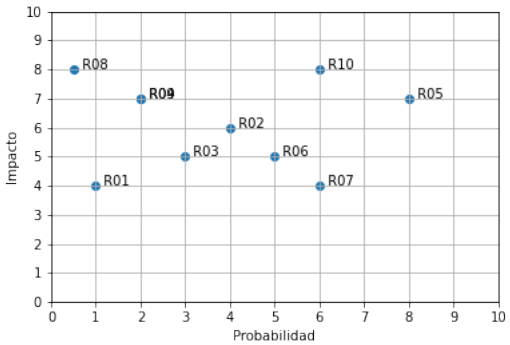


Figura 2.3: Gráfica de probabilidad frente a impacto de cada riesgo.

Una vez definidos los riesgos a los que nos enfrentamos hay que concretar un plan para cada uno. Estas respuestas a cada riesgo entran dentro de cuatro categorías más generales.

1. **Aceptación del riesgo** : No se hace nada respecto al riego, se aceptan las consecuencias.

2. **Evitación del riesgo**: Se evita en su totalidad realizar la actividad que produce este riesgo.

3. **Mitigación del riego**: Se realiza la actividad que produce el riesgo pero se introducen medidas para disminuir la probabilidad de que ocurra.

4. **Transferencia del riesgo**: El riego es transferido a otra persona u organización.

ID	Plan de actuación
R01	**Aceptar el riesgo.** Aparte de cumplir las medidas de seguridad propuestas por el gobierno no hay muchas más opciones para evitar el contagio
R02	**Mitigar el riego.** Asegurarnos de disponer de un equipo secundario que podamos utilizar y usar herramientas de control de versiones como GitHub para evitar perder los avances.
R03	**Mitigar el riesgo.** No subestimar el tiempo necesario para realizar cada actividad.Para ello añadir colchones de seguridad en cada una.
R04	**Mitigar el riesgo.** Los objetivos serán revisados por el tutor académico para comprobar que no son poco realistas.
R05	**Mitigar el riesgo.** Asegurarnos de comprender correctamente cada concepto antes de continuar.
R06	**Mitigar el riesgo.** Realizar pruebas con los ejemplos proporcionados por las librerías para comprobar la capacidad de usarlas.
R07	**Mitigar el riesgo.** Asegurarnos de comprender bien los conceptos teóricos y haber consultado la documentación de las librerías antes de realizar la implementación.
R08	**Aceptar el riesgo.** Un bug en la librerías que se van a usar es poco probable ya que cada cambio es cuidadosamente introducido y son manejadas por expertos en el campo.
R09	**Mitigar el riego.** Asegurarnos de cumplir con las fechas estimadas para cada actividad.
R10	**Evitar el riesgo.** Pese a que se puedan obtener resultados mejores implementando conceptos aún exclusivos del mundo académico es muy probable fracasar y por tanto se evitará por completo.

Cuadro 2.3: Planes para cada riesgo presentado en la Tabla 2.2

En la Tabla 2.3 se puede observar que no se ha incluido ningún riesgo en la categoría de transferencia. Esto es debido a que el proyecto debe estar completado en su totalidad por el autor y por tanto no se puede realizar una sub-contratación o transferir el riego a otra persona.

2.2.8. Asignación de recursos

Al haber exclusivamente una persona trabajando en este proyecto es fácil asignar las tareas a la persona encargada ya que para todas es el autor. Pero también se debe ver cómo se asignan los recursos temporales y como encajan en la planificación. Para ello se hará uso del diagrama de Gantt presente en la Figura 2.4.

2.2.9. Revisión, publicación, modificación y ejecución del plan

Una vez descrito el plan de acción y habiendo planificado cada aspecto de este es el momento de revisar cada uno de los pasos y publicar el plan. Habitualmente el proceso de revisión del plan debería ser realizado en conjunto con miembros de la organización que no han formado parte de la creación del plan pero tienen claros los objetivos. Esto es debido a que mientras se está realizando una actividad es habitual tener ciertos fallos que no se han detectado. Una nueva perspectiva sobre el resultado puede detectar estos errores y evitar problemas futuros.

Este proceso no es viable en esta situación debido a que la única persona encargada del proyecto es el autor. Por ello la revisión, publicación, modificación y ejecución del plan son tareas que no pueden ejecutarse de la forma ideal sino que deben ser adaptadas a la situación existente.

Esto no es una situación extraña debido a que *Step-wise* es una metodología que funciona igualmente para proyectos de una sola persona. La única diferencia es que estos últimos pasos se realizará por la misma persona.

Por último se debe mencionar que existe la posibilidad de hacer ciertas modificaciones en el plan a medida que se está ejecutando. En caso de que haya cambios de este estilo en el proyecto estos serán mencionados en las conclusiones ya que no pueden indicarse antes de la ejecución del plan.

2.2.10. Cálculo de presupuesto

Para realizar el cálculo del presupuesto existen múltiples enfoques. Algunos de estos enfoques están más orientados al trabajo cercano con el cliente donde se realizan pago cada vez que hay un bloque funcional de software. Otros en cambio buscan obtener un contrato inalterable donde el precio ya está definido al inicio del proyecto.

En este caso se usará un enfoque que estimará el pago dividiéndolo en dos categorías, los materiales usados y las unidades de esfuerzo necesarias para completar el proyecto.

Las estimación del precio de los materiales es sencilla en este caso ya que la totalidad del software usado es de código abierto y por tanto gratuito. Respecto al hardware usado, detallado en el Capítulo 1, este ya era propiedad del desarrollador y por tanto supone un coste de 0€.

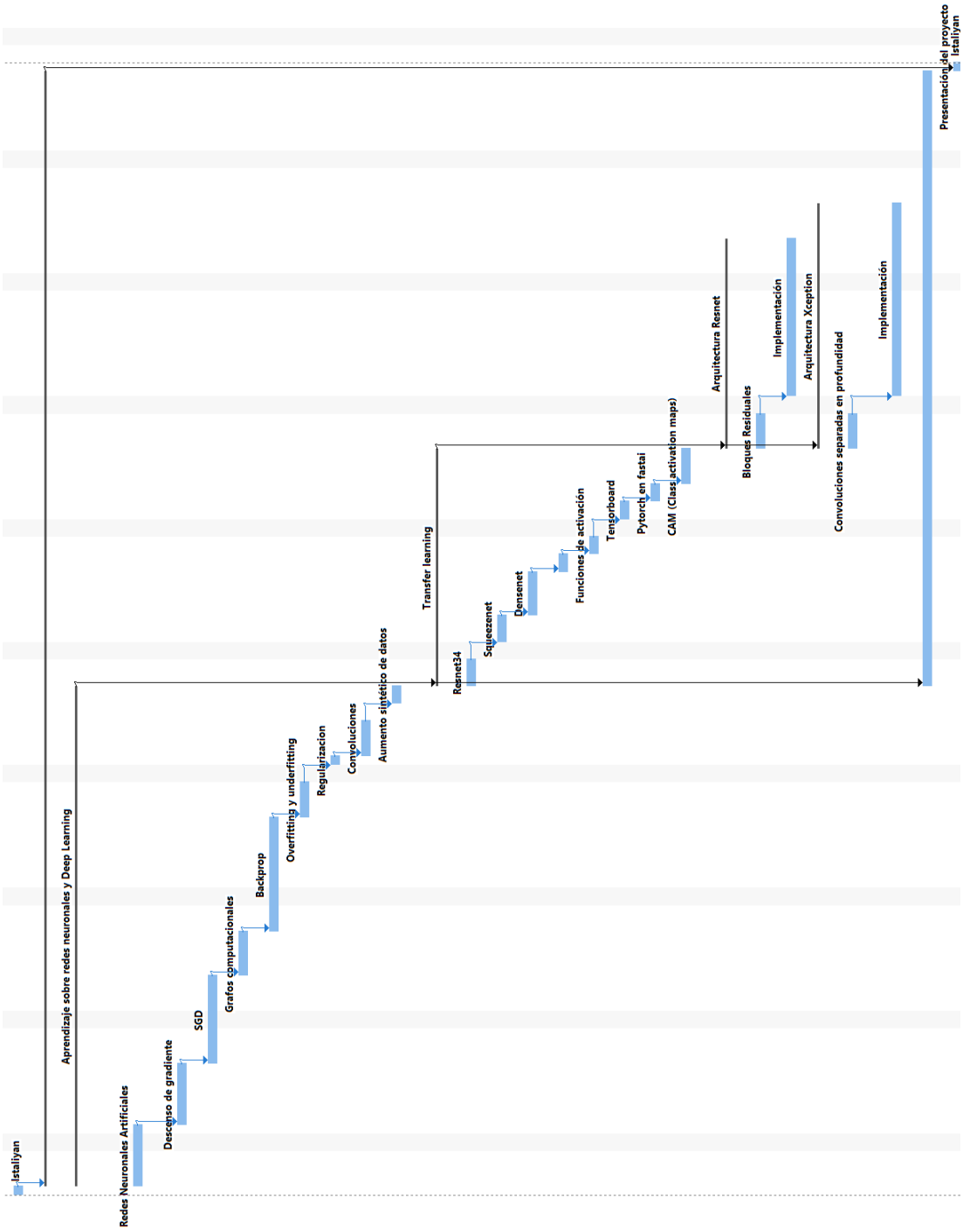


Figura 2.4: Diagrama Gantt en el que se indica la duración relativa de cada tarea.

En este apartado cabe destacar que las aplicaciones basadas en Deep Learning suelen ser computacionalmente muy costosas. Realizar estos cálculos en la CPU casi siempre resulta inadecuado y por tanto se deben usar GPUs, y mas recientemente TPUs (Tensorial Processing Unit) que debido a su capacidad de paralelización permiten realizar estos cómputos. Durante este proyecto se ha utilizado la GPU detallada en el Capitulo 1 pero su rendimiento es modesto.

Muchos de los resultado obtenidos podrían haber sido mejores, al menos respecto al tiempo de ejecución necesario para obtenerlos, en caso de disponer de una GPU mejor. Los precios de este tipo de hardware pueden variar entre los 500 €- 1000 €para GPUs que puedan obtener un buen rendimiento en modelos similares a los presentados en este proyecto hasta los 10000 €para GPUs y TPUs usados en datacenters de forma profesional.

Pasemos ahora a la estimación del pago basado en las unidades de esfuerzo. En este caso una unidad de esfuerzo supondrá una hora de trabajo realizado por el desarrollador. El proyecto tiene una carga de 12 créditos ECTS. Un crédito ECTS equivale a 25 horas de trabajo, por tanto obtenemos 300 horas de trabajo totales para la realización del proyecto.

El sueldo medio de un ingeniero informático recién graduado oscila entre los 18.000€ y los 22.000€ anuales brutos. En este caso supondremos el valor mínimo de este intervalo. De esta forma obtenemos un sueldo bruto de 1500€ y por tanto un pago de 68,18€por jornada laboral completa (8 horas) o aproximadamente 8,5€ por hora de trabajo.

Por tanto si multiplicamos el precio por hora obtenido por las horas totales que hay que realizar obtenemos un coste total de 2556,81€ para la realización del proyecto. Como no hay costes materiales que cubrir este es el precio final.

Capítulo 3

Contexto tecnológico

3.1. Introducción

En este capítulo se va a realizar una exposición mas detallada de las librerías usadas así como herramientas que las complementan de modo que en el Capítulo 5 no sea necesario centrarse en estos detalles.

3.2. Fastai

Como ya se ha indicado, fastai es una librería de aprendizaje profundo que permite al usuario usar componentes de alto nivel para diseñar modelos de forma rápida y obtener resultados que sean comparables a los obtenidos por gigantes de la tecnología como Google o Facebook. En la mayoría de casos obviamente esto no ocurrirá pero ha habido momentos en los que se ha probado el potencial de esta librería, como en 2017 [11] cuando un equipo de estudiantes consiguió obtener mejores resultados que Google e Intel en la competición *DAWNBench*.

Fastai también permite realizar modificaciones a bajo nivel lo que hace que la librería sea apta para un amplio espectro de usuarios. Para ver como se consigue esto debemos fijarnos en la arquitectura de la librería.

En la Figura 3.1 se pueden ver los bloques que componen la API de fastai. Los bloques superiores indican las aplicaciones típicas para las que están configuradas las clases de fastai: visión artificial, procesamiento del lenguaje natural...

Esta configuración base incluye parámetros predefinidos que han sido probados con anterioridad y se ha llegado a la conclusión de que producen buenos resultados para un amplio número de situaciones. Por ejemplo, el learning rate predefinido en la clase Learner tiene un

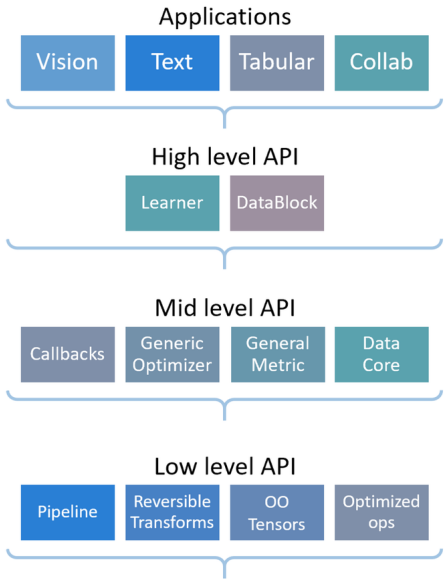


Figura 3.1: Arquitectura de la librería fastai.

valor de 0,001. Este valor probablemente será modificado y es recomendable que sea así pero es un buen lugar de inicio.

Además de los parámetros también se incluyen decisiones ya probadas sobre distintos aspectos de las arquitecturas. Por ejemplo, una parte muy importante del proceso de aprendizaje de un modelo es el algoritmo mediante el que se realiza la optimización. El algoritmo básico para realizar esta optimización es el descenso de gradiente, explicado en más profundidad en el Capítulo 4. Pero existen variantes como el descenso de gradiente estocástico (también explicado en el Capítulo 4), Adam [12] o el descenso de gradiente con momento [13] (o simplemente Momentum).

Fastai por defecto usa el algoritmo Adam que produce resultados satisfactorios habitualmente. Lógicamente este algoritmo no será el que obtenga los mejores resultados siempre, si fuese así se usaría siempre. Dependiendo del dataset que se esté utilizando, las transformaciones que se puedan realizar sobre las imágenes, la arquitectura que se ha implementado, etc... se pueden obtener mejores resultados adaptando esta selección. El descenso de gradiente estocástico suele tardar más tiempo pero puede producir un modelo que generalice mejor [14].

En la segunda fila de la Figura 3.1 se pueden ver algunos ejemplos de clases de alto nivel. La clase Learner tiene un papel fundamental en cualquier aplicación que use fastai. En ella se introduce un modelo, un *Dataloader* y una función de pérdida (también llamada función de coste).

Antes de continuar con esta clase veamos las principales características de un *Dataloader*.

La clase *Dataloader* nos permite trabajar con el dataset que se va a usar para realizar el aprendizaje. Se debe especificar un lugar desde el que se cargan los elementos, si se está trabajando con imágenes se especifica la carpeta, si en cambio son datos tabulares se puede especificar la ruta de un fichero CSV o TSV, también existe la posibilidad de especificar la URL de la que descargar alguno de los datasets ofrecidos por fastai como CIFAR o MNIST.

Aparte de especificar el dataset se puede definir el número de elementos que queremos en cada batch, el número de procesos que se van a usar para cargar los datos, si queremos mezclar los elementos (esto puede ayudar a obtener una mejor generalización al aplicar el modelo al dataset de validación).

Volviendo a la clase *Learner*, esta nos permite definir aparte de la tasa de aprendizaje, las métricas de las que queremos realizar un seguimiento, callbacks que queremos introducir para modificar el bucle de aprendizaje etc.

El bucle de aprendizaje es la función principal del Learner ya que es el momento en el que se modifican los valores de los parámetros del modelo con el objetivo de minimizar los valores de la función de pérdida que se ha seleccionado.

```
from fastai.vision.all import *

learner = cnn_learner(dls, resnet34, opt_func=SGD,
      loss_func= CrossEntropyLossFlat(),
      metrics = accuracy,
      cbs=Mixup)

learner.fit(10)
```

Listing 3.1: Ejemplo de un Learner en fastai

En el Listing 3.1 se puede observar la inicialización de un Learner para una arquitectura *Resnet 34* que usa un Dataloader del que obtiene los datos y descenso de gradiente estocástico como método de optimización, una función de coste propia de fastai que supone una modificación de la entropía cruzada, la especificación de las métricas que en este caso es la precisión de clasificación y un callback para realizar aumento sintético de datos.

A continuación se llama al bucle de aprendizaje que se ejecuta durante 10 epochs. Todo esto en unas pocas líneas de código. Esto supone una gran simplificación de complejidad con respecto a otras librerías de Deep Learning como Pytorch que se verá en la siguiente sección.

```
from fastai.vision.all import *

def correcta(nombre_imagen):
   return nombre_imagen[0] == 'C'

dls = ImageDataLoaders.from_name_func(path, get_image_files(path),
    valid_pct=0.3, seed=42, label_func=correcta(),
    item_tfms=Resize(100, ResizeMethod.Pad, pad_mode='zeros)
)
```

Listing 3.2: Ejemplo de un Dataloader

En el Listing 3.2 se puede observar la inicialización de un Dataloader de imágenes donde se especifica el path en el que se encuentran, el porcentaje de validación que nos permite indicar la cantidad de imágenes sobre las que el modelo probará su capacidad de generalización, una semilla de modo que los batchs contengan siempre las mismas imágenes si volvemos a entrenar el modelo, una función que determina si la imagen es correcta dependiendo de si la primera letra del nombre es el caracter 'C' y una transformación donde las imágenes van a pasar a tener un tamaño de 100 pixels de ancho por 100 pixels de alto.

Y estas son solo algunas de las clases usadas para visión artificial, existen otras muchas como *TextDataLoaders* o *LMLearner* para procesamiento de lenguaje natural, *TabularDataLoaders* para datos tabulares, etc.

Pero es posible que las necesidades del programador sean mas específicas. Para ello se dispone de las clases que forman el nivel medio y bajo de la arquitectura. Los callbacks como ya se ha indicado modifican el bucle de aprendizaje insertando el código del callback en distintos momentos del bucle, por ejemplo al empezar cada epoch, al empezar el aprendizaje, después de realizar las predicciones, etc.

```
1 class ModelResetter(Callback):
2   def begin_train(self): self.model.reset()
3   def begin_validate(self): self.model.reset()
```

Listing 3.3: Ejemplo sencillo de un Callback

En el Listing 3.3 se puede ver un ejemplo sencillo de como puede ser un callback personalizado que hereda de la clase Callback de fastai. En este caso se eliminan los valores calculados en las diferentes capas del modelo al principio del entrenamiento y al principio de la validación [15].

Es posible también que sea necesario implementar un nuevo algoritmo mediante el que realizar la optimización ya que los ya implementados no se adaptan a las necesidades del programador. Para ello se puede usar la clase *Optimizer* de la cual pueden heredar los optimizadores personalizados.

Este tipo de optimizador genérico no existe en otras librerías de Deep Learning donde se debería programar el algoritmo desde cero.Como todos los algoritmos están basados en la idea del descenso de gradiente en esencia lo que proporciona esta abstracción es la posibilidad de definir la forma en la que se da el paso hacia los valores de la función que la minimizan (más en detalle en el Capítulo 4) y la forma en la que se guardan las estadísticas.

Por último al nivel mas bajo se pueden crear *Pipelines* personalizadas que permiten al programador definir una serie de funciones que serán aplicadas en serie a un elemento. Este elemento puede ser una imagen en forma de tensor, un objeto Image, objetos TensorText, etc.

Si volvemos al Listing 3.2 podemos ver que en la penúltima línea se realiza la operación Resize sobre el dataset que estamos cargando. Mediante las *Pipelines* podemos sustituir las funciones predeterminadas que permite usar fastai. De esta forma se dispone de total libertad para modificar los datos que estemos cargando para ser utilizados por el modelo y además

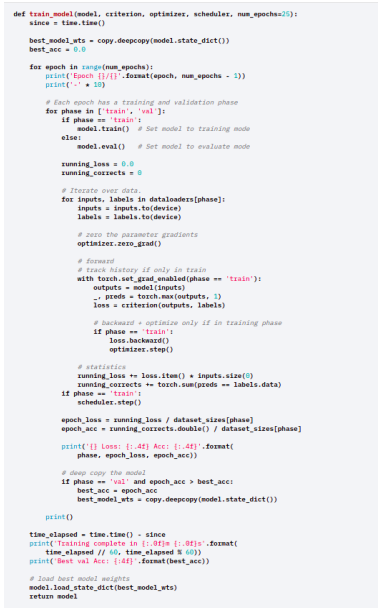
el tipo de datos que modifiquemos puede ser el que deseemos siempre y cuando funcione correctamente con las funciones que se definan.

Como se ha visto a lo largo de esta sección fastai es una librería diseñada para ser apta en múltiples niveles de uso y adaptarse a toda clase de necesidades. El diseño de la arquitectura que da la posibilidad de usar abstracciones o implementar mucha de la funcionalidad hace que sea idónea para un proyecto como este.

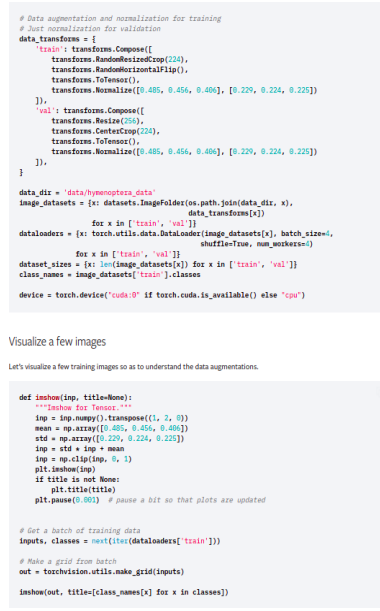
3.3. PyTorch

Como fastai está construida sobre Pytorch, así como otras muchas librerías, es conveniente dedicar una breve sección a esta librería así como hacer una comparación entre algunos de los elementos más habitualmente usados.

PyTorch es una librería de aprendizaje profundo que a su vez está basada en Torch. Como base sobre la que se implementa fastai, PyTorch proporciona un cómputo eficiente de tensores y distintas herramientas con las que construir redes neuronales profundas.



(a) Bucle de aprendizaje



(b) Carga de datos

Figura 3.2: Muestras de código haciendo uso de PyTorch [16]

En la la imagen *a* de la Figura 3.2 podemos ver el código necesario en PyTorch para realizar el bucle de aprendizaje para un modelo definido previamente en una aplicación de Transfer Learning. Si recordamos el Listing 3.1 vemos como en fastai definimos el bucle con todos los parámetros necesario en dos líneas cuando en PyTorch se requiere definir a mano

todos los pasos. Sobre todo en los casos en los que se usa Transfer Learning la implementación del bucle la mayoría de veces supone una fuente de errores aportando muy poco en cuanto a modificaciones funcionales de valor.

Al hacer zoom en la imagen (se ha reducido el tamaño para no ocupar varias páginas con este código) podemos ver que manejamos los mismos elementos que en fastai, un modelo en el que ajustaremos los pesos y los *bias*, una función de activación (*criterion*), un algoritmo de optimización y el número de epochs que se traduce en iteraciones del bucle.

El único elemento novedoso es el *scheduler* que básicamente es una función que indica cómo debe variar la tasa de aprendizaje a lo largo del proceso ya que habitualmente se quiere empezar con una tasa mayor e ir decreciendo según se acerca la función que se está optimizando al valor mínimo. Para ello en fastai también se dispone de una forma muy cómoda para definir este comportamiento.

En la imagen *b* de la Figura 3.2 se implementa una funcionalidad similar a la del Listing 3.2 donde se realizan transformaciones a los datos de entrada,se llevan a GPU si está disponible y se implementa una función de visualización. De nuevo toda esta funcionalidad se implementa en fastai en dos líneas sin perder expresividad.

3.4. Tensorboard

Por último veremos una herramienta para visualización de grafos de modelos, métricas, imágenes, etc. Forma parte de la librería Tensorflow y ofrece una interfaz muy fácil de usar además de cierta funcionalidad que no se encuentra en otras librerías e implementar por completo sería un trabajo considerable.

Tensorboard está integrado en PyTorch y por tanto resulta muy sencillo transferir todos los datos necesarios para crear las gráficas, visualizar las imágenes, etc.

Únicamente se debe instanciar una clase *SummaryWriter()* que incluye los métodos necesarios mediante los que transferir los datos y poder visualizarlos.

```
1 from torch.utils.tensorboard import SummaryWriter
2
3 writer = SummaryWriter(log_dir=path)
4
5 for epoch in range(epochs):
6       ...
7   writer.add_scalar('Perdida', perdida, epoch)
8       ...
9 writer.close()
```

Listing 3.4: Ejemplo de la funcionalidad básica de Tensorboard

Como vemos en el Listing 3.4 en cada iteración del bucle de aprendizaje podemos introducir una línea en la que enviamos las estadísticas necesarias a Tensorboard, en este caso la pérdida acumulada hasta el momento, de modo que la variable *perdida* contendrá los valores de la variable dependiente y la variable *epoch* los de la variable independiente.

Una peculiaridad a tener en cuenta es que cada vez que trasladamos algún tipo de dato a Tensorboard debemos pasarlo en forma de Tensor que es el tipo básico con el que trabaja PyTorch. Por tanto no se puede introducir una imagen, incluso un video o audio que no hayan sido convertidos a tensores previamente.

Capítulo 4

Introducción a las redes neuronales

En este capítulo se van a describir los principales conceptos teóricos necesarios para la realización de este proyecto. En primer lugar se explicarán los conceptos mas generales de aprendizaje automático sobre los que se apoyan muchas de las ideas usadas en aprendizaje profundo.

4.1. Conceptos básicos de aprendizaje automático

El término *Deep Learning* surgió por primera vez en 1986, en el artículo de conferencia *Learning While Searching in Constraint-Satisfaction-Problems* [17] escrito por Rina Dechter. Pero para llegar al Deep Learning hubo un largo camino que empezó con avances en el campo de la Inteligencia Artificial (IA). Conseguir máquinas con la capacidad de razonar y aprender para poder imitar la inteligencia natural de los seres humanos ha cautivado la atención de la humanidad desde la antigüedad y este es el problemas que intenta resolver la IA. Con la evolución de este campo han surgido diversas disciplinas, englobadas dentro del mismo, cada una con un enfoque diferente.

Una de las principales disciplinas cuyos avances han destacado en los últimos años ha sido el Aprendizaje Automático (en inglés, *Machine Learning*). La manera de abordar el problema de la IA por parte del Machine Learning ha sido introducir un concepto clave en el aprendizaje natural de los seres humanos: la experiencia. Hasta este momento los algoritmos diseñados para resolver los problemas que se planteaban usaban reglas definidas por los programadores para obtener los resultados. En Machine Learning el programador crea un algoritmo que sea capaz de aprender mediante la experiencia, por tanto el resultado de este algoritmo es otro algoritmo del que obtendremos los resultados finales. Pero ¿que significa aprender en términos de Machine Learning?

Según Mitchell [18, p. 2] se dice que " un programa aprende de una experiencia E con respecto a alguna clase de tareas T y una medida de rendimiento P, si el rendimiento sobre las tareas en T, medido mediante P, mejora con la experiencia E".

Esta definición nos indica que el algoritmo que se obtiene mediante técnicas de Machine Learning obtendrá mejores resultados a medida que es sometido a más experiencia. Y esto ocurre de manera análoga con los humanos. A medida que obtenemos más experiencia en un campo solemos obtener mejores resultados cuando repetimos la tarea.

Mitchell también menciona las clases de tareas. A continuación se encuentran algunos ejemplos de estas:

1. **Clasificación:** Trata de obtener la categoría a la que pertenece una entrada de nuestro algoritmo, por ejemplo, un algoritmo que diferencie a gatos y a perros. En este caso tendríamos 2 categorías pero estas pueden ser muchas más. ImageNet es una base de datos con imágenes que pertenecen a 20.000 categorías. Esta es la clase de tarea es la que más nos interesa ya que el problema que hemos planteado en este TFG pertenece a esta clase.

2. **Regresión:** Trata de estimar la relación de una variable dependiente con una o variar variables independientes. Un ejemplo puede ser predecir el valor de una hipoteca en función del número de metros cuadrados de la vivienda y el numero de habitaciones.

3. **Transcripción:** Trata de transformar datos en formato de audio, imagen o vídeo a texto. Este tipo de tarea tienen su propia disciplina, Procesamiento de Lenguaje Natural (en inglés, Natural Language Processing).Un ejemplo de este tipo de tareas puede ser pasar a texto la matrícula de un coche a partir de la fotografía del mismo.

4. **Traducción:** Trata de traducir texto de un lenguaje natural a otro. De nuevo esta es una tarea que pertenece al campo del Procesamiento de Lenguaje Natural. La complejidad de esta tarea puede engañar ya que no es una cuestión de sustituir las palabras por su traducción mediante una búsqueda en una base de datos. Las traducciones de este tipo casi siempre resultan de baja calidad. La diferencia de las traducciones basadas en Machine Learning es que se usa aprendizaje sobre grandes cantidades de datos que permiten incluir el contexto y reglas gramáticas en la traducción.

5. **Reducción de ruido:** Trata de eliminar el ruido de una señal. Habitualmente la señal se trata de una imagen o una señal de audio. En este tipo de tareas hay que tener en cuenta que la modificación de la señal puede resultar en ocasiones en pérdida de información, lo que se traduciría a pérdida en detalle en las imágenes, dependiendo de los algoritmos que se usen.

Pero, ¿donde encaja el Deep Learning en lo que se ha descrito hasta ahora? Ciertos problemas que se intentaron resolver con Machine Learning demostraron ser demasiado complejos o no se pudo llegar a una solución suficientemente buena con la tecnología existente.

El Deep Learning comparte muchas de las características citadas anteriormente, los algoritmos también aprenden y las técnicas creadas en esta disciplina también sirven para resolver

problemas pertenecientes al tipo de tareas que se han enumerado. La novedad que aporta es la capacidad de abstracción. Esta es la cualidad que permite a los algoritmos basados en técnicas de Deep Learning obtener resultados considerablemente mejores en problemas como la clasificación de imágenes.

Cuando se usa el término abstracción en el párrafo anterior se está haciendo referencia a la capacidad para distinguir una característica y poder analizarla separadamente del resto de elementos. La forma en la que se consiguió esto fue con la introducción de las redes neuronales profundas. Pero no fue únicamente esto lo que permitió dar el salto a Deep Learning.

Otro de los problemas a los que se enfrentaba el Machine Learning era la generalización [19], sobre todo en el momento en el que hay que trabajar con datos de espacios de múltiples, en ocasiones millones, de dimensiones. Este problema se denomina la *maldición de la dimensión* [20]. Este concepto se es fácil de entender mediante un ejemplo. En el caso de que queramos expresar un color podemos usar el sistema RGB (hay otros sistemas) donde cada color viene determinado por la intensidad de cada uno de los colores primarios de la luz: rojo (en ingles, Red), verde (en inglés, Green) y azul (en inglés, Blue). Por lo tanto cada color esta definido por tres números que lo identifican unívocamente. Al tener tres números decimos que el espacio de los colores RGB es tridimensional.

Pero fácilmente podemos ver como la dimensionalidad puede subir en el caso de enfrentarnos a datos como el estado de salud de pacientes de un hospital que puede depender de múltiples factores como análisis de sangre, enfermedades previas, consumo de drogas, operaciones, tratamientos previos, etc.

Los avances tecnológicos permitieron usar algoritmos que necesitaban una gran cantidad de computación. Por otro lado las soluciones basadas en Deep Learning usan una gran cantidad de datos, los cuales hace 30 años no se podían conseguir con la misma facilidad que hoy en día, no debemos olvidar la importancia que ha adquirido hoy en día el campo de los Macrodatos (en inglés, Big Data).

El surgimiento del Deep Learning no ha sido cuestión de un único descubrimiento, más bien han sido innovaciones sobre elementos preexistentes en Machine Learning acompañados de las innovaciones tecnológicas que han permitido usar estas ideas de forma efectiva.

Esto sitúa al Deep Learning como una subdisciplina del Machine Learning que a su vez pertenece al campo de la Inteligencia Artificial. Se puede visualizar lo descrito en la siguiente figura.

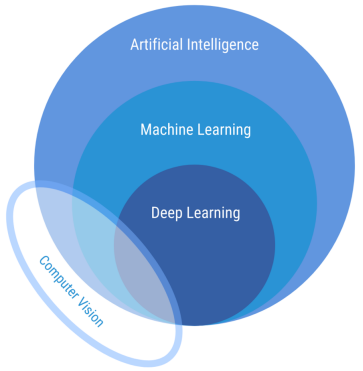


Figura 4.1: Campo y subcampos de la Inteligencia Artificial (IA). Fuente: [21]

En esta figura se pueden apreciar las separaciones entre las distintas disciplinas tal y como hemos descrito hasta ahora. Pero hay una disciplina más que no se ha mencionado hasta este punto. Se trata de la Visión Artificial (en inglés, Computer Vision).

Este campo, en esencia, trata de obtener información útil de las imágenes [22]. Para ello se empezó imitando el ojo humano aunque después según evolucionaba el campo se introdujeron elementos que no estaban directamente relacionados con la visión humana. Este es un buen punto para reflexionar sobre como muchas de las ideas que surgen para resolver problemas imitan estructuras que ya existen. La visión artificial es un ejemplo pero las redes neuronales artificiales, sobre las que se profundizará mas adelante, son otro. Los inicios de este campo están estrechamente vinculados a los de la Inteligencia Artificial. Ambos se formaron en los años sesenta aunque lógicamente ya se habían sugerido ideas en estas líneas muchos años antes.

El motivo por el que se ha incluido este campo en la Figura 4.1 es la naturaleza de este TFG. La clasificación de imágenes utiliza algunos conceptos de Visión Artificial que son usados junto con técnicas de Deep Learning y por lo tanto parecía justo incluirlo. Como se puede apreciar en la figura la Visión Artifical no es un subcampo propio de la Inteligencia Artificial, más bien tiene usos en algunas de las tareas que intenta resolver la IA pero hay otras ideas que no están vinculadas a esta.

Una vez descrito el mapa del campo en el que estamos trabajando estamos en posición para explicar los conceptos principales del aprendizaje automático. Pero antes debemos hacer una pequeña clasificación entre los distintos tipos de aprendizaje automático que existen. Esto es lo que se describe en la siguiente sección.

4.1.1. Tipos de aprendizaje automático

En esta sección se revisará brevemente los principales tipos de aprendizaje que existen en el campo del Machine Learning. Las principales diferencias surgen de la cantidad de datos que se dispone a la hora de realizar el aprendizaje, como vimos este punto tiene gran importancia en este campo, el orden en el que el algoritmo pasa a disponer de estos datos y las características que tiene el conjunto de datos de prueba.

Debido a que en este punto todavía no se ha explicado el significado de un conjunto de datos de prueba se hará un pequeño inciso para aclarar algunos conceptos antes de continuar.

Ya se ha mencionado antes que los algoritmos basados en Machine Learning, en especial los basados en Deep Learning, necesitan de una gran cantidad de datos que se usarán para realizar el aprendizaje.

Estos datos con los que se va a trabajar reciben el nombre de **conjunto de datos** (en inglés, dataset). Los conjuntos de datos pueden tener muchas características que le hagan buenos candidatos para un problema de Machine Learning o no. Estas son algunas de las mas relevantes a la hora de evaluar si un conjunto de datos es adecuado o no.

- **Precisión:** El conjunto de datos no debe contener datos erróneos. Por ejemplo, en un problema de clasificación de imágenes cada imagen debe estar acompañada de su categoría. Si hay muchas imágenes que se encuentran en la categoría incorrecta el modelo que se está entrenando clasificará las imágenes en las categorías incorrectas.
- **Tamaño:** El conjunto de datos debe contener suficientes elementos como para poder realizar el entrenamiento. De forma aproximada debemos entrenar el modelo sobre al menos un numero de datos que se encuentre en un orden de magnitud por encima de los parámetros que se vayan a entrenar. En este punto no se ha descrito todavía el concepto de parámetro en el contexto de Machine Learning pero es una regla que podemos tener en mente.
- **Representación:** El conjunto de datos debe representar de manera adecuada el problema que se está intentando resolver. Por ejemplo, usando el ejemplo de las hipotecas que usamos en la sección anterior, no resulta de provecho usar un conjunto de datos donde se estudien exclusivamente hipotecas para casas con 4 dormitorios, a no ser que este sea exactamente el problema que estemos estudiando. Las hipotecas de las casas con menos dormitorios serán mas baratas pero si no las incluimos nuestro conjunto de datos no será representativo y las predicciones de nuestro modelo no serán adecuadas.

Una vez vistas las características deseables de un dataset pasaremos a ver la forma en la que se trabaja con los conjuntos de datos en Machine Learning. Habitualmente se divide el conjunto de datos en tres subconjuntos, cada uno con un propósito específico.

- **Conjunto de entrenamiento:** Estos son los datos que se van a usar para que se pueda llevar el aprendizaje, o lo que es igual, los datos que van a permitir que cambien los parámetros del modelo.

- **Conjunto de prueba:** Estos son los datos sobre los que se va a probar el funcionamiento del modelo que ha sido entrenado con los datos del conjunto de entrenamiento. Los datos del conjunto de prueba nunca deben haber sido usados en el entrenamiento ya que el objetivo de este conjunto de datos es ver como se va a comportar el modelo antes ejemplos que no ha visto hasta ahora, dicho de otra manera, se va a probar como de bien generaliza el modelo.
- **Conjunto de validación:** Este conjunto de datos pertenece a una categoría intermedia entre el conjunto de prueba y el de entrenamiento. Las razones por las que se creó este conjunto de datos tendrán más sentido más adelante pero de forma simplificada este conjunto de datos se usa para evaluar de forma imparcial el éxito del modelo y poder de esta forma modificar los hiperparámetros del modelo sin usar el conjunto de prueba. Es una forma de evitar usar el conjunto de datos de prueba para modificar los hiperparámetros y en cierto sentido engañar al modelo.

En este punto se han explicado las principales características que buscamos en los conjuntos de datos y cual es la forma habitual de trabajar con ellos. Volvamos a los principales tipos de aprendizaje automático.

- **Aprendizaje supervisado:** En este tipo de aprendizaje el modelo va a recibir un conjunto de datos junto con la etiqueta que le corresponde a ese dato. Lo que aprenderá el modelo de esta forma es la relación que existe entre los datos y sus etiquetas para después poder generalizar y aplicarlo a datos nuevos donde no va a existir una etiqueta para ese dato ya que va a ser el modelo el que la proporcione. La tarea de clasificar imágenes se encuentra en este campo ya que las imágenes del conjunto de datos están etiquetadas y el modelo aprende a predecir (en caso de que pueda generalizar de forma adecuada) la etiqueta de nuevas imágenes.
- **Aprendizaje no supervisado:** En este caso los datos que se proporcionan al modelo no están etiquetados y el objetivo es conseguir que el modelo identifique relaciones entre los datos pero sin que tengamos la ayuda que suponen las etiquetas de los datos. El agrupamiento (en inglés, clustering) es un problema típico que se encuentra en el campo del aprendizaje no supervisado. En este tipo de problemas el modelo debe agrupar los datos con características similares en grupos y separar los que tengan características contrarias. Esto puede hacer difícil evaluar el rendimiento del modelo ya que el clustering suele ser subjetivo por naturaleza.
- **Aprendizaje por refuerzo:** En este caso el aprendizaje se basa en las acciones que debe tomar un agente. Se introduce aquí este nuevo concepto que no se ha mencionado antes. Como indican Russell y Norvig [23] : “un agente es algo que razona y un agente racional es aquel que actúa con la intención de alcanzar un mejor resultado”. El aprendizaje por refuerzo se diferencia de los otros paradigmas mencionados en el uso de un agente al que se le aplica una función que devuelve un valor numérico que representa la recompensa que obtiene el agente por realizar una acción. El objetivo del agente es maximizar esta función. Una característica relevante de este tipo de aprendizaje es que existe un enfrentamiento entre tomar decisiones ya probadas que han funcionado anteriormente y tomar decisiones novedosas que pueden conllevar una mayor recompensa pero también pueden suponer una penalización.

Una vez descritos los principales paradigmas podemos empezar introduciendo algunos de los conceptos propios del aprendizaje automático y empezaremos por el modelo computacional que revolucionó este campo: las redes neuronales artificiales.

4.1.2. Redes neuronales artificiales

Las redes neuronales artificiales surgieron en los años 60, momento en el que se consolidó el campo de la Inteligencia Artificial. De manera similar a la Visión Artificial, las primeros desarrollos en redes neuronales artificiales surgieron con la intención de desarrollar un modelo computacional del aprendizaje biológico. Para lograr este objetivo se intentó reconstruir el cerebro humano partiendo de sus componente más básicos, las neuronas y las sinapsis que las conectan. Al igual que en la Visión Artificial no se consiguió el objetivo por completo pero resultó en grandes avances para la IA.

Las neuronas biológicas son muy complejas pero su funcionamiento básico es relativamente sencillo. Estas células se encargan de transmitir los impulsos nerviosos cuando reciben algún tipo de estímulo. Una vez estimulada una neurona esta se activa, lo que produce una corriente eléctrica que viaja a través de las sinapsis.

Las primeras neuronas artificiales fueron desarrolladas por Frank Rosenbatt en 1957 y se llamaron perceptrones. Este tipo de neuronas actúan como una función con un numero variable de entradas y una salida binaria. La salida binaria nos indica si la neurona ha sido activada o no.

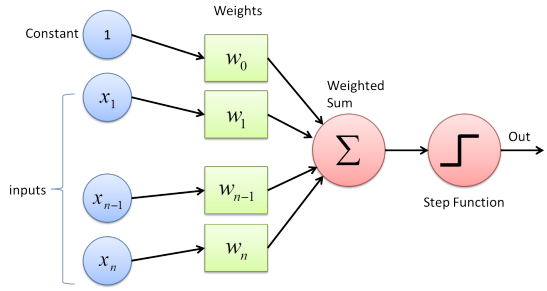


Figura 4.2: Diagrama de un perceptrón. Fuente:[24]

En la Figura 2.2 podemos ver las entradas del perceptrón $x_1, x_2, ..., x_n$. Para calcular la salida se realiza la suma ponderada usando los pesos correspondientes y esta suma será evaluada mediante una función de activación la cual indica si la neurona ha sido activada o no.La primera entrada es una constante con valor 1 y por lo tanto el peso w_0 nunca será cancelado, algo que puede ocurrir con los demás pesos. A este peso se le suele denominar sesgo (en inglés, bias). Este comportamiento se expresa mejor mediante la siguiente función definida a trozos.

$$salida = \begin{cases} 0 & \text{si } \sum_{n=1}^{n} w_n x_n + bias \leq 0 \\ 1 & \text{si } \sum_{n=1}^{n} w_n x_n + bias > 0 \end{cases} \tag{4.1}$$

Un ejemplo muy simple del uso del perceptrón podría ser decidir si deberíamos quedarnos en casa o no en función del tiempo que vemos por la ventana. Podremos observar distintos fenómenos, está nublado, hay viento, hay lluvia, etc. Cada uno de estos fenómenos será una entrada para el perceptrón. En caso de que esté nublado la entrada correspondiente tendrá el valor 1. Los pesos indicarán la importancia que le damos a cada fenómeno,si no nos importa el viento pero odiamos mojarnos entonces el peso de la lluvia será mayor que el del viento.

Por otro lado el bias corresponde a las ganas que tenemos de quedarnos en casa. Si el bias tiene un valor muy alto entonces será fácil que al sumarlo a la suma ponderada de los demás pesos obtengamos un valor mayor que 0 y se active el perceptrón, indicando que deberíamos quedarnos en casa. Con un valor negativo del bias se podría obtener un valor negativo en la suma y que el perceptrón devuelva un 0. Con este ejemplo podemos ver que el perceptrón tiene ciertas similitudes a como tomamos decisiones los humanos.

Pero en este concepto hay un problema. Observando la ecuación 2.1 podemos ver que en casos en los que la suma ponderada tenga valores muy cercanos a 0 una pequeña variación en el valor de los pesos o las entradas puede suponer un cambio en la activación del perceptrón.

Este cambio tan brusco al hacer una pequeña modificación puede resultar problemático a la hora de modificar los pesos del perceptrón. Pero ¿qué motivo habría para que cambiasen los pesos del perceptrón? Los perceptrones no fueron creados para ser usados de forma independiente, si no para ser los bloques con los que se construyen las redes neuronales artificiales. Cada una de las salidas de un perceptrón puede ser una de las entradas de un perceptrón de la capa siguiente. Una capa de una red neuronal esta formada por múltiples neuronas artificiales que no se encuentran conectadas entre sí pero en cambio sí que están conectadas con las neuronas de la siguiente capa. Las salidas de las neuronas de una capa son las entradas de la capa siguiente.

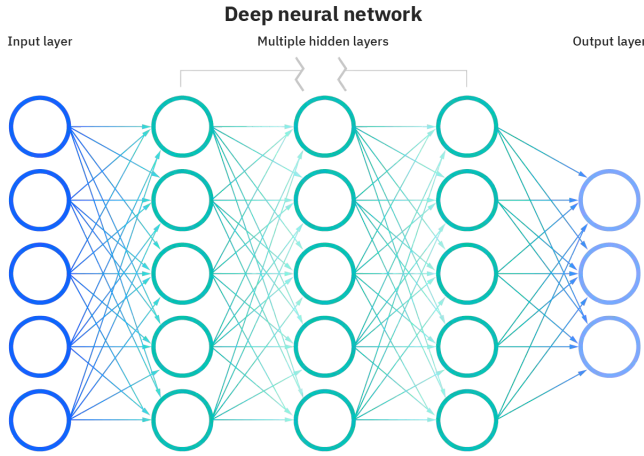


Figura 4.3: Diagrama de una red neuronal. Fuente: [25]

Existen dos capas especiales que son las de color azul en la figura 2.3. Estas son la capa de entrada y la capa de salida. Las neuronas de la capa de entrada no tienen a su vez entradas.Si

no hay entradas en la ecuación 2.1 obtendremos $w_0 + bias$ ya que el resto de términos se anularán. Se comprobará si este valor es mayor o menor que 0 y el perceptrón tendrá una salida en función a ello. Esto significa que al perceptrón se le creará con el peso w_0 y *bias* adecuado para que su salida sea 1 o 0 en dependiendo de la entrada que se quiera definir para la red neuronal.A efectos prácticos podemos pensar que simplemente estamos introduciendo un 0 o un 1 a las entradas de la segunda capa de la red.

Las capas de salida no tienen salida ya que representan el resultado que obtiene la red. Estos perceptrones en realidad sí que tienen salida pero normalmente no se representa en los diagramas ya que no es la entrada de otra capa.

Volvamos a la pregunta que habíamos planteado sobre la modificación de los pesos del perceptrón. Estos pueden cambiar ya que esta es la manera en la que va a ocurrir el aprendizaje. En la capa de entrada se van a introducir los valores a partir de los cuales se va a calcular la salida. Pero para obtener la salida que esperamos los pesos y *bias* deben ser los adecuados. Por ello si no devuelven la salida correcta hay que modificarlos. Esta modificación puede resultar problemática ya que un pequeño cambio puede modificar la salida de forma que el perceptrón pase de activado a desactivado o viceversa. Pero muchas veces se requiere que este paso sea progresivo en lugar de instantáneo.

Este es el punto donde se introducen las neuronas sigmoidales. Lo que queremos obtener de estas neuronas es que una pequeña modificación en los pesos y *bias* se traduzca en una pequeña modificación en la salida. De esta manera se pueden ajustar los valores para evitar que haya cambios tan drásticos.

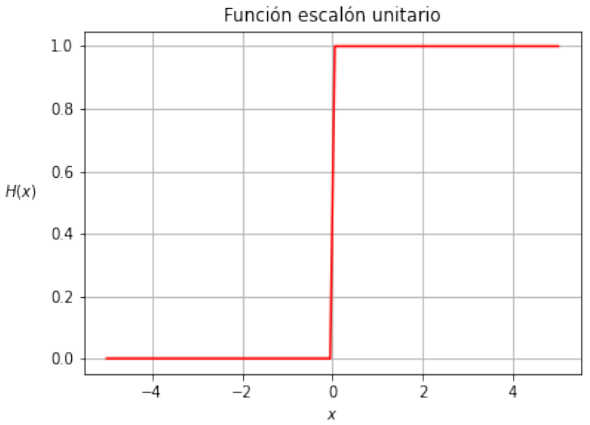
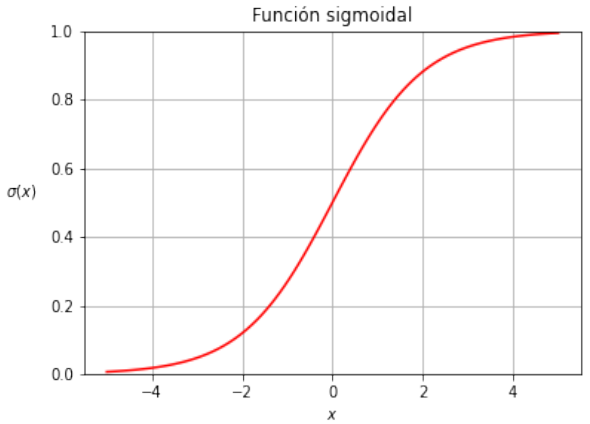
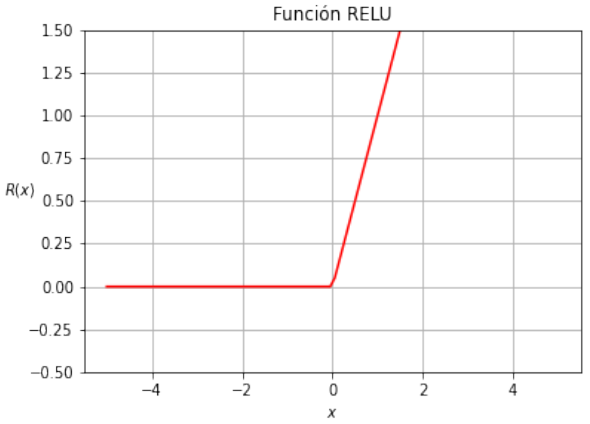
La neuronas sigmoidales se comportan de la misma forma que los perceptrones con la única diferencia encontrándose en la evaluación de la suma ponderada.

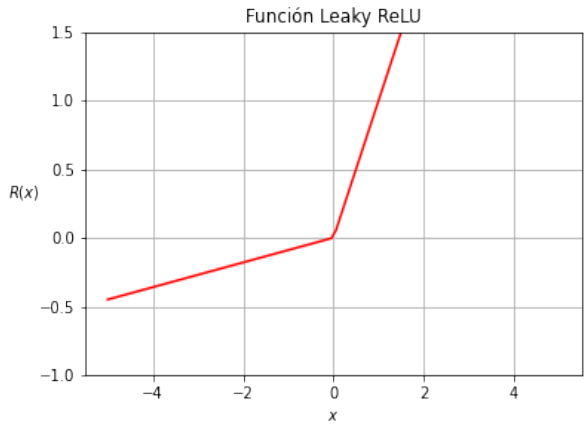
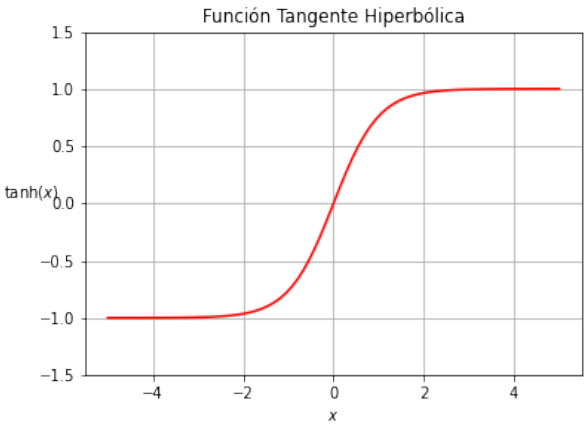
$$salida = \frac{1}{1 + e^{-\sum_{n=1}^{n} x_n w_n + bias}} \tag{4.2}$$

De esta forma no se compara la salida de la suma ponderada con otro valor sino que se introduce el resultado de la suma en la función sigmoide. Se puede ver de forma mas simplificada en la ecuación 2.3 donde z es el resultado de la suma ponderada:

$$\sigma(z) = \frac{1}{1 + e^{-z}} \tag{4.3}$$

La función sigmoide y la función escalón unitario, este es el nombre de la función 2.1, no son las únicas que se usan para determinar como se comporta la salida de una neurona artificial. Lo que cambia en esencia son las funciones de activación, la estructura de los pesos, entradas y *bias* se mantiene.

Gráfica de la función	Función de activación
	$H(x) = \begin{cases} 0 & \text{si } x \leq 0 \\ 1 & \text{si } x > 0 \end{cases}$
	$\sigma(x) = \frac{1}{1+e^{-x}}$
	$R(x) = \begin{cases} 0 & \text{si } x \leq 0 \\ x & \text{si } x > 0 \end{cases}$

	$R(x) = \begin{cases} \alpha x & \text{si } x \leq 0 \\ x & \text{si } x > 0 \end{cases}$ *En este caso $\alpha = 0{,}3$
	$\tanh(x) = \frac{e^x - e^{-x}}{e^x + e^{-x}}$

Cuadro 4.1: Principales funciones de activación usadas en Machine Learning.

En el cuadro 2.1 se introducen algunas funciones de activación que no se han mencionado hasta ahora.

- **ReLU :** El nombre completo es (en inglés, Rectified Linear Unit). Esta función es la recomendada para usar en redes neuronales actuales [19]. La razón por la que se da el paso de usar la función sigmoide a la ReLU es por el problema del desvanecimiento del gradiente. En este punto no se han introducido todavía los conceptos necesarios para explicar el problema, por lo tanto se profundizará en ello más adelante.

- **Leaky ReLU :** Modificación de la función de activación ReLU. La Leaky ReLU evita que las salida sea 0 en caso de que la entrada sea negativa. Resuelve el problema presente en la ReLU donde el gradiente es 0 en los casos en los que la entrada de la neurona es negativa. Al existir un gradiente con valor 0 esto anula el aprendizaje que ocurre en las próximas capas de la red. Esto se verá de forma mas clara cuando se introduzca el

algoritmo de propagación hacia atrás. No hay consenso sobre si siempre es mejor usar Leaky ReLU en vez de ReLU pero se ha demostrado que en muchas situaciones puede obtener resultados mejores [26].

- **Tangente Hiperbólica :** Variante basada en la función sigmoide. La principal ventaja que tiene esta función es que debido a que tiene un $Rango(\tanh(x)) = (-1, 1)$ la pendiente en los puntos cercanos a 0 produce derivadas con valores mayores. Esto hace que aplicando el algoritmo de propagación hacia atrás se converja de forma más rápida a una solución [27]. También tiene la ventaja de que las entradas cercanas a 0 producen salidas cercanas a 0 a diferencia de la sigmoide donde $\sigma(x) = 0{,}5$.

Existen otras funciones de activación con diferentes propósitos pero estas son las que se usarán principalmente en el desarrollo de este trabajo.

4.1.3. Descenso de gradiente

En esta sección vamos a introducir algunos conceptos que constituyen la base del funcionamiento de las redes neuronales artificiales. Esta sección en conjunto con la siguiente, que describe el funcionamiento del algoritmo de propagación hacia atrás, se centra en cómo ocurre el aprendizaje.

Para comprobar si una red neuronal está aprendiendo en primer lugar necesitamos una forma de evaluar su salida. Como se explicó al principio del capítulo, los resultados obtenidos mejoran a través de la experiencia a la que se somete al algoritmo. De modo que para saber en que medida está mejorando, o por el contrario empeorando, la red neuronal hay que poder cuantificar como de correcta es la salida de la red.

En este punto se introducen las funciones de coste. Este tipo de funciones nos van a indicar el error que está cometiendo la red neuronal dada una entrada y la salida que debería obtener. Si usamos el ejemplo de la clasificación de imágenes en este caso se compararía el resultado de la imagen que se quiere clasificar y la categoría a la que pertenece realmente esta imagen. La función más usada es el error cuadrático medio (en inglés, mean squared error).

$$ECM = \frac{1}{n} \sum_{x} \|f(\boldsymbol{x}, \boldsymbol{w}) - \boldsymbol{y}\|^2 \tag{4.4}$$

La ecuación 2.4 nos indica cómo calculamos este error. El valor de n se corresponde con el número de entradas que vamos a utilizar. Este debe ser equivalente al número de elementos que hay en el dataset de entrenamiento. De esta forma haremos la media entre del coste entre todos los elementos que se van a introducir en la red.

La función f tiene como entradas el vector que describe un elemento del dataset. Es importante indicar que $\boldsymbol{x} \in \mathbb{R}^n$ ya que este vector puede tener una dimensión, en este caso lo denominaremos escalar, y ser usado en un problema de regresión lineal o puede representar

una imagen, como es el caso en este proyecto, donde tendrá las dimensiones necesarias para contener la información.

Se puede denominar a la salida de la función como $\hat{\boldsymbol{y}} = f(\boldsymbol{x}, \boldsymbol{w})$. A partir de este momento nos referiremos a la salida obtenida de la red de esta forma.

Como vemos en esencia al calcular la norma de la diferencia de vectores obtenemos la distancia euclidiana entre ambos. Podemos visualizar geométricamente cómo en caso de obtener vectores similares esta distancia se acorta y al obtener vectores dispares aumenta.

En la clasificación de imágenes $\hat{\boldsymbol{y}}$ puede ser un vector donde el elemento $\hat{\boldsymbol{y}}_i \in \mathbb{R}$ represente la "confianza" de la red neuronal de que la entrada pertenezca a esta categoría. El elemento $\boldsymbol{y_i} \in \mathbb{R}$ representa la categoría a la que realmente pertenece la imagen. Como debemos saber la categoría a la que pertenece realmente la imagen en realidad solo uno de los elementos de $\boldsymbol{y}$ va a tener el valor máximo y el resto tendrán el valor mínimo ya que estaremos seguros de la categoría de la imagen. Como indicamos en la sección anterior los elementos del dataset deben estar etiquetados.

De esta manera al computar la diferencia entre los valores obtenidos como salida de la red y los valores reales se obtiene un valor numérico que indica cuanto se acerca el resultado a los valores que queríamos obtener. En el caso de que $\hat{\boldsymbol{y}}_i = \boldsymbol{y}_i$ obtendremos un valor de 0 en la ecuación 2.4. De esta forma sabemos que cuando ambos vectores son iguales y el resultado es 0 hemos obtenido el resultado perfecto y cuanto mayor sea este número peor se ha comportado la red. Por tanto lo que trataremos de hacer es minimizar el valor de la función de coste.

El error cuadrático medio no es la única función de coste. Mas adelante introduciremos la entropía cruzada ya que la usaremos durante el desarrollo del trabajo. Existen otras muchas funciones pero solo mencionaremos la divergencia de Kullback-Leibler [28] ya que deriva de la entropía cruzada.

Para minimizar la función de coste usaremos la técnica de descenso de gradiente.

$$\nabla f(x_1, x_2, \ldots, x_n) = \begin{bmatrix} \dfrac{\partial f}{\partial x_1}(x_1, x_2, \ldots, x_n) \\ \dfrac{\partial f}{\partial x_2}(x_1, x_2, \ldots, x_n) \\ \vdots \\ \dfrac{\partial f}{\partial x_n}(x_1, x_2, \ldots, x_n) \end{bmatrix} \tag{4.5}$$

En la ecuación 2.5 se define el gradiente de una función multivariable. Para obtener el vector gradiente de una función debemos obtener las derivadas parciales respecto a cada una de las variables de la misma. Cada una de estas derivadas parciales va a indicarnos si la función que estamos evaluando tiene una pendiente positiva o negativa si modificamos la variable respecto a la que estamos calculando la derivada y mantenemos el resto de variables fijas.

De esta manera si se evalúa $\frac{\partial f}{\partial x_1}$ en los puntos de la entrada de la función $x_1, x_2, ..., x_n$, derivada parcial respecto de la variable x_1, podemos obtener la pendiente del hiperplano tangente a f en el punto determinado por los valores usados respecto de x_1. Si realizamos este calculo para cada una de las dimensiones de entrada, en este caso n, obtendremos la pendiente del hiperplano tangente. En caso de haber tenido solo una variable, es decir $f(x)$, habríamos obtenido la pendiente de la línea tangente. De esta manera vemos como el gradiente es en realidad la generalización del concepto de la derivada para funciones multivariables.

Pero, ¿cual es la utilidad de calcular el gradiente de nuestra función de coste? Ya hemos mencionado que el objetivo es minimizar esta función ya que de esta manera la red neuronal devolverá valores mas cercanos a los deseados.

En primer lugar vamos a comprobar que evaluar el gradiente de la función en los puntos de entrada, recordar que estos valores van a ser los valores que forman el vector w que contiene los pesos, nos va a devolver la dirección en la que la función nos va a proporcionar el valor mas alto si nos dirigimos a esta nueva dirección.

Esto no es exactamente lo que queremos ya que no queremos maximizar la función para obtener valores mas altos, si no minimizarla. Usando la dirección en la que obtendrían el máximo incremento de la función, si nos dirigimos en la dirección opuesta vamos a obtener la dirección en la que se obtiene el máximo decremento. Esto último es lo que nos interesa principalmente para minimizar la función de coste.

Vamos a usar una función de dos variables para ver que el gradiente corresponde a la dirección en la que se obtiene el máximo incremento. Para ellos haremos uso de las derivadas direccionales.

Para una función $f(x, y)$ queremos saber cual es la dirección del vector $\boldsymbol{v}$ que va a suponer un mayor incremento en el resultado de la función. En este caso supondremos que es un vector con longitud 1, y por lo tanto $\|\boldsymbol{v}\| = 1$.

Supongamos que nos encontramos en el punto con coordenadas (x_0, y_0). Si calculamos la derivada direccional $D_{\boldsymbol{v}} f(x_0, y_0)$ obtendremos el cambio que se produce en la función f si moviésemos los puntos en los que evaluamos f en esa dirección.

$$D_{\vec{v}} f(x) = \nabla f(x) \cdot \vec{v} \tag{4.6}$$

La ecuación 2.6 es la forma general de la derivada direccional.En este caso el vector se ha representado de la forma $\vec{v}$ aunque durante este texto se ha usado $\boldsymbol{v}$ para referirnos a vectores. Una vez que sabemos lo que nos indica la derivada direccional vamos a maximizar este valor ya que queremos saber como obtener el máximo incremento en la función.

$$\max_{\boldsymbol{v}} \nabla f(x, y) \cdot \boldsymbol{v} \tag{4.7}$$

En las ecuaciones 2.6 y 2.7 vemos que para calcular la derivada direccional hacemos un producto escalar entre el gradiente de la función y el vector $\boldsymbol{v}$. Como el producto escalar entre dos vectores, recordemos que el gradiente es un vector también, se define como

$$a \cdot b = \|a\|\|b\| \cos\theta \tag{4.8}$$

entonces el valor de la derivada direccional es :

$$D_{\boldsymbol{v}} f(x_0, y_0) = \nabla f(x_0, y_0) \cdot \boldsymbol{v} = \|\nabla f(x_0, y_0)\| \cos\theta \tag{4.9}$$

En la ecuación 2.9 podemos ver que el vector $\boldsymbol{v}$ desaparece ya que como se indicó este tiene longitud 1 y por lo tanto $\|v\| = 1$. Recordemos que estamos maximizando el valor de la ecuación 2.9. En este caso el valor máximo se obtendrá cuando $\cos\theta = 1$, ya que $Rango(\cos(x)) = [-1, 1]$, por lo tanto $\theta = 0$.

De esta forma obtenemos que el valor de la derivada direccional que estábamos calculando,

$$D_{\boldsymbol{v}} f(x_0, y_0) = \|\nabla f(x_0, y_0)\| \tag{4.10}$$

corresponde a la longitud del vector gradiente.

Pero lo que se trataba de obtener no era el valor del cambio que se produce en la función f si no la dirección en la que tendríamos que "mover" los puntos x_0 e y_0 para obtener este valor. En esencia la dirección en la que apunta el vector $\boldsymbol{v}$.

Volvamos a la ecuación 2.9. Como hemos indicado el valor de θ que hace máximo el valor de $D_{\boldsymbol{v}} f(x_0, y_0)$ es $\theta = 0$. Esto nos indica que el ángulo entre el vector $\nabla f(x_0, y_0)$ es 0 y por tanto ambos vectores son paralelos. De esta manera vemos que la forma de obtener el mayor incremento en el resultado de la función es modificar los valores de entrada en la misma dirección que el gradiente de la función. Por lo tanto el vector $\boldsymbol{v}$ debe ser :

$$\boldsymbol{v} = \frac{\nabla f(x_0, y_0)}{\|f(x_0, y_0)\|} \tag{4.11}$$

De esta forma el vector $\boldsymbol{v}$ mantiene la dirección del vector gradiente. La única diferencia es que se encuentra normalizado, es decir, se divide el vector gradiente entre la longitud del mismo. Esto es debido a que ya indicamos que este vector es un vector unitario, por lo tanto se necesita de esta normalización para cumplir la condición impuesta.

Pero hasta este punto averiguado la dirección de un vector v que no indica la dirección hacia la que obtendríamos el mayor cambio en la salida de la función f. Pero en este caso se trata de minimizar la salida de la función de coste. Por lo tanto si regresamos a la ecuación 2.9 vemos que el valor mínimo se obtendría si $\cos\theta = -1$ y para ello es necesario que $\theta = \pi$.

Esto significa que la dirección del vector $\boldsymbol{v}$ será opuesta a la dirección del vector gradiente. Mas en concreto:

$$\boldsymbol{v} = -\frac{\nabla f(x_0, y_0)}{\|f(x_0, y_0)\|} \tag{4.12}$$

De esta forma vemos que los valores de entrada deber ser modificados de tal forma que los nuevos valores se encuentren en la dirección señalada por el vector v y por tanto en dirección contraria al gradiente.

Ahora veamos como deben actualizarse los valores de los pesos dado que ya se ha obtenido la dirección en la que hay que modificarlos. El algoritmo de descenso de gradiente es un algoritmo iterativo. Empezaremos con unos valores iniciales en el vector de pesos $\boldsymbol{w}$ y modificaremos estos pesos en la dirección opuesta al vector gradiente ya que como hemos visto es la modificación que va a resultar en un valor menor en el resultado de la función de coste.

Para realizar una iteración del algoritmo los nuevos valores del vector de pesos serán modificados siguiendo la siguiente ecuación:

$$\boldsymbol{w}^{n+1} = \boldsymbol{w}^n - \eta \nabla_{\boldsymbol{w}} ECM \tag{4.13}$$

En ella se nos indica que la siguiente iteración del vector viene definida por los valores actuales del vector a los que se resta el vector gradiente, ya que esta es la forma que hemos indicado que nos va a llevar al punto mínimo, multiplicado por la constante η.

Esta constante η se denomina tasa de aprendizaje. Como vemos en la ecuación 2.13, al estar multiplicada por el vector gradiente lo que hace es modificar la longitud del vector. Cuanto mayor sea la tasa de aprendizaje mayor será la longitud del vector y por tanto mayor será la diferencia entre los nuevos puntos del vector y los antiguos. Por el contrario si la tasa de aprendizaje es muy pequeña también lo será el cambio.

En muchas de las fuentes consultadas a la tasa de aprendizaje se la denomina con el nombre *step-size* ya que indica el tamaño del "paso" que se va a dar al modificar los valores del vector de pesos.

En la ecuación 2.13 se usa la función de coste del error cuadrático medio pero no es necesario que se use esta función para que se cumpla la igualdad presentada en la ecuación. Como ya se ha indicado hay otras funciones de coste y cualquiera de ellas podría sustituir a la función ECM. Por último se va a introducir un ejemplo gráfico del descenso de gradiente.

En la Figura 2.4 podemos ver la gráfica de una función multivariable. En este caso solo hay dos variables pero en el caso de los vectores de pesos pueden ser millones. En en diagrama aparece también marcado un punto. Este punto será el punto inicial desde el que aplicaremos descenso de gradiente. Para ello como hemos descrito modificaremos los valores del punto en dirección contraria al gradiente. En este caso el gradiente debe apuntar hacia la parte azul de la gráfica de la función ya que esa sería la dirección en la que se obtendrían los valores máximos de z.

Como aplicamos descenso de gradiente la dirección en la que se encontrará el nuevo punto será en el sentido contrario al gradiente. Dependiendo del valor que asignemos a la tasa de aprendizaje obtendremos modificaciones más o menos grandes.

Después de la primera iteración puede que nuestro nuevo punto sea (0,0,0). Aplicando

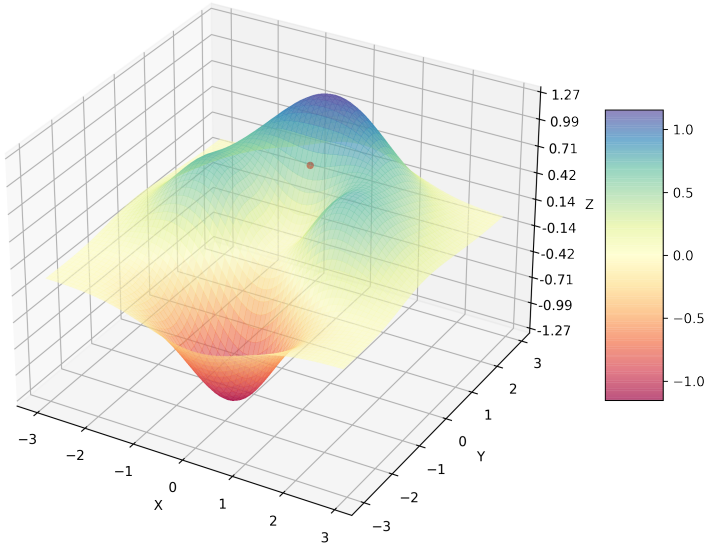


Figura 4.4: Diagrama de la función $z(x, y) = (x^2 + y^3)e^{\frac{x^2+y^2}{2}}$ y el punto (0,1,0.606)

múltiples veces la ecuación 2.13 obtendremos puntos que se acercarán cada vez mas a la zona roja de la función, donde se encuentran los valores mínimos de z

En este punto puede que surjan dudas sobre como se escoge la tasa de aprendizaje ya que solo se ha comentado que describe la longitud del vector que determina la nueva posición del punto.

La elección de una tasa de aprendizaje adecuada puede ser una tarea complicada. En esta sección se van a indicar tres situaciones a tener en cuenta.

1. **Tasa de aprendizaje baja**: Si la tasa de aprendizaje es muy baja el vector, mediante el que determinamos hacia donde y cuanto vamos a modificar los pesos, tendrá una longitud muy corta. Esto significa que con cada actualización de los puntos, podemos usar la Figura 2.4 para visualizarlo, el nuevo punto se va a encontrar bastante cerca del punto inicial. En cada iteración del algoritmos la dirección nos va a llevar hacia los puntos que disminuyen el valor de la función pero al ser modificaciones pequeñas vamos a necesitar muchas de ellas para llegar al mínimo y debemos recordar que el vector de pesos puede tener millones de elementos.

2. **Tasa de aprendizaje elevada**: Si la tasa de aprendizaje es muy alta obtendremos un vector que puede "saltar" la zona de los valores mínimos. Si el punto inicial en la Figura 2.4 se encontrase en la posición (-2,-3, $z(-2, -3)$) el gradiente apuntaría hacia los valores positivos de x, ya que es donde se haría mínima z. Pero si el vector es demasiado largo puede que el nuevo punto tenga un valor x positivo y se salte $x = 0$ donde z tendría valores todavía mas bajos. En este caso también hay que tener en

cuenta que para hallar el nuevo punto no hacen falta tanta iteraciones como en el caso anterior.

3. **Tasa de aprendizaje óptima**: En este caso el valor de la tasa de aprendizaje es lo suficientemente alta como para llegar al resultado en un número razonable de iteraciones y lo suficientemente bajo como para no saltar los valores que minimizarían la función de coste.

Por lo tanto solo disponemos de unas pautas generales sobre como elegir la tasa de aprendizaje pero no unas reglas claras o un algoritmo mediante el que obtener la tasa de aprendizaje óptima. Existen técnicas mediante las que se pueden obtener rangos de valores que pueden tener un buen rendimiento para la red neuronal a la que se aplican. Estas técnicas están basadas en el artículo " Cyclical Learning Rates for Training Neural Networks" Smith(2017) [29]. Estas técnicas están implementadas en fastai, serán usadas durante el desarrollo del proyecto pero no se discutirán más en profundidad de forma teórica.

El descenso de gradiente parece que nos proporciona las herramientas necesarias para resolver el problema que habíamos planteado al principio de la sección. Un método que nos permita minimizar la función de coste que nos indica cuanto error está cometiendo la red neuronal. Hasta este punto parece que el descenso de gradiente cumple con ello. Pero hay una cuestión sobre la que no se ha mencionado nada y esto es cuanto poder de computación vamos a requerir para hacer estos cálculos.

Si revisamos el descenso de gradiente, mas en concreto la ecuación 2.4, podemos ver la forma en la que obtenemos el error es iterando sobre todos los elementos del dataset. Cada uno de estos elementos $\boldsymbol{x}$ nos proporciona el error que produce la red al clasificarlo. Para calcular el gradiente vamos a tener que calcular todas las derivadas parciales cuyo numero depende del número de entradas de la función.

En muchas ocasiones, sobre todo en Deep Learning debido al uso de redes profundas, va a ser necesaria una cantidad elevada de datos y para obtener el nuevo punto que indica el gradiente es necesario calcular el coste sobre estos datos muchas veces.

De esta forma se introduce el descenso de gradiente estocástico (en inglés, stochastic gradient descent (SGD)). Este algoritmo cumple con la misma función que el descenso de gradiente normal excepto que se usará un número mucho menor de entradas.

En primer lugar se seleccionará un subconjunto de entradas $A = \{x^{(0)}, x^{(1)}, ..., x^{(m)}\}$ donde $m < n$. A este subconjunto se le denomina minibatch. En caso de que $n = m$ estaríamos aplicando la versión normal del descenso de gradiente.

De esta forma los nuevos puntos se generarán respetando la siguiente ecuación:

$$\boldsymbol{w}^{(i+1)} = \boldsymbol{w}^{i} - \eta \nabla_{\boldsymbol{w}} \frac{1}{m} \sum_{x} \|f(\boldsymbol{x}, \boldsymbol{w}) - \boldsymbol{y}\|^2 \tag{4.14}$$

La ecuación 2.14 es muy similar a la ecuación 2.13. Es la misma ecuación en esencia solo que el número de elementos que se disponen para calcular el coste es menor y por tanto se

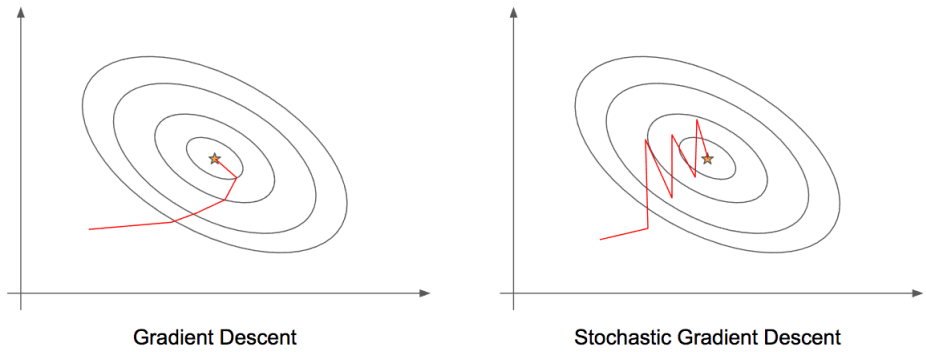


Figura 4.5: Descenso de gradiente (*izq*) y descenso de gradiente estocástico (*dcha*). Fuente: [30]

pierde precisión a la hora de determinar el nuevo punto. Pero esta pérdida de precisión no es relevante mientras nos estemos dirigiendo hacia el mínimo de la función.

En la figura 2.5 se puede ver como el descenso de gradiente sigue una ruta más directa que la variante estocástica pero para cada nuevo punto el descenso de gradiente normal debe evaluar todos los elementos de entrada. Evaluar todas las entradas resulta en una mayor precisión pero esta precisión no es indispensable si podemos obtener una aproximación lo suficientemente buena evaluando muchas menos entradas, que es justo lo que nos proporciona el descenso de gradiente estocástico.

4.1.4. Propagación hacia atrás

Una vez comprendido el descenso de gradiente y su versión estocástica (SGD) podemos continuar con el algoritmo que probablemente sea el más importante a la hora de construir redes neuronales artificiales.

En la sección anterior hemos visto como minimizar el error que comete la red neuronal dados los pesos y las entradas de la red. No debemos olvidar que mediante el descenso de gradiente actualizamos los pesos mientras que podemos considerar las entradas de la red constantes ya que el dataset sobre el que estamos entrenando es siempre el mismo.

Si volvemos a la Figura 2.3 recordamos que la red neuronal usada como ejemplo tiene una capa de entrada, múltiples capas intermedias y una capa de salida en la que obtenemos el resultado. Una vez obtenido el resultado podemos usar la ecuación 2.4 para calcular el coste de la red y usar descenso de gradiente para normal o estocástico para minimizar el resultado actualizando los valores del vector de pesos.

Para llevar a cabo la actualización de los pesos debemos calcular el gradiente de la ecuación de coste respecto de los parámetros de la red, como recordamos de la ecuación 2.13 si se está aplicando la versión normal del descenso de gradiente o la ecuación 2.14 si se está aplicando SGD.

Debido a la gran cantidad de derivadas que se van a tener que realizar debemos tener un método eficiente para realizar estos cálculos. Para esta tarea no podemos obtener las derivadas parciales de forma simbólica debido a la ineficiencia de esta técnica si es usada por programas informáticos. Para un ejemplo de esta ineficiencia se puede consultar la siguiente fuente [31].

En cuanto a técnicas de diferenciación podemos distinguir tres categorías:

1. **Diferenciación numérica:** Usando la definición de la derivada $f'(x) = \lim_{h\to 0} \frac{f(x+h)-f(x)}{h}$ se obtiene el valor en el punto aunque requiere una gran cantidad de cómputo para funciones multivariables.

2. **Diferenciación simbólica:** En este caso las derivadas se obtienen obteniendo primero la fórmula de la derivada y a continuación sustituyendo los valores. Esta es la forma habitual en la que pensamos cuando se requiere derivar una expresión.

3. **Diferenciación automática:** Se divide la función que se va a derivar en bloques que nos permitirán calcular de forma numérica, no simbólica, variables intermedias que podremos usar para aplicar la regla de la cadena. Este proceso se puede visualizar de forma sencilla en grafos de computación, que es en esencia un grafo dirigido donde los nodos son operaciones o variables.

La ventaja que ofrece la diferenciación automática es que esta forma de calcular las derivadas es mucho mas eficiente que los otros dos métodos. El algoritmo de propagación hacia atrás (en inglés, backpropagation), es en realidad un caso especial de diferenciación automática y por tanto a veces se referencia a este algoritmo como *autodiff*.

Uno de los componentes principales de la propagación hacia atrás es la regla de la cadena que para funciones multivariables tiene la siguiente forma.

Si suponemos una función diferenciable de la forma $w = f(x, y)$, donde $x = x(t)$ e $y = y(t)$ son también diferenciables

$$\frac{dw}{dt} = \frac{\partial f}{\partial x}\frac{dx}{dt} + \frac{\partial f}{\partial y}\frac{dy}{dt} \tag{4.15}$$

nos da la derivada de la función respecto de la variable t.

En este caso se ha usado una función de dos variables para evitar la complejidad adicional de la notación de la forma general [32].

Usemos un ejemplo sencillo para ver como la regla de la cadena puede ser usada. Supongamos una función de coste $y = (4x - 2)^2$. Esta expresión se asemeja bastante a la ecuación 2.4. Para la función que acabamos de introducir la variable x tiene el papel que tendrían los vectores $\boldsymbol{w}$ y $\boldsymbol{x}$ donde se introducen los pesos y las entradas a la red. A este valor se le resta otro que en este caso es simplemente una constante y elevamos todo ello al cuadrado para evitar valores negativos.

Usemos la regla de la cadena para calcular $\frac{\partial y}{\partial x}$. Para ello primero vamos a separar en varias funciones intermedias la función principal de forma que $f = 4x$, $g = f - 2$ e $y = g^2$.

Calculando la derivada parcial de cada función intermedia, $\frac{\partial f}{\partial x} = 4$, $\frac{\partial g}{\partial f} = 1$ y $\frac{\partial y}{\partial g} = 2g$, resulta sencillo obtener la derivada $\frac{dy}{dx}$.

Si aplicamos la regla de la cadena obtenemos que $\frac{dy}{dx} = \frac{\partial y}{\partial g}\frac{\partial g}{\partial f}\frac{\partial f}{\partial x} = 8(4x - 2)$.

Es muy sencillo imaginar como se vería el grafo computacional de esta función, donde se realizan varias operaciones sobre x que sería considerado el predecesor de y [33]. Veamos como hace uso el algoritmo de propagación hacia atrás de la regla de la cadena.

En primer lugar debemos saber qué es lo que estamos propagando. Al realizar el primer cálculo del error que está cometiendo la red se ha hecho lo que en términos del algoritmo de propagación se denomina la propagación hacia adelante ya que se han propagado las entradas hasta obtener los valores de salida. Una vez calculado el error debemos usarlo para modificar los pesos y *bias* de cada una de las capas anteriores para intentar lograr un error menor en el siguiente cálculo.

Pero, ¿como se realiza la propagación hacia adelante si los pesos vienen determinados por el error obtenido al final de esta fase? Para esta explicación supondremos que los pesos y *bias* se inicializan con valores aleatorios pero existen técnicas para obtener valores que van a necesitar menos iteraciones para producir el mismo resultado. Algunas de estas técnicas son inicialización Xavier [34], incializazacón He [35], etc.

Si pensamos sobre lo que ocurre al hacer la propagación hacia adelante vemos como se calcula la entrada de una neurona en la capa l, este valor sirve como entrada a la función de activación y la salida de la neurona en la capa l permite calcular la entrada de la neurona en la capa $l + 1$. Este comportamiento se puede expresar fácilmente mediante composición de funciones.

En la sección anterior se ha usado ECM como función de coste pero como ya hemos indicado puede ser otra función que cumpla ciertos requisitos por lo tanto para simplificar la notación en esta sección usaremos C para referirnos a la función de coste.

Situémonos en la capa de salida de la red. Como el objetivo es disminuir el error obtenido debemos comprobar como afecta a cada uno de los pesos y bias a este error con la idea de que si introducimos un pequeño cambio Δw esto afectará a la salida de la neurona que propagará hacia delante este cambio hasta modificar ligeramente el error total.

Recordemos que z es el resultado de la suma ponderada de la ecuación 4.1 y al aplicar una función de activación sobre este valor obtenemos la salida de la neurona. Llamaremos a a este valor donde si usamos la función de activación sigmoidal $a = \sigma(z)$.

De esta manera lo que hemos formulado hasta ahora de forma escrita se traduce en la siguiente ecuación:

$$\frac{\partial C}{\partial w_{ji}^l} = \frac{\partial C}{\partial a_j^l} \frac{\partial a_j^l}{\partial z_j^l} \frac{\partial z_j^l}{\partial w_{ji}^l} \tag{4.16}$$

En la ecuación 4.16 vemos la dependencia entre el coste total y uno de los pesos conectados desde la neurona i en la capa $l-1$ a la neurona j de la capa l.

Una alternativa ampliamente usada para obtener una notación mas compacta es introducir el "error en la neurona". De este modo solo se evaluará cómo depende el coste total de la salida de la neurona y cómo depende esta salida de la entrada. En la ecuación 4.16 ya usamos este error aunque lo relacionamos también con el peso entrante. Por lo tanto este error se define por la siguiente ecuación:

$$\delta_j^l = \frac{\partial C}{\partial a_j^l} \frac{\partial a_j^l}{\partial z_j^l} \tag{4.17}$$

Usando la ecuación 4.17 podemos volver a 4.16 y realizar la sustitución. En este punto es recomendable prestar atención al último elemento de la ecuación 4.16. Vemos que relaciona la entrada de la neurona con uno de los pesos. Pero esta misma ecuación sería válida para los bias, solo que se debe tener cuidado con algunos aspectos.

Como vimos en la ecuación 4.1 cada peso se multiplica por un valor x que en la primera capa es obtenido de la entrada de la red y en cada capa posterior l es obtenido de las salidas de las neuronas en la capa $l-1$, a cuyo valor hemos designado la variable a. Por lo tanto ahora la entrada de cada neurona tendrá la siguiente forma:

$$z_j^l = \sum_i a_i^{l-1} w_{ji}^l + bias^l \tag{4.18}$$

Si ahora realizamos la derivada de la ecuación 4.18 respecto de w, $\frac{\partial z_j^l}{\partial w_{ji}}$ obtendremos la activación de la neurona correspondiente en la capa $l-1$, a_i^{l-1}. Pero si realizamos la derivada respecto del bias obtenemos 1.

Mediante un poco de manipulación algebraica podemos ver como calcular la derivada parcial del coste respecto de los pesos y bias en función del error de la neurona, δ.

$$\frac{\partial C}{\partial w_{ji}^l} = \delta^l \frac{\partial z_j^l}{\partial w_{ji}^l} = \delta^l a_j^{l-1} \tag{4.19}$$

$$\frac{\partial C}{\partial b^l} = \delta^l \frac{\partial z_j^l}{\partial b^l} = \delta^l \tag{4.20}$$

Mediante las ecuaciones 4.19 y 4.20 podemos calcular el efecto que está teniendo un peso o bias de una neurona en el coste total si conocemos el error en esa neurona. Mediante

la ecuación 4.17 podemos calcular el error de la neurona que necesitamos para aplicar la ecuaciones 4.19 y 4.20. Pero hay una cuestión sin resolver.

Tal y como hemos definido el coste en la neurona vemos que se aplica la regla de la cadena y que la primera derivada parcial relaciona el coste total con la activación, por lo tanto esta ecuación solo será válida para la última capa de la red. El coste total solo depende directamente de la activación de las neuronas en la última capa pero depende indirectamente de todas las activaciones en las capas anteriores.

Por lo tanto si podemos obtener el error en una neurona en función del error de la neurona correspondiente de la siguiente capa tendremos todas las herramientas necesarias para realizar la propagación hacia atrás.

De nuevo podemos hacer uso de la regla de la cadena. Introduzcamos otra capa $l+1$ a continuación de la capa final l. Este cambio afectaría a la forma de calcular el error en una neurona de la capa l. Veamos como debería ser reformulada la ecuación 4.17 si se produjese este cambio.

$$\delta_j^l = \sum_i \frac{\partial C}{\partial z_i^{l+1}} \frac{\partial z_i^{l+1}}{\partial z_j^l} \tag{4.21}$$

El cambio de la capa adicional queda reflejado en la ecuación 4.21. Como queremos ver cómo afecta la entrada de una neurona al coste total, debemos tener en cuenta que la neurona de la capa l está conectada con todas las neuronas de la capa $l+1$.Por lo tanto el coste total depende de las entradas de todas las neuronas de la capa $l+1$ ya que todas ellas dependen de la entrada de la neurona en la capa l.

Si ahora modificamos la ecuación 4.18 para representar la entrada de la capa $l+1$ (es la misma ecuación, solo se deben ajustar los índices) y realizamos la derivada respecto de la entrada de la neurona j en la capa l, $\frac{\partial z_i^{l+1}}{\partial z_j^l}$ obtendremos la tasa de cambio de la neurona j si modificamos la entrada de la neurona i. Podemos hacer esta derivada porque la activación depende de la entrada de esa capa como vimos en 4.3.

Al modificar la ecuación 4.18 para representar las entradas de la capa $l+1$ vemos que ahora la activación que multiplicará a los pesos deberá ser la de la capa l.

De esta forma sustituyendo esta derivada, $\frac{\partial z_i^{l+1}}{\partial z_j^l}$, en la ecuación 4.21 obtenemos:

$$\delta_j^l = \sum_i \frac{\partial C}{\partial z_i^{l+1}} w_{ji}^{l+1} \frac{\partial a_j^l}{\partial z_j^l} \tag{4.22}$$

En esta ecuación 4.22 podemos ver que la primera derivada parcial en realidad corresponde con lo que hemos llamado el error en la neurona. La única diferencia es que la neurona se encuentra en la capa $l+1$. Por lo tanto se podría sustituir esta derivada por δ_i^{l+1}.

Si nos fijamos en la ecuación 4.22, lo que nos indica en realidad es bastante intuitivo. El error en una neurona de la capa l viene determinado por la suma de errores de la capa $l+1$ multiplicados por los pesos que unen cada neurona de la capa $l+1$ con la neurona correspondiente de la capa l, de la misma forma que al realizar la propagación hacia delante estábamos multiplicando las activaciones de las neuronas por los pesos para moverlos a la siguiente capa.A su vez este valor es multiplicado por la derivada de la activación de la capa l con respecto de la entrada z de esa misma capa y mediante la regla de la cadena nos indica la forma en la que esta entrada modifica la activación.

Por lo tanto podemos ver un flujo muy natural, el coste total es relacionado con el error de la neurona de la capa $l+1$, este error es propagado hacia atrás mediante los pesos que unen la capa con la anterior,la capa l, y este valor es relacionado con la entrada de la neurona de la capa l. Esta es la ecuación que mejor expresa la noción de "propagación hacia atrás" de los errores que da el nombre al algoritmo.

De este modo hemos obtenido todas las ecuaciones necesarias para modificar todos los pesos y bias de la red neuronal. Veamos como sería este proceso. Mediante la ecuación 4.17 calculamos el error en la neurona para cada una de las neuronas de la capa final. Una vez obtenidos estos errores usaremos las ecuaciones 4.19 y 4.20 para calcular las derivadas del coste total respecto a los pesos y bias de la capa l. Estos valores serán sustituidos en la ecuación 4.14 de forma que el peso será actualizado a uno que reducirá en una pequeña cantidad el coste total. Lo mismo ocurrirá con los bias.

Una vez actualizados los valores usaremos la ecuación 4.22 para calcular los errores de las neuronas de la penúltima capa. Para ello usaremos el error en las neuronas de la capa l que ahora pasará a llamarse capa $l+1$ y la penúltima capa es la que ahora consideraremos como capa l, la capa en la que nos estamos situando.

Obtenidos los errores de la capa l, ahora la penúltima capa, podemos volver a calcular las derivadas de los pesos y bias mediante 4.19 y 4.20 y actualizar estos valores mediante 4.14. Repitiendo este bucle iremos actualizando los valores en cada capa hasta llegar a la primera. Cuando lleguemos a este punto todos los pesos y bias de la red habrán sido ya actualizados y habremos terminado el proceso.

Como vemos es un algoritmo simple que puede volverse complicado por la notación al introducir distintas capas pero donde cada ecuación es bastante intuitiva.

4.1.5. Overfitting y underfitting

Una vez que se ha realizado el entrenamiento, aplicando descenso de gradiente y realizando propagación hacia atrás, y se ha conseguido una pérdida que se considera aceptable sobre el conjunto de entrenamiento es el momento de comprobar como se comporta el modelo frente al conjunto de validación. Este es el punto crítico donde se determina si funciona adecuadamente ya que de nada sirve obtener un coste muy bajo en el entrenamiento si cuando exponemos al modelo a nuevas entradas este no es capaz de clasificarlas correctamente.

A esta capacidad de obtener buenos resultados sobre entradas que el modelo no ha "visto"

(no ha modificado sus pesos y bias para esta entrada en particular) se llama generalización. Ya se ha mencionado este concepto y en esta sección se profundizará en los motivos por los que es tan importante.

El sobreaprendizaje (en inglés, overfitting) se produce cuando al entrenar el modelo sobre el conjunto de entrenamiento se modifican los pesos y bias de tal manera que permiten clasificar casi a la perfección cualquier elemento de este conjunto pero a su vez obtiene resultados peores cuando una vez entrenado se enfrenta al conjunto de validación.

Imaginemos un ejemplo simple. Queremos crear un modelo que clasifique correctamente imágenes que contienen un gato o un perro. Es posible que después de entrenar un modelo así y probarlo con imágenes del conjunto de validación obtengamos resultados donde un gato con ojos verdes sea incorrectamente clasificado. Esto puede resultar extraño ya que probablemente al ver la imagen un humano reconozca enseguida que se trata de un gato.

Lo que ocurre en este caso es que se ha producido overfitting. El modelo se ha ajustado tanto a las imágenes del conjunto de entrenamiento que una pequeña variación como el color de los ojos puede producir este error aunque la anatomía del gato sea claramente reconocible.

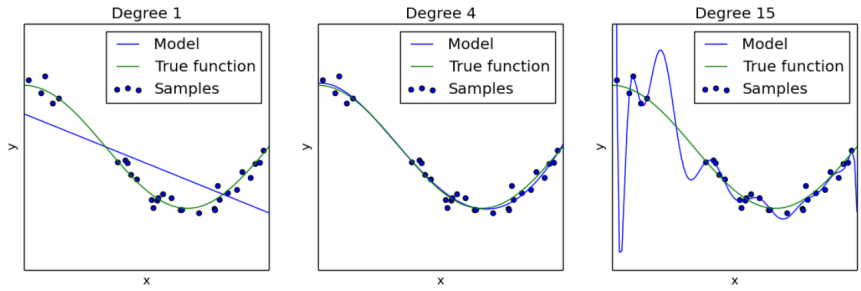


Figura 4.6: Ejemplos de underfitting (*izda*), ajuste correcto y overfitting (*dcha*). Fuente: [36]

En el ejemplo presentado en la Figura 4.6 se está resolviendo un problema de regresión. En este tipo de problemas se desea predecir un valor numérico dado un valor de entrada. El sobreajuste se representa en la figura de la derecha donde el modelo está usando un polinomio de orden quince. Las predicciones sobre los datos existentes serán correctas, como vemos el polinomio se ajusta a todos los puntos pero si se obtiene la predicción del modelo para un valor de x muy bajo obtendremos un valor de y también muy bajo que no sigue la tendencia y no se ajusta a la función real que siguen estos valores.

En el caso del ajuste correcto, la imagen central de la Figura 4.6, hay puntos para los que el modelo no devuelve el resultado correcto, pero devuelve un valor lo suficientemente cercano. Este comportamiento es exactamente el que queremos emular. Los valores que predice el modelo siguen claramente la tendencia de los puntos y por tanto se ajustan a la función real.

Por último podemos tener el caso contrario al overfitting, el underfitting. En este caso la red neuronal no se ha entrenado lo suficiente y por tanto el modelo no ha tenido las iteraciones necesarias para modificar los pesos y bias para hacer un buen ajuste.

En el ejemplo de la Figura 4.6, en la imagen de la izquierda, vemos como el modelo está

usando una recta para predecir los resultados. En este caso el modelo es demasiado simple y no sigue la tendencia que marcan los puntos representados. En este caso podemos decir que se está generalizando en exceso. Como vemos la recta tiene cierta pendiente, no es un modelo totalmente ilógico y probablemente sea la recta que mejores predicciones pueda dar pero para ajustarse realmente a la función el modelo debe comportase como un polinomio de orden superior.

Para detectar si se están produciendo estos efectos al entrenar el modelo debemos prestar atención a la tasa de error sobre el conjunto de entrenamiento y sobre el conjunto de validación. Se pueden dar varias situaciones:

- Si ambas métricas están bajando es que el modelo está aprendiendo y si paramos el entrenamiento en este punto podemos obtener un modelo que sufra de underfitting.
- Si la tasa de error en el conjunto de entrenamiento está bajando pero la tasa de error sobre el conjunto de validación está subiendo es que se está produciendo overfitting, se están clasificando mejor las entradas con las que se realiza el aprendizaje pero la generalización será peor.
- Si ambas métricas se mantienen o aumentan es que el modelo no está aprendiendo o directamente tiene un comportamiento peor que si no se hubiese entrenado. Esta situación suele ser indicativo de un error en el modelo.

4.1.6. Regularización

En la sección anterior se ha visto en qué consisten el overfitting y el underfitting y cómo detectar estos problemas. La solución al problema del overfitting es clara, se debe continuar el entrenamiento de la red. Pero la solución al overfitting no es tan sencilla.

Existen estrategias que nos permiten reducir el error que obtenemos en el conjunto de validación pero con la intención de no aumentar el error en el conjunto de entrenamiento (con la intención debido a que no siempre es posible). A estas técnicas se las denomina regularización y son la respuesta al problema del sobreaprendizaje.

Estas técnicas se dividen en distintas categorías ya que existen distintas formas de atacar el problema. La categoría más habitualmente usada es la regularización mediante modificación de la función de coste.

Existen múltiples formas de modificar la función de coste para reducir el overfitting pero casi todas siguen la misma estrategia general. En este sección continuaremos usando C para referirnos a la función de coste.

$$C'(\boldsymbol{x}, \boldsymbol{w}, \boldsymbol{y}) = C(\boldsymbol{x}, \boldsymbol{w}, \boldsymbol{y}) + \lambda R(\boldsymbol{w}) \tag{4.23}$$

Como vemos en la ecuación 4.23 la función de coste regularizada, C', proviene del coste sin regularizar, C, al que se le aplica una penalización $\lambda R(\boldsymbol{w})$.

En esta penalización el valor λ indica la importancia que le estamos dando a la penalización, de una forma muy similar a cómo funciona la tasa de aprendizaje en la ecuación 4.13.

Por último se aplica una función a los pesos de la red. Esta es la función que suele cambiar para las distintas variantes de este tipo de penalizaciones. A continuación veremos algunas de ellas.

- **Regularización L2**: También conocida bajo el nombre de regresión de arista o decaimiento de pesos. En este caso la ecuación de penalización tiene la siguiente forma:

$$R(\boldsymbol{w}) = \frac{1}{2}\|\boldsymbol{w}\|_2^2 \tag{4.24}$$

  El nombre, L2, proviene de que se está aplicando la norma L2 sobre el vector de pesos. Si recordamos esta norma nos permite obtener la longitud del vector en el espacio euclídeo. El propósito de la constante $\frac{1}{2}$ es eliminar la constante 2 que obtendremos al realizar la derivada de la norma que será necesario al calcular el gradiente de la función de coste.

  Pero, ¿cómo puede ayudarnos esta penalización a reducir el overfitting? La respuesta está en que la ecuación 4.23 se sustituye en la ecuación mediante la que se actualizan los pesos, 4.13. Al realizar esta sustitución vemos que el signo negativo de la ecuación 4.13 hace que la penalización también sea negativa. Con cada nueva actualización del peso su valor tenderá a 0 ya que solo se está restando.

  La única forma de que un peso no termine siendo 0 es que función de coste dependa mucho de este peso, que averiguaremos al realizar el gradiente de la función de coste. De esta forma solo los pesos que realmente deben tener valores elevados los mantienen y el resto de pesos terminan teniendo valores bajos debido a la penalización y esto permite "reducir la complejidad" de la solución que está dando nuestro modelo.

  Si volvemos a la figura aplicando regularización estaríamos forzando al modelo a obtener resultados similares a los obtenidos por un polinomio de orden bajo.

- **Regularización L1**: En este caso de nuevo el nombre de la técnica nos indica cómo se va a calcular la penalización. Usando la norma L1 obtenemos la siguiente ecuación:

$$R(\boldsymbol{w}) = \|\boldsymbol{w}\|_1 \tag{4.25}$$

  En este caso el cálculo de la penalización es incluso más sencillo. Lo que debemos tener en cuenta es que cuando se realiza el cálculo del gradiente y se deriva la penalización respecto de cada peso obtendremos $\lambda signo(w_k)$ ya que la norma L1 es la suma de todos los pesos. Por lo tanto en cada iteración el peso va a tener una penalización fija $|\lambda|$ que se sumará o restará en función del signo del peso.

  La razón por la que este tipo de regularización ayuda con el problema del overfitting es similar a la regularización L2, con la diferencia de que la regularización L1 produce una solución más dispersa, es decir, un mayor número de pesos terminan valiendo cero mientras que en la L2 los pesos tienden a reducirse pero mantienen valores pequeños distintos de cero.

Este comportamiento se produce debido a que como la penalización en la regularización L1 es un valor constante el nuevo peso tenderá a cero más despacio si es un valor grande y más rápido si es un valor menor.

En estas técnicas los cambios que se realizan son sobre la función de coste y por tanto afectan directamente a los cálculos del modelo. Existe otra categoría de técnicas de regularización que reduce el overfitting modificando la estructura o el tiempo de aprendizaje. Estas son las principales:

- **Dropout**: Esta técnica asigna a cada neurona de la red una probabilidad p de mantenerse activada. Habitualmente se selecciona $p = 0{,}5$ y se recorre la red donde cada neurona queda activada con esa probabilidad y queda desactivada con una probabilidad de $1 - p$. Si realizamos el proceso con estos valores aproximadamente la mitad de neuronas quedarán desactivadas. Cuando una neurona quede desactivada los pesos de entrada y salida también quedarán anulados. Las únicas capas que no pueden ser modificadas son la de entrada y la de salida. Una vez modificada la estructura de la red se realiza propagación hacia delante y hacia atrás.

  Después de haber realizado la iteración se vuelven a activar todas las neuronas y se vuelven a desactivar la mitad. Las que se han desactivado en esta nueva iteración no tiene por que ser las mismas que en la anterior, de hecho lo mas habitual es que no lo sean. Este bucle se repita hasta completar el entrenamiento.

  De esta forma el overfitting se reduce ya que en cada iteración se están entrenando neuronas distintas y por tanto pesos distintos. Al no entrenarlos de forma continua resulta más complicado para el modelo adaptarse excesivamente bien al conjunto de entrenamiento y que se produzca sobreaprendizaje.

- **Early stopping**: Esta técnica se basa en detener el entrenamiento antes de que haya terminado. Como se indicó en la sección anterior el overfitting se puede detectar si el error sobre el conjunto de entrenamiento continua bajando pero el error sobre el conjunto de validación empieza a aumentar después de haber bajado.

  Si detenemos el aprendizaje justo en el momento en el que se detecta que el error sobre el conjunto de validación está aumentando obtendremos la versión del modelo que mejor se comporta. Pero esto no siempre es tan sencillo, muchas veces los valores tendrán cambios erráticos y el error del conjunto de validación subirá para luego volver a bajar.

  Es difícil prever si los valores seguirán la tendencia que detectamos. Solo podremos estar seguros de ello cuando veamos los valores del error en los distintos conjuntos una vez haya terminado el entrenamiento y sepamos como se habría comportado el error de validación. Pero si esperamos a que termine el entrenamiento se pierde el sentido de usar esta técnica.

4.2. Redes neuronales convolucionales

Las redes neuronales convolucionales son un tipo especial de redes neuronales que funcionan particularmente bien con distintos tipos de señales. En el campo del procesamiento de la señal se define a esta como una función que expresa cierta información sobre un fenómeno. Esta definición es bastante general pero se aclarará con un ejemplo.

Para este proyecto nos centraremos en un ejemplo en particular, las imágenes. Pero ¿que tienen que ver la imágenes con las señales? Bastante en realidad. Una imagen puede ser interpretada como una función $f(x, y)$ donde los valores de x e y representan las coordenadas de cada píxel y la función devuelve la intensidad del píxel en estas coordenadas.

La mayoría de imágenes que vemos a diario no son monocromáticas, es decir, no están compuestas por solo un tono. Por tanto cada píxel de estas imágenes con múltiples colores no puede estar definido por solo una intensidad o brillo. Existen diferentes modelos de representación para los colores que nos permiten representarlos de forma numérica.

El más habitualmente usado es el modelo RGB donde cada píxel tiene asociados tres valores numéricos. Cada uno indica la intensidad del pixel para uno de los colores primarios, rojo (**R**ed), verde (**G**reen) y azul (**B**lue).

De esta forma podemos pensar en cada color primario como una matriz $\boldsymbol{A} \in \mathbb{R}^{m \times n}$ donde el elemento $\boldsymbol{A}_{ij}$ representa la intensidad del píxel.

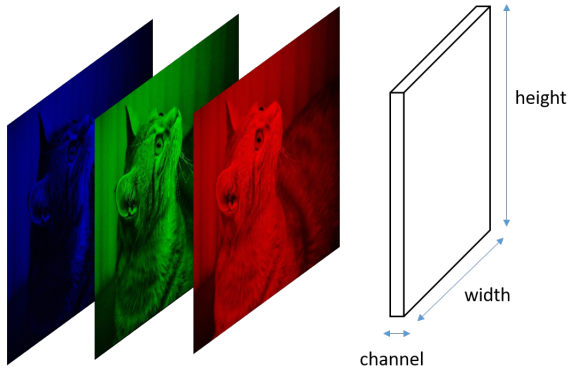


Figura 4.7: Composición de una imagen RGB mediante los tres colores primarios. Fuente: [37]

Esta descripción se asemeja bastante a la descripción de señal que hemos dado al principio de la sección. Cada una de las matrices representa la señal de cada color primario de la imagen. En la Figura 4.7 podemos ver que la imagen final se forma mediante la composición de las tres señales, también llamadas canales.

Una forma en la que se puede incorporar toda esta información en una entidad algebraica es mediante los tensores [38]. Un tensor es la generalización de los vectores y las matrices. Esta es una definición muy simplificada pero es suficiente para este contexto.

El tensor generaliza el concepto de matriz de forma que podemos añadir una tercera dimensión (o más). De esta forma un tensor de dimensión 3 estará formado por múltiples matrices de la misma forma que una matriz está formada por múltiples vectores y un vector por múltiples elementos individuales.

Figura 4.8: Ejemplos de tensores con distintas dimensiones. Fuente: [39]

En la Figura 4.8 se puede observar cómo cada elemento se obtiene al expandir en una dimensión el el elemento anterior. De esta forma ahora cada imagen pasará a ser representada por un tensor de dimensión 3 donde un elemento individual, la intensidad de un píxel en un canal concreto, será referenciado por $\mathtt{A}_{i,j,k}$. De esta forma el canal viene determinado por el elemento k mientras que i, j determinan las coordenadas sobre esa matriz en particular.

Puede parecer que introduciendo todos estos elementos se está complicando más de lo necesario el tratamiento de las imágenes. Pero debemos recordar que cada imagen será una entrada en la red neuronal y debemos disponer de una forma adecuada para introducir la información presente en la imagen y poder computar el resultado.

Una vez vista la forma en la que se van a tratar la imágenes en una red convolucional veamos los cambios que introducen este tipo de redes respecto del tipo de red que hemos visto a lo largo de este capítulo.

El nombre obviamente deriva de la operación de convolución. En la siguiente sección la describiremos más en detalle pero de momento veamos como es la arquitectura de estas redes.

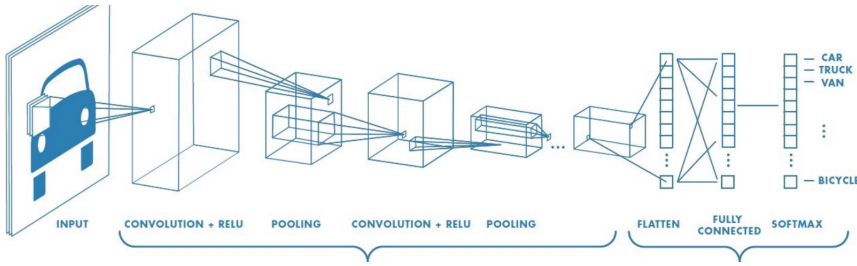


Figura 4.9: Arquitectura de una red neuronal convolucional. Fuente: [40]

En la Figura 4.9 vemos un ejemplo de una red convolucional. La entrada es el tensor de la imagen con sus respectivos canales. La matriz de cada canal es introducida como una entrada

separada en la red donde sobre cada matriz se realizarán convoluciones mediante las que se extraen *features* de la imagen. A medida que se avanza en estos niveles y se van aplicando más convoluciones y otras operaciones como el *pooling* que describiremos mas adelante se obtienen características de más alto nivel.¿Que significa esto?

Podemos pensar en las característica de bajo nivel como los elementos mas básicos de detección como bordes o esquinas de objetos y a medida que se avanza en la red se obtienen características más abstractas a partir de las cuales se hace la decisión sobre la categoría a la que pertenece la imagen. Algunos ejemplos interesantes de las características que detecta una red convolucional pueden ser encontrados en "Visualizing and Understanding Convolutional Networks" [41].

Una vez obtenidas unas características lo suficientemente abstractas se realiza la operación de *flattening*. De esta se reduce en una dimensión la matriz que ha resultado de las convoluciones y se obtiene un vector con toda la información de la matriz. Este vector es el que servirá de entrada a la segunda parte de la red convolucional que es una red con las capas conectadas al igual que en la Figura 4.3.

4.2.1. Convoluciones

En la sección anterior se ha mencionado que las convoluciones son usadas para extraer *features* o características de una imagen. Este es un proceso que en este contexto se puede entender mejor de forma visual aunque las explicaciones se apoyarán en sus respectivas ecuaciones. Las *features* son el resultado de las convoluciones por tanto veamos primero como funciona la operación de convolución.

$$s(t) = \int_{-\infty}^{+\infty} f(a)w(t-a)da \tag{4.26}$$

En la ecuación 4.26 se puede observar la formula para una variable continua. Debido a que habitualmente se usa la versión para variables discretas en este contexto nos centraremos en esta última.

Veamos mediante un ejemplo lo que indica la ecuación 4.26. Imaginemos que la función $f(x)$ nos devuelve la distancia en metros desde la salida en el instante x de un coche de carreras. Este resultado se obtiene mediante un láser pero al comprobar las mediciones vemos que hay cierto ruido en ellas, otra señal está interfiriendo con el láser. Para minimizar el efecto de esta segunda señal se aplica una convolución a las mediciones.

Mediante la convolución se dará mas peso a las mediciones recientes que a las anteriores. De esta forma si queremos obtener la posición del coche en el instante t mediante la convolución,$s(t)$, se obtiene la posición en el momento a, donde a indica el tiempo transcurrido desde que se ha tomado la medición hasta el momento t y $w(t-a)$ indica el peso de la medición. Si nos encontramos en el momento $t = 3$ y una medición se ha efectuado en el instante $t = 0$ (el tiempo transcurrido desde que se hizo la medición es 3) obtendremos un peso bajo $w(3-3) = w(0)$ para esta.

Si la medición se ha hecho en el instante $t = 3$ entonces el peso será el máximo ya que $w(3-0) = w(3)$ tal y como indicamos al principio de la explicación.

Para pasar a variables discretas y acercarnos más a una formulación de las convoluciones adecuada para el tratamiento de imágenes debemos introducir primero el concepto de kernel. Podemos pensar en este como el elemento que determina la importancia de cada valor y en la ecuación 4.26 la función $w(t-a)$ es el *kernel*.

En el tratamiento de imágenes, al representarlas como matrices, la función f de la ecuación 4.26 determinará la intensidad del píxel en unas coordenadas x, y. El kernel por otro lado será también otra matriz, donde los valores de este determinarán los pesos por los que serán multiplicados los píxeles de la imagen y por tanto el resultado de la convolución para cada píxel de la imagen.

$$S(i,j) = (K * I)(i,j) = \sum_m \sum_n I(i-m, j-n) K(m,n) \tag{4.27}$$

En la ecuación 4.27 se presenta el procedimiento descrito. Al aplicar una convolución sobre un pixel de una imagen, $S(i,j)$ se realiza la multiplicación entre los valores correspondientes en la matriz de la imagen I y el *kernel* K.

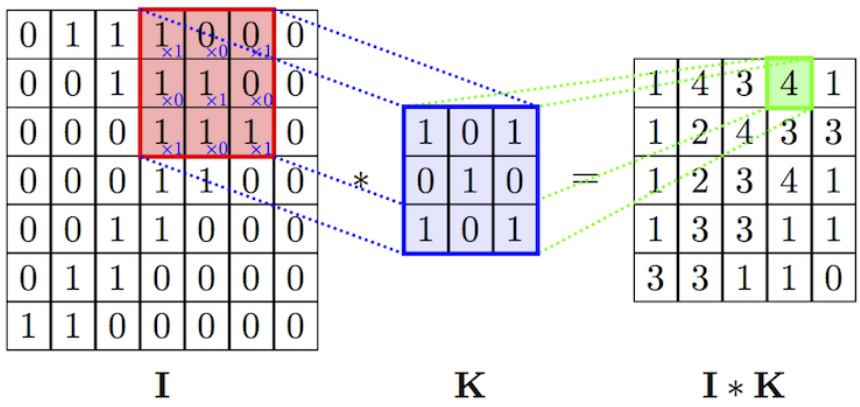


Figura 4.10: Ejemplo de la operación de convolución en variables discretas. Fuente: [40]

Mediante ambos sumatorios se recorre cada fila del *kernel* de izquierda a derecha. Pero si nos fijamos en la función de la imagen I vemos que al restar las posiciones del *kernel* puede haber valores negativos. Este efecto significa que se multiplica un elemento del kernel con una posición fuera de la matriz de la imagen. Como no hay valores fuera de la matriz esto supone un problema. Este se resuelve mediante la introducción de valores adicionales normalmente inicializados a 0 que forman un borde alrededor de la matriz original. A este borde se le denomina *padding*.

En la Figura 4.10 se puede ver como se está aplicando un *kernel* K de tamaño 3×3 sobre una matriz I de tamaño 7×7. Un efecto que se debe tener en cuenta es que al aplicar el

kernel se obtiene una matriz de tamaño 5×5. Esto ocurre debido a que al aplicar el *kernel* se obtiene un escalar y por tanto si se usa un *kernel* de tamaño 3×3 se reduce una fila y una columna a cada lado, si es un *kernel* de 4×4 se pierden dos filas y columnas, etc.

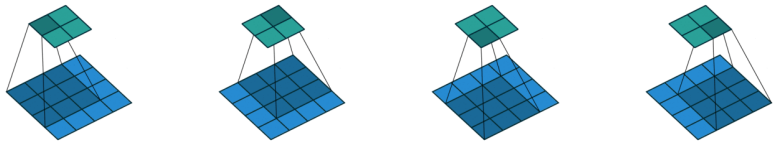


Figura 4.11: Ejemplo de convolución al aplicar un *kernel* de tamaño 3×3. Fuente: [42]

En la Figura 4.11 se puede apreciar mejor este efecto de reducción de dimensionalidad al aplicar un kernel. En color azul claro podemos observar la matriz de entrada, la imagen, en color azul oscuro el *kernel* y en color verde la matriz resultante.

Este efecto de reducción de dimensionalidad puede ser anulado mediante un *padding*. Si es un *padding* de un elemento y se aplica un *kernel* de tamaño 3×3 la matriz resultante tendrá las mismas dimensiones que al inicio, si es un *padding* de dos elementos y un *kernel* de tamaño 4×4 ocurre lo mismo, etc.

Por último se debe introducir el concepto de *stride*. El *stride* nos indica el número de píxeles que se moverá el *kernel*. De esta forma el *kernel* no estará centrado en el pixel adyacente sino en el pixel $(i + x, j)$ donde x es el valor del *stride*.

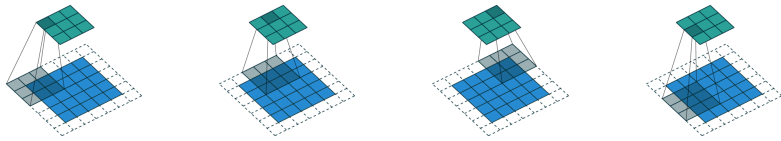


Figura 4.12: Ejemplo de convolución con *kernel* 3×3, *padding* 1 y *stride* 2. Fuente: [42]

En la Figura 4.12 se puede observar que un *stride* superior a 1 también puede reducir el tamaño de la matriz final.

Las convoluciones normalmente vienen acompañadas de la operación *pooling*, mas en concreto *max-pooling* ya que existen múltiples variantes.

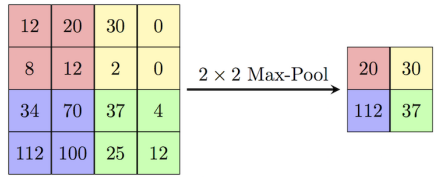


Figura 4.13: Ejemplo de la operación *max-pooling* de tamaño 2×2 . Fuente: [43]

Al aplicar *max-pooling* se divide la matriz a la que se aplica en tantos sectores como indique el tipo de operación, *max-pooling* 2×2 divide la matriz en 4 sectores. De estos sectores se obtiene el valor máximo y se produce una matriz del tamaño indicado, 2×2 en el caso de *max-pooling* 2×2.En la figura 4.13 podemos ver el mismo ejemplo que el descrito.

De esta forma, aplicando *kernels* previamente diseñados se pueden extraer las *features* que se indicaron al principio de la sección.

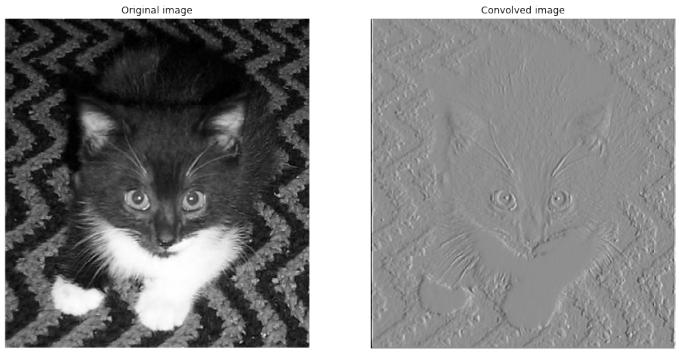


Figura 4.14: Ejemplo de *feature* de una red convolucional. Fuente: [41]

4.2.2. Aumento sintético de datos

En la sección anterior se describieron algunas técnicas generales mediante las que se puede hacer frente al problema del overfitting. Una vez explicada la forma en la que se va a trabajar con las imágenes es posible introducir una nueva técnica mediante la que no solo es posible reducir el *overfitting* sino también incrementar el dataset actual.

Esta técnica es el aumento sintético de datos. En esencia lo que busca conseguir es incrementar el número de elementos del dataset realizando transformaciones sobre los datos pertenecientes a este. Dependiendo del tipo de datos con los que se esté trabajando estas transformaciones tendrán una estructura distinta. Por ejemplo, si se está trabajando con un modelo de procesamiento de lenguaje natural y los datos usados para entrenar el modelo son frases en un idioma determinado, los nuevos datos generados serán frases derivadas de las presentes en el dataset.

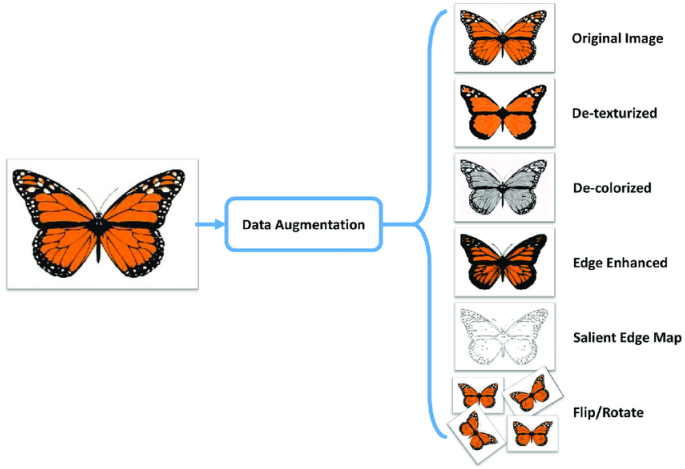


Figura 4.15: Ejemplo de transformaciones aplicadas a una imagen RGB. Fuente: [44]

Si en el dataset existe una frase como "Me gustan los coches", una frase creada mediante aumento sintético puede ser "Me gustan los aviones". Este ejemplo de forma indirecta introduce un componente muy importante del aumento sintético. Los datos creados mediante esta técnica deben "tener sentido". Esto significa que las transformaciones deben producir datos que sean válidos. En los problemas donde se realiza clasificación multiclase, un dato generado sintéticamente debe pertenecer a la misma clase que el datos original sobre el que se ha realizado la transformación.

En la Figura 4.15 podemos ver un ejemplo de algunas de las principales transformaciones que son usadas en el aumento sintético de imágenes. Algunas de estas son rotaciones de la imagen, modificaciones realizadas sobre los distintos canales en la imágenes RGB como incrementos y decrementos del brillo, modificaciones en la saturación, aplicaciones de kernels específicos para obtener los bordes de las mismas y muchas otras.

El aumento sintético de datos ayuda en el problema del overfitting debido a que al incrementar el número de elementos del dataset se vuelve más difícil para el modelo adaptarse a los datos proporcionados. Esto implica que mediante este aumento en los datos el modelo obtendrá una mejor habilidad de generalización.

4.2.3. Transfer Learning

Hasta este momento, cuando se ha hablado de entrenar un modelo la estrategia que se ha seguido ha sido el entrenamiento de todas las capas mediante la entradas proporcionadas del dataset. Pero este método de entrenamiento no es el único método posible para entrenar los modelos.

La técnica de Transfer Learning es una técnica muy extendida donde la idea principal es que un modelo sin entrenar es entrenado sobre un dataset genérico. En la clasificación de

imágenes un dataset de este tipo sería ImageNet [45] que está compuesto por millones de imágenes cada una perteneciendo a una categoría de objetos.

Una vez entrenado el modelo sobre este dataset se vuelven a entrenar las últimas capas de este modelo de forma que se ajusten al dataset con el que se está trabajando.De esta forma las primeras capas del modelo mantienen los pesos obtenidos en la primera fase y la segunda parte obtiene nuevos pesos respectivos al dataset sobre el que estamos interesado en entrenar el modelo.

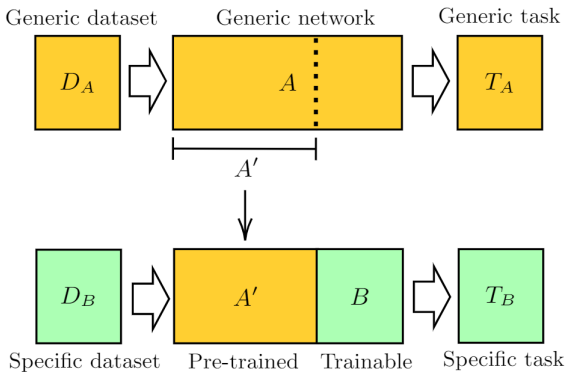


Figura 4.16: Diagrama de la técnica Transfer Learning. Fuente: [46]

En la Figura 4.16 se puede observar como el modelo A es entrenado sobre el dataset genérico D_A.Una vez terminada esta tarea las primeras capas de este modelo son mantenidas iguales y solo son entrenadas las últimas capas sobre el dataset específico D_B para el problema que estamos interesados en resolver. De esta forma no solo se consigue entrenar el modelo final en menos tiempo, ya que las primeras capas ya han sido entrenadas, sino que se obtiene una mejora en el rendimiento del modelo debido que los datasets genéricos en la gran mayoría de casos contienen muchos mas datos que los datasets específicos y por tanto la generalización obtenida es mejor.

Capítulo 5

Clasificación mediante Deep Learning

En este capítulo se pasará a detallar el proceso mediante el que se ha llegado a las soluciones al problema planteado. En primer lugar se va a describir el dataset, o conjunto de entrenamiento, que se ha utilizado para realizar las pruebas. Sobre este dataset se han realizado ciertas modificaciones que también serán descritas.

A continuación se indicará en que consisten los experimentos realizados mediante la técnica de transfer learning y los resultados obtenidos. Después se detallará la implementación de una arquitectura Resnet así como sus componentes característicos y los resultados obtenidos de esta forma. De la misma manera se describirá una arquitectura Xception y sus resultados. Por último se cotejarán los datos obtenidos de todas las soluciones y se dará fin al capítulo.

5.1. Dataset

Ya sabemos que para realizar el entrenamiento de una red neuronal en un problema de clasificación hacen falta datos que estén correctamente etiquetados. Obtener un dataset óptimo en es una tarea que puede ser muy complicada por varias razones.

- **Tamaño del dataset:** Pese a que hoy en día se dispone de más datos que en ningún otro momento de la historia dependiendo de campo en el que se sitúe el problema esto no siempre puede ser así.
- **Calidad de los datos:** No solo hace falta una cantidad sustancial de datos pero también deben ser datos que sean consistentes, por ejemplo no tener una imagen etiquetada como correcta y una copia de la misma como incorrecta. También deben ser fiables, es decir, que representen correctamente el espacio del problema y cuanto mas recien-

tes sean los datos, más probable es que se ajusten mejor al escenario donde va a ser desplegado el modelo.

- **Etiquetado:** Aunque se consigan los datos necesarios si estos no tienen asociada la categoría a la que pertenecen no se puede realizar el entrenamiento.

Estos últimos son solo algunos de los problemas que pueden ocurrir al crear un dataset. Por suerte, para este proyecto, el tutor académico ha proporcionado un dataset que no presenta estos problemas. De esta forma se ha evitado tener que pasar por el proceso de recopilación de imágenes y etiquetado a mano que podría suponer un coste temporal muy alto.

La categoría de "piezas industriales" es muy amplia. Desde piezas simples como tuercas o pernos hasta piezas más complejas como intercambiadores de calor. En este caso las imágenes del dataset corresponden a soldaduras. Detectar soldaduras incorrectas permite descubrir riesgos estructurales que pueden darse en otras piezas más complejas ya que supone el punto de unión entre dos o más materiales.

Veamos ahora las características de los datos proporcionados.

- **Tipos de soldaduras:** Los datos están divididos en dos grandes categorías.Al crear el dataset las categorías a las que pertenecían las soldaduras son "Tipo 2" y "Tipo 3". Estos nombres en al práctica no aportan información relevante y pueden ser sustituidos por otros identificadores. De esta manera las soldaduras de "Tipo 2" pasará a ser el *Dataset 1* y las soldaduras de "Tipo 3" de *Dataset 2*.
- **División:** Para cada tipo de soldadura hay una subdivisión en soldaduras correctas e incorrectas. De esta forma las imágenes ya se encuentran divididas en dos carpetas para cada tipo de soldadura.
- **Tamaño de las imágenes:** Cada una de las imágenes tiene un tamaño de 80 píxeles de altura y 480 píxeles de anchura. Esta resolución puede parecer baja para la tarea en la que serán usadas. En realidad para las tareas de aprendizaje automático no son necesarias resoluciones muy altas, debe ser una resolución adecuada para permitir al modelo obtener las características necesarias que determinarán la clasificación. Existen también casos en los que, paradójicamente, incrementar la resolución de la imagen puede incrementar el error de validación [47].
- **Color en las imágenes:** Todas las imágenes están representadas mediante una escala de grises, denominadas comúnmente "en blanco y negro". De esta forma cada píxel toma un valor de 0 a 255 que indica su brillo. Si es un valor cercano a 0 se obtiene un píxel oscuro y si es un valor cercano a 255, un píxel casi blanco.

En la Figura 5.1 se pueden ver algunos ejemplos de soldaduras correctas e incorrectas de ambos tipos. Si nos fijamos en las imágenes (han sido seleccionadas de forma aleatoria) podemos ver que las soldaduras incorrectas en ambos casos tienen una sección en el centro de la misma donde el grosor es menor.

Este tipo de anomalías son las que un humano percibe de forma instantánea y en las siguientes secciones veremos cómo de bien pueden detectarlos los modelos implementados.

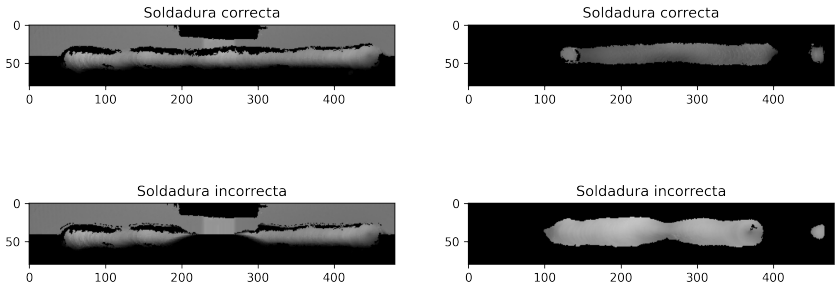


Figura 5.1: Muestras de imágenes de soldaduras del Dataset 1 (*izda*) y el Dataset 2 (*dcha*) .

Ya se ha indicado que las soldaduras se dividen en correctas e incorrectas pero muchas veces el dataset no tiene una distribución uniforme de elementos, en este caso imágenes, en las distintas categorías (suponiendo un problema de clasificación). Cuando esta distribución no es uniforme nos encontramos ante lo que se denomina un dataset desequilibrado.

Un dataset de este tipo puede resultar problemático. Si decidimos realizar el entrenamiento del modelo sobre este dataset, al existir más datos de una categoría que de otra, el modelo adaptará los pesos de tal forma que se clasifiquen la mayoría de elementos del conjunto de validación como pertenecientes a este categoría mayoritaria.

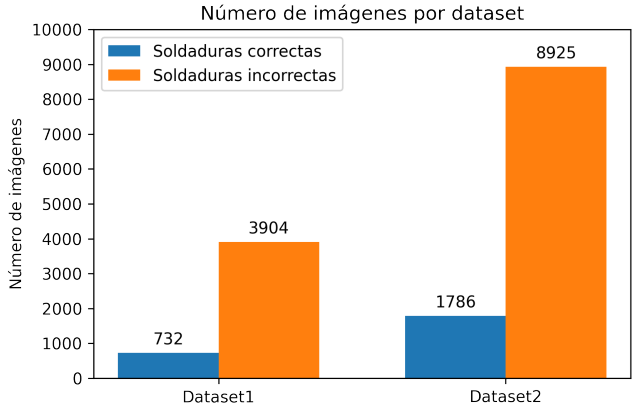


Figura 5.2: Número de soldaduras correctas e incorrectas de cada tipo de soldadura.

De esta forma el modelo en realidad no está clasificando correctamente las imágenes debido a las características o features que detecta sino que los pesos se ajustan para clasificar los datos del conjunto de validación en la categoría mayoritaria. De esta forma el modelo

obtiene un coste muy bajo ya que casi con total seguridad el dato va a pertenecer a esta categoría y el modelo no nos aporta información útil.

Como vemos en la figura 5.2 ambos dataset tienen más imágenes que muestran una soldadura errónea que imágenes de soldaduras correctas. Este es en realidad un problema muy común. Como se ha indicado antes al encontrarnos en una época en la el Big Data tiene tanta importancia es muy sencillo obtener una gran cantidad de datos pero lo que nos se asegura es cómo van a estar estructurados. Hay muchos sectores donde por la naturaleza del proceso del que se está recopilando información estos van a producir un dataset desequilibrado como del que disponemos.

Pongamos un ejemplo de una situación en la que puede surgir un dataset de este tipo. Se está desarrollando un estudio en un banco sobre las transacciones de los clientes. El objetivo es detectar automáticamente cuando se va a producir una transacción errónea. Para detectarlas se ha construido un dataset con miles de transacciones que pueden ser analizadas. Pero los errores en transacciones bancarias no suelen ocurrir muy a menudo.

Si suponemos que una de cada diez operaciones fracasa (una tasa mucho mas alta de lo que suele ocurrir en la realidad) tendremos solo un 10 % de datos en una de las categorías y un 90 % en la otra. Si suponemos una tasa de error más realista (una de cada mil operaciones) la situación empeora todavía más.

Para lidiar con este problema se van a proponer varias técnicas, los cuales ambos tienes ventajas y desventajas.

1. **Reducir la categoría predominante:** También denominada *undersampling*, es la más sencilla de las técnicas ya que solo es necesario eliminar el número suficiente de imágenes hasta conseguir un dataset equilibrado. Para reducir la posibilidad de que se eliminen todas las imágenes que tienen presente cierta característica este proceso se realizará de forma aleatoria. Esto no elimina por completo esa posibilidad pero la reduce.

   El principal problema con este enfoque es que el modelo pierde la posibilidad de ser entrenado sobre estas imágenes reduciendo de esta forma la eficacia del mismo.

2. **Aumentar la categoría minoritaria:** Para al creación de nuevas imágenes usaremos las técnicas de *data augmentation* vistas en el capítulo anterior. De esta forma solo se aplicarán transformaciones a las soldaduras correctas y podremos incrementar artificialmente el dataset. Esta técnica también denominada *oversampling* permite obtener un dataset equilibrado pero puede incrementar la posibilidad de obtener overfitting.

   Existen algunas técnicas como SMOTE (*Synthetic Minority Oversampling Technique*) [48] para la creación artificial de datos en conjuntos donde hay desequilibrio entre las clases. Esta técnica en concreto se aplica casi exclusivamente a datos tabulares y por tanto no puede ser usada en este problema.

3. **Usar herramientas de control de resultados:** Cuando se realice el entrenamiento se usarán distintas métricas para determinar el rendimiento del modelo. Estas métricas

serán explicadas con más detalle en la sección siguiente pero una de las más habitualmente usadas es la exactitud (en inglés, *accuracy*) que indica el porcentaje de imágenes del conjunto de validación que han sido clasificadas correctamente.

Esta métrica puede ser un buen indicador del rendimiento del modelo cuando se este realizando el entrenamiento sobre un dataset equilibrado, pero ante un dataset como el que disponemos puede inducir a error en la interpretación de los resultados. Como se ha mencionado si hay un gran desequilibrio entre las categorías el modelo termina clasificando casi todos los elementos en el conjunto de validación como pertenecientes a la clase predominante. Esto producirá un alto valor de *accuracy* pero no es un resultado verdadero de la capacidad del modelo. Es un producto del dataset.

Para controlar este problema usaremos métricas como *Precision* y *F1* y herramientas como las matrices de confusión que nos permitirán saber si el *accuracy* obtenido es producto del modelo o derivado del dataset.

Una vez aplicadas estas técnicas se obtienen dos datasets con la siguiente distribución de imágenes.

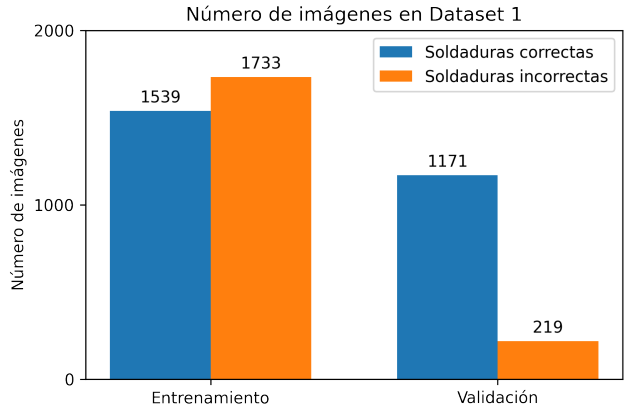


Figura 5.3: Número de soldaduras correctas e incorrectas en cada conjunto para el Dataset 1.

En la Figura 5.3 podemos ver esta distribución para el Dataset 1. Para crear el conjunto de validación habitualmente se usa la regla "30-70". Esta regla indica que el 30 % de las imágenes iniciales con las que se dispone deben ser usadas en el conjunto de validación. El resto de imágenes son las usadas en el conjunto de entrenamiento.

Para este Dataset 1 se ha usado esta regla de modo que en el conjunto de validación las imágenes suponen el 30 % del dataset original. Tanto para las imágenes correctas como para incorrectas.

Las imágenes correctas en el conjunto de entrenamiento han sufrido *augmentations* y por tanto se ha incrementado el número de imágenes en esa categoría. Las incorrectas han sido reducidas para poder equipararse a las correctas.

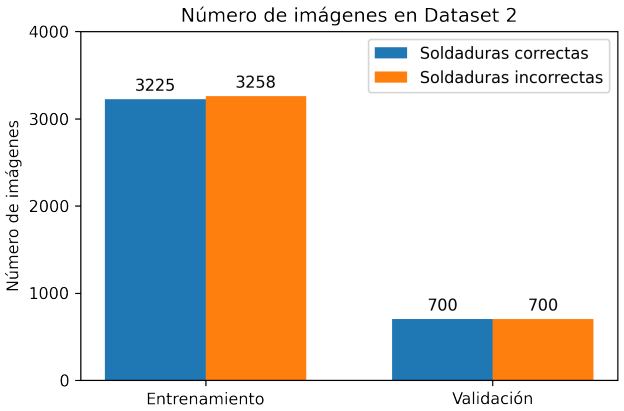


Figura 5.4: Número de soldaduras correctas e incorrectas en cada conjunto para el Dataset 1.

En la Figura 5.4 se dispone de la distribución del Dataset 2. En este caso el dataset ha sido creado de la misma forma que en el caso anterior con la excepción de que se ha calculado el 30 % sobre las imágenes correctas y se ha usado esa cantidad para obtener las imágenes de ambas categorías en el conjunto de validación.

5.1.1. Augmentations

Como vimos en el capítulo anterior existen muchas formas de modificar las imágenes del conjunto de entrenamiento para aumentar su número.

En este caso debido a las imágenes usadas no es posible aplicar muchos de los tipos de modificaciones que se han descrito previamente. En primer lugar como ya se ha indicado las imágenes son representadas mediante una escala de grises (en inglés, *greyscale*). Por ello todas las transformaciones en las que se modifica de alguna manera los distinto canales de la imagen no pueden ser aplicados en este contexto debido a que en estas imágenes se dispone solo de un canal y estas transformaciones usan los tres canales presentes en las imágenes RGB de forma simultánea.

Las transformaciones basadas en rotaciones de las imágenes tampoco pueden ser usadas debido a que puede perjudicar más al modelo de lo que ayude en el entrenamiento. Una soldadura girada un cierto número de grados puede provenir de una soldadura correcta. Pero las imágenes que serán evaluadas son sensibles a los giros, de modo que una soldadura etiquetada como correcta dada la vuelta se consideraría incorrecta.

Esto ocurre en muchos otros casos como por ejemplo el famoso dataset MNIST que contiene imágenes de dígitos escritos a mano. En este caso si aplicamos rotaciones de 180 grados a las imágenes que contienen el número 6, estaremos introduciendo en el conjunto de

entrenamiento una imagen que representa un 9 pero tiene como etiqueta un 6. Estas imágenes junto con las del dígito 9 harán que se modifiquen de forma errónea los pesos del modelo.

Existen otros casos en los que no existe este problema. Si se está entrenando un clasificador de gatos y perros, un gato sigue siendo un gato por mucho que se rote la imagen.

1. **Ruido Gaussiano:** El nombre proviene de la distribución Gaussiana o más habitualmente llamada distribución normal. De esta forma cada valor para cada pixel del filtro proviene de esta distribución. Modificando la media μ y la desviación estándar σ se puede obtener un tono mas oscuro del ruido o la distorsión que produce en la imagen respectivamente. El efecto obtenido es similar al ruido "blanco" presente en las televisiones.El ruido gaussiano habitualmente usado debido a que es muy común de encontrar en situaciones reales y además es un modelo matemáticamente simple con el que trabajar.

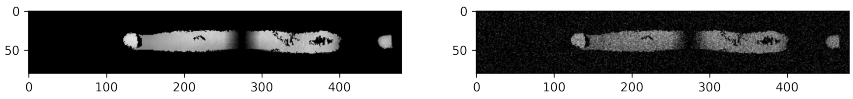


Figura 5.5: Ejemplo de ruido Gaussiano aplicado a un elemento del dataset.Imagen original (*izda*) e imagen transformada (*dcha*).

2. **Cambio de tamaño:** Modificar el tamaño de la imagen es una forma de reducir el procesamiento necesario por la máquina que realiza el cómputo. Un dataset con imágenes de alta resolución implica un mayor número de cálculos para entrenar el modelo. Debido a que la tarjeta gráfica usada para este proceso no es un modelo con grandes capacidades esta transformación ayudará al tiempo de entrenamiento.

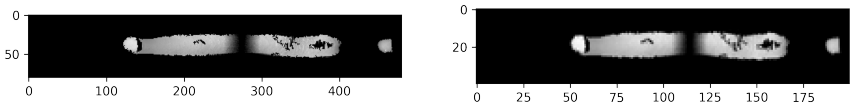


Figura 5.6: Ejemplo de cambio de tamaño aplicado a un elemento del dataset.Imagen original (*izda*) e imagen transformada (*dcha*).

En la Figura 5.6 debemos fijarnos en la escala de las imágenes. La imagen de la derecha tiene un tamaño de 200×40 pixeles frente los 480×80 de la imagen original a la izquierda.

3. **Correción Gamma:** Esta modificación de la imagen actúa sobre la luminiscencia de la misma. La luminiscencia en imágenes RGB es calculada en base a las intensidades de los tres canales y en imágenes basadas en escalas de grises en base al único canal.

Mediante la fórmula $V_{salida} = V_{entrada}^{\gamma}$ se modifica cada píxel para obtener la imagen resultante.

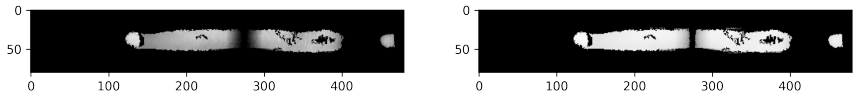


Figura 5.7: Ejemplo de corrección gamma con valor $\gamma = 0{,}2$ aplicada a un elemento del dataset.Imagen original (*izda*) e imagen transformada (*dcha*).

5.2. Transfer Learning

Una vez determinados los datos con los que se va a trabajar es el momento de realizar los primeros experimentos. Para estos se usará aprendizaje de transferencia donde se aplicarán las técnicas y conceptos adquiridos en el Capítulo 4.

Para realizar estos experimentos y obtener resultados mediante los que se puede llegar a una conclusión lógica se debe seguir una metodología. En primer lugar se hará una descripción de las métricas usadas para la evaluación de cada experimento. A continuación se realizará un experimento base mediante el que determinaremos cuales son los resultados que se obtienen usando una configuración tradicional en los problemas de clasificación de imágenes.

Una vez se disponga de esta línea base se realizarán modificaciones sobre ciertos elementos como la arquitectura del modelo, el optimizador, las funciones de activación, etc. En cada uno de estos experimentos nuevos que se realizan se realizará la modificación exclusivamente de un elemento del clasificador, es decir, en los experimentos en los que se usan arquitecturas diferentes únicamente se modificará la arquitectura y el resto de parámetros se mantendrán.

Usando esta metodología podremos ver cómo afecta cada cambio a los resultados y sabremos que es debido a la modificación introducida. De no seguir este procedimiento y modificar varios parámetros a la vez se podrían obtener conclusiones erróneas sobre la causa del cambio en los resultados.

5.2.1. Diseño de experimentos

Una vez descrito el procedimiento a seguir podemos describir la métricas que se usarán en cada experimento y los motivos por los que han sido seleccionadas.

1. **Coste de entrenamiento:** Este es el coste con el que se ha trabajado durante todo el Capítulo 4. Por ello su funcionamiento ya ha sido descrito y sabemos que nos da un

valor numérico mediante el que interpretar lo que está ocurriendo cuando se modifican los pesos y bias de la red neuronal. En estos experimentos se va a usar la entropía cruzada, ya mencionada en el Capítulo 4, como función de coste en vez de el error cuadrático medio.

$$C = -\frac{1}{n}\sum_{x}[\boldsymbol{y}\ln a + (1-\boldsymbol{y})\ln(1-a)] \tag{5.1}$$

En la Figura 5.1 $\boldsymbol{y}$ es el vector de valores que se desean obtener, n es el número de elementos sobre los que se calcula la función, x es el numero de elementos del vector de entrada y a es la salida que produce la red, el valor en el coste cuadrático medio se había definido como $\hat{\boldsymbol{y}}$. La razón por la que se usará esta función de coste es porque actualmente supone el estándar para las funciones de coste. El error cuadrático medio tiene una fórmula donde el cálculo es mas intuitivo y por tanto supone una mejor opción a la hora de realizar la explicación.

2. **Coste de validación:** Al igual que en el caso anterior se usará la entropía cruzada. Como su nombre indica mediante esta métrica se comprobará el coste al evaluar el modelo sobre el conjunto de validación, lo que indicará cómo se comportan los pesos y bias antes imágenes que no han pasado por el modelo.

3. **Accuracy:** Como se indicó en la sección anterior, indica el porcentaje de imágenes que se clasifican correctamente. Es una métrica que permite ver de forma muy clara la eficacia del modelo pero por las razones que mencionamos anteriormente debe ir acompañado de las dos siguiente métricas.

4. **Precision:** Debido a que se estará usando un dataset desequilibrado esta métrica servirá para controlar si el valor que se está obteniendo en *accuracy* es producido por el clasificador y no es un valor artificialmente aumentado por este desequilibrio.

$$Precision = \frac{Verdaderos\ Positivos}{(Verdaderos\ Positvos + Falsos\ Positivos)} \tag{5.2}$$

En la ecuación 5.2 podemos ver la forma en la que se calcula esta métrica. De esta forma se determina la habilidad del clasificador de no clasificar un elemento de una categoría como perteneciente a otra. El mejor valor es 1 y el peor es 0.

5. **F1 Score:** De una manera similar a *Precision* la métrica *F1* combina la exhaustividad (en inglés,*Recall*) y *Precision* en un solo valor. El *Recall* definido en la ecuación 5.3 indica la proporción de verdaderos positivos que se han identificado correctamente.

$$Recall = \frac{VerdaderosPositivos}{VerdaderosPositivos + FalsosNegativos} \tag{5.3}$$

$$F1 = 2 \cdot \frac{Precision \cdot Recall}{Precision + Recall} \tag{5.4}$$

En la ecuación 5.4 podemos observar la forma de calcular el valor F1. Al igual que en el caso de *Precision* el mejor valor es 1 y el peor 0.

6. **Tiempo:** Uno de los problemas más comunes que suelen aparecer al entrenar modelos mediante Deep Learning es que este aprendizaje puede tardar mucho tiempo. Es importante tener en cuenta que no solo se requiere un modelo que obtenga buenos resultados sino que este debe poder entrenarse en un tiempo aceptable. Es posible que usando descenso de gradiente estocástico y una tasa de aprendizaje lo suficientemente baja se obtenga un clasificador con un *Accuracy* muy alto pero de nada sirve si se debe estar entrenando muchos meses.

   Cabe destacar que muchos modelos muy profundos en efecto deben entrenarse en ocasiones durante meses pero estos son habitualmente modelos del estado del arte o modelos complejos entrenados en centros de procesamiento de datos.

Acompañando estas métricas se usarán matrices de confusión. Estas matrices nos aportarán una mejor visualización de los resultados obtenidos. Cada columna de esta matriz representa el número de predicciones de cada clase del modelo mientras que las filas representan el número de elementos reales en cada categoría.

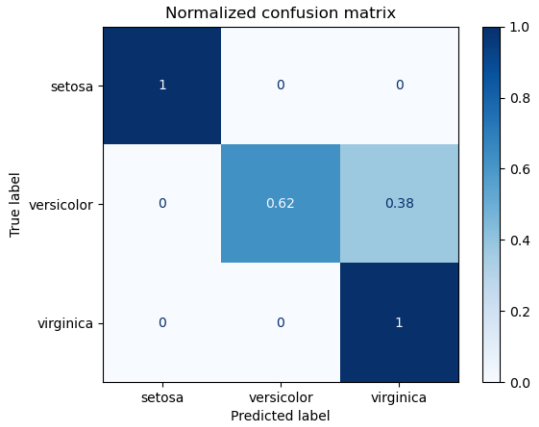


Figura 5.8: Ejemplo de matriz de confusión normalizada. Fuente:[36]

Como se puede ver en la Figura 5.8 en las posiciones $(0, 0)$ y $(2, 2)$) de la matriz se ha conseguido una clasificación perfecta ya que el 100 % de imágenes clasificadas coinciden con la categoría a la que realmente pertenecen.

Por otro lado en la posición $(1, 1)$ vemos que se obtiene un 62 % de éxito al clasificar la categoría "versicolor". Por otro lado lo que nos indica la casilla $(1, 2)$ es que un 38 % de los elementos que realmente pertenecen a la categoría "versicolor" fueron clasificados como "virginica".

Esta información nos ayuda a entender mejor como se esta comportando el modelo en las distintas categorías ya que como vemos aquí hay solo una que realmente está resultando problemática. Por último se debe destacar que matriz de confusión ideal es la que tiene los elementos de la diagonal en su valor máximo y el resto de casillas con el valor 0.

5.2.2. Experimento base

Para el experimento principal como se ha indicado se usará una configuración que se puede considerar "estándar" en la clasificación de imágenes. La configuración usada es la siguiente:

Característica	Selección
Arquitectura del modelo	ResNet-34
Modelo entrenado de antemano	Si
Normalización	Si
Tasa de aprendizaje	Determinada en ejecución.
Función de activación	*Rectified Linear Unit*
Función de coste	Determinada en ejecución.
Optimizador	Entropía cruzada

Cuadro 5.1: Resultado del entrenamiento sobre el Dataset 2 mediante Transfer Learning de una arquitectura ResNet-34

De este modo en cada uno de los experimentos a continuación se realizará un solo cambio sobre esta configuración. En primer lugar se ha elegido una arquitectura ResNet-34 ya que no es demasiado profunda, como su nombre indica está formada por 34 capas, y por lo tanto no se necesita un tiempo demasiado elevado para entrenarla.

La normalización de los datos es una tarea importante a realizar sobre el conjunto de datos antes de poder entrenar el modelo. Cuando los datos están normalizados estos tienen una media de 0 y una desviación estándar de 1. Como indicamos antes, los valores de cada píxel en una imagen estarán en el intervalo $[0, 255]$ para imágenes RGB o $[0, 1]$ para imágenes basadas en escalas de grises.

En fastai cuando se usa un modelo entrenado de antemano se puede realizar una normalización a cada batch en base a las estadísticas de estos modelos. De esta forma, al haber sido entrenados mediante datasets como ImageNet, se usan estos valores para realizar la normalización sobre los datos introducidos por el programador para realizar Transfer Learning.

Epochs	Coste Entren.	Coste Valid.	*Accuracy*	*Precision*	*F1*	Tiempo
0	0.366318	0.295735	0.905755	0.866667	0.613569	00:20
1	0.271967	0.125145	0.958993	0.797794	0.883910	00:20
2	0.148291	0.130217	0.963309	0.830709	0.892178	00:20
3	0.111168	0.042447	0.982014	0.980198	0.940618	00:20
4	0.063857	0.073505	0.984892	0.938053	0.952809	00:20

Cuadro 5.2: Resultado del entrenamiento sobre el Dataset 1 mediante Transfer Learning de una arquitectura ResNet-34

Epochs	Coste Entren.	Coste Valid.	*Accuracy*	*Precision*	*F1*	Tiempo
0	0.431884	0.467951	0.779286	0.943311	0.729185	00:36
1	0.310500	0.274525	0.872857	0.948454	0.861154	00:36
2	0.208080	0.188792	0.915000	0.914408	0.915061	00:37
3	0.239416	0.170864	0.933571	0.929279	0.933902	00:37
4	0.191354	0.184898	0.927143	0.942308	0.925872	00:37

Cuadro 5.3: Resultado del entrenamiento sobre el Dataset 2 mediante Transfer Learning de una arquitectura ResNet-34

En la tabla 5.3 se pueden ver que el clasificador obtiene un *Accuracy* de 98 %. Este es un buen valor aunque se debe tener en cuenta el valor de *Precision* y *F1*. Estos valores también indican que el clasificador no suele atribuir la categoría errónea a las imágenes aunque el máximo de estos valores es 0.95 y por tanto todavía hay posibilidad de mejora.

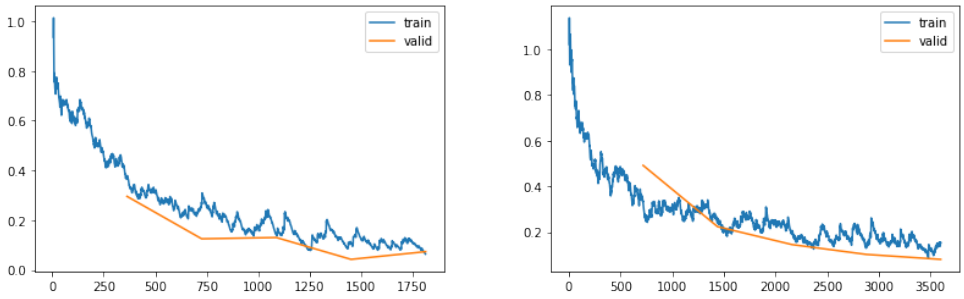


Figura 5.9: Gráficas de coste frente a número de batches evaluados para el Dataset 1 (*izda.*) y Dataset 2 (*dcha*).

En la Figura 5.9 se puede observar los motivos por los que se detuvo en entrenamiento para ambos datasets. En el caso del Dataset 1 se puede ver como el coste sobre el conjunto de validación estaba empezando a aumentar y por tanto la situación empezaba a parecer propia de una caso de *overfitting*. Por ello se detuvo el entrenamiento pese a que el coste de entrenamiento seguía bajando.

En el caso de Dataset 2 el entrenamiento se detuvo por el aumento del coste de entrenamiento en la última epoch. En este caso no se estaba detectando ningún aumento del coste de validación y el coste pese a indicar un aumento estaba siguiendo la tendencia marcada. En este caso detener el entrenamiento es más dudoso ya que no hay señal de overfitting aunque haya un incremento del coste de entrenamiento.

En la Figura 5.10 se pueden ver las matrices de confusión mediante las que podemos ver las clasificaciones en las que falla el modelo. Para el Dataset 1 se han obtenido muy pocas imágenes clasificadas incorrectamente aunque hay cierto desequilibrio entre ellas. Un mayor número de imágenes han sido clasificadas como correctas pese a que no lo son. Si observamos

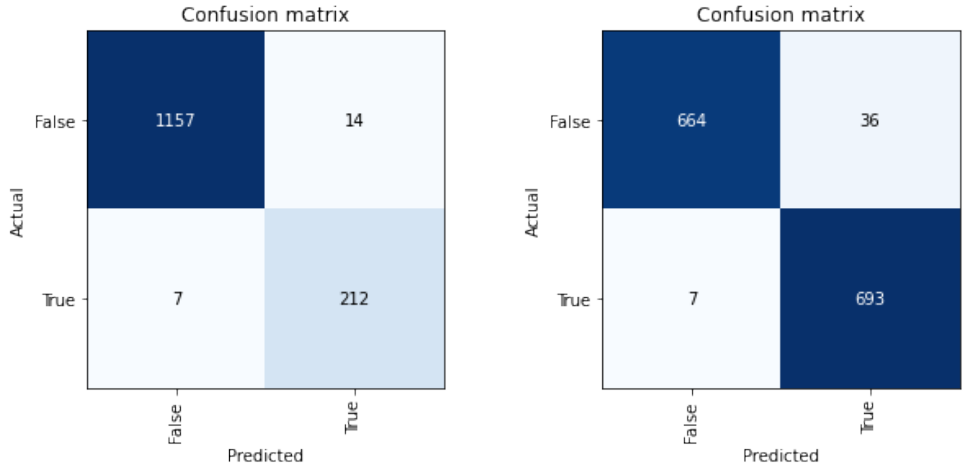


Figura 5.10: Matriz de confusión para el Dataset 1 (*izda.*) y Dataset 2 (*dcha*).

la matriz correspondiente al Dataset 2 vemos que se puede observar un efecto similar solo que en este caso el número de imágenes clasificadas incorrectamente es mayor.

La razón por la que puede ocurrir esto es que en primer lugar el clasificador del Dataset 2 obtuvo un *Accuracy* menor. Por ello tiene sentido que haya más imágenes incorrectas. Pero respecto al desequilibrio entre las categorías para las imágenes erróneamente clasificadas se debe tener en cuenta la forma en la que fueron construidos los datasets.

En ambos casos las *augmentations* de imágenes fueron aplicadas en su mayoría al conjunto de imágenes correctas ya que esta era la categoría de la que se disponían menos imágenes. De esta forma el conjunto de imágenes incorrectas dispone de más imágenes originales y por tanto puede generalizar mejor. Por ello el clasificador predice en muchas ocasiones que una soldadura es correcta cuando en realidad no lo és, generaliza peor la categoría de imágenes correctas.

En esta sección también se incluirán algunos experimentos adicionales con el objetivo de ilustrar conceptos que se han explicado en el Capítulo 4. Estos experimentos no modifican el experimento realizado y por tanto solo se describirán aquí ya que no afectan al rendimiento del clasificador.

La detección de bordes fue uno de los principales elementos en la descripción de las convoluciones en el Capítulo 4. Como ya se ha detallado el funcionamiento de estas en esta sección se presentarán algunos de los resultados al aplicar ciertos *kernels* sobre imágenes de ambos datasets.

$$\boldsymbol{K}_{inf} = \begin{bmatrix} 1 & 1 & 1 \\ 0 & 0 & 0 \\ -1 & -1 & -1 \end{bmatrix} \qquad \boldsymbol{K}_{sup} = \begin{bmatrix} -1 & -1 & -1 \\ 0 & 0 & 0 \\ 1 & 1 & 1 \end{bmatrix} \tag{5.5}$$

$$\boldsymbol{K}_{izq} = \begin{bmatrix} -1 & 0 & 1 \\ -1 & 0 & 1 \\ -1 & 0 & 1 \end{bmatrix} \qquad \boldsymbol{K}_{der} = \begin{bmatrix} 1 & 0 & -1 \\ 1 & 0 & -1 \\ 1 & 0 & -1 \end{bmatrix} \tag{5.6}$$

Las expresiones 5.5 representan las matrices de los *kernels* que será usados en las convoluciones. Estos *kernels* son denominados operadores *Prewitt* [49]. El diseño de estos *kernels* es muy sencillo. Supongamos que realizamos la convolución con el *kernel* $\boldsymbol{K}_{sup}$. Al realizar las operaciones obtendremos valores altos para las zonas de la imagen en las que hay valores bajos en la zona superior del espacio, donde actuarán los valores -1 del *kernel*, y valores altos en la zona inferior, donde actuarán los valores 1 del *kernel*.

De esta forma se obtienen los bordes superiores, se detectan las zonas donde se pasa de una zona de píxeles de baja intensidad, fuera del contorno del objeto, a una zona donde los píxeles tienen mayor intensidad, dentro del contorno del objeto. De forma análoga se obtienen los bordes laterales.

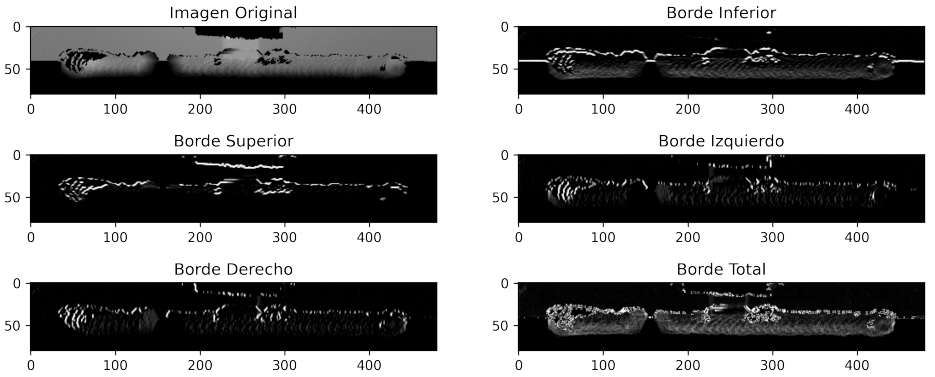


Figura 5.11: Ejemplo de convoluciones para detección de bordes sobre imagen del Dataset 1.

En las figuras 5.11 y 5.12 se pueden ver los bordes obtenidos mediante estos *kernels*. Recordemos que al aplicar cada kernel el resultado final sigue siendo una matriz (o *tensor*) donde destacan los píxeles que pertenezcan a los bordes, de esta manera se puede realizar una suma de las cuatro matrices resultantes de aplicar cada *kernel* y de esta forma obtener una matriz donde se destaca el borde completo de la figura. Este borde completo es el representado en la imagen "Borde Total" de cada figura.

Existen múltiples otros *kernels* y métodos mediante los que obtener representaciones de

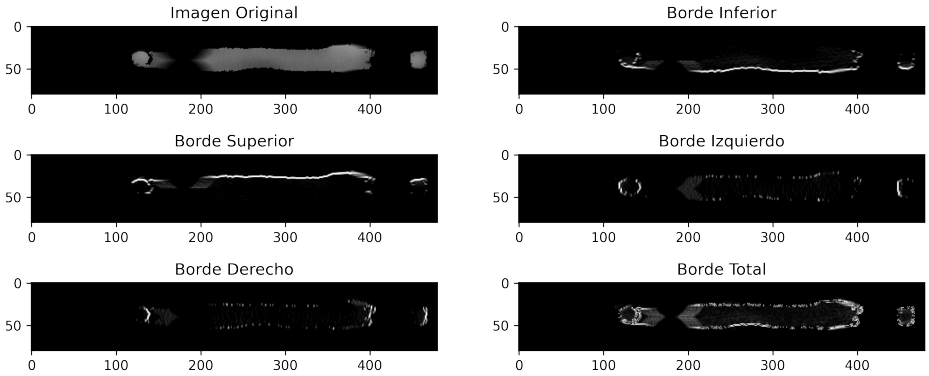


Figura 5.12: Ejemplo de convoluciones para detección de bordes sobre imagen del Dataset 2.

los bordes, gradientes, etc. Algunos de los más destacados son los operadores *Sobel* [50] o el operador discreto de Laplace [51].

Por último se van a presentar algunos ejemplos de mapas de activación de clase (en inglés, *Class Activation Map* (CAM)) [52] para los modelos Resnet-34 que se han entrenado en este experimento. Al realizar el entrenamiento es posible guardar el estado de una capa cualquiera del modelo. Para los mapas de activación de clase se guardará la última capa convolucional de la ResNet-34.

De esta forma se pueden obtener los *features* producidos en este última convolución y multiplicando cada píxel de este *feature* por el peso que después se le asigna a ese píxel, la entrada de las capas completamente conectadas como en la figura 4.3, se obtiene el mapa de activación de clase donde se puede observar las partes de la imagen que más han contribuido a que se la clasifique en la categoría que se le ha asignado.

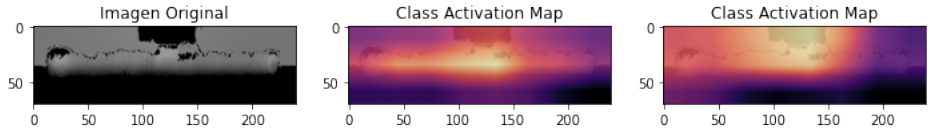


Figura 5.13: Ejemplo de mapas de activación sobre imagen del Dataset 1.

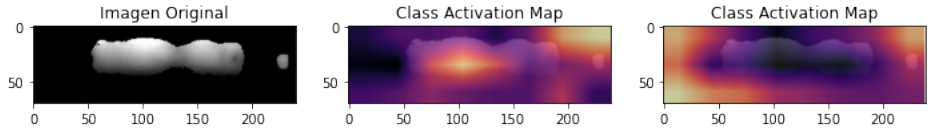


Figura 5.14: Ejemplo de mapas de activación sobre imagen del Dataset 2.

Se puede observar en la imagen central de la Figura 5.13 cómo los puntos de interés para la red coinciden con el la posición de la soldadura. En la imagen derecha de la misma figura se puede observar que en este caso ha tenido más relevancia la parte superior izquierda de la soldadura estando el foco en la parte central superior donde hay una sección rectangular vacía.

En la Figura 5.14 en cambio se puede ver que en la imagen central las activaciones se concentran sobre la soldadura misma pero también sobre la esquina derecha, y en la última imagen de esta figura son exclusivamente las esquinas las que han determinado la categoría. Como vemos el modelo no siempre clasifica de una manera intuitiva para un humano.

Los mapas de activación son útiles a la hora de realizar *debugging* sobre un modelo ya que permiten visualizar de forma clara las regiones que resultan de mayor importancia para la clasificación.

5.2.3. Arquitecturas

Una vez se ha realizado el experimento base se pueden realizar las modificaciones que se indicaron anteriormente. En esta sección se modificará la arquitectura del modelo para comprobar cómo influenciará esta modificación a los resultados.

5.2.3.1. Alexnet

Debido a la importancia histórica de esta arquitectura, esta supone la primera modificación que se probará. Alexnet [53] fue creada en 2012 y tuvo gran relevancia en el campo de Deep Learning ya que introdujo el uso de GPUs para su entrenamiento y consiguió resultados significativamente superiores al resto de implementaciones en el concurso "ImageNet Large Scale Visual Recognition Challenge". Superó por $10,2\,\%$ de *accuracy* al segundo puesto.

Compuesta por 5 capas convolucionales y 60 millones de parámetros no es una red muy profunda pero no obstante puede producir buenos resultados en un dataset como del que se dispone.

Como vemos en las tablas 5.4 y 5.4 los valores de *accurracy* que se obtienen son un poco mejores para el Dataset 2 y un poco peores para el Dataset 1. Lo que debemos tener en cuenta al observar la tabla 5.4 son las epochs necesarias para conseguir un resultado que ha sido peor que si se hubiese usado la arquitectura ResNet-34.

Por otro lado también se debe destacar que pese a que se han usado menos epochs para el segundo dataset el tiempo total ha sido de 4 minutos y 6 segundos, más que el tiempo usado para el primer dataset y bastante más que el usado en ambos casos mediante la arquitectura ResNet-34.

Por último haciendo la comparación entre los valores obtenidos para *Precision* y *F1*, estos son menores que en la arquitectura ResNet-34 lo que indica que pese a obtener mejor

Epochs	Coste Entren.	Coste Valid.	*Accuracy*	*Precision*	*F1*	Tiempo
0	0.351745	0.296439	0.865468	0.921053	0.272374	00:15
1	0.244465	0.253541	0.910072	0.670290	0.747475	00:14
2	0.162918	0.161781	0.933813	0.880240	0.761658	00:14
3	0.128058	0.157127	0.938129	0.754789	0.820833	00:14
4	0.091344	0.105320	0.963309	0.868421	0.885906	00:14
5	0.072286	0.094672	0.966906	0.881057	0.896861	00:14
6	0.054766	0.084473	0.969784	0.889868	0.905830	00:15
7	0.034284	0.085034	0.972662	0.898678	0.914798	00:15
8	0.051779	0.090943	0.964029	0.836653	0.893617	00:15
9	0.030898	0.086115	0.970504	0.893805	0.907865	00:14
10	0.030625	0.064118	0.976259	0.930556	0.924138	00:15
11	0.023718	0.066999	0.976259	0.915179	0.925508	00:15

Cuadro 5.4: Resultado del entrenamiento sobre el Dataset 1 mediante Transfer Learning de una arquitectura Alexnet

Epochs	Coste Entren.	Coste Valid.	*Accuracy*	*Precision*	*F1*	Tiempo
0	0.140589	0.224055	0.830000	0.965726	0.801003	01:02
1	0.143367	0.298342	0.866429	0.955595	0.851940	01:02
2	0.117903	0.514092	0.805714	1.000000	0.758865	01:02
3	0.103965	0.154422	0.943571	0.912351	0.945630	01:02

Cuadro 5.5: Resultado del entrenamiento sobre el Dataset 2 mediante Transfer Learning de una arquitectura Alexnet

accuracy este puede ser debido en mayor medida al dataset con el que se trabaja y no tanto al modelo en sí.

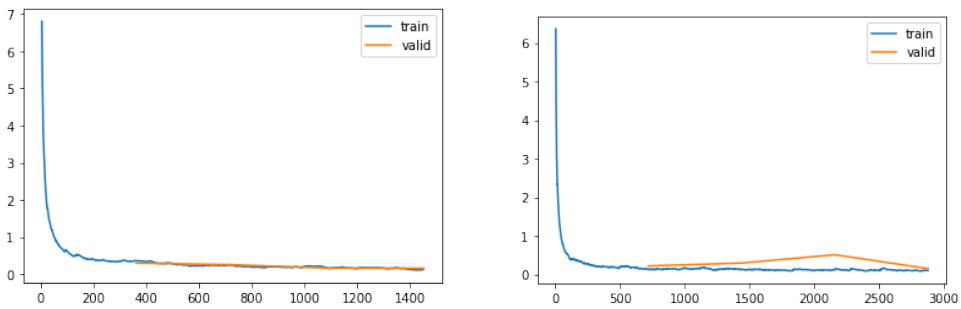


Figura 5.15: Gráficas de coste frente a número de batches evaluados para el Dataset 1 (*izda.*) y Dataset 2 (*dcha*) para la arquitectura Alexnet.

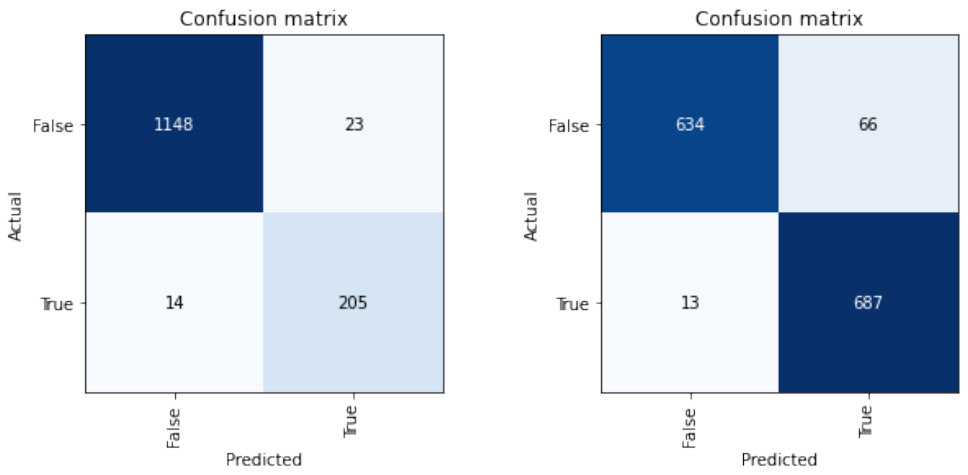


Figura 5.16: Matriz de confusión para el Dataset 1 (*izda.*) y Dataset 2 (*dcha*) en la arquitectura Alexnet.

En la figura 5.15 se puede ver el ligero incremento en el coste de validación al entrenar el Dataset 2 aunque este decrece a continuación y no resulta indicio suficiente de *overfitting*. En la Figura 5.16 se puede observar que en efecto el clasificador clasifica una cantidad elevada de imágenes de forma incorrecta, sobre todo indicando que las imágenes son correctas cuando en realidad no lo son.

5.2.3.2. Squeezenet

Derivada de la aquitectura Alexnet, Squeezenet fue creada en 2016 con el propósito de obtener una red con menos parámetros que entrenar, por tanto reduciendo el espacio que ocupa, pero manteniendo un rendimiento similar la de Alexnet.

Para conseguir esto Squeezenet usa tres estrategias. Reduce los *kernels* de las convoluciones de 3×3 a 1×1, es decir, *kernels* que evalúan cada píxel de uno en uno para reducir de esta forma los parámetros. Reduce el número de entradas en las capas convolucionales y usa *strides* mayores a 1 para mover el *kernel* un mayor número de unidades en la matriz.Estas técnicas si se aplican sin cuidado reducirán reducirán drásticamente el rendimiento de la red.

Epochs	Coste Entren.	Coste Valid.	*Accuracy*	*Precision*	*F1*	Tiempo
0	0.292407	0.256671	0.874101	0.978261	0.339623	00:10
1	0.155679	0.157326	0.943885	0.768061	0.838174	00:10
2	0.099644	0.157917	0.943885	0.747368	0.845238	00:10
3	0.062119	0.058490	0.982734	0.957746	0.944444	00:10
4	0.049992	0.038943	0.990647	0.963964	0.970522	00:10

Cuadro 5.6: Resultado del entrenamiento sobre el Dataset 1 mediante Transfer Learning de una arquitectura SqueezeNet

Epochs	Coste Entren.	Coste Valid.	*Accuracy*	*Precision*	*F1*	Tiempo
0	0.234516	0.532940	0.759286	0.989218	0.685341	00:19
1	0.151767	0.385971	0.808571	0.988688	0.765324	00:19
2	0.105862	0.235683	0.865000	0.982987	0.846216	00:19
3	0.101148	0.117174	0.964286	0.962963	0.964337	00:19

Cuadro 5.7: Resultado del entrenamiento sobre el Dataset 2 mediante Transfer Learning de una arquitectura SqueezeNet

Como se puede observar en la tabla 5.6 se ha conseguido un valor bastante alto de *accuracy* para el Dataset 1.

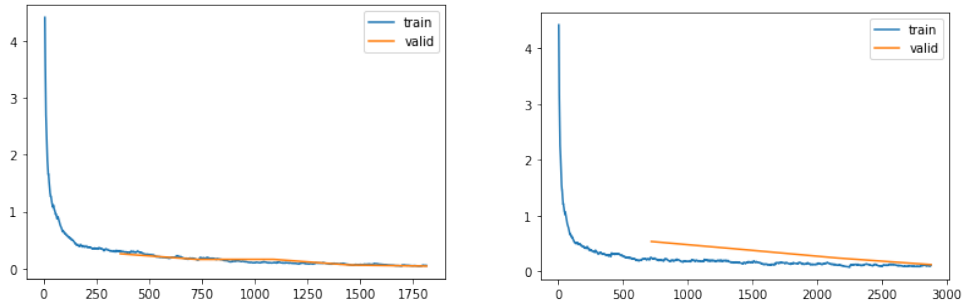


Figura 5.17: Gráficas de coste frente a número de batches evaluados para el Dataset 1 (*izda.*) y Dataset 2 (*dcha*) para la arquitectura SqueezeNet.

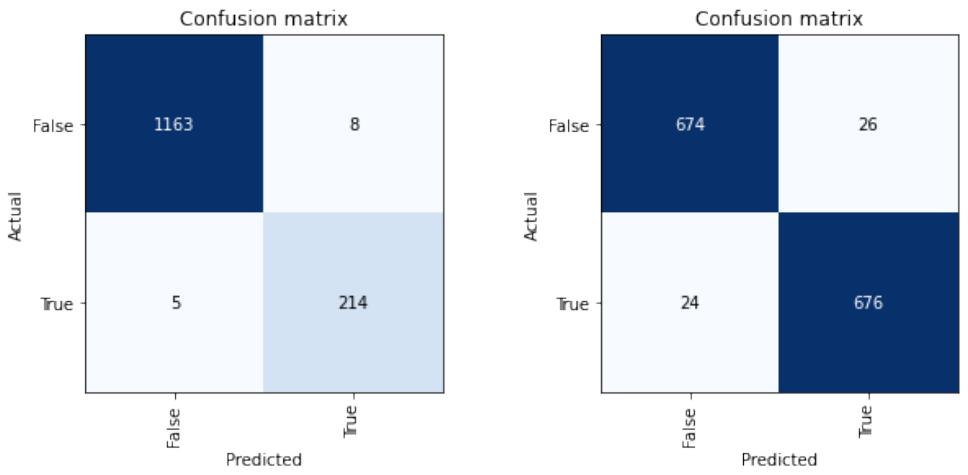


Figura 5.18: Matriz de confusión para el Dataset 1 (*izda.*) y Dataset 2 (*dcha*) en la arquitectura SqueezeNet.

5.2.3.3. Densenet

Epochs	Coste Entren.	Coste Valid.	*Accuracy*	*Precision*	*F1*	Tiempo
0	0.148052	0.023313	0.991367	1.000000	0.971831	00:44
1	0.075187	0.037015	0.989928	0.939914	0.969027	00:42
2	0.020780	0.004736	0.997842	0.986486	0.993197	00:43

Cuadro 5.8: Resultado del entrenamiento sobre el Dataset 1 mediante Transfer Learning de una arquitectura Densenet

Epochs	Coste Entren.	Coste Valid.	*Accuracy*	*Precision*	*F1*	Tiempo
0	0.156233	0.100949	0.965000	0.985097	0.964260	01:20
1	0.038815	0.040692	0.987143	0.991354	0.987088	01:20
2	0.026688	0.024610	0.992143	0.988652	0.992171	01:20

Cuadro 5.9: Resultado del entrenamiento sobre el Dataset 2 mediante Transfer Learning de una arquitectura Densenet

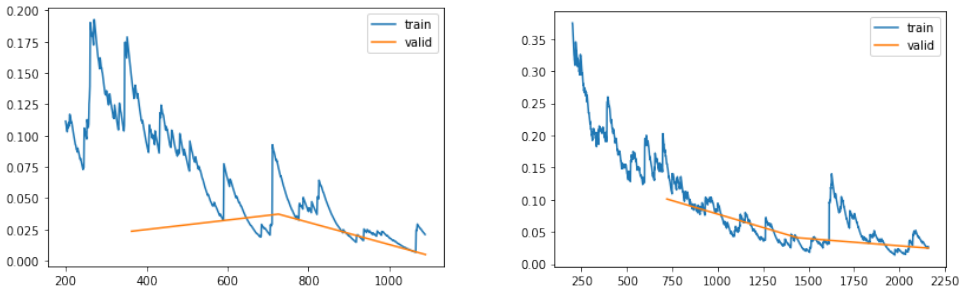


Figura 5.19: Gráficas de coste frente a número de batches evaluados para el Dataset 1 (*izda.*) y Dataset 2 (*dcha*) para la arquitectura Densenet.

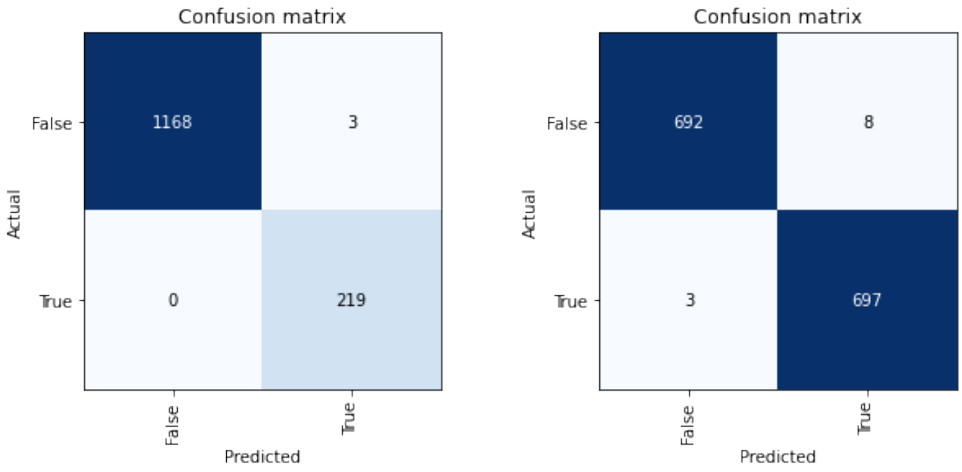


Figura 5.20: Matriz de confusión para el Dataset 1 (*izda.*) y Dataset 2 (*dcha*) en la arquitectura Densenet.

5.2.4. Funciones de activación

Al inicio del Capítulo 4 se introdujeron varias funciones de activación para las neuronas. Se vieron algunas ventajas y desventajas de cada función y los problemas que llevaron a introducir nuevas funciones de activación. En esta sección no se van a explorar todas las opciones posibles, al igual que en el caso anterior, debido a la gran cantidad de variantes que existen pero se usarán las mas representativas.

Como indicamos en el experimento base, la función de activación *Rectified Linear Unit* es la más habitual en los modelos de clasificación de imágenes. Algunas de las principales variantes son la *Leaky ReLU* y la tangente hiperbólica como se indicó en la tabla 4.1.

5.2.4.1. Leaky ReLU

La *Rectified Linear Unit* modifica ligeramente la *ReLU* original. Para ello añade una pendiente a la recta para los valores negativos. En este caso esta pendiente será $\alpha = 0{,}3$ al igual que en el ejemplo presentado en la tabla 4.1.

Epochs	Coste Entren.	Coste Valid.	*Accuracy*	*Precision*	*F1*	Tiempo
0	0.526406	0.286520	0.871942	0.691589	0.453988	00:19
1	0.413646	0.239462	0.911511	0.726415	0.714617	00:19
2	0.304298	0.198353	0.929496	0.776256	0.776256	00:20
3	0.349882	0.166995	0.933813	0.835979	0.774510	00:19
4	0.290800	0.142947	0.946043	0.875000	0.817518	00:19
5	0.207232	0.138282	0.951799	0.808943	0.855914	00:19
6	0.204663	0.119103	0.958273	0.877934	0.865741	00:19
7	0.190375	0.114922	0.961151	0.845188	0.882096	00:19

Cuadro 5.10: Resultado del entrenamiento sobre el Dataset 1 mediante Transfer Learning mediante la función de activatión *Leaky ReLU*

Epochs	Coste Entren.	Coste Valid.	*Accuracy*	*Precision*	*F1*	Tiempo
0	0.480763	0.476295	0.765000	0.892178	0.719523	00:35
1	0.430693	0.364718	0.835000	0.877617	0.825132	00:35
2	0.351411	0.309018	0.870000	0.832905	0.876861	00:36
3	0.369141	0.302007	0.870000	0.871060	0.869814	00:37
4	0.299343	0.308212	0.866429	0.907790	0.859293	00:36
5	0.302916	0.266288	0.887143	0.881690	0.887943	00:36
6	0.272487	0.258699	0.893571	0.884240	0.894848	00:36
7	0.304161	0.258005	0.881429	0.899701	0.878655	00:36

Cuadro 5.11: Resultado del entrenamiento sobre el Dataset 2 mediante Transfer Learning mediante la función de activación *Leaky ReLU*

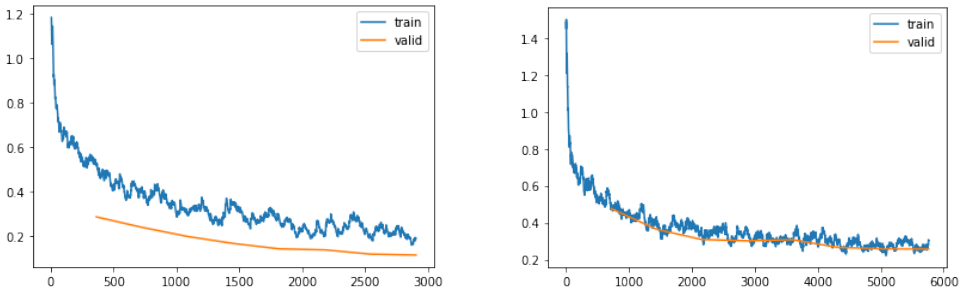


Figura 5.21: Gráficas de coste frente a número de batches evaluados para el Dataset 1 (*izda.*) y Dataset 2 (*dcha*) para la función de activación *Leaky ReLU*.

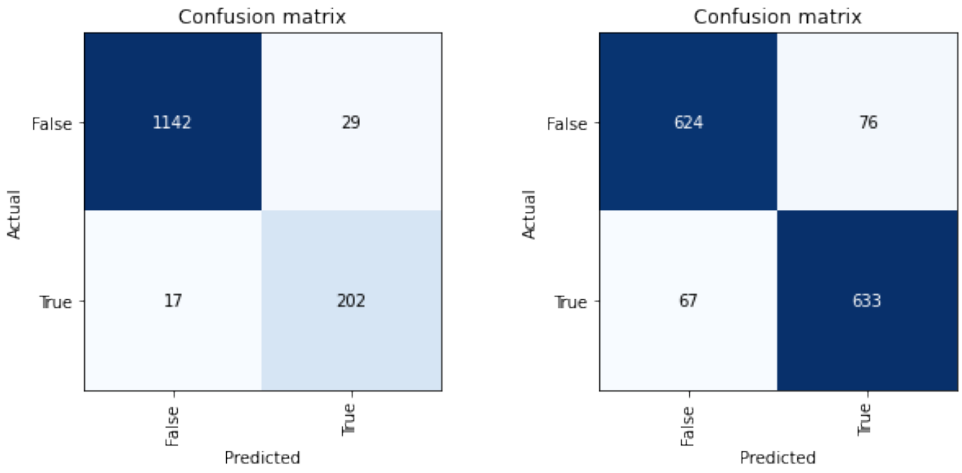


Figura 5.22: Matriz de confusión para el Dataset 1 (*izda.*) y Dataset 2 (*dcha*) para la función de activación *Leaky ReLU*.

5.2.5. Tangente Hiperbólica

La tangente hiperbólica tiene un planteamiento similar a la función sigmoidal pero con la modificación de que el rango de la tangente hiperbólica es $Rango(tanh(x)) = (-1, 1)$.

Epochs	Coste Entren.	Coste Valid.	*Accuracy*	*Precision*	*F1*	Tiempo
0	0.380454	0.336206	0.856115	0.524675	0.668874	00:19
1	0.334365	0.637376	0.702158	0.344992	0.511792	00:19
2	0.230722	0.128542	0.951079	0.810700	0.852814	00:19
3	0.217470	0.129883	0.939568	0.935484	0.775401	00:19
4	0.183844	0.088452	0.961870	0.915000	0.873508	00:19

Cuadro 5.12: Resultado del entrenamiento sobre el Dataset 1 mediante Transfer Learning mediante la función de activatión tangente hiperbólica.

Epochs	Coste Entren.	Coste Valid.	*Accuracy*	*Precision*	*F1*	Tiempo
0	0.487487	0.733565	0.654286	0.925197	0.492662	00:34
1	0.400536	0.574124	0.754286	0.888646	0.702936	00:34
2	0.466444	0.538625	0.767143	0.930876	0.712522	00:34
3	0.414304	0.477895	0.789286	0.943107	0.745030	00:34
4	0.342137	0.390516	0.805714	0.889091	0.782400	00:34
5	0.296038	0.389545	0.815714	0.931641	0.787129	00:34
6	0.331221	0.406116	0.797857	0.937107	0.759558	00:49
7	0.302105	0.322748	0.845714	0.924561	0.829921	00:50
8	0.293008	0.660999	0.722143	0.972644	0.621963	00:50
9	0.307370	0.325813	0.855714	0.788194	0.870844	00:50
10	0.268099	0.362787	0.827857	0.965517	0.797988	00:50
11	0.249897	0.295717	0.860000	0.943662	0.845426	00:50
12	0.254041	0.268182	0.872143	0.934891	0.862202	00:50

Cuadro 5.13: Resultado del entrenamiento sobre el Dataset 2 mediante Transfer Learning mediante la función de activatión tangente hiperbólica.

Como se puede observar en la tabla 5.12 el *Accuracy* obtenido es aproximadamente equivalente al obtenido mediante la función *Leaky ReLU* para el Dataset 1. Pero observando la tabla 5.13 se puede ver que fueron necesarias bastantes epochs más para conseguir el mismo resultado para el Dataset 2. Ninguno de estos resultados es excesivamente bueno y por tanto no hay razón aparente para sustituir la función de activación.

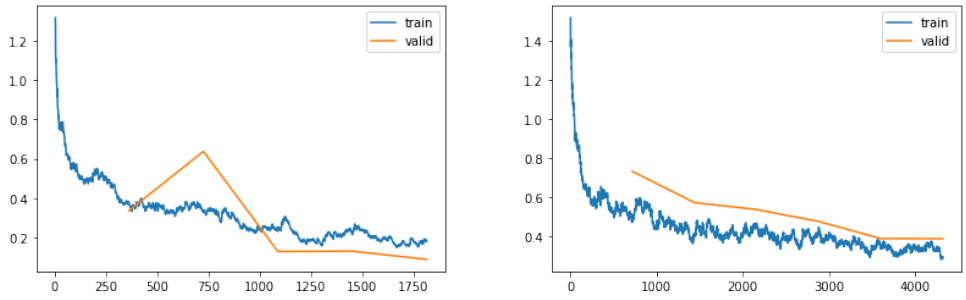


Figura 5.23: Gráficas de coste frente a número de batches evaluados para el Dataset 1 (*izda.*) y Dataset 2 (*dcha*) para la función de activación tangente hiperbólica.

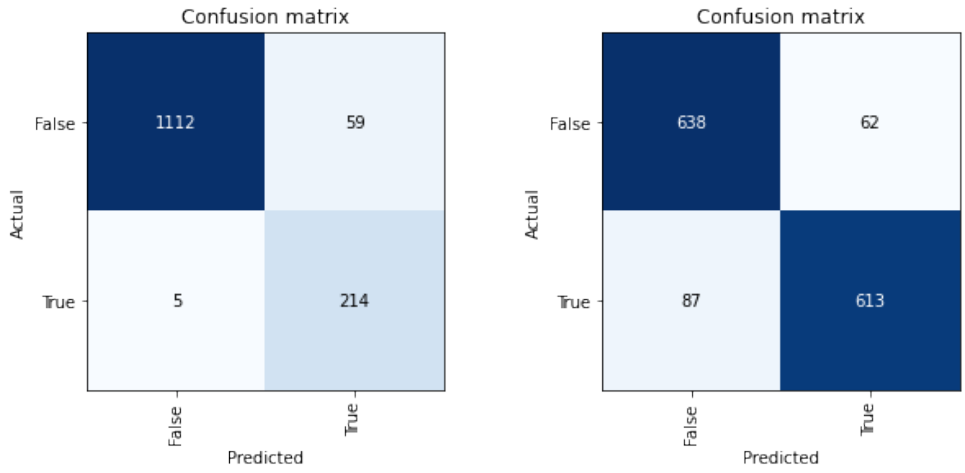


Figura 5.24: Matriz de confusión para el Dataset 1 (*izda.*) y Dataset 2 (*dcha*) para la función de activación tangente hiperbólica.

5.2.6. Optimizadores

En el capítulo anterior se indicó que el descenso de gradiente o su variante, el descenso de gradiente estocástico, no son las únicas funciones de optimización. Una de las principales variantes es el algoritmo Adam [12]. La principal diferencia con respecto del descenso de gradiente reside en la tasa de aprendizaje.

Si recordamos la forma en la que se aplica el descenso de gradiente,ecuación 4.14, se dispone de una tasa de aprendizaje η que es aplicada en cada peso que se quiera actualizar y esta tasa no se modifica durante el entrenamiento.Estas son exactamente las características que modifica Adam. Los pesos pasan a tener una tasa de aprendizaje propia que se adapta durante el entrenamiento.Las principales ventajas de Adam es que obtiene buenos resultados y los obtiene en un periodo de tiempo inferior a otros algoritmos. Por ello es el algoritmo más habitualmente usado frente a otras versiones de descenso de gradiente como AdaGrad o RMSProp.

5.2.7. Adam

Epochs	Coste Entren.	Coste Valid.	*Accuracy*	*Precision*	*F1*	Tiempo
0	0.269734	0.131270	0.961151	0.828685	0.885106	00:20
1	0.129166	0.087160	0.977698	0.931193	0.929062	00:20
2	0.150118	0.123427	0.976978	0.915556	0.927928	00:20
3	0.095234	0.084932	0.982734	0.941176	0.945455	00:20
4	0.111284	0.057855	0.982734	0.901235	0.948052	00:20

Cuadro 5.14: Resultado del entrenamiento sobre el Dataset 1 usando el optimizador Adam.

Epochs	Coste Entren.	Coste Valid.	*Accuracy*	*Precision*	*F1*	Tiempo
0	0.260515	0.300169	0.857857	0.921008	0.846332	00:38
1	0.184113	0.166343	0.937143	0.943478	0.936691	00:38
2	0.186013	0.188064	0.920000	0.985149	0.914242	00:39
3	0.075394	0.077928	0.970000	0.958217	0.970381	00:38
4	0.118693	0.056200	0.983571	0.976090	0.983700	00:40
5	0.133843	0.080815	0.980714	0.974612	0.980837	01:07
6	0.095562	0.066396	0.983571	0.968188	0.983837	01:07
7	0.030110	0.029938	0.991429	0.988636	0.991453	01:06

Cuadro 5.15: Resultado del entrenamiento sobre el Dataset 2 usando el optimizador Adam.

En las tablas 5.14 y 5.15 se puede observar que el resultado final obtenido en *Accuracy* es bastante bueno en especial para el Dataset 2 donde se obtienen valores cercanos a 1 para las métricas Precision y *F1*. Al comprobar la matriz de confusión 5.26 para el Dataset 2 podemos ver que el modelo ha fallado en un total de 12 soldaduras de un total de 1400.

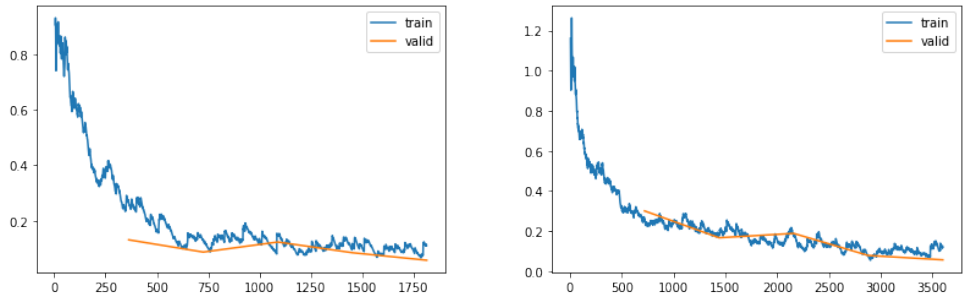


Figura 5.25: Gráficas de coste frente a número de batches evaluados para el Dataset 1 (*izda.*) y Dataset 2 (*dcha*) para el algoritmo de optimización Adam.

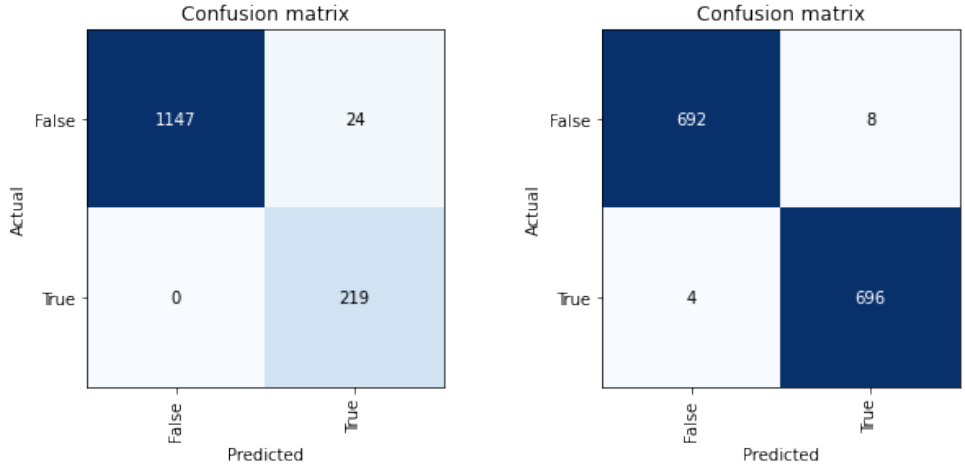


Figura 5.26: Matriz de confusión para el Dataset 1 (*izda.*) y Dataset 2 (*dcha*) para para el algoritmo de optimización Adam.

5.2.8. Integración con Pytorch

Uno de los aspectos de fastai que se comentaron el Capítulo 3 es que al ser una librería basada en Pytorch es posible implementar el modelo en Pytorch y al mismo tiempo usar las funcionalidades de fastai como normalización, métricas, herramientas de visualización, etc. De esta forma cualquier modelo definido en Pytorch puede ser entrenado de forma muy sencilla mediante la clase *Learner* de la que hay un ejemplo en el Listing 3.1.

Para ello se ha implementado en Pytorch un modelo muy sencillo pero con una gran relevancia histórica.La arquitectura LeNet fue creada 1989 por Yann LeCun y fue una de las primeras redes neuronales convolucionales.

```
class ModeloSimple(nn.Module):
  def __init__(self):
    super(ModeloSimple, self).__init__()
    # 1 canal de entrada, 6 canales de salida, kernels de 5x5 que
    # se aplican en la convolucion
    self.conv1 = nn.Conv2d(3, 6, 5)
    self.conv2 = nn.Conv2d(6, 16, 5)
    # (80/2)/2 - 3 = 17
    # (480/2)/2 - 3 = 17
    self.fc1 = nn.Linear(16 * 17 * 117 , 120)
    self.fc2 = nn.Linear(120, 10)
    # Al tener 2 categorias la ultima capa debe reducirse
    # a tamano 2
    self.fc3 = nn.Linear(10, 2)

  def forward(self, x):
    # En este caso el max pooling se hace a traves de una ventana de 2x2
    x = F.max_pool2d(F.relu(self.conv1(x)), (2, 2))
    x = F.max_pool2d(F.relu(self.conv2(x)), 2)
    #Se necesita aplanar el tensor en una dimension
    x = torch.flatten(x, 1)
    x = F.relu(self.fc1(x))
    x = F.relu(self.fc2(x))
    x = self.fc3(x)
    return x
```

Listing 5.1: Implementación de la arquitectura LeNet en Pytorch

Epochs	Coste Entren.	Coste Valid.	*Accuracy*	*Precision*	*F1*	Tiempo
0	0.467757	0.424918	0.763699	0.320513	0.345125	00:14
1	0.282258	0.380975	0.787360	0.341880	0.319043	00:14
2	0.251809	0.300623	0.857410	0.570128	0.577491	00:14
3	0.204535	0.241626	0.883562	0.749226	0.564103	00:14
4	0.135226	0.253853	0.891968	0.735000	0.628877	00:15
5	0.071201	0.275354	0.901930	0.700730	0.709141	00:15
6	0.055140	0.317369	0.896949	0.654079	0.723475	00:16
7	0.032290	0.293445	0.909091	0.717352	0.733090	00:15
8	0.026392	0.309668	0.907534	0.708042	0.731707	00:15

Cuadro 5.16: Resultado del entrenamiento sobre el Dataset 1 usando la arquitectura LeNet implementada en Pytorch.

5.2.9. Tensorboard

Esta herramienta ya fue mencionada en el Capítulo 3 y por lo tanto es esta sección solo se mostrarán algunos de los resultados obtenidos. Debido a que es una herramienta de visualización principalmente, mediante Tensorboard puede ser usado para construir gráficas con los resultados obtenidos de un modelo, representaciones de las imágenes clasificadas, etc.

En este caso en primer lugar se ha usado la herramienta de visualización de grafos para

representar la arquitectura LeNet implementada en la sección anterior. Poder visualizar estos grafos muchas veces supone una gran ayuda a la hora de realizar *debugging*.

Por otro lado se ha usado la herramienta *Projector* para obtener una representación visual de las imágenes clasificadas donde la distancia entre las imágenes es representativa de la similitud encontrada entre ellas. Para este propósito se usan las activaciones de la última capa de la red.

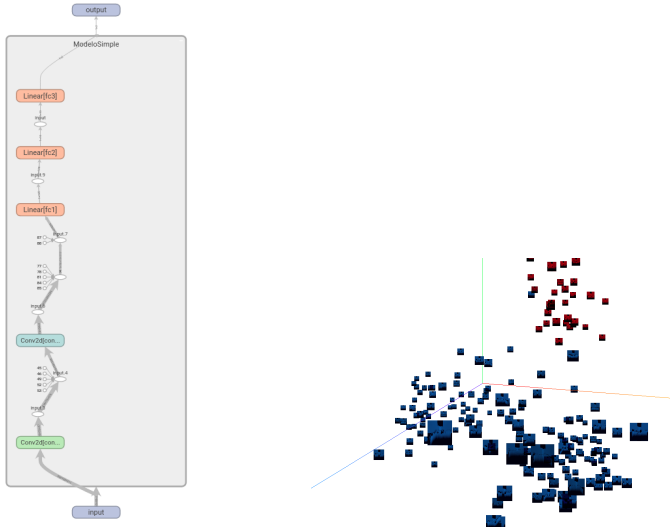


Figura 5.27: Grafo de arquitectura LeNet (*izda*) y representación en 3D de las imágenes clasificadas (*dcha*).

5.3. Arquitectura Resnet

En la sección anterior se ha trabajado con varias arquitecturas para los modelos. Se ha usado Transfer Learning para entrenar las últimas capas del modelo acorde a los datos de los que se dispone en el dataset. Una vez hecho esto hemos podido comprobar que la arquitectura ResNet-34 puede proporcionar resultados bastante buenos a la hora de clasificar las imágenes y además realizar el entrenamiento en un tiempo aceptable.

Pero es posible que no se desee realizar Transfer Learning y que deseemos implementar nosotros mismos la arquitectura que se va a usar y entrenar el modelo por completo, sin que este haya sido entrenado en otro dataset previamente. Con el dataset que se dispone esto no sería aconsejable debido a que probablemente se obtendrían resultados peores y además se necesitaría bastante más tiempo debido que hay que entrenar la red por completo.

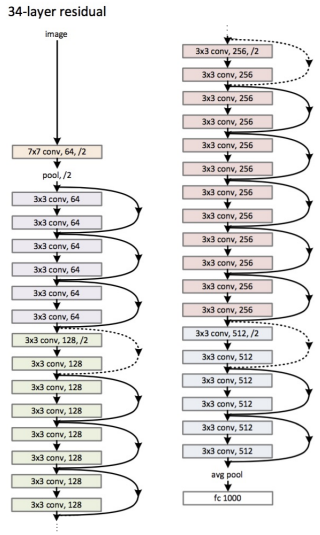


Figura 5.28: Arquitecura ResNet-34. Fuente: [54]

Puede haber múltiples razones por las que se requiera implementar la arquitectura. Los modelos que están entrenados sobre ImageNet son limitados y entre ellos no se encuentran todas las arquitecturas que existen y por tanto estas deberán ser implementadas y entrenadas desde cero.

Por otro lado se debe tener en cuenta el propósito para el que ha sido entrenada la red, clasificación de imágenes. Para cualquier otro propósito no es posible usar los pesos obtenidos mediante Transfer Learning y por tanto la red debe ser entrenada por completo.

En esta sección se van a mostrar los resultados obtenidos del entrenamiento de la implementación de una arquitectura ResNet-34 como la que se puede observar en la Figura 5.28.

Pero antes de mostrar los resultados se describirán brevemente los bloques residuales de cuyo nombre proviene ResNet y la forma en la que se han inicializado de antemano los pesos.

5.3.1. Inicialización Kaiming

Puede parecer contradictorio inicializar los pesos de una red neuronal. ¿No es el objetivo mismo del entrenamiento producir los valores óptimos para estos? Esto es correcto pero debemos pensar en lo que ocurre al dejar que todas las actualizaciones de los pesos se realicen en el entrenamiento. Si cada peso tiene un valor aleatorio entonces se necesitarán un mayor número de iteraciones para obtener los mejores valores posibles.

Además de un mayor tiempo de entrenamiento una inicialización poco cuidadosa de los pesos puede provocar que se manifiesten los problemas de desvanecimiento de gradiente y gradiente explosivo. Esto provocaría gradientes con valores demasiado elevados o demasiado bajos como para ser útiles en el entrenamiento ya que si recordamos la ecuación 4.14, el gradiente determina la actualización del peso.

Un gradiente demasiado bajo anularía la actualización y obtendríamos el mismo valor, por lo tanto el entrenamiento sería inútil. Un gradiente demasiado alto resultaría en un nuevo peso con valor 0.La inicialización Kaiming evita en gran medida estos problemas. Para ello los pesos con los que se va a inicializar la red deben cumplir la siguiente condición:

$$\frac{1}{2}n^l \mathrm{Var}[w^l] = 1,\ \forall l \tag{5.7}$$

$$w_i^l \sim \mathcal{N}(0, \frac{2}{n^l}) \tag{5.8}$$

En la ecuación 5.7 n_l indica el número de conexiones entre la capa $l-1$ y la capa l y w_l es una variable aleatoria que determina los pesos en esa capa. En la ecuación 5.8 podemos observar cómo cada peso es obtenido de una distribución normal de media $\mu = 0$ y varianza $\sigma^2 = \frac{2}{n_l}$.

5.3.2. Bloques residuales

Los bloques residuales son la principal aportación de la arquitectura ResNet. Como ya se describió en el Capítulo 4 un modelo pretende realizar el rol de una función donde a cada entrada le es asignada una salida. De esta forma el modelo, de forma simplificada, realiza $M(x) = y$. En cada bloque de la red tendremos de forma similar una función $f(x)$ que deberá optimizar el resultado para su entreda x. Pero lo bloques residuales usan la función $f(x) - x$ que es más fácil de optimizar. Una vez optimizada esta parte se suma con la entrada del bloque de forma que la salida obtenida pasa a ser $f(x) - x + x = f(x)$, al igual que en los bloques normales.

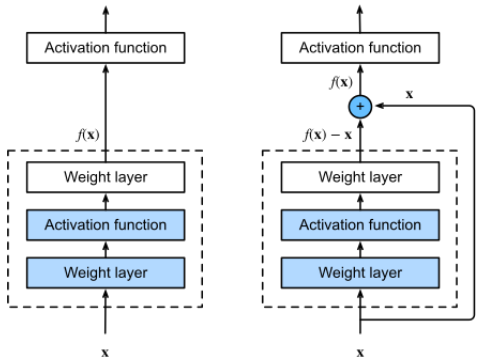


Figura 5.29: Bloque normal (*izda*) y bloque residual (*dcha*) usados en la arquitectura ResNet-34. Fuente: [55]

En la Figura 5.29 se puede observar que el bloque residual descompone la entrada y la suma al final del mismo, esta es la denominada conexión residual.

```
class ResidualBlock(nn.Module):
  def __init__(self, canales_entrada, canales_salida, stride=1):
    super().__init__()
    self.canales_entrada = canales_entrada
    self.canales_salida = canales_salida
    self.operaciones = nn.Sequential(
      nn.Conv2d(canales_entrada, canales_salida, stride=stride,
      kernel_size=3, padding=1, bias=False),
      nn.BatchNorm2d(canales_salida),
      nn.ReLU(),
      nn.Conv2d(canales_salida, canales_salida, stride=1,
      kernel_size=3, padding=1, bias=False),
      nn.BatchNorm2d(canales_salida)
      )
    self.res = []


    if stride != 1:
      self.res = [nn.AvgPool2d(kernel_size = 2, stride=stride, ceil_mode=True)
    ] + self.res

    if canales_entrada != canales_salida:
      self.res = self.res + [nn.Conv2d(canales_entrada, canales_salida,
      kernel_size=1, padding=0, bias=False),
      nn.BatchNorm2d(canales_salida)]


    self.res = nn.Sequential(*self.res)

  def forward(self, entrada):
    a = self.operaciones(entrada)
    b = self.res(entrada)
    return nn.ReLU()(a + b)
```

Listing 5.2: Implementación de un bloque residual

5.3.3. Resultados

Epochs	Coste Entren.	Coste Valid.	*Accuracy*	*Precision*	*F1*	Tiempo
0	0.184297	0.358088	0.866255	0.565714	0.603659	01:18
1	0.224891	0.254476	0.887860	0.608911	0.692958	01:18
2	0.214195	0.255739	0.912551	0.717949	0.724919	01:18
3	0.119537	0.144518	0.943416	0.922414	0.795539	01:18
4	0.046317	0.132362	0.967078	0.890323	0.896104	01:18
5	0.088215	0.063346	0.981481	0.965517	0.939597	01:18
6	0.032694	0.045827	0.985597	0.972789	0.953333	01:18
7	0.038589	0.039887	0.986625	0.943038	0.958199	01:18
8	0.008905	0.038573	0.987654	0.949045	0.961290	01:18
9	0.018976	0.039797	0.986625	0.948718	0.957929	01:18

Cuadro 5.17: Resultado del entrenamiento sobre el Dataset 1 mediante una red ResNet-34 implementada y entrenada exclusivamente sobre el dataset.

Epochs	Coste Entren.	Coste Valid.	*Accuracy*	*Precision*	*F1*	Tiempo
0	0.516891	0.469428	0.830635	0.200000	0.010909	02:50
1	0.351829	0.276965	0.911582	0.779510	0.711382	02:40
2	0.227684	0.183086	0.936488	0.976945	0.768707	02:41
3	0.184860	0.163082	0.940847	1.000000	0.784091	02:43
4	0.170361	0.175334	0.929328	0.901042	0.752992	02:43
5	0.090327	0.144145	0.919054	0.757009	0.757009	02:43
6	0.112840	0.122089	0.937422	0.956284	0.776915	02:43
7	0.094076	0.121067	0.923724	0.757092	0.777070	02:43
8	0.093862	0.120873	0.925903	0.782857	0.775472	02:43
9	0.103361	0.119527	0.926837	0.765018	0.786558	02:44

Cuadro 5.18: Resultado del entrenamiento sobre el Dataset 2 mediante una red ResNet-34 implementada y entrenada exclusivamente sobre el dataset

En las tablas 5.17 y 5.17 se puede observar como el aprendizaje es mas lento en estos casos. Pese a obtener un buen resultado en el Dataset 1, en el Dataset 2 se obtiene un resultado bastante peor que mediante otros experimentos. Además de este resultado vemos como el tiempo de entrenamiento es muy superior a los casos en los que se usa Transfer Learning.

En las figuras 5.32 y 5.33 se pueden ver los resultados que se obtienen al introducir algunas imágenes en la arquitectura ResNet. Estos son algunos de los ejemplos mediante los que la red determina los features. En la Figura 5.32 estos pueden ser más claros para la vista humana pero a medida se realizan convoluciones sobre estas imágenes los features se vuelven más abstractos y las imágenes mas pequeñas.

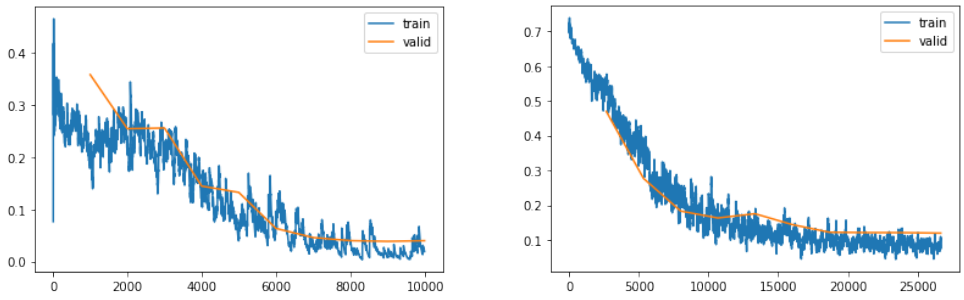


Figura 5.30: Gráficas de coste frente a número de batches evaluados para el Dataset 1 (*izda.*) y Dataset 2 (*dcha*) usando la implementación ResNet-34.

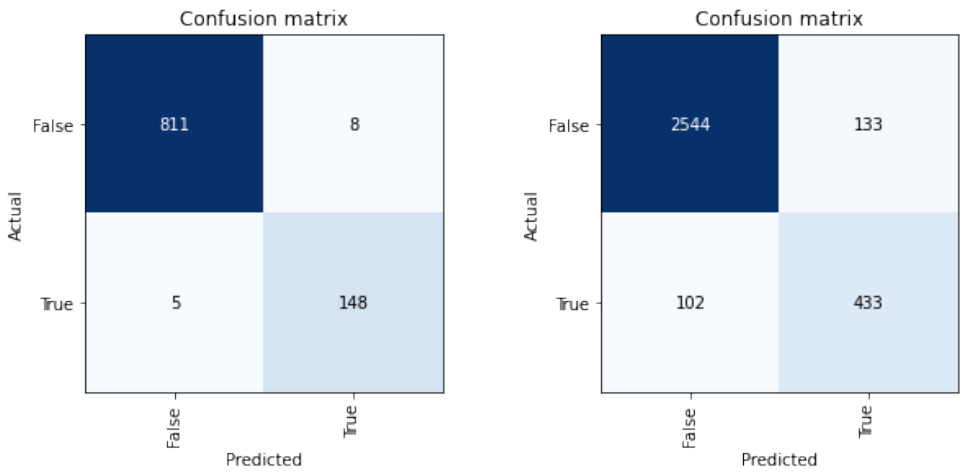


Figura 5.31: Matriz de confusión para el Dataset 1 (*izda.*) y Dataset 2 (*dcha*) usando la implementación de ResNet-34.

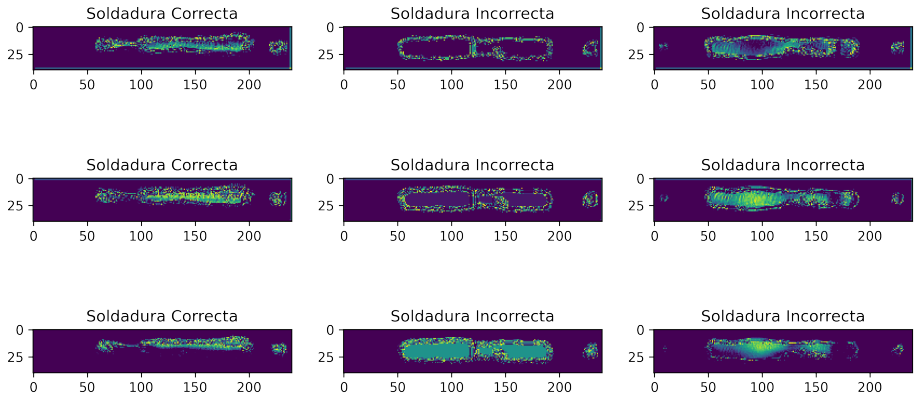


Figura 5.32: Imágenes transformadas mediante el primer nivel de la arquitectura ResNet-34.

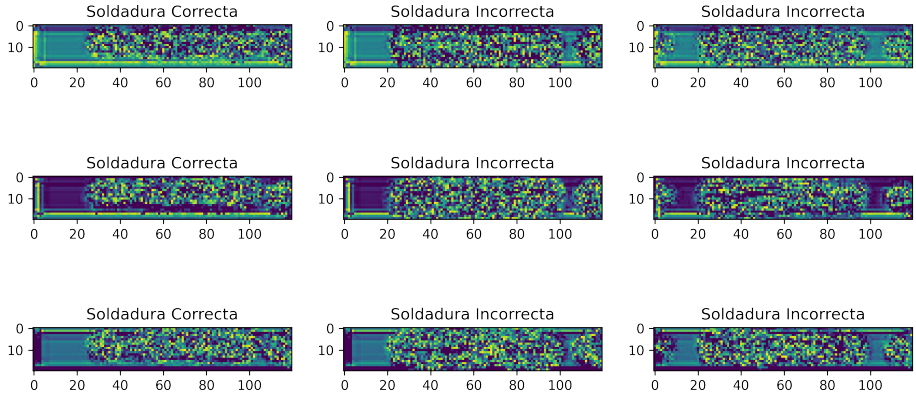


Figura 5.33: Imágenes transformadas mediante el segundo nivel de la arquitectura ResNet-34

5.4. Arquitectura Xception

Como ya se mencionó el la última sección, existe una gran variedad de arquitecturas y no todas son entrenadas de antemano y pueden ser usadas para realizar Transfer Learning.En la sección anterior se ha implementado una arquitectura ya conocida y por tanto en esta sección se implementará una arquitectura relativamente reciente.

La arquitecura Xception [56] fue diseñada en 2017 por François Chollet, creador de la librería de aprendizaje automático *Keras*, y supone una modificación de la arquitectura Inception. Las modificaciones intentan conseguir una disminución de los parámetros, con un objetivo similar a las arquitectura SqueezeNet, mediante las convoluciones separables en profundidad y los bloques Inception.

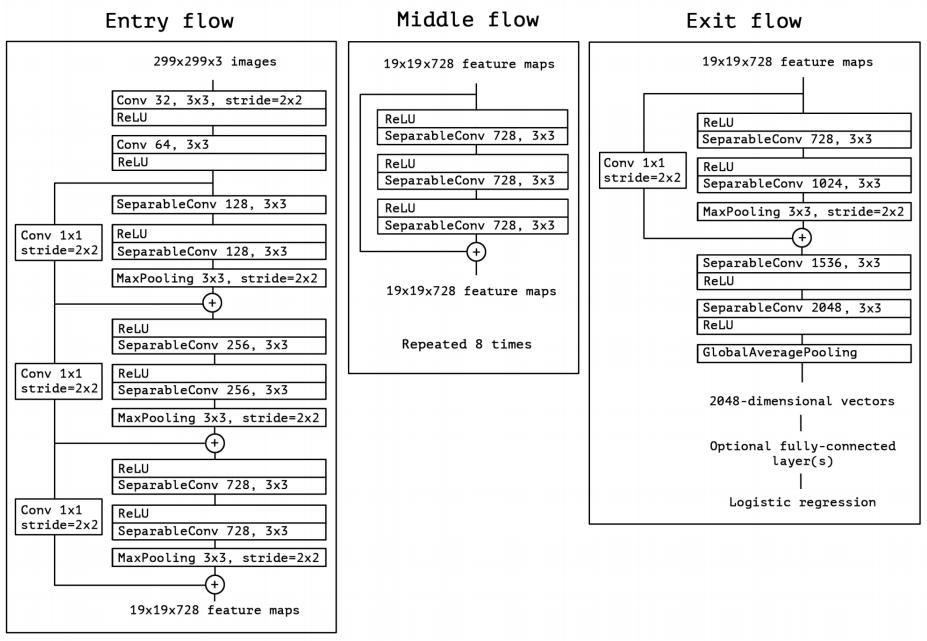


Figura 5.34: Arquitectura Xception. Fuente: [56]

En la Figura 5.34 se puede observar de izquierda a derecha las etapas por las que pasan los tensores de las imágenes. Cada uno de los bloques en los que se usan las convoluciones separables en profundidad tiene una estructura residual, en algunos de estos bloques las conexiones residuales se forman usando convoluciones con *kernels* 1×1 y *strides* 2×2 que recordemos reducen el tamaño del feature a la mitad.

5.4.1. Convoluciones separables en profundidad

Hasta este momento se ha estado usando el modelo de convoluciones explicadas en el Capitulo 4 pero esta no es la única forma de realizar esta operación. La idea básica detrás

de las convoluciones separables es que un kernel puede ser separado en dos kernels de menor dimensión mediante la multiplicación de matrices.

$$\begin{bmatrix} 3 & 6 & 9 \\ 4 & 8 & 12 \\ 5 & 10 & 15 \end{bmatrix} = \begin{bmatrix} 3 \\ 4 \\ 5 \end{bmatrix} \times \begin{bmatrix} 1 & 2 & 3 \end{bmatrix} \tag{5.9}$$

En la ecuación 5.9 se puede observar como el kernel es descompuesto en dos kernels separados que son aplicados de forma independiente. De esta forma se ejecutan 6 multiplicaciones frente a las 9 que se requerían antes para aplicar la convolución. Esto funciona exclusivamente en los casos en los que el kernel puede ser separado de esta manera, ya que no siempre es así.

Las convoluciones separables en profundidad se separan en dos etapas. En la primera se realiza una convolución en profundidad. Si recordamos cada imagen RGB está representada por un tensor de profundidad 3. De esta forma al realizar una convolución sobre la imagen se aplica un kernel de profundidad 3, donde cada uno de los kernels de profundidad 1 es aplicado a un solo canal de la imagen.

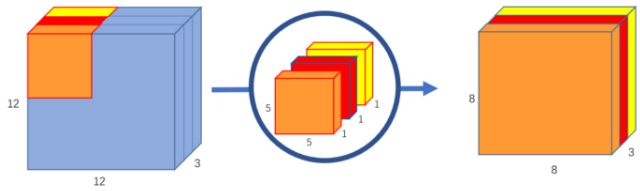


Figura 5.35: Convolución en profundidad. Fuente: [57]

En la Figura 5.35 vemos el procedimiento descrito en esta primera parte. Una vez se dispone de este tensor se aplican tantas convoluciones, n, como features se quiera obtener. El kernel en este caso es de tamaño $1 \times 1 \times 3$. En la Figura 5.36 se puede observar el procedimiento de esta segunda parte del proceso.

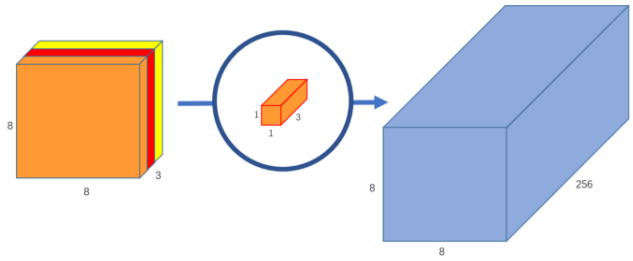


Figura 5.36: Convolución *pointwise*. Fuente: [57]

Si usamos los tamaños proporcionados en las imágenes y queremos obtener 256 features sobre una imagen RGB de tamaño 12×12 se realizarán $5 \times 5 \times 3 \times 8 \times 8$ operaciones en la

primera fase y $3 \times 256 \times 8 \times 8$ en la segunda sumando un total de 53,952 operaciones. Pese a que el tamaño de la imagen es 12 en la primera fase supondremos que no hay padding y por tanto se realizan solo 8 movimientos. Si en cambio hubiésemos aplicado la convolución normal habríamos necesitado $256 \times 5 \times 5 \times 3 \times 8 \times 8 = 1{,}228{,}800$ operaciones.

```
class SeparableConvolution(nn.Module):
  def __init__(self, canales_entrada, canales_salida, tam_kernel = 3,
           padding = 1, bias= False):
    super(SeparableConvolution, self).__init__()
    self.prof = nn.Conv2d(canales_entrada, canales_entrada,
                kernel_size=tam_kernel,
                padding=padding,
                groups=canales_entrada,
                bias=bias)

    self.indiv = nn.Conv2d(canales_entrada, canales_salida,
                 kernel_size=1,
                 bias=bias)

  def forward(self, x):
    salida = self.prof(x)
    salida = self.indiv(salida)
    return salida
```

Listing 5.3: Implementación de la convolución separable en profundidad.

5.4.2. Resultados

Epochs	Coste Entren.	Coste Valid.	*Accuracy*	*Precision*	*F1*	Tiempo
0	0.207981	0.278268	0.893004	0.929825	0.504762	02:21
1	0.293328	0.204756	0.922840	0.943182	0.688797	02:22
2	0.207463	0.276800	0.925926	0.795620	0.751724	02:22
3	0.126168	0.130882	0.961934	0.946154	0.869258	02:22
4	0.056194	0.082213	0.981481	0.919255	0.942675	02:23
5	0.027386	0.051235	0.990741	0.961538	0.970874	02:22
6	0.015694	0.001314	0.998971	0.993506	0.996743	02:22
7	0.001512	0.029305	0.993827	0.962264	0.980769	02:23
8	0.000049	0.004463	0.998971	0.993506	0.996743	02:23
9	0.000076	0.003380	0.997942	0.993464	0.993464	02:23

Cuadro 5.19: Resultado del entrenamiento sobre el Dataset 1 mediante una red Xception implementada y entrenada exclusivamente sobre el dataset.

Epochs	Coste Entren.	Coste Valid.	*Accuracy*	*Precision*	*F1*	Tiempo
0	0.404314	0.311186	0.832503	0.491018	0.233618	07:23
1	0.240169	0.292848	0.891345	0.841912	0.567534	07:22
2	0.167310	0.188567	0.926837	0.977707	0.723204	07:24
3	0.167255	0.153510	0.932752	0.907928	0.766739	07:24
4	0.054878	0.203926	0.882005	0.587054	0.735150	07:25
5	0.028047	0.188031	0.959838	0.809451	0.891688	07:25
6	0.032658	0.031891	0.991283	0.953488	0.974406	07:25
7	0.009228	0.011756	0.996264	0.981584	0.988868	07:25
8	0.001768	0.013319	0.995953	0.981550	0.987929	07:25
9	0.000897	0.014044	0.995641	0.976234	0.987061	07:26

Cuadro 5.20: Resultado del entrenamiento sobre el Dataset 2 mediante una red Xception implementada y entrenada exclusivamente sobre el dataset.

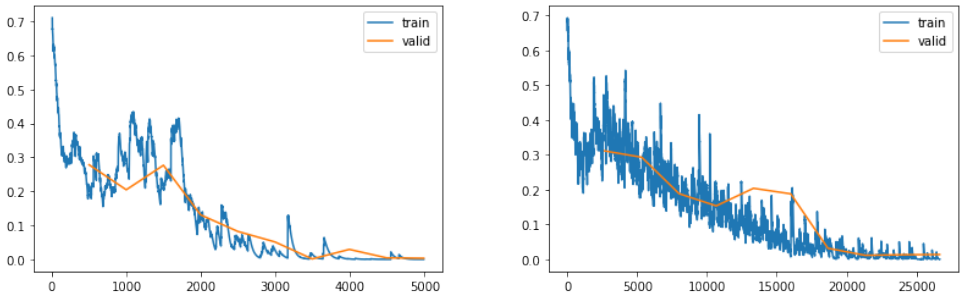


Figura 5.37: Gráficas de coste frente a número de batches evaluados para el Dataset 1 (*izda.*) y Dataset 2 (*dcha*) usando la implementación Xception.

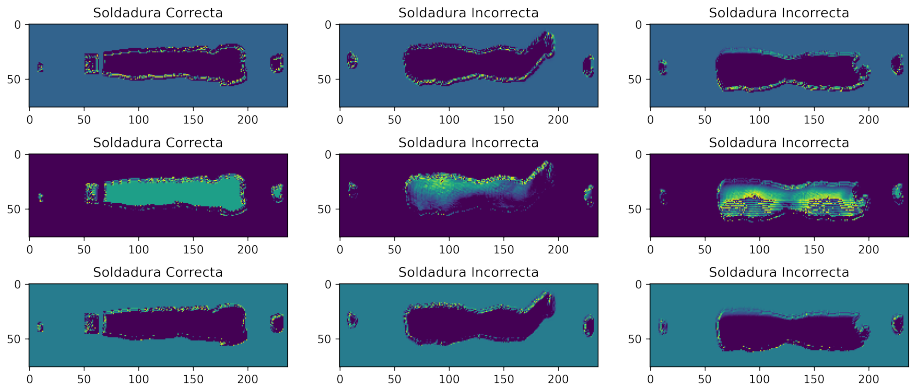


Figura 5.39: Imágenes transformadas mediante el Bloque 1.

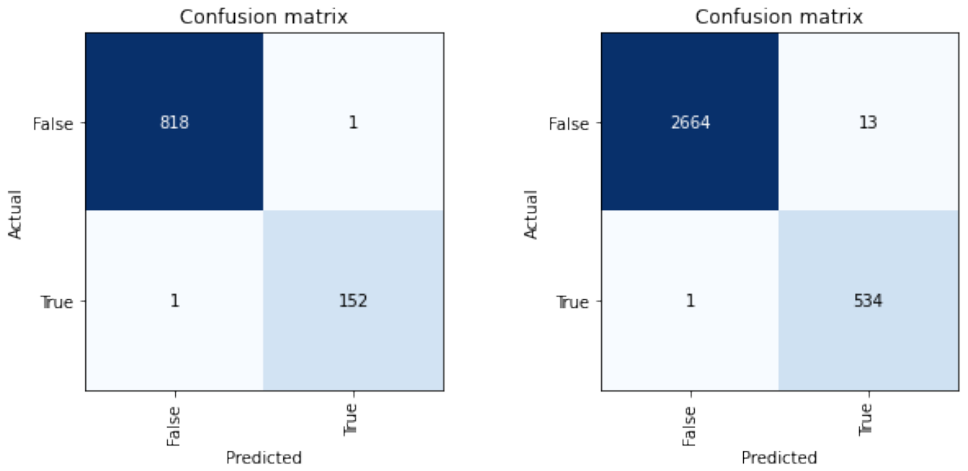


Figura 5.38: Matriz de confusión para el Dataset 1 (*izda.*) y Dataset 2 (*dcha*) usando la implementación de Xception.

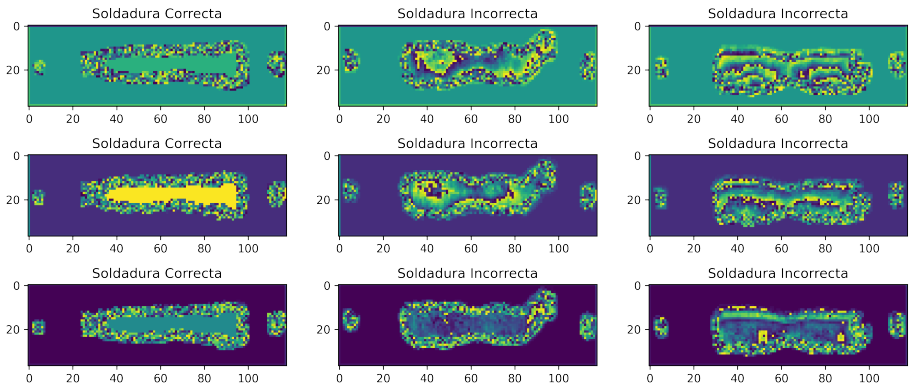


Figura 5.40: Imágenes transformadas mediante bloques residuales con convoluciones separables.

5.5. Análisis comparativo

Por último en esta sección se compararán los resultados obtenidos mediante los distintos métodos para comprobar cuál sería la configuración óptima.

En la tabla 5.21 se puede observar que para el Dataset 1 se han obtenido los mejores

Experimento	Coste Entren.	Coste Valid.	*Accuracy*	*Precision*	*F1*
TL-ResNet34	0.063857	0.073505	0.984892	0.938053	0.952809
TL-Alexnet	0.023718	0.066999	0.976259	0.915179	0.925508
TL-Squeezenet	0.049992	0.038943	0.990647	0.963964	0.970522
TL-Densenet	0.026688	0.024610	0.992143	0.988652	0.992171
FA-LeakyReLU	0.190375	0.114922	0.961151	0.845188	0.882096
FA-TanH	0.183844	0.088452	0.961870	0.915000	0.873508
OPT-Adam	0.111284	0.057855	0.982734	0.901235	0.948052
Resnet	0.018976	0.039797	0.986625	0.948718	0.957929
Xception	**0.000076**	**0.003380**	**0.997942**	**0.993464**	**0.993464**

Cuadro 5.21: Comparación entre los resultados obtenidos en los distintos experimentos para el Dataset 1 donde TL(*Transfer Learning*), FA(*Funciones de Activación*) y OP(*Optimizador*).

valores mediante la arquitectura Xception entrenada por completo sobre el Dataset 1 sin realizar Transfer Learning aunque por un margen muy pequeño ya que los modelos Squeezenet y sobre todo Densenet entrenados mediante Transfer Learning obtienen unos valores muy similares de *Accuracy*.

Experimento	Coste Entren.	Coste Valid.	*Accuracy*	*Precision*	*F1*
TL-ResNet34	0.191354	0.184898	0.927143	0.942308	0.925872
TL-Alexnet	0.103965	0.154422	0.943571	0.912351	0.945630
TL-Squeezenet	0.101148	0.117174	0.964286	0.962963	0.964337
TL-Densenet	0.026688	0.024610	0.992143	**0.988652**	**0.992171**
FA-LeakyReLU	0.304161	0.258005	0.881429	0.899701	0.878655
FA-TanH	0.254041	0.268182	0.872143	0.934891	0.862202
OPT-Adam	0.030110	0.029938	0.991429	0.988636	0.991453
Resnet	0.103361	0.119527	0.926837	0.765018	0.786558
Xception	**0.000897**	**0.014044**	**0.995641**	0.976234	0.987061

Cuadro 5.22: Comparación entre los resultados obtenidos en los distintos experimentos para el Dataset 2 donde TL(*Transfer Learning*), FA(*Funciones de Activación*) y OP(*Optimizador*).

En la tabla 5.22 se puede observar que de nuevo la arquitectura Xception ha obtenido los mejores resultados en cuanto a *Accuracy* aunque en cuanto a *Precision* y F1 se han obtenido los mejores valores en la arquitectura Densenet, la cual tiene el segundo mejor valor en cuanto a *Accuracy*.

Una vez vistos los resultados podemos se ha comprobado que para este dataset la mejor arquitectura en Xception. En cuanto a Transfer Learning frente a entrenar la red por completo no hay una decisión clara ya que mediante ambos métodos se pueden obtener resultados buenos.En este caso los resultados indican que es mejor entrenar el modelo por completo pero la diferencia con los mejores resultados de Transfer Learning es mínima.

En cuanto a las funciones de activación se ha visto como los resultados empeoran si

no se usa la función de activación ReLU y por tanto esta es la mejor opción.En cuanto al optimizador, hay ocasiones en las que hay grandes mejores con respecto al rendimiento y el tiempo necesario para entrenar el modelo. Por lo tanto es recomendable hacer uso de Adam en lugar de descenso de gradiente estocástico.

Capítulo 6

Conclusiones y lineas futuras

Al comienzo de este Trabajo de Fin de Grado, en el Capítulo 1, se indicaron una serie de objetivos a cumplir a los largo de proyecto. A continuación se comprobará si se ha cumplido cada uno de estos objetivos.

1. **Aprendizaje teórico de los conceptos básicos usados en Aprendizaje Automático:** Este TFG ha tenido un componente teórico muy amplio. El Aprendizaje Automático es uno de los componentes principales sobre los que se ha basado este proyecto y por tanto es de gran importancia. Casi todos lo elementos desarrollados en el Capítulo 5 usan en cierta medida estos conocimientos y por tanto el objetivo ha sido cumplido.

2. **Aprendizaje teórico de conceptos usados en Deep Learning:** Al igual que en el caso anterior los conceptos sobre Deep Learning han supuesto el fundamento sobre el que construir los modelos convolucionales. De nuevo, este objetivo también ha sido cumplido.

3. **Aprendizaje teórico y práctico de la librería fast.ai:** Una vez se dispuso de la base teórica sobre la que construir estos modelos hacía falta una librería mediante la que construirlos. Debido a que esta es usada en todos los modelos presentados se considera el objetivo como cumplido.

4. **Puesta en práctica de los conocimientos adquiridos mediante un caso real:** Por último, una vez implementados los modelos y revisados los resultados se puede comprobar que los valores obtenidos son aceptables, superando el 99 % de éxito de clasificación de las imágenes en algunos de los modelos. Pese a esto, es posible mejorar estos resultado debido a que en un proceso de control de calidad se debe procurar minimizar el número de piezas defectuosas. Pese a que no se ha obtenido un 100 % de éxito de clasificación los resultados son lo suficientemente buenos como para considerar el objetivo como cumplido.

Como se ha indicado, una de las líneas futuras en las que puede ser continuado este trabajo es la implementación de otros modelos que todavía no se han explorado y otras técnicas mediante las que mejorar los resultados obtenidos.

Apéndice A

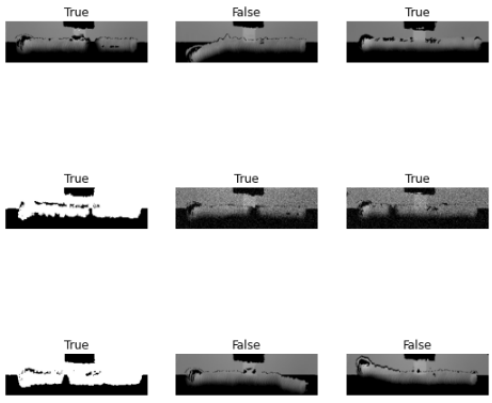
Notebooks

A continuación se inluirán algunos ejemplos de algunos de los Notebooks desarrollados. Este primer ejemplo constituye el experimento base sobre el que se han realizado el resto de modificaciones indicadas en el Capítulo 5. Todos los Notebooks se encuentran en el repositorio : `https://github.com/Istaliyan/TFG`

In [1]:
```
import os
import matplotlib.pyplot as plt
import numpy as np
from fastai.vision.all import *
import torch
import skimage
from PIL import Image
```

In [3]:
```
path = Path('/home/yani/Pruebas TFG/DatasetConIncorrectas/Tipo2Oversampling/Completo')
dls = ImageDataLoaders.from_folder(path, train='train', valid='valid', bs=9, seed=43)
```

In [4]:
```
dls.show_batch()
```

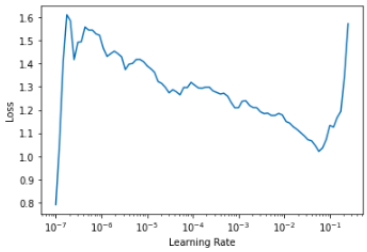


In [5]:
```
precision = Precision()
f1 = F1Score()
```

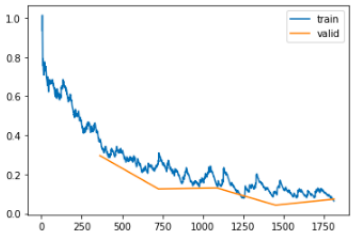
In [19]:
```
learn = cnn_learner(dls, models.resnet34,normalize=True,pretrained=True,
                    metrics=[accuracy, precision, f1],opt_func=SGD)
```

In [20]:
```
learn.lr_find()
```

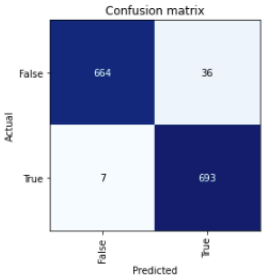
Out[20]:
```
SuggestedLRs(lr_min=0.005754399299621582, lr_steep=9.12010818865383e-07)
```



In [22]:
```
learn.recorder.plot_loss()
```



In [23]:
```
interp = ClassificationInterpretation.from_learner(learn)
interp.plot_confusion_matrix()
```



In [82]:
```
path = Path('/home/yani/Pruebas TFG/DatasetConIncorrectas/' +
            'Tipo3Oversampling/Completo/train/False/' +
            'Mdia_1_Pieza_40_Soldadura_79_7.png')

path2 = Path('/home/yani/Pruebas TFG/DatasetConIncorrectas/' +
            'Tipo2Oversampling/Completo/train/False/' +
            'Mdia_1_Pieza_1_Soldadura_42_9.png')

```

In [109]:
```
path = Path('/home/yani/Pruebas TFG/DatasetConIncorrectas/' +
            'Tipo3Oversampling/Completo/train/False/' +
            'Mdia_1_Pieza_40_Soldadura_79_7.png')
img = PILImage.create(path)
x, = first(dls.test_dl([img]))
```

In [110]:
```
class Hook():
    def hook_func(self, m, i, o): self.stored = o.detach().clone()
```

In [111]:
```
salida_hook = Hook()
hook = learn.model[0].register_forward_hook(salida_hook.hook_func)
```

In [112]:
```
with torch.no_grad(): output = learn.model.eval()(x)
```

In [113]:
```
act = salida_hook.stored[0]
salida = F.softmax(output, dim=-1)
```

In [114]:
```
dls.vocab
```

Out[114]:
```
['False', 'True']
```

In [115]:
```
x.shape
```

Out[115]:
```
torch.Size([1, 3, 70, 240])
```

In [116]:
```
mapaCAM = torch.einsum('ck,kij->cij', learn.model[1][-1].weight, act)
mapaCAM.shape
```

Out[116]:
```
torch.Size([2, 3, 8])
```

In [121]:
```
x_dec = TensorImage(dls.train.decode((x,))[0][0])
_,ax = plt.subplots()
x_dec.show(ctx=ax)
ax.imshow(mapaCAM[1].detach().cpu(), alpha=0.8, extent=(0,240,70,0),
interpolation='bilinear', cmap='magma', aspect='equal');
```

In [53]:
```
ax.imshow(mapaCAM[0].detach().cpu(), alpha=0.8, extent=(0,240,70,0),
interpolation='bilinear', cmap='magma');

```

In [1]:
```
import numpy as np
import matplotlib.pyplot as plt
from PIL import Image
```

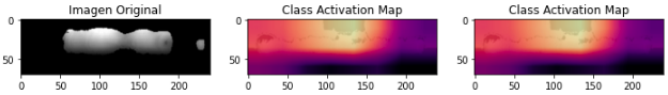
In [10]:
```
img = Image.open('/home/yani/TFG-Memoria/img/base/output-onlinepngtools2.png')
img = img.resize((240,70))

```

In [11]:
```
img2 = Image.open('/home/yani/TFG-Memoria/img/base/output-onlinepngtools.png')

img2 = img2.resize((240,70))
```

In [12]:
```
display(img)

```

In [23]:
```
import matplotlib.pyplot as plt
from mpl_toolkits.axes_grid1 import make_axes_locatable
import numpy as np

w = 280
h = 70
lista = []
path2 = Path('/home/yani/Pruebas TFG/DatasetConIncorrectas/' +
             'Tipo3Oversampling/Completo/train/False/' +
             'Mdia_1_Pieza_40_Soldadura_79_7.png')

lista.append(path2)
lista.append(img)
lista.append(img2)

fig = plt.figure(figsize=(12,5))

columns = 3
rows = 1
ax = []

for i in range(3):
    if i == 0:
        img = Image.open(lista[i]).convert('LA')
    else:
        img = lista[i]
    ax.append(fig.add_subplot(rows, columns, i+1))
    plt.imshow(img)
```

```
ax[0].title.set_text('Imagen Original')
ax[1].title.set_text('Class Activation Map')
ax[2].title.set_text('Class Activation Map')

plt.show()
```

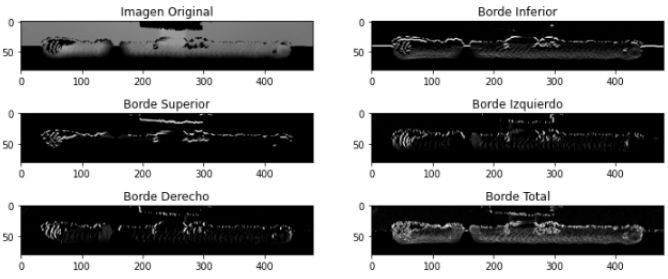


In [2]:

```
import cv2
import numpy as np
from PIL import Image
```

In [39]:

```
img = cv2.imread('/home/yani/Pruebas TFG/DatasetConIncorrectas/' +
            'Tipo2Oversampling/Soldadura_Buena/dia_1_Pieza_2_Soldadura_45_cortv.png')

kernelx_1 = np.array([[1,1,1],[0,0,0],[-1,-1,-1]])
kernelx_2 = np.array([[-1,-1,-1],[0,0,0],[1,1,1]])

kernely_1 = np.array([[-1,0,1],[-1,0,1],[-1,0,1]])
kernely_2 = np.array([[1,0,-1],[1,0,-1],[1,0,-1]])

salida1 = cv2.filter2D(img, -1, kernelx_1)
salida2 = cv2.filter2D(img, -1, kernelx_2)

salida3 = cv2.filter2D(img, -1, kernely_1)
salida4 = cv2.filter2D(img, -1, kernely_2)

salida5 = salida1 + salida2 + salida3 + salida4

salida1 = Image.fromarray(salida1)
salida2 = Image.fromarray(salida2)
salida3 = Image.fromarray(salida3)
salida4 = Image.fromarray(salida4)
salida5 = Image.fromarray(salida5)

w = 280
h = 70
lista = []
lista.append(img)
lista.append(salida1)
lista.append(salida2)
lista.append(salida3)
lista.append(salida4)
lista.append(salida5)
fig = plt.figure(figsize=(12,5))

columns = 2
rows = 3
ax = []

for i in range(len(lista)):
    img = lista[i]
    ax.append(fig.add_subplot(rows, columns, i+1))
    plt.imshow(img)
ax[0].title.set_text('Imagen Original')
ax[1].title.set_text('Borde Inferior')

ax[2].title.set_text('Borde Superior')
ax[3].title.set_text('Borde Izquierdo')

ax[4].title.set_text('Borde Derecho')
ax[5].title.set_text('Borde Total')

plt.show()

```

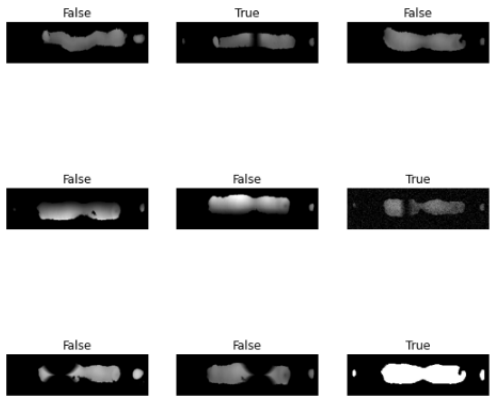


Ejemplo del Notebook correspondiente al experimento del optimizador Adam en el Capítulo 5.

In [1]:
```
import os
import matplotlib.pyplot as plt
import numpy as np
from fastai.vision.all import *
import torch
import skimage
import torch.nn.functional as F
from PIL import Image
```

In [2]:
```
path = Path('/home/yani/Pruebas TFG/DatasetConIncorrectas/Tipo3Oversampling/Completo')
dls = ImageDataLoaders.from_folder(path, train='train', valid='valid', bs=9, seed=41)
```

In [3]:
```
dls.show_batch()
```

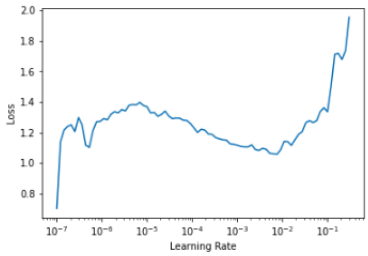


In [4]:
```
precision = Precision()
f1 = F1Score()
```

In [5]:
```
learn = cnn_learner(dls, models.resnet34,normalize=True,pretrained=True,
                    metrics=[accuracy, precision, f1],opt_func=Adam)
```

In [6]:
```
learn.lr_find()
```

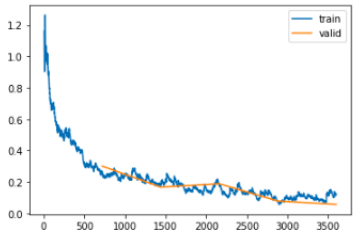
Out[6]:
```
SuggestedLRs(lr_min=0.0007585775572806596, lr_steep=9.999999747378752e-06)
```



In [7]:
```
learn.fit(5, lr=1e-03)
```

epoch	train_loss	valid_loss	accuracy	precision_score	f1_score	time
0	0.260515	0.300169	0.857857	0.921008	0.846332	00:38
1	0.184113	0.166343	0.937143	0.943478	0.936691	00:38
2	0.186013	0.188064	0.920000	0.985149	0.914242	00:39
3	0.075394	0.077928	0.970000	0.958217	0.970381	00:38
4	0.118693	0.056200	0.983571	0.976090	0.983700	00:40

In [8]:
```
1 learn.recorder.plot_loss()
```



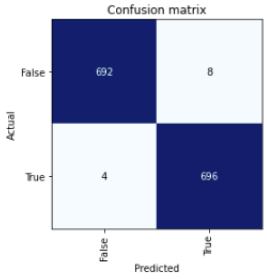
In [10]:
```
1 learn.freeze()
```

In [11]:
```
1 learn.fine_tune(3)
```

epoch	train_loss	valid_loss	accuracy	precision_score	f1_score	time
0	0.098640	0.089296	0.966429	0.993949	0.965467	00:39

epoch	train_loss	valid_loss	accuracy	precision_score	f1_score	time
0	0.133843	0.080815	0.980714	0.974612	0.980837	01:07
1	0.095562	0.066396	0.983571	0.968188	0.983837	01:07
2	0.030110	0.029938	0.991429	0.988636	0.991453	01:06

In [12]:
```
1 interp = ClassificationInterpretation.from_learner(learn)
2 interp.plot_confusion_matrix()
```



Bibliografía

[1] Jeremy Howard et al. fastai. https://github.com/fastai/fastai, 2020.

[2] Wikipedia. Python — wikipedia, la enciclopedia libre, 2021. [Internet; descargado 17-mayo-2021].

[3] Wikipedia. Proyecto jupyter — wikipedia, la enciclopedia libre, 2021.

[4] Stéfan van der Walt, Johannes L. Schönberger, Juan Nunez-Iglesias, François Boulogne, Joshua D. Warner, Neil Yager, Emmanuelle Gouillart, Tony Yu, and the scikit-image contributors. scikit-image: image processing in Python. *PeerJ*, 2:e453, 6 2014.

[5] Frank Turley. Prince2. https://prince2.wiki/es/, n.d.

[6] Wikipedia. Método de análisis y diseño de sistemas estructurados — wikipedia, la enciclopedia libre. https://es.wikipedia.org/w/index.php?title=M%C3%A9todo_de_an%C3%A1lisis_y_dise%C3%B1o_de_sistemas_estructurados&oldid=128308412, 2020.

[7] Robert Hughes. The 'step wise' planning approach to software projects. 1996.

[8] Marcos Sacristán Represa. Boletín oficial de castilla y león. https://www.uva.es/export/sites/uva/2.docencia/2.01.grados/2.01.05.areaestudiantes/_documentos/Normativa-trabajo-fin-de-grado.pdf, 2013.

[9] Guía del alumno. https://www.inf.uva.es/wp-content/uploads/2013/01/00-GuiaAlumnoTFG_2017.pdf, 2012.

[10] Wikipedia. Guía de los fundamentos para la dirección de proyectos — wikipedia, la enciclopedia libre. https://es.wikipedia.org/w/index.php?title=Gu%C3%ADa_de_los_fundamentos_para_la_direcci%C3%B3n_de_proyectos&oldid=136233531, 2021.

[11] James Vincent. An ai speed test shows clever coders can still beat tech giants like google and intel. https://www.theverge.com/2018/5/7/17316010/fast-ai-speed-test-stanford-dawnbench-google-intel, May 2017.

[12] Diederik P. Kingma and Jimmy Ba. Adam: A method for stochastic optimization, 2017.

[13] Ning Qian. On the momentum term in gradient descent learning algorithms, 1999.

[14] Pan Zhou, Jiashi Feng, Chao Ma, Caiming Xiong, Steven HOI, and Weinan E. Towards theoretically understanding why sgd generalizes better than adam in deep learning, 2020.

[15] J. Howard and S. Gugger. *Deep Learning for Coders with Fastai and Pytorch: AI Applications Without a PhD*. O'Reilly Media, Incorporated, 2020.

[16] Sasank Chilamkurthy. Transfer learning for computer vision tutorial. `https://pytorch.org/tutorials/beginner/transfer_learning_tutorial.html`, 2017.

[17] Rina Dechter. Learning while searching in constraint-satisfaction-problems. pages 178–185, 01 1986.

[18] Thomas M Mitchell. *Machine Learning*. McGraw-Hill, 1997.

[19] Ian Goodfellow, Yoshua Bengio, and Aaron Courville. *Deep Learning*. MIT Press, 2016. `http://www.deeplearningbook.org`.

[20] Wikipedia. Maldición de la dimensión — wikipedia, la enciclopedia libre, 2021.

[21] RE2 Robotics. *AI fields*.

[22] Simon J.D Prince. *Computer Vision*. Cambridge University Press, 2012.

[23] Stuart J Russell and Peter Norvig. *Inteligencia Artificial: Un enfoque moderno*. Pearson Prentice Hall, 2 edition, 2004.

[24] Himanshu Singh and Yunis Ahmad Lone. *Artificial Neural Networks*, pages 157–198. Apress, Berkeley, CA, 2020.

[25] IBM. What is artificial intelligence (ai)?, Jun 2020.

[26] Bing Xu, Naiyan Wang, Tianqi Chen, and Mu Li. Empirical evaluation of rectified activations in convolutional network, 2015.

[27] Yann Lecun, Leon Bottou, Genevieve Orr, and Klaus-Robert Müller. Efficient backprop. 08 2000.

[28] Wikipedia. Divergencia de kullback-leibler — wikipedia, la enciclopedia libre, 2020.

[29] Leslie N. Smith. Cyclical learning rates for training neural networks, 2017.

[30] Mohit Deshpande. Complete guide to deep neural networks. `https://pythonmachinelearning.pro/complete-guide-to-deep-neural-networks-part-2/#Backpropagation`, 2017.

[31] SeHyoun Ahn. Speed test of using symbolic vs automatic differentiation. `https://sehyoun.com/EXAMPLE_test_symbolic.html`, Apr 2017.

[32] Wikipedia contributors. Chain rule — wikipedia, the free encyclopedia. `https://en.wikipedia.org/wiki/Chain_rule`, 2021.

[33] Wikipedia. Grafo dirigido — wikipedia, la enciclopedia libre. `https://es.wikipedia.org/w/index.php?title=Grafo_dirigido&oldid=136059346`, 2021.

[34] Xavier Glorot and Yoshua Bengio. Understanding the difficulty of training deep feedforward neural networks. In *Proceedings of the Thirteenth International Conference on Artificial Intelligence and Statistics*, pages 249–256, 2010.

[35] Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. Delving deep into rectifiers: Surpassing human-level performance on imagenet classification, 2015.

[36] F. Pedregosa, G. Varoquaux, A. Gramfort, V. Michel, B. Thirion, O. Grisel, M. Blondel, P. Prettenhofer, R. Weiss, V. Dubourg, J. Vanderplas, A. Passos, D. Cournapeau, M. Brucher, M. Perrot, and E. Duchesnay. Scikit-learn: Machine learning in Python. *Journal of Machine Learning Research*, 12:2825–2830, 2011.

[37] Mathanraj Sharma. *Composition of an RGB image*. 2019.

[38] Wikipedia. Cálculo tensorial — wikipedia, la enciclopedia libre. `https://es.wikipedia.org/w/index.php?title=C%C3%A1lculo_tensorial&oldid=136258877`, 2021.

[39] Mable Wilderman. *Tensor examples*. 2020.

[40] Sumit Saha. *A Comprehensive Guide to Convolutional Neural Networks*. 2018.

[41] Matthew D Zeiler and Rob Fergus. Visualizing and understanding convolutional networks, 2013.

[42] Vincent Dumoulin and Francesco Visin. A guide to convolution arithmetic for deep learning, 2018.

[43] Computer Science Wiki. Max-pooling / pooling — computer science wiki,, 2018. [Online; accessed 11-July-2021].

[44] Jamil Ahmad, Khan Muhammad, and Sung Baik. Data augmentation-assisted deep learning of hand-drawn partially colored sketches for visual search. *PLOS ONE*, 12:e0183838, 08 2017.

[45] Jia Deng, Wei Dong, Richard Socher, Li-Jia Li, Kai Li, and Li Fei-Fei. Imagenet: A large-scale hierarchical image database. In *2009 IEEE conference on computer vision and pattern recognition*, pages 248–255. Ieee, 2009.

[46] Andrea Mari, Thomas R. Bromley, Josh Izaac, Maria Schuld, and Nathan Killoran. Transfer learning in hybrid classical-quantum neural networks. *Quantum*, 4:340, Oct 2020.

[47] Carl F. Sabottke and Bradley M. Spieler. The effect of image resolution on deep learning in radiography. *Radiology: Artificial Intelligence*, 2(1):e190015, 2020.

[48] N. V. Chawla, K. W. Bowyer, L. O. Hall, and W. P. Kegelmeyer. Smote: Synthetic minority over-sampling technique. *Journal of Artificial Intelligence Research*, 16:321–357, Jun 2002.

[49] Wikipedia contributors. Prewitt operator — Wikipedia, the free encyclopedia. `https://en.wikipedia.org/w/index.php?title=Prewitt_operator&oldid=1025793582`, 2021.

[50] Wikipedia contributors. Sobel operator — Wikipedia, the free encyclopedia. `https://en.wikipedia.org/w/index.php?title=Sobel_operator&oldid=1031067706`, 2021.

[51] Wikipedia contributors. Discrete laplace operator — Wikipedia, the free encyclopedia. `https://en.wikipedia.org/w/index.php?title=Discrete_Laplace_operator&oldid=1019714497`, 2021.

[52] Bolei Zhou, Aditya Khosla, Agata Lapedriza, Aude Oliva, and Antonio Torralba. Learning deep features for discriminative localization, 2015.

[53] Alex Krizhevsky, Ilya Sutskever, and Geoffrey E Hinton. Imagenet classification with deep convolutional neural networks. In F. Pereira, C. J. C. Burges, L. Bottou, and K. Q. Weinberger, editors, *Advances in Neural Information Processing Systems*, volume 25. Curran Associates, Inc., 2012.

[54] Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. Deep residual learning for image recognition, 2015.

[55] Aston Zhang, Zachary C. Lipton, Mu Li, and Alexander J. Smola. Dive into deep learning. *arXiv preprint arXiv:2106.11342*, 2021.

[56] François Chollet. Xception: Deep learning with depthwise separable convolutions, 2017.

[57] Chi-Feng Wang. A basic introduction to separable convolutions, Aug 2018.