Universidad de Valladolid

Trabajo de Fin de Grado

Análisis de entrenamiento en deep learning para gestión de calidad en fabricación automática

Autor: Andrés Trigueros Vega

Tutor: Benjamín Sahelices Fernández

Agradecimientos

Este proyecto no se habría llevado a cabo de no ser por todas esas personas que han estado a mi lado durante el transcurso de la carrera. En especial:

A los compañeros y amigos, los que ya tenía y los que he ido conociendo, por su compañía y apoyo en los peores momentos y su indispensable presencia en los mejores momentos.

A los profesores, en especial a mi tutor Benjamín por darme los conocimientos y guiarme durante el desarrollo de este proyecto.

Y por supuesto, a las personas más importantes de mi vida: mis padres y mis hermanos, que me han dedicado todo su tiempo, todo su esfuerzo y todos sus recursos con tal de educarme y formarme lo mejor posible para afrontar la vida.

Resumen

El objetivo de este Trabajo de Fin de Grado es desarrollar redes neuronales que sean capaces de distinguir imágenes de soldaduras buenas (sin defectos) de imágenes de soldaduras malas (con defectos).

Para entrenar las redes ha sido necesario realizar programas que simulen defectos, para lo cual se ha utilizado Python 3.9.5.

Para la creación de las redes, se han utilizado tanto anaconda, que es una distribución utilizada en ciencia de datos y machine learning, como colab, que es una herramienta de Google que da acceso gratuito a GPUs.

Además, se ha utilizado la librería de python fast.ai, la cual está especializada en deep learning y es la librería que más rápido ha crecido desde su creación en 2017.

Por último, se realizará un estudio en el que se identificará la mejor manera de entrenar una red que no es capaz de clasificar correctamente una imagen concreta, en cuanto a tiempo y resultados.

Para la realización del proyecto, se ha seguido el marco de trabajo para el desarrollo ágil SCRUM.

Índice general

Índice de figuras

Capítulo 1

Introducción y objetivos

Imaginemos una empresa que busca un programa que sea capaz de identificar soldaduras con defectos. Hoy en día, esto es relativamente fácil de solucionar con la utilización de una red neuronal. El principal problema que surge es que para poder entrenar una red neuronal, hacen falta una gran cantidad de datos (en este caso, imágenes), cosa que en muchas ocasiones, es difícil o costoso de obtener.

El objetivo del trabajo de fin de grado es conseguir redes que clasifiquen imágenes de diferentes tipos de soldaduras en soldaduras buenas (sin defectos) y soldaduras malas (con defectos). A continuación, se realizará un estudio en el que se buscará la manera más optimizada de reentrenar una red ya entrenada que no es capaz de clasificar correctamente una imagen concreta, con el objetivo de poder utilizar la información para reentrenar progresivamente una red con las imágenes que no es capaz de clasificar correctamente.

De esta manera, haría falta una red inicial con pocas imágenes, y por lo tanto, con una tasa de acierto bastante reducida. Poco a poco, habría que nutrir a la red con imágenes que no sea capaz de clasificar inicialmente, mejorando así su tasa de acierto a largo plazo, y utilizando la menor cantidad de imágenes posible. Así, la empresa ahorra una gran cantidad de dinero y tiempo en obtener muchas imágenes para conseguir una red con una tasa de acierto suficientemente grande.

Para la realización de este proyecto, se ha obtenido información que ha sido de vital importancia de:

- "Neural Networks and Deep Learning", por Michael Nielsen. [Nie]

  Es un libro online gratuito que explica de forma teórica el funcionamiento de las redes neuronales y del deep learning. Está escrito por el escritor australiano Michael Nielsen, quien además de ser escritor, es profesor en la Universidad de

Quieensland y del Instituto Perimeter de Física teórica, tiene un doctorado de física y es experto en redes neuronales. [Wikb]

- "Fastai / fastbook" de github, escrito por "Jeremy", "Sylvain " y "Alexis Gallagher" [Syl]

  Es una guía online gratuita que explica como funcionan las redes neuronales y cómo programarlas con la librería "fastai". Además, puede servir como manual de uso de la misma librería. Está escrita por los escritores "Jeremy", que es un profesor de aprendizaje automático desde hace 30 años, y es uno de los creadores de "fast.ai", "Sylvain ", que es un escritor que ya ha escrito más de 10 libros de matemáticas y "Alexis Gallagher", que es un investigador en biología matemática y guionista.

Con esta base teórica del funcionamiento de redes neuronales y el manual de uso de la librería que utilizaremos para llevar a cabo los experimentos, se tratará de responder a estas y más cuestiones que se presenten a lo largo de la realización del proyecto.

Capítulo 2

Planificación

2.1. Seguimiento del proyecto

Durante la realización de este proyecto se ha utilizado la metodología SCRUM, que es un proceso en el que se aplican de manera regular un conjunto de buenas prácticas para trabajar y obtener el mejor resultado posible de un proyecto. Consiste en realizar entregas parciales y regulares del producto final, priorizadas por el beneficio que aportan al receptor del proyecto, o en este caso, al tutor del proyecto.[pro]

La planificación ha variado ligeramente de lo que se pensaba inicialmente, pero contábamos con tiempo para resolver estos contratiempos.

Se ha dividido el tiempo en 18 "sprints". En cada uno de ellos se llevan a cabo unas cuantas tareas, en las que se estima el tiempo de realización de la tarea y se escribe el tiempo real que ha supuesto realizar dicha tarea.

- Sprint 1 (22/02/2021 - 01/03/2021)

	Descripción	Tiempo estimado	Tiempo empleado	Estado
Tarea 1	Estructura general del documento	30 min	20 min	Completada
Tarea 2	Creación del proyecto en overleaf	10 min	15 min	Completada
Tarea 3	Obtención plantilla TFG	10 min	10 min	Completada
Tarea 4	Obtención libro "Neural Networks and Deep Learning"	10 min	5 min	Completada
Tarea 5	Obtención link "Github fastai fastbook"	10 min	2 min	Completada

Tiempo total del sprint:	52 minutos
Tiempo total del proyecto:	52 minutos

Figura 2.1: Sprint 1

Durante este sprint dejo listo todo el material que necesito para empezar el trabajo. Evidentemente no hay ninguna tarea que no haya terminado porque todas son muy cortas.

- Sprint 2 (01/03/2021 - 08/03/2021)

	Descripción	Tiempo estimado	Tiempo empleado	Estado
Tarea 6	Lectura inicial del libro "Neural Networls and Deep Learning"	4 horas	10 horas	Pendiente

Tiempo total del sprint:	10 h
Tiempo total del proyecto:	10 h 52 min

Figura 2.2: Sprint 2

Durante este sprint he de hacer una lectura inicial para entender bien como funcionan las redes neuronales y el deep learning. Aunque ya tenía algunos conocimientos de como funcionaba la inteligencia artificial, me faltaban conocimientos para hacer un TFG sobre ello.

- Sprint 3 (08/03/2021 - 15/03/2021)

	Descripción	Tiempo estimado	Tiempo empleado	Estado
Tarea 6	Lectura inicial del libro “Neural Networks and Deep Learning”	5 horas	5 horas	Completada
Tarea 7	Introducción a las redes neuronales	30 horas	10 horas	Pendiente

Tiempo total del sprint:	15 h
Tiempo total del proyecto:	25 h 52 min

Figura 2.3: Sprint 3

Durante este sprint acabo la lectura del libro y expreso en el primer apartado un resumen de lo que he aprendido.

- Sprint 4 (15/03/2021-22/03/2021)

	Descripción	Tiempo estimado	Tiempo empleado	Estado
Tarea 7	Introducción a las redes neuronales	30 horas	20 horas	Completada

Tiempo total del sprint:	20 h
Tiempo total del proyecto:	45 h 52 min

Figura 2.4: Sprint 4

Durante este sprint tan solo finalizo la introducción a las redes neuronales.

- Sprint 5 (22/03/2021-29/03/2021)

	Descripción	Tiempo estimado	Tiempo empleado	Estado
Tarea 8	Descargar python y pycharm (crear defectos)	30 minutos	1 hora	Completada
Tarea 9	Programación con python defectos de soldaduras	80 horas	35 horas	Pendiente

Tiempo total del sprint:	36 h
Tiempo total del proyecto:	81 h 52 min

Figura 2.5: Sprint 5

Durante este sprint empiezo a preparar el dataset que utilizaré para llevar a cabo los experimentos del estudio.

- Sprint 6 (29/03/2021-05/04/2021)

	Descripción	Tiempo estimado	Tiempo empleado	Estado
Tarea 9	Programación con python defectos de soldaduras	80 horas	35 horas	Pendiente

Tiempo total del sprint:	35 h
Tiempo total del proyecto:	116 h 52 min

Figura 2.6: Sprint 6

Durante este sprint sigo programando los defectos de todos los tipos de soldaduras.

- Sprint 7 (05/04/2021-12/04/2021)

	Descripción	Tiempo estimado	Tiempo empleado	Estado
Tarea 9	Programación con python defectos de soldaduras	80 horas	35 horas	Pendiente

Tiempo total del sprint:	35 h
Tiempo total del proyecto:	151 h 52 min

Figura 2.7: Sprint 7

Durante este sprint termino de programar los defectos de todos los tipos de soldaduras.

- Sprint 8 (12/04/2021-19/04/2021)

	Descripción	Tiempo estimado	Tiempo empleado	Estado
Tarea 10	Ejecución de las primeras redes neuronales	10 horas	35 horas	Completada

Tiempo total del sprint:	35 h
Tiempo total del proyecto:	186 h 52 min

Figura 2.8: Sprint 8

Durante este sprint ejecuto las primeras redes neuronales y realizo las primeras pruebas. Como son muchas redes neuronales y el equipo con el que cuento no es el óptimo, tardo bastante más de lo que esperaba. Me doy cuenta del desbalanceo de imágenes buenas y malas.

- Sprint 9 (19/04/2021-26/04/2021)

	Descripción	Tiempo estimado	Tiempo empleado	Estado
Tarea 11	Programación augmentation soldaduras buenas	4 horas	5 horas	Completada
Tarea 12	Ejecución de las segundas redes neuronales	35 horas	30 horas	Completada

Tiempo total del sprint:	35 h
Tiempo total del proyecto:	221 h 52 min

Figura 2.9: Sprint 9

Durante este sprint hago augmentation de las soldaduras buenas para balancear el número de imágenes buenas y malas en los datasets y vuelvo a realizar las pruebas. Me doy cuenta de que hay que limpiar los datasets de imágenes que no están bien clasificadas.

- Sprint 10 (26/04/2021-03/05/2021)

	Descripción	Tiempo estimado	Tiempo empleado	Estado
Tarea 13	Limpieza de los datasets	10 horas	7 horas	Completada
Tarea 14	Ejecución de las terceras redes neuronales	30 horas	30 horas	Completada

Tiempo total del sprint:	37 h
Tiempo total del proyecto:	258 h 52 min

Figura 2.10: Sprint 10

Durante este sprint limpio los datasets de imágenes que estaban mal clasificadas y vuelvo a ejecutar las redes neuronales. Esta vez consigo tasas de error aceptables.

- Sprint 11 (03/05/2021-10/05/2021)

	Descripción	Tiempo estimado	Tiempo empleado	Estado
Tarea 15	Escritura análisis del clasificador	15 horas	9 horas	Completada
Tarea 16	Lectura github fastai fastbook	40 horas	10 horas	Pendiente

Tiempo total del sprint:	19 h
Tiempo total del proyecto:	277 h 52 min

Figura 2.11: Sprint 11

Durante este sprint escribo el análisis del clasificador, donde explico lo que he estado haciendo las últimas semanas. Comienzo la segunda lectura, el github de fastai fastbook.

- Sprint 12 (10/05/2021-17/05/2021)

	Descripción	Tiempo estimado	Tiempo empleado	Estado
Tarea 16	Lectura github fastai fastbook	30 horas	20 horas	Pendiente

Tiempo total del sprint:	20 h
Tiempo total del proyecto:	297 h 52 min

Figura 2.12: Sprint 12

Durante este sprint sigo estudiando fastai.

- Sprint 13 (17/05/2021-24/05/2021)

	Descripción	Tiempo estimado	Tiempo empleado	Estado
Tarea 16	Lectura github fastai fastbook	10 horas	15 horas	Completada

Tiempo total del sprint:	15 h
Tiempo total del proyecto:	312 h 52 min

Figura 2.13: Sprint 13

Durante este sprint termino de estudiar fastai.

- Sprint 14 (24/05/2021-31/05/2021)

	Descripción	Tiempo estimado	Tiempo empleado	Estado
Tarea 17	Escritura contexto tecnológico, pytorch, fastai	15 horas	10 horas	Completada

Tiempo total del sprint:	10 h
Tiempo total del proyecto:	322 h 52 min

Figura 2.14: Sprint 14

Durante este sprint realizo la escritura del contexto tecnológico.

- Sprint 15 (31/05/2021-07/06/2021)

	Descripción	Tiempo estimado	Tiempo empleado	Estado
Tarea 18	Experimentos evaluación del aprendizaje	20 horas	22 horas	Pendiente

Tiempo total del sprint:	22 h
Tiempo total del proyecto:	344 h 52 min

Figura 2.15: Sprint 15

Durante este sprint realizo los experimentos de evaluación del aprendizaje. Como no dispongo del mismo material del que contaba anteriormente, tardo más de lo que esperaba.

- Sprint 16 (07/06/2021-14/06/2021)

	Descripción	Tiempo estimado	Tiempo empleado	Estado
Tarea 18	Experimentos evaluación del aprendizaje	5 horas	5 horas	Completada
Tarea 19	Escritura evaluación del aprendizaje	7 horas	10 horas	Completada

Tiempo total del sprint:	15 h
Tiempo total del proyecto:	359 h 52 min

Figura 2.16: Sprint 16

Durante este sprint finalizo los experimentos de la evaluación del aprendizaje y realizo la escritura de esta sección.

- Sprint 17 (14/06/2021-21/06/2021)

	Descripción	Tiempo estimado	Tiempo empleado	Estado
Tarea 20	Escritura planificación	10 horas	14 horas	Completada
Tarea 21	Escritura introducción y objetivos	2 horas	2 horas	Completada

Tiempo total del sprint:	16 h
Tiempo total del proyecto:	375 h 52 min

Figura 2.17: Sprint 17

Durante este sprint realizo la escritura de la planificación (la cual estaba escribiendo en sucio en un programa diferente), y todas las subsecciones, y escribo la introducción y los objetivos.

- Sprint 18 (21/06/2021-28/06/2021)

	Descripción	Tiempo estimado	Tiempo empleado	Estado
Tarea 22	Lectura final del proyecto y corrección de erratas	8 horas	10 horas	Completada

Tiempo total del sprint:	10 h
Tiempo total del proyecto:	385 h 52 min

Figura 2.18: Sprint 18

Durante este sprint realizo una lectura final del proyecto y corrijo errores y erratas que encuentro.

2.2. Diagrama de Gantt

Los diagramas de Gantt permiten comunicar visualmente el cronograma del proyecto de forma simple y comprensible.[ven]

El diagrama de Gantt del proyecto sería el siguiente:

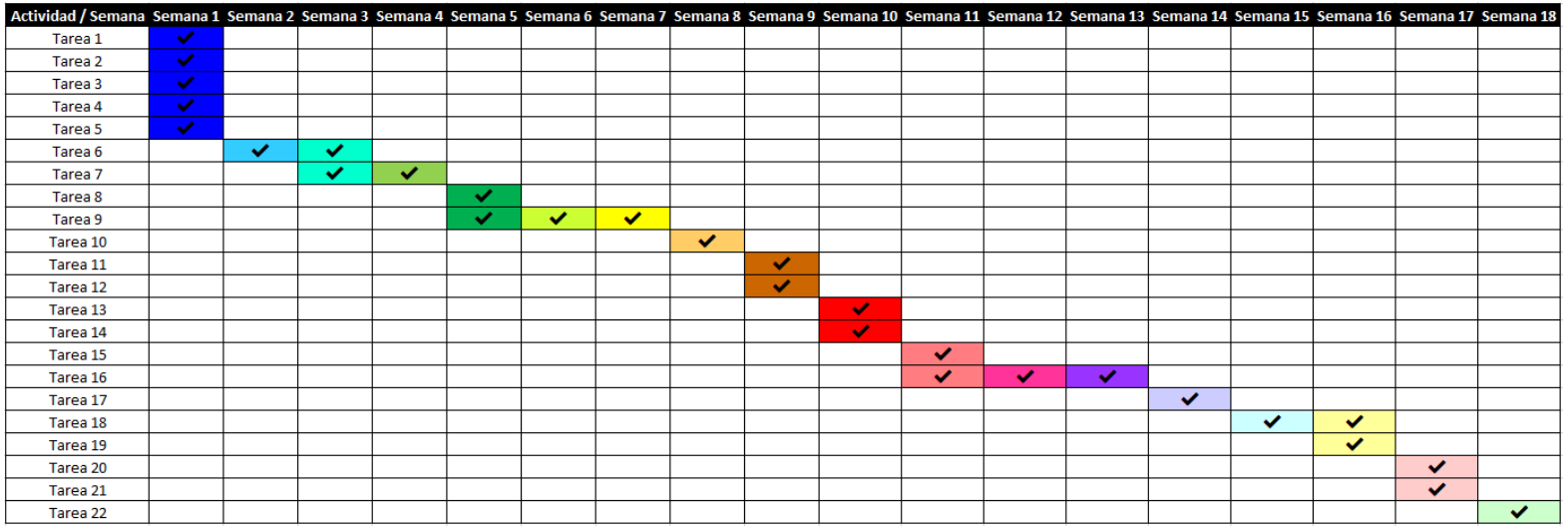

Figura 2.19: Diagrama de Gantt

2.3. Presupuesto económico

En esta sección se recoge el presupuesto económico del proyecto, de acuerdo a la planificación previamente mencionada.

2.3.1. Recursos humanos

Categoría	Número de personas	Nº horas	Precio unitario	Importe total
Ingeniero informático	1	385 horas 52 minutos	12,00 €	4.630,40 €
Profesor de universidad	1	15 horas	18,60 €	279 €
			TOTAL	4.909,40 €

Figura 2.20: Presupuesto para los recursos humanos del proyecto

2.3.2. Recursos para el desarrollo

Descripción	Unidades	Precio unitario	Importe total
Ordenador personal	2	800,00 €	1.600,00 €
Windows 10 Home	2	145,00 €	290,00 €
Microsoft office 2019	2	149,00 €	298,00 €
		TOTAL	2.188,00 €

Figura 2.21: Presupuesto para los recursos para el desarrollo

2.4. Análisis de riesgos

Se van a listar los principales riesgos a los que está sometido el proyecto, junto con una pequeña descripción, la probabilidad de que suceda, el impacto que tendría en el proyecto, una lista con acciones de mitigación y una lista de acciones de contingencia.

Riesgo 1: Pérdida de datos o de trabajo	
Descripción	La pérdida del trabajo elaborado es uno de los riesgos más claros de un proyecto. Ya sea por un fallo humano o de alguna máquina, conviene tener en cuenta que este riesgo es posible que ocurra.
Probabilidad	Media (depende de la persona)
Impacto	Bajo-Alto (dependiendo de la pérdida)
Acciones de mitigación	Crear copias de seguridad
Acciones de contingencia	Descarga de la copia de seguridad

Figura 2.22: Riesgo 1: Pérdida de datos o de trabajo

Riesgo 2: Enfermedad	
Descripción	Un proyecto de esta magnitud tiene una duración prolongada a lo largo del tiempo, y durante ese tiempo, el autor podría contraer alguna enfermedad que dificultase el proceso de creación del mismo.
Probabilidad	Baja
Impacto	Bajo-Alto (dependiendo de la enfermedad)
Acciones de mitigación	Uso de mascarillas y distancia social. Autocuidado.
Acciones de contingencia	Replanificación del proyecto.

Figura 2.23: Riesgo 2: Enfermedad

Riesgo 3: Fallo en las máquinas	
Descripción	Para la realización del proyecto, son necesarias varias máquinas: las que toman las imágenes, las que crean las redes neuronales... Una avería en cualquiera de ellas podría retasar el proyecto.
Probabilidad	Baja
Impacto	Bajo-Alto (dependiendo del fallo)
Acciones de mitigación	Realización de un correcto mantenimiento
Acciones de contingencia	Reparación o sustitución de la máquina

Figura 2.24: Riesgo 3: Fallo en las máquinas

Riesgo 4: Planificación irreal	
Descripción	Hay diversas razones por las que se puede realizar una planificación irreal. Ya sea por optimismo, o simplemente por inexperiencia o ineptitud, la duración del proyecto puede variar enormemente.
Probabilidad	Medio-Alto
Impacto	Bajo
Acciones de mitigación	Tener en cuenta posibles retrasos
Acciones de contingencia	Postponer la fecha de fin del proyecto o aumentar el ritmo de creación del proyecto

Figura 2.25: Riesgo 4: Planificación irreal

Capítulo 3

Introducción a las redes neuronales

Una red neuronal es un modelo computacional que consiste en un conjunto de unidades llamadas perceptrones, conectadas entre sí para transmitirse señales. La información de entrada atraviesa la red neuronal (donde se somete a diversas operaciones) y produce unos valores de salida.[Wikd]

3.1. Perceptrón

Los perceptrones son la unidad básica de inferencia en forma de discriminador lineal, a partir de lo cual se desarrolla un algoritmo capaz de generar un criterio para seleccionar un sub-grupo a partir de un grupo de componentes más grande.[Wikc]

Fueron creados en los años 50 y 60 por el científico Frank Rosenblatt, que a su vez se inspiró del trabajo realizado previamente por los científicos Warren McCulloch y Walter Pitts.

Hay diferentes tipos de perceptrones. El más simple y el primero que se creó, recibe varios inputs binarios (0 o 1) y produce un solo output binario. Cada input (x1, x2...) tiene un peso (w1, w2... $\in$ R), y el output es 0 o 1, en función de si la función $\sum_j w_j x_j$ es mayor o menor que un umbral ($\in$ R).

Es decir, que el resultado de un perceptrón, se elige de la siguiente forma:

De esta manera, un perceptrón podría calcular, problemas como el que se enuncia a continuación:

Imagina que pretendes ir a una fiesta, y hay tres factores que quieres tener en cuenta:

1. Si va a asistir la chica/chico que te gusta

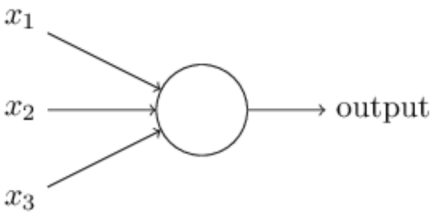

Figura 3.1: Esquema básico de un perceptrón

$$\text{output} = \begin{cases} 0 & \text{if } \sum_j w_j x_j \leq \text{ threshold} \\ 1 & \text{if } \sum_j w_j x_j > \text{ threshold} \end{cases}$$

Figura 3.2: Fórmula básica salida perceptrón

2. Si va a ir tu grupo de amigos
3. Si el día que se celebra la fiesta es fin de semana

Podemos representar estos tres factores de una manera binaria, de tal forma que $x_1 = 1$ si la chica/chico que te gusta va a asistir y $x_1 = 0$ si no asistirá, $x_2 = 1$ si tu grupo de amigos asistirá y $x_2 = 0$ si no asistirá, y $x_3 = 1$ si la fiesta se celebra durante un fin de semana y $x_3 = 0$ si la fiesta se celebra entre semana.

Ahora, imagina que, para ti, es más importante que vayan tus amigos a que vaya la chica/chico que te gusta o el día de la semana en el que se celebre. Para ti, en este caso, el peso que le des a la x_1, debe ser mayor al peso que le des a x_2 y a x_3. Por ejemplo, podríamos dar al factor x_1, un peso $w_1 = 4$, y a los factores x_2 y x_3, unos pesos $w_2 = 2$ y $w_3 = 1$.

Por último, imagina que eliges un umbral de 3.

Para estos datos, una fiesta a la que van tus amigos, es suficiente para pasar el umbral, y por lo tanto, el hecho de que vayan tus amigos es condición suficiente para que tú asistas a la fiesta, independientemente de si va la chica/chico que te gusta o el día de la semana que sea, pero también, si la fiesta es en un fin de semana y asiste la chica/chico que te gusta, se dan las condiciones suficientes para que asistas a la fiesta, aunque no vayan a ir tus amigos.

Variando los pesos y el umbral, se podrían modelizar las tomas de decisiones de los seres humanos.

Evidentemente, un perceptrón, no es un modelo completo de la toma de decisiones

humana, pero empieza a ser una aproximación bastante realista. Obviamente, para resolver problemas más complejos, harán falta soluciones más complejas, como la mostrada a continuación.

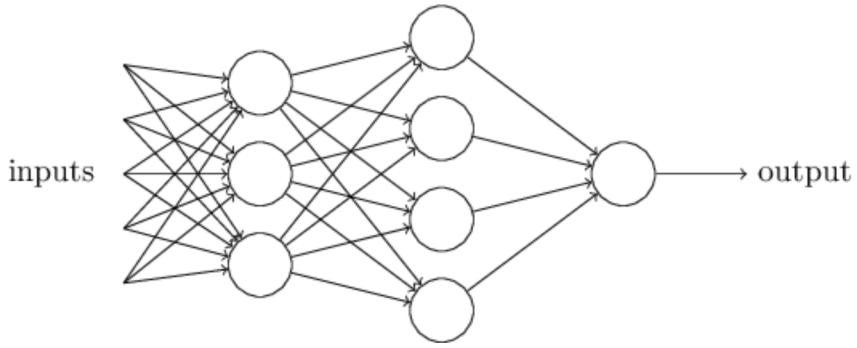

Figura 3.3: Esquema red neuronal simple

En esta red, a la primera columna, se la llama primera capa de perceptrones, a la segunda columna se la llama segunda capa de perceptrones, y así sucesivamente.

Hay algo que puede llevar a confusión en este esquema, y es que cada perceptrón tiene una sola salida, no varias, como muestra la imagen, pero esa misma salida es la que entrará como input en cada uno de los perceptrones de la siguiente capa.

Para simplificar la fórmula del cálculo de la salida de los perceptrones, podemos escribir la suma de productos como un producto escalar, y cambiando el umbral (threshold) por el bias (b), de tal manera que b $\equiv -$threshold, podríamos reescribir la fórmula como:

$$\text{output} = \begin{cases} 0 & \text{if } w \cdot x + b \leq 0 \\ 1 & \text{if } w \cdot x + b > 0 \end{cases}$$

Figura 3.4: Fórmula resumida salida perceptrón

De esta manera, el bias es una medida que indica la facilidad de que un perceptrón tenga salida 1. Con un bias grande, es más fácil que la salida sea 1, y con un bias pequeño, es más difícil que la salida sea 1.

El único problema de este modelo de perceptrón es que un pequeño cambio en un peso o bias, cambia totalmente la red, sobre todo si es en las primeras capas. Para resolver este problema, se crearon las neuronas sigmoides.

3.2. Neurona sigmoide

La principal diferencia entre el perceptrón mencionado anteriormente y la neurona sigmoide es que el perceptrón solo admitían inputs y outputs binarios, pero la neurona sigmoide admite valores entre el 0 y el 1. De esta manera, un pequeño cambio en un peso o en un bias, realiza un pequeño cambio en la red.

Además, el output, no se calculará de la manera previamente mencionada, sino que se calculará utilizando la función sigmoide:

$$\sigma(z) \equiv \frac{1}{1+e^{-z}}.$$

Figura 3.5: Fórmula básica neurona sigmoide

Siendo z = w·x + b, por lo que para simplificarla, se puede escribir de la siguiente manera:

$$\frac{1}{1+\exp(-\sum_j w_j x_j - b)}.$$

Figura 3.6: Fórmula simplificada neurona sigmoide

De esta manera, la forma de la función pasa de tener esta forma:

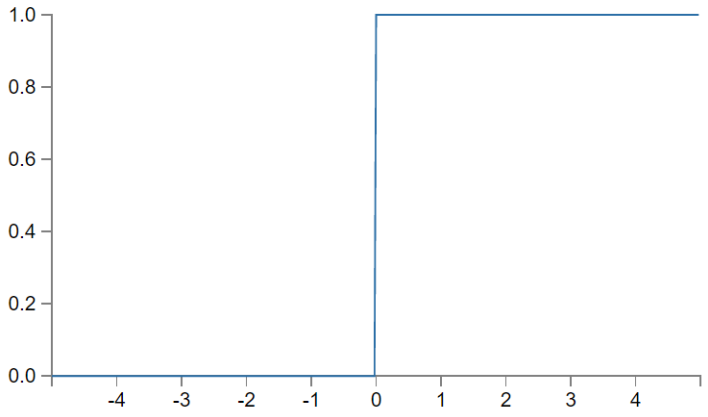

Figura 3.7: Forma función perceptrón

A tener la siguiente forma:

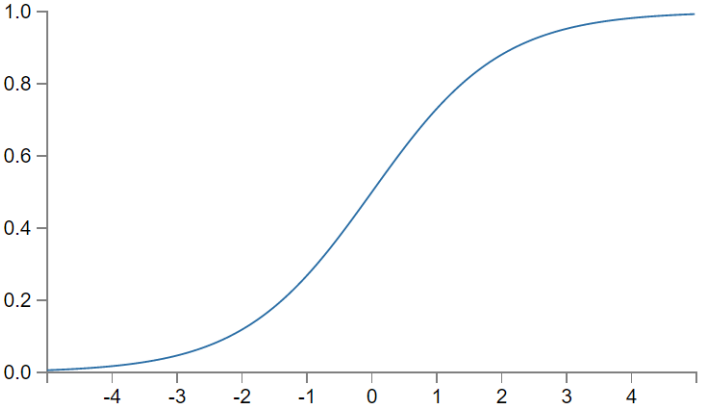

Figura 3.8: Forma función neurona sigmoide

Este suavizado de la función es el que produce pequeños cambios en la salida cuando se realizan pequeños cambios en los bias y los pesos.

De hecho, gracias al cálculo, podemos aproximar con relativa precisión estos cambios en la salida gracias a la siguiente fórmula:

$$\Delta \text{output} \approx \sum_j \frac{\partial \, \text{output}}{\partial w_j} \Delta w_j + \frac{\partial \, \text{output}}{\partial b} \Delta b,$$

Figura 3.9: Formula básica salida neurona sigmoide

Al utilizar las neuronas sigmoides, como hemos explicado anteriormente, no solo el input pasa de poder tener valores únicamente binarios (0 o 1) a tener valores intermedios, como el 0.716. También el output podrá tomar estos valores intermedios.

Por esta razón, es posible que la salida de una red en la que se identifican números manuscritos (ejemplo que utilizaremos más adelante), dé como resultado que una imagen contenga un 4 con una probabilidad de 0.98, un 9 con una probabilidad de un 0.01, y así con todos los dígitos. Es más difícil (aunque más práctico y preciso) interpretar esta salida, que la salida de un perceptrón, en el que sale un dígito con una probabilidad de 1, y el resto con una probabilidad de 0.

3.3. Arquitectura de las redes neuronales

Como he mencionado previamente, en una red como la que se muestra a continuación, a las neuronas que forman la primera capa (o capa de entrada), se las denomina neuronas de entrada y a la neurona o las neuronas que forman la última capa (o capa de salida), se las llaman neuronas de salida. Todas las demás neuronas se encuentran en las capas ocultas (que es simplemente, una manera de dar nombre a las capas que no son ni de entrada ni de salida).

Por razones históricas, a estas redes de muchas capas se las llama MLPs (que son las siglas en inglés de "multilayer perceptrons", que significa perceptrones de muchas capas), aunque esto puede llegar a ser confuso, ya que no están formadas por perceptrones, sino por neuronas sigmoides.

El diseño de las capas de entrada y salida suele ser siempre bastante parecido. El número de neuronas que tiene la capa de entrada suele ser igual al número de píxeles que tienen las imágenes que estamos tratando de clasificar (por ejemplo, si es de 64 píxeles de alto y 64 píxeles de ancho, la red tendrá una capa de entrada de 4096 neuronas sigmoides, y la de salida, dependerá de lo que se busque con la red. Si simplemente se quiere diferenciar si en una imagen hay un perro o no, solamente tendrá una neurona sigmoide, que tendrá un output en el que si la salida es mayor

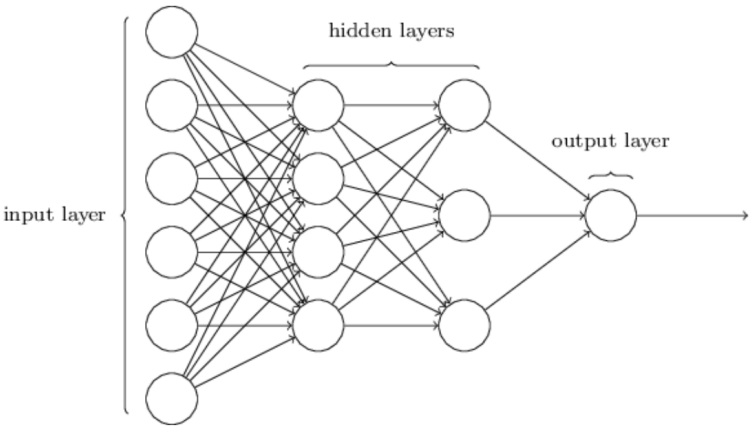

Figura 3.10: Esquema partes de una red neuronal

que 0.5, el resultado es que la imagen sí contendrá un perro, y menor que 0.5, que significará que la imagen no contiene un perro.

Al igual que estas capas suelen ser bastante sencillas, las capas ocultas suelen ser más misteriosas. De hecho, no es posible resumir el proceso de diseño de las capas ocultas con unas reglas sencillas. De hecho, los investigadores de redes neuronales, han desarrollado numerosos diseños heurísticos para las capas intermedias, que ayudan a la gente a entender el comportamiento que quieren que tengan sus redes.

De hecho, hasta ahora todas las redes utilizaban la salida de la capa anterior como entrada de la capa, y la salida de la capa, como entrada de la capa siguiente. A este tipo de redes se las llama redes neuronales feedforward, que significa que no tienen bucles en el interior de la red. La información siempre va hacia adelante (forward, en inglés), y nunca hacia atrás.

Sin embargo, hay modelos de redes neuronales en los que los bucles, sí que son posibles. A estos modelos, se les llama redes neuronales recurrentes, y en estas redes, las neuronas se activan y desactivan temporalmente en forma de cascada. De esta manera, los bucles no causan ningún problema, ya que las salidas solo afectan a las entradas un tiempo después, no instantáneamente.

De momento, las redes neuronales recurrentes han sido menos influyentes que las feedforward, ya que de momento, son menos potentes, aunque siguen siendo muy interesantes, y mucho más parecidas a como funcionan las neuronas de nuestro cerebro. Probablemente, estas redes acabarán resolviendo problemas que las redes feedforward no sean capaces de resolver, o tarden demasiado tiempo, o simplemente con más facilidad que estas, pero como de momento se utilizan menos, y están menos desarrolladas, utilizaré solo las feedforward.

3.4. Ejemplo: Red simple para clasificar dígitos manuscritos

Para realizar una primera aproximación a este ejemplo, que ampliaremos y mejoraremos más adelante, vamos a hacer nuestra primera red neuronal que clasifique dígitos manuscritos del 0 al 10, es decir, que la red obtenga como imagen de input lo siguiente:

Figura 3.11: Ejemplo dígito manuscrito

E identifique que se trata de un "5".

Para que la red sea capaz de clasificar correctamente este tipo de imágenes, crearemos una red convolucional que estará formada por 3 capas:

1. La capa de entrada (input layer), estará formada por 764 neuronas sigmoides (ya que las imágenes que vamos a utilizar tienen 28 píxeles de ancho y 28 píxeles de alto, y eso crea un cuadrado de 764 píxeles), y como cada píxel almacena un número, que representa la intensidad del píxel en blanco y negro (0.0 representando el blanco, 1.0 representando el negro y los valores intermedios representando la escala de grises).

   No tendremos las 3 capas (rojo, verde y azul) que tienen todos los píxeles usualmente en las imágenes, de manera que nos ahorramos neuronas gracias a esto.

2. En las capas ocultas (hidden layers), elegiremos nosotros de manera arbitraria el número de neuronas sigmoides que queremos que tengan. En este caso, he decidido que haya una sola capa que tenga quince neuronas, pero no por ninguna razón en particular.

3. Por último, la capa de salida (output layer), tendrá el mismo número de neuronas como posibles resultados tenga la red. En este caso, tendrá diez neuronas sigmoides, representando los diez valores que pueden tomar los dígitos (del 0 al 9).

De esta manera, la red que generaremos tendrá 764 - 15 - 10 neuronas sigmoides, y más o menos, tendrá la siguiente forma:

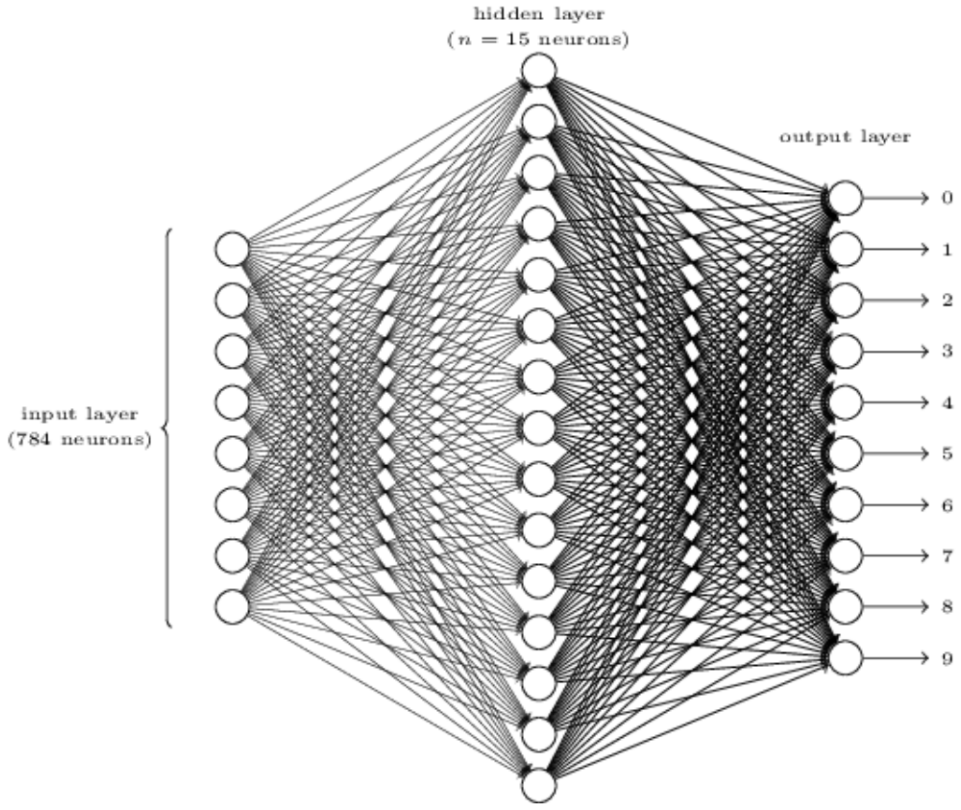

Figura 3.12: Esquema ejemplo red neuronal dígitos manuscritos

3.5. Aprendizaje con el gradiente descendiente

Lo primero que necesitaremos para hacer que la red "aprenda", es tener un dataset que sirva para entrenar la red, con las imágenes bien clasificadas, y un dataset (independiente al anterior), que sirva para probar la red, con la que obtendremos su tasa de error.

Para hacer que esta red "aprenda", utilizaremos una función llamada "función cuadrática de coste":

Donde "C" es la función cuadrática de coste, "w" es la colección de todos los pesos de la red, "b", la colección de todos los bias, "n" es el número de inputs de entrenamiento,

$$C(w,b) \equiv \frac{1}{2n} \sum_x \|y(x) - a\|^2$$

Figura 3.13: Función cuadrática de coste

"a" es el vector de salida de la red cuando "x" es el input, e "y" es la salida deseada (y real) para el input "x".

Para el ejemplo anterior utilizado de los dígitos manuscritos, las imágenes tienen 28x28 píxeles, por lo que cada "x" es un vector con 784 valores, cada uno de ellos representando el valor de la escala de grises de cada píxel de la imagen. El resultado deseado, como se ha expresado antes, viene dado por "y", que para este caso, se trata de un vector que siempre contiene 10 valores, de tal manera que si el resultado deseado para el un input "x" es 6, entonces y(x) = $(0,0,0,0,0,0,1,0,0,0)^T$.

Como es evidente, el resultado de esta función jamás puede ser negativo, y cuanto más se parezcan los resultados de salida de la red a los deseados, "C" será menor, y por lo tanto, si "C" es grande, los resultados de salida de la red distan de los resultados deseados. De esta manera, se ve que el objetivo es minimizar la función "C", es decir, que buscamos un set de pesos y bias que hagan que la función tome el menor valor posible. Esto se hará usando un algoritmo llamado gradiente descendiente.

Para explicar esto, imaginemos primero simplemente una función con muchas variables, por ejemplo $C(v)$, que puede ser cualquier función de valores reales donde $v = v_1, v_2...$ y queremos minimizar dicha función.

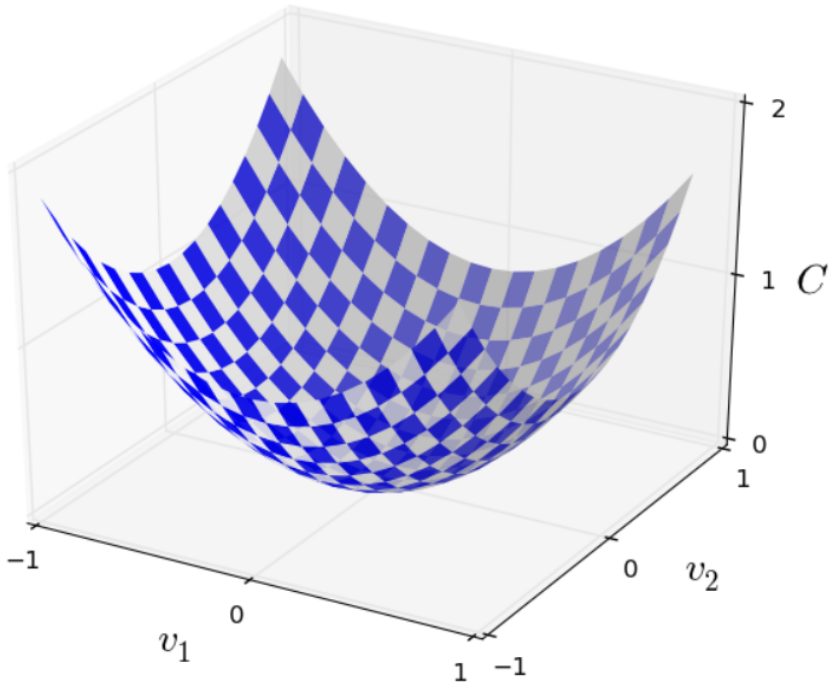

Figura 3.14: Gráfico de ejemplo funcionamiento gradiente descendiente 1

Por supuesto, en este ejemplo, es muy sencillo de ver, pero utilizaremos este ejemplo para explicar el gradiente descendiente.

Una manera de hacer esto es utilizando cálculo, calculando las derivadas y después buscar dónde C es un extremo. Este método es muy efectivo con pocas variables, pero en cuanto hay más de 5 o 6 variables este método tarda demasiado, y no digamos en las redes neuronales más grandes que se utilizan hoy en día, con billones de pesos y bias. Esta manera es inadmisible.

Por suerte para nosotros, existe otra manera de hallar los mínimos, y una analogía con la que es muy sencillo de entender.

Imaginemos por un momento que la función es un valle. Imagina también que ponemos una bola en algún punto de la función. Si imaginamos que existe la gravedad, evidentemente esta bola acabará en algún mínimo en algún momento.

Por lo tanto, esto es lo que vamos a utilizar. Elegimos un punto aleatorio para poner una pelota (imaginaria), y simulamos el movimiento que tomaría la pelota rodando hasta la parte más baja del valle (esto lo conseguimos calculando la primera o segunda derivada de C).

El cálculo nos ayuda a calcular ΔC conociendo Δv_1 y Δv_2:

$$\Delta C \approx \frac{\partial C}{\partial v_1}\Delta v_1 + \frac{\partial C}{\partial v_2}\Delta v_2$$

Figura 3.15: Fórmula gradiente descendiente 1

Tenemos de encontrar las Δv_1 y Δv_2 que hagan ΔC negativa (para que vaya hacia abajo). Para esto, definimos el vector $\Delta v \equiv (\Delta v_1, \Delta v_2)^T$ y el gradiente como:

$$\nabla C \equiv \left(\frac{\partial C}{\partial v_1}, \frac{\partial C}{\partial v_2}\right)^T$$

Figura 3.16: Fórmula gradiente descendiente 2

Y de esta manera, ΔC puede escribirse como $\Delta C \approx \nabla C \cdot \Delta v$.

Gracias a esto, imaginando un η pequeño, podemos calcular el $\Delta v = -\eta \nabla C$.

Y así podemos ir calculando la v mediante:

$$v \rightarrow v' = v - \eta \nabla C$$

Figura 3.17: Fórmula gradiente descendiente 3

Lo que creará un movimiento en la pelota, con lo que cambiará su posición. Después de esto habrá que volver a calcular su v y de esta manera bajará siempre, hasta que llegue al mínimo.

Y así podemos ir calculando la v mediante:

Y esto, ¿Cómo se aplica al aprendizaje de una red neuronal?

La idea es usar el gradiente descendiente para encontrar los pesos y los bias que minimicen el coste C de la función cuadrática de coste. Si reemplazamos en la anterior fórmula las variables de v_j por las variables w_k y b_l, obtenemos las siguientes fórmulas:

De esta manera se consiguen los pesos y los bias que minimizan al máximo el coste C de la función cuadrática de coste, y con ello, aumentaremos el número de resultados bien clasificados.

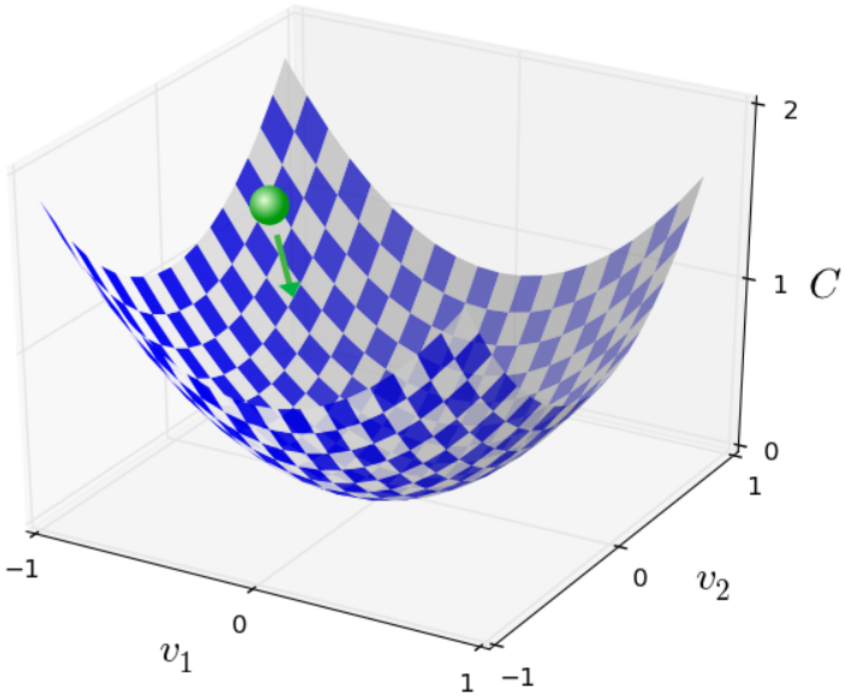

Figura 3.18: Gráfico de ejemplo funcionamiento gradiente descendiente 2

$$w_k \to w'_k = w_k - \eta \frac{\partial C}{\partial w_k}$$
$$b_l \to b'_l = b_l - \eta \frac{\partial C}{\partial b_l}.$$

Figura 3.19: Fórmula gradiente descendiente 4

Sin embargo, esto va muy lento, ya que para computar ∇C, tenemos que calcular todos los ∇C_x y después calcular su media. Para hacer esto más rápido, utilizamos el gradiente de descenso estocástico. Lo que se hace es calcular algunos ∇C_x de manera aleatoria, y así obtenemos un ∇C bastante bueno. Para esto se cogen de manera aleatoria m inputs, a los que llamaremos mini-lotes y calcularemos su media, que debería ser parecida a la obtenida si utilizaremos todos los ∇C_x. Se van cogiendo subconjuntos aleatorios (mini lotes) y se van entrenando los pesos y los bias:

Cuando se han utilizado todos los elementos del conjunto de entrenamiento, se acaba

$$w_k \to w'_k = w_k - \frac{\eta}{m} \sum_j \frac{\partial C_{X_j}}{\partial w_k}$$

$$b_l \to b'_l = b_l - \frac{\eta}{m} \sum_j \frac{\partial C_{X_j}}{\partial b_l},$$

Figura 3.20: Fórmula gradiente descendiente 5

la epoch, y se empieza la siguiente época de entrenamiento. Esto del gradiente y la razón por la que se coge un mini-lote es porque solo queremos saber la dirección aproximada a la que tiene que "rodar la pelota".

3.6. Código que clasifica dígitos

Utilizaremos un pequeño ejemplo para dar a entender cómo funciona un red neuronal que clasifica dígitos manuscritos. Lo escribiremos en Python (2.7) ya que se puede programar una red relativamente sencilla (tan solo 74 líneas de código), con facilidad.

Para entrenarla, vamos a utilizar un dataset llamado MNIST, que contiene 50000 imágenes para el conjunto de entrenamiento (con lo que se ajustan los pesos y los bias), 10000 para el conjunto de validación (con los que se va comprobando que los pesos y los bias se van ajustando correctamente) y 10000 para el conjunto de prueba (con los que obtendremos una tasa de error que nos servirá para comprobar la bondad de nuestra red neuronal).

Vamos a ir explicando poco a poco como funciona el código:

Este es el código que se utilizará para inicializar el objeto "Network".

El array "sizes" contiene el número de neuronas de cada capa, de tal manera que si se llama a la clase Network de la siguiente manera: "net = Network([2, 3, 1])", significará que la primera capa tendrá 2 neuronas, la segunda capa, 3 y la tercera y última, 1.

Los bias y los pesos de todas las capas sin contar la primera, se inicializan de manera aleatoria utilizando "np.random.randn", que es una función que genera un número aleatorio con media 0 y desviación estándar 1, con una distribución Gausiana. De esta manera, tenemos datos con los que empezar a trabajar, aunque hay manera mejores de hacerlo.

```
class Network(object):

    def __init__(self, sizes):
        self.num_layers = len(sizes)
        self.sizes = sizes
        self.biases = [np.random.randn(y, 1) for y in sizes[1:]]
        self.weights = [np.random.randn(y, x)
                        for x, y in zip(sizes[:-1], sizes[1:])]
```

Figura 3.21: Código parte 1

Cabe destacar también, que los bias y los pesos están almacenados en matrices, y que, por ejemplo, net.weights[1], contiene el peso de la conexión entre la segunda y la tercera capa de neuronas. Para entenderlo mejor, simplemente imaginemos la matriz "w_{jk}". Esta, representa el peso de la conexión entre la neurona k de la segunda capa, y la neurona j de la tercera capa. De esta manera, puede parecer que j y k se han intercambiado, pero en realidad, esta nomenclatura nos va a ayudar a calcular el vector de activación de la tercera capa de neuronas, ya que para calcularla, se utiliza la siguiente fórmula:

$$a' = \sigma(wa + b)$$

Figura 3.22: Formula vector de activación

En esta ecuación, a representa el vector de activación de la segunda capa de neuronas, a' el de la tercera capa de neuronas, w, la matriz de pesos, b, la matriz de bias, y σ, es la función que se va a aplicar a cada elemento del vector $wa + b$ [3.5]. Esta función, devuelve el mismo resultado que [3.6].

A continuación, escribimos esta misma función, la función sigmoide, en un método de la siguiente manera:

```
def sigmoid(z):
    return 1.0/(1.0+np.exp(-z))
```

Figura 3.23: Código parte 2

Otro método que vamos a utilizar es el método "feedforward", en el que simplemente aplica la ecuación anterior [3.23] a cada capa:

```
def feedforward(self, a):
    """Return the output of the network if "a" is input."""
    for b, w in zip(self.biases, self.weights):
        a = sigmoid(np.dot(w, a)+b)
    return a
```

Figura 3.24: Código parte 3

Hasta ahora, tenemos las funciones de activación de las capas, pero nuestra red, todavía no realiza su función más importante, aprender. Para eso utilizaremos el siguiente método, cuyo objetivo es implementar el gradiente descendiente estocástico:

```
def SGD(self, training_data, epochs, mini_batch_size, eta,
        test_data=None):
    """Train the neural network using mini-batch stochastic
    gradient descent.  The "training_data" is a list of tuples
    "(x, y)" representing the training inputs and the desired
    outputs.  The other non-optional parameters are
    self-explanatory.  If "test_data" is provided then the
    network will be evaluated against the test data after each
    epoch, and partial progress printed out.  This is useful for
    tracking progress, but slows things down substantially."""
    if test_data: n_test = len(test_data)
    n = len(training_data)
    for j in xrange(epochs):
        random.shuffle(training_data)
        mini_batches = [
            training_data[k:k+mini_batch_size]
            for k in xrange(0, n, mini_batch_size)]
        for mini_batch in mini_batches:
            self.update_mini_batch(mini_batch, eta)
        if test_data:
            print "Epoch {0}: {1} / {2}".format(
                j, self.evaluate(test_data), n_test)
        else:
            print "Epoch {0} complete".format(j)
```

Figura 3.25: Código parte 4

El “training_data” es una lista de tuplas (x, y), que representan los inputs y sus correspondientes outputs.

La variable “epoch” representa el número de épocas en las que se va a entrenar.

Un “mini_batch_size” representa el tamaño que van a tener los lotes de las muestras.

La variable “eta” contiene la tasa de aprendizaje (η).

El parámetro opcional “test_data” se introduce cuando el programa evalúa la red después de cada época de entrenamiento y se quiere que imprima el progreso parcial.

Esto es útil para llevar una constancia del progreso, aunque ralentiza el proceso sustancialmente.

El funcionamiento del programa es el siguiente:

1. Comienza tomando de manera aleatoria unos cuantos inputs del conjunto de entrenamiento.
2. Divide esos inputs en los diferentes mini-lotes, que tendrán el tamaño apropiado.
3. Aplicamos un paso del gradiente descendiente a cada mini-lote mediante la línea self.update_mini_batch(mini_batch, eta), la cual actualiza la red de pesos y bias de acuerdo con una iteración del gradiente descendiente, utilizando los inputs del training data del mini-lote.<s

El método update_mini_batch, que realiza lo previamente mencionado, es el siguiente:

```
def update_mini_batch(self, mini_batch, eta):
    """Update the network's weights and biases by applying
    gradient descent using backpropagation to a single mini batch.
    The "mini_batch" is a list of tuples "(x, y)", and "eta"
    is the learning rate."""
    nabla_b = [np.zeros(b.shape) for b in self.biases]
    nabla_w = [np.zeros(w.shape) for w in self.weights]
    for x, y in mini_batch:
        delta_nabla_b, delta_nabla_w = self.backprop(x, y)
        nabla_b = [nb+dnb for nb, dnb in zip(nabla_b, delta_nabla_b)]
        nabla_w = [nw+dnw for nw, dnw in zip(nabla_w, delta_nabla_w)]
    self.weights = [w-(eta/len(mini_batch))*nw
                    for w, nw in zip(self.weights, nabla_w)]
    self.biases = [b-(eta/len(mini_batch))*nb
                   for b, nb in zip(self.biases, nabla_b)]
```

Figura 3.26: Código parte 5

Casi todo el trabajo está hecho por la línea: delta_nabla_b, delta_nabla_w = self.backprop(x, y).

Lo que hace esta línea es llamar a un algoritmo de “backpropagation”. Lo que hace el

método es computar los gradientes de cada ejemplo de entrenamiento del mini-lote y va actualizando self.weights y self.biases. El código completo es el siguiente:

```
"""
network.py
~~~~~~~~~~

A module to implement the stochastic gradient descent learning
algorithm for a feedforward neural network.  Gradients are calculated
using backpropagation.  Note that I have focused on making the code
simple, easily readable, and easily modifiable.  It is not optimized,
and omits many desirable features.
"""

#### Libraries
# Standard library
import random

# Third-party libraries
import numpy as np

class Network(object):

    def __init__(self, sizes):
        """The list ``sizes`` contains the number of neurons in the
        respective layers of the network.  For example, if the list
        was [2, 3, 1] then it would be a three-layer network, with the
        first layer containing 2 neurons, the second layer 3 neurons,
        and the third layer 1 neuron.  The biases and weights for the
        network are initialized randomly, using a Gaussian
        distribution with mean 0, and variance 1.  Note that the first
        layer is assumed to be an input layer, and by convention we
        won't set any biases for those neurons, since biases are only
        ever used in computing the outputs from later layers."""
        self.num_layers = len(sizes)
        self.sizes = sizes
        self.biases = [np.random.randn(y, 1) for y in sizes[1:]]
        self.weights = [np.random.randn(y, x)
                        for x, y in zip(sizes[:-1], sizes[1:])]

    def feedforward(self, a):
        """Return the output of the network if ``a`` is input."""
        for b, w in zip(self.biases, self.weights):
            a = sigmoid(np.dot(w, a)+b)
        return a

    def SGD(self, training_data, epochs, mini_batch_size, eta,
            test_data=None):
        """Train the neural network using mini-batch stochastic
        gradient descent.  The ``training_data`` is a list of tuples
        ``(x, y)`` representing the training inputs and the desired
        outputs.  The other non-optional parameters are
        self-explanatory.  If ``test_data`` is provided then the
        network will be evaluated against the test data after each
        epoch, and partial progress printed out.  This is useful for
        tracking progress, but slows things down substantially."""
        if test_data: n_test = len(test_data)
        n = len(training_data)
        for j in xrange(epochs):
            random.shuffle(training_data)
            mini_batches = [
```

```
                training_data[k:k+mini_batch_size]
                for k in xrange(0, n, mini_batch_size)]
            for mini_batch in mini_batches:
                self.update_mini_batch(mini_batch, eta)
            if test_data:
                print "Epoch {0}: {1} / {2}".format(
                    j, self.evaluate(test_data), n_test)
            else:
                print "Epoch {0} complete".format(j)

    def update_mini_batch(self, mini_batch, eta):
        """Update the network's weights and biases by applying
        gradient descent using backpropagation to a single mini batch.
        The ``mini_batch`` is a list of tuples ``(x, y)``, and ``eta``
        is the learning rate."""
        nabla_b = [np.zeros(b.shape) for b in self.biases]
        nabla_w = [np.zeros(w.shape) for w in self.weights]
        for x, y in mini_batch:
            delta_nabla_b, delta_nabla_w = self.backprop(x, y)
            nabla_b = [nb+dnb for nb, dnb in zip(nabla_b, delta_nabla_b)]
            nabla_w = [nw+dnw for nw, dnw in zip(nabla_w, delta_nabla_w)]
        self.weights = [w-(eta/len(mini_batch))*nw
                        for w, nw in zip(self.weights, nabla_w)]
        self.biases = [b-(eta/len(mini_batch))*nb
                       for b, nb in zip(self.biases, nabla_b)]

    def backprop(self, x, y):
        """Return a tuple ``(nabla_b, nabla_w)`` representing the
        gradient for the cost function C_x.  ``nabla_b`` and
        ``nabla_w`` are layer-by-layer lists of numpy arrays, similar
        to ``self.biases`` and ``self.weights``."""
        nabla_b = [np.zeros(b.shape) for b in self.biases]
        nabla_w = [np.zeros(w.shape) for w in self.weights]
        # feedforward
        activation = x
        activations = [x] # list to store all the activations, layer by layer
        zs = [] # list to store all the z vectors, layer by layer
        for b, w in zip(self.biases, self.weights):
            z = np.dot(w, activation)+b
            zs.append(z)
            activation = sigmoid(z)
            activations.append(activation)
        # backward pass
        delta = self.cost_derivative(activations[-1], y) * \
            sigmoid_prime(zs[-1])
        nabla_b[-1] = delta
        nabla_w[-1] = np.dot(delta, activations[-2].transpose())
        # Note that the variable l in the loop below is used a little
        # differently to the notation in Chapter 2 of the book.  Here,
        # l = 1 means the last layer of neurons, l = 2 is the
        # second-last layer, and so on.  It's a renumbering of the
        # scheme in the book, used here to take advantage of the fact
        # that Python can use negative indices in lists.
        for l in xrange(2, self.num_layers):
            z = zs[-l]
            sp = sigmoid_prime(z)
            delta = np.dot(self.weights[-l+1].transpose(), delta) * sp
            nabla_b[-l] = delta
            nabla_w[-l] = np.dot(delta, activations[-l-1].transpose())
        return (nabla_b, nabla_w)

```

```
120     def evaluate(self, test_data):
121         """Return the number of test inputs for which the neural
122         network outputs the correct result. Note that the neural
123         network's output is assumed to be the index of whichever
124         neuron in the final layer has the highest activation."""
125         test_results = [(np.argmax(self.feedforward(x)), y)
126                         for (x, y) in test_data]
127         return sum(int(x == y) for (x, y) in test_results)
128 
129     def cost_derivative(self, output_activations, y):
130         """Return the vector of partial derivatives \partial C_x /
131         \partial a for the output activations."""
132         return (output_activations-y)
133 
134 #### Miscellaneous functions
135 def sigmoid(z):
136     """The sigmoid function."""
137     return 1.0/(1.0+np.exp(-z))
138 
139 def sigmoid_prime(z):
140     """Derivative of the sigmoid function."""
141     return sigmoid(z)*(1-sigmoid(z))
```

Listing 3.1: Código completo

Para ejecutar el programa, se puede utilizar el programa de ayuda mnist_loader:

```
>>> import mnist_loader
>>> training_data, validation_data, test_data = \
... mnist_loader.load_data_wrapper()
```

Figura 3.27: Código parte 6

En este ejemplo, se van a utilizar 30 neuronas en la capa oculta, por lo que se llamará a la red de la siguiente manera:

```
>>> import network
>>> net = network.Network([784, 30, 10])
```

Figura 3.28: Código parte 7

Y el descenso de gradiente estocástico, lo seleccionamos con 30 épocas (epoch), los tamaños de los mini-lotes de 10 y una tasa de aprendizaje (η) de 3.0:

```
>>> net.SGD(training_data, 30, 10, 3.0, test_data=test_data)
```

Figura 3.29: Código parte 8

El resultado del código tiene el siguiente aspecto:

```
Epoch 0: 9129 / 10000
Epoch 1: 9295 / 10000
Epoch 2: 9348 / 10000
...
Epoch 27: 9528 / 10000
Epoch 28: 9542 / 10000
Epoch 29: 9534 / 10000
```

Figura 3.30: Resultado código

Como puede observarse, muestra, en función de las épocas, el número de imágenes bien clasificadas de las totales. En la primera época, ya clasifica correctamente 9129 de las 10000, y en la última, 9534, lo que quiere decir, que con esta red, y con estos datos, la red clasifica correctamente en torno a un 95 % de las imágenes.

Si volvemos a ejecutar la red utilizando 100 neuronas en la capa intermedia, evidentemente, tardará más tiempo en ejecutarse, pero conseguirá eventualmente un resultado mejor (más de un 96 %), aunque puede que haya ejecuciones que empeoren el porcentaje, debido a la varianza.

Los hiperparámetros de la red (el número de épocas de entrenamiento, el tamaño de los mini-lotes y la tasa de aprendizaje (η)) es importante que sean todos razonables. Por ejemplo, si ponemos una tasa de aprendizaje de 0.001, conseguiremos los siguientes resultados:

```
Epoch 0: 1139 / 10000
Epoch 1: 1136 / 10000
Epoch 2: 1135 / 10000
...
Epoch 27: 2101 / 10000
Epoch 28: 2123 / 10000
Epoch 29: 2142 / 10000
```

Figura 3.31: Resultado código 2

Como puede observarse, se obtiene un resultado notablemente peor, aunque va mejorando lentamente a lo largo de las “epoch”.

Este no es el caso de este otro ejemplo, en el que se ejecuta el programa con 30 neuronas en la capa oculta y una tasa de aprendizaje (η) de 100.0, donde obtengo los siguientes resultados:

```
Epoch 0: 1009 / 10000
Epoch 1: 1009 / 10000
Epoch 2: 1009 / 10000
Epoch 3: 1009 / 10000
...
Epoch 27: 982 / 10000
Epoch 28: 982 / 10000
Epoch 29: 982 / 10000
```

Figura 3.32: Resultado código 3

Y viendo estos resultados, y sin saber que con una tasa de aprendizaje de 3.0 salen buenos resultados, no sabemos qué le pasa a la red. No sabemos si hemos inicializado los pesos y bias de tal manera que hace difícil a la red aprender, o no tenemos suficientes datos en el dataset de entrenamiento, o no tenemos suficientes épocas, o la tasa de aprendizaje es demasiado baja, o demasiado alta... Por estas razones, debuguear una red neuronal, es difícil.

3.7. Un acercamiento al deep learning

Es cierto que el interior de una red neuronal es bastante misteriosa ahora mismo. Para que nosotros podamos entender cómo funciona, vamos a poner un ejemplo en el que se entiende bastante bien cómo funciona por dentro.

Imagina que queremos crear una red neuronal que descubra si en una imagen hay caras humanas o no. Para esto, una heurística que podemos utilizar para resolver el problema es separarlo en subproblemas, como si hay un ojo en la parte superior izquierda, un ojo en la parte superior derecha, una nariz en el medio, una boca en medio abajo, pelo en la parte de arriba...Si se dan esos casos, probablemente estemos ante una cara.

Por supuesto, esta heurística tiene defectos. Por ejemplo, si la persona es calva, o por el ángulo no se ve un ojo... pero esto es solo un ejemplo para que podamos entender cómo funciona, así que obviaremos estos defectos.

A su vez, el ojo se puede subdividir en pestañas, ceja, iris, pupila... y así sucesivamente hasta que se subdividan en elementos cada vez más simples hasta llegar a los píxeles, que es nuestra entrada a la red.

De esta manera, una pregunta que parece muy difícil de responder, como si en una imagen hay una cara humana o no, se puede subdividir en preguntas cada vez más sencillas, hasta llegar a los píxeles, con lo que acaba siendo evidente.

3.8. Mejoras

Como hemos visto, en el mejor resultado que hemos obtenido, la solución con mejor tasa de acierto era de en torno al 95 %. ¿Esto se puede mejorar?

Por suerte, la respuesta es sí, y aquí enunciamos algunas de las maneras que se han encontrado para mejorar la velocidad o el porcentaje de acierto de la red.

3.8.1. Función de coste de entropía cruzada

Los seres humanos, aprendemos rápido de nuestros errores cuanto más lejos están del acierto. Sin embargo, las redes neuronales que estamos estudiando no funcionan así.

Imagina una neurona sigmoide con un solo input, de la que queremos que, sea cual sea el input, el output sea 0. Si escribimos los pesos y bias manualmente, es muy fácil de hacer. Sin embargo, si inicializamos estos valores de manera aleatoria y ponemos a la red a trabajar como si fuera una red de una sola capa y una sola neurona, no es tan evidente. Si inicializamos el valor del peso del input a 0.6, y el valor del bias a 0.9 (por ejemplo), el output inicial será 0.82 (esto es fácil de calcular utilizando la fórmula [3.6].

Pues bien, utilizando la función de coste [3.13] y una tasa de aprendizaje $\eta = 0.15$, tarda 300 épocas en llegar a un output de 0.09.

Sin embargo, si inicializamos el peso a 2.0 y el bias a 2.0, el output inicial es de 0.98, y en 300 épocas, se baja a 0.2.

Esto puede parecer “no tan malo” a primera vista, pero si nos fijamos en la forma que se crea en la función que relaciona el output con el número de épocas, nos damos cuenta de algo muy importante. Por la forma que tienen, se ve que cuanto más lejos están los pesos y los bias de los “ideales”, más épocas le cuesta a la red obtener buenos resultados.

Esto se puede ver fácilmente en estas dos funciones, con los pesos y bias mencionados previamente, y la misma tasa de aprendizaje ($\eta = 0.15$):

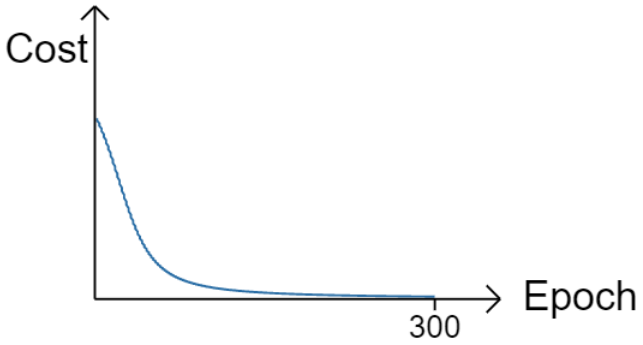

Figura 3.33: Relación output - épocas (peso 0.6, bias 0.9)

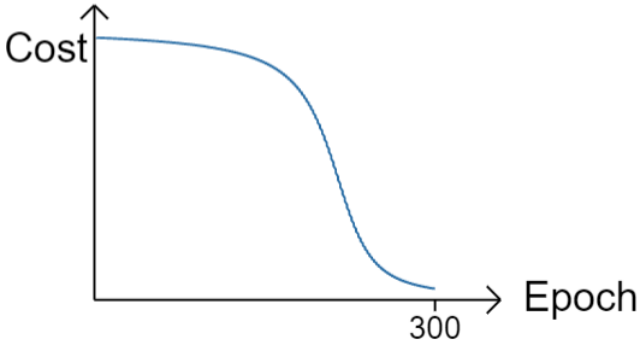

Figura 3.34: Relación output - épocas (peso 2.0, bias 2.0)

Para mejorar esto, se empezó a utilizar un método para calcular el coste que evitara esto. Voy a omitir la demostración matemática, porque no es el objetivo del trabajo, pero el que quiera profundizar, dejaré en la bibliografía lugares donde esto se explica de una forma mucho más detallada. La función que se utiliza, pasa de ser [3.13] a ser la siguiente:

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

Figura 3.35: Relación output - épocas (peso 0.6, bias 0.9)

En esta función, las variables son las mismas que en la función anterior, pero utilizando este método, se evita esa ralentización que se produce al distar mucho los pesos y bias originales de los finales. Los resultados de estos dos mismos ejemplos utilizando este método, son los siguientes:

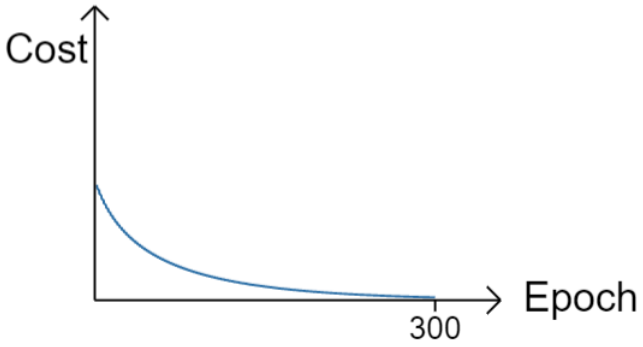

Figura 3.36: Relación output - épocas (peso 0.6, bias 0.9)

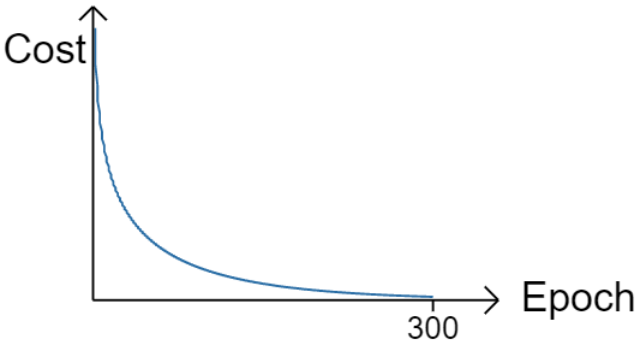

Figura 3.37: Relación output - épocas (peso 2.0, bias 2.0)

Como curiosidad, para estos dos últimos ejemplos, la tasa de aprendizaje (η), es de 0.005, aunque esto no es una reducción, ya que ambas magnitudes no son comparables. Es como comparar peras con olmos.

En el fondo, da igual cómo de bueno acabe siendo o la rapidez de mejora, lo importante es la diferencia entre las formas de las gráficas, y lo que ello conlleva.

3.8.2. Sobreajuste y regularización

El sobreajuste es el efecto de sobreentrenar un algoritmo de aprendizaje. El algoritmo de aprendizaje debe alcanzar un estado en el que será capaz de predecir el resultado utilizando la generalización. Esto se entiende muy fácilmente con la siguiente imagen:

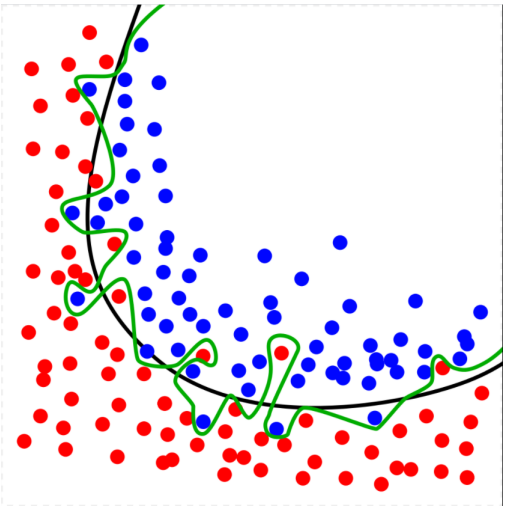

Figura 3.38: Ejemplo sobreajuste

Como puede observarse, la línea negra probablemente tendrá menos errores en datos futuros que la verde, pese a que la verde diferencie perfectamente los datos con los que contamos.[Wike]

Este es uno de los mayores problemas de las redes neuronales, y necesitamos una manera de detectar cuando se está produciendo un sobreajuste para no sobreentrenar la red, y desarrollar técnicas que reduzcan los efectos del sobreentrenamiento.

Una de las maneras más evidentes es llevar un seguimiento de la tasa de acierto mientras nuestra red se va entrenando, y utilizar el test de prueba, es decir, que si vemos que la tasa de error no está mejorando, debemos dejar de entrenar.

Otra manera es utilizando el dataset de validación. Hasta ahora, solo hemos hecho incapié en los dataset de entrenamiento, con los que se entrenaba la red y se configuraban los pesos y bias, y el dataset de prueba, con el que se obtenía la bondad de la red con la tasa de error. Si hacemos un poco de memoria, dije que el dataset que estábamos utilizando tenía 70000 imágenes, de las cuales 50000 iban a ser utilizadas para el dataset de entrenamiento, 10000 para el dataset de prueba y 10000 para el dataset de validación, pero nunca llegué a explicar para qué valía este.

Esta segunda manera, utilizando el dataset de validación, es significativamente mejor que la anterior. Para realizar esta comprobación, comprobamos la precisión del dataset de validación al final de cada época. Cuando la precisión se sature, paramos el entrenamiento en esa época. A esta técnica se la llama “early stopping”, que en inglés significa “paro rápido”.

En la práctica no sabemos cuando se ha saturado, así que lo que hacemos es seguir

entrenando hasta que estemos seguros de que se ha saturado.

Los hiperparámetros (el número de épocas, la tasa de aprendizaje, la arquitectura, el tamaño de los mini-lotes...), se seleccionan utilizando el dataset de validación, y esto es así, y no utilizando el dataset de prueba porque si se utilizara directamente el dataset de prueba, podría darse el caso que la red encaje de una manera correcta para los datos del dataset de prueba concreto, pero que no llegue a generalizar. De esta manera, tenemos una última prueba para asegurar que se ha generalizado correctamente. A este método se le suele llamar "hold out", ya que el dataset de validación se aparta, que es lo que significa "hold out" en inglés.

Hay maneras de reducir el sobreentrenamiento, ya sea con datasets más grandes, utilizando redes más pequeñas, o utilizando las técnicas de regularización. No ahondaré en estas técnicas de regularización, pero dejo en la sección de bibliografía más información por si se quiere profundizar.

Para obtener datasets más grandes, se pueden conseguir más datos, lo que suele ser bastante caro, pero también existen otras maneras de conseguir más datos, como por ejemplo, realizar pequeñas rotaciones a las imágenes, por ejemplo:

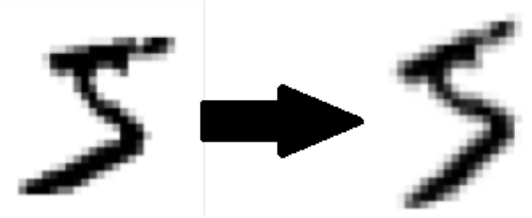

Figura 3.39: Ejemplo expansión de dataset barata

Ya que el dígito sigue siendo reconocible, pero a nivel de píxeles, es diferente.

También se pueden realizar recortes de imágenes en las que siga siendo reconocible el dígito, traslaciones... con los que conseguimos mejores resultados en nuestras redes. De hecho, en casi todos los casos y para casi todos los algoritmos, cuantos más datos se tengan para entrenar, mejor será la red (mejor tasa de acierto tendrá).

3.8.3. Inicialización de los pesos

Hasta ahora, inicializábamos los pesos y los bias mediante variables aleatorias gaussianas independientes, con media 0 y desviación estándar 1, sin embargo, existen maneras mejores de inicializar estas variables.

La razón por la que esto no es lo más óptimo es porque para ciertos inputs (solo unos y ceros, por ejemplo), se acaban generando salidas de neuronas sigmoides muy cercanas a los dos extremos, ya sea 1 o 0, y un pequeño cambio en un peso o un bias, hará un cambio minúsculo en la siguiente capa, y así sucesivamente. La red se satura y tardará muchas épocas en aprender.

Este es un problema similar al que resolvimos con la función de coste de entropía cruzada, en el que veíamos que cuanto más se alejara un dato del dato óptimo para la red, más tiempo tardaba en optimizarse. Sin embargo, esta función solo nos ayudaba para la capa de salida, no para las capas ocultas.

La manera que se ha encontrado de mejorar esto es utilizar una distribución gausiana aleatoria con la misma media (0), pero una desviación típica diferente ($1/\sqrt{n_{in}}$, siendo n_{in} el número de pesos de entrada).

De esta manera, la distribución Gausiana pasa de tener la siguiente forma:

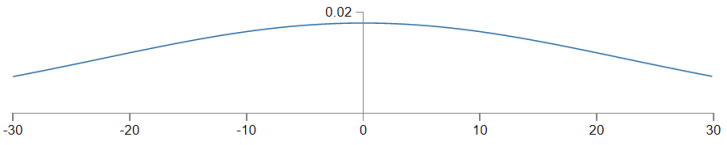

Figura 3.40: Curva distribución Gausiana 1

A tener la siguiente forma:

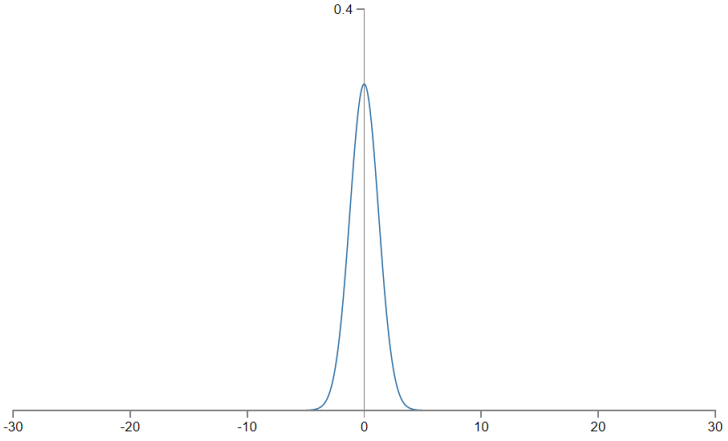

Figura 3.41: Curva distribución Gausiana 2

En el caso de los bias, no generan saturación, así que podemos seguir utilizando el

mismo código para generarlos. Hay quien los inicializa todos a 0, pero no genera diferencias.

Como puede observarse en esta gráfica en la que se representan las tasas de acierto en función de las épocas de entrenamiento, esta manera de inicializar los pesos es notablemente mejor, pese a que al final se llegue al mismo resultado:

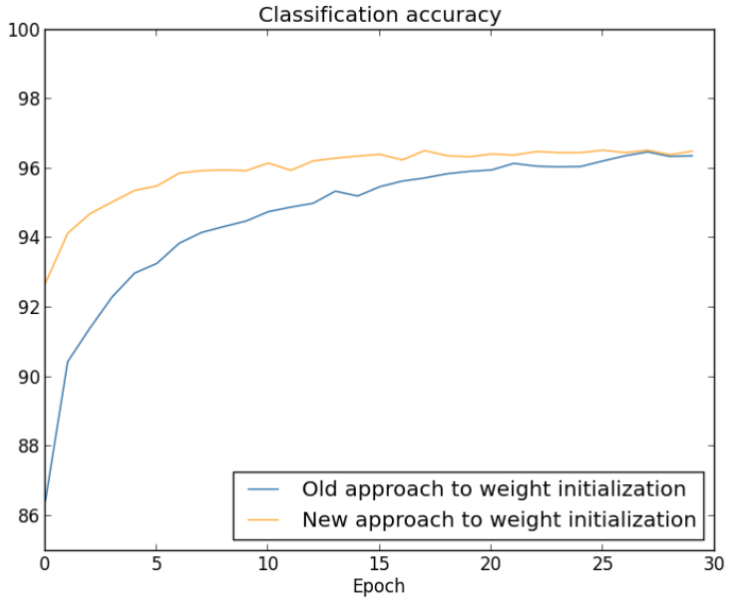

Figura 3.42: Diferencias en tasa de acierto en función de la inicialización pesos

3.8.4. Cómo elegir los hiperparámetros

Como hemos explicado previamente, los hiperparámetros son: la tasa de aprendizaje (η), el tamaño de los mini-lotes, el número de épocas, el número de neuronas ocultas, el parámetro de regularización (λ)...

Imagina que estamos en la siguiente situación: estamos haciendo la red que clasifica dígitos y nos sabemos qué parámetros utilizar. Por casualidad, conseguimos buenos parámetros para el número de neuronas ocultas (30), el tamaño de los mini-lotes (10), el número de épocas (30), pero elegimos mal la tasa de aprendizaje (10), y el parámetro de regularización (1000). Con estos datos obtenemos los siguientes resultados:

Como puede observarse, estos resultados no son aceptables. Hasta un bebé sin ningún conocimiento, pulsando botones al azar, puede obtener un resultado similar o mejor.

```
>>> import mnist_loader
>>> training_data, validation_data, test_data = \
... mnist_loader.load_data_wrapper()
>>> import network2
>>> net = network2.Network([784, 30, 10])
>>> net.SGD(training_data, 30, 10, 10.0, lmbda = 1000.0,
... evaluation_data=validation_data, monitor_evaluation_accuracy=True)
Epoch 0 training complete
Accuracy on evaluation data: 1030 / 10000

Epoch 1 training complete
Accuracy on evaluation data: 990 / 10000

Epoch 2 training complete
Accuracy on evaluation data: 1009 / 10000

...

Epoch 27 training complete
Accuracy on evaluation data: 1009 / 10000

Epoch 28 training complete
Accuracy on evaluation data: 983 / 10000

Epoch 29 training complete
Accuracy on evaluation data: 967 / 10000
```

Figura 3.43: Ejemplo elección hiperparámetros

Un acercamiento que se suele hacer para establecer los hiperparámetros consiste en subdividir el problema inicial. En el ejemplo que estábamos mostrando, se podría, por ejemplo, hacer una red neuronal que diferencia "0" de "1", y de esta manera, será más fácil encontrar unos hiperparámetros que los diferencien y aumentamos el número de pruebas que se pueden hacer en la misma cantidad de tiempo, ya que tenemos un 20 % de datos de los que teníamos inicialmente. Existen otras maneras de acelerar el proceso, como reduciendo el número de capas ocultas, o reduciendo la impresión de resultados intermedios, pero son menos relevantes.

Poco a poco, se pueden ir variando los hiperparámetros hasta conseguir unos hiperparámetros con los que se consiga una tasa de acierto mayor a la aleatoria.

Una vez hemos conseguido esto, vamos realizando pequeñas modificaciones hacia

arriba y hacia abajo en η, en λ, en el número de neuronas de las capas ocultas... Hasta conseguir la mejor combinación.

No es una manera muy efectiva, pero es la mejor que se ha encontrado hasta ahora, y como es evidente, como casi todo en la vida, lo más difícil es empezar. Sin embargo, sí que hay trucos que pueden acelerar este proceso:

- Tasa de aprendizaje (η): Imaginamos tres valores diferentes para η: $\eta = 0.025$, $\eta = 0.25$ y $\eta = 2.5$. Utilizando el resto de hiperparámetros que ya hemos escrito antes (30 épocas, mini-lotes de 10, $\lambda = 5.0$ y 50000 imágenes). Las gráficas de coste en función de las épocas que se genera es el siguiente:

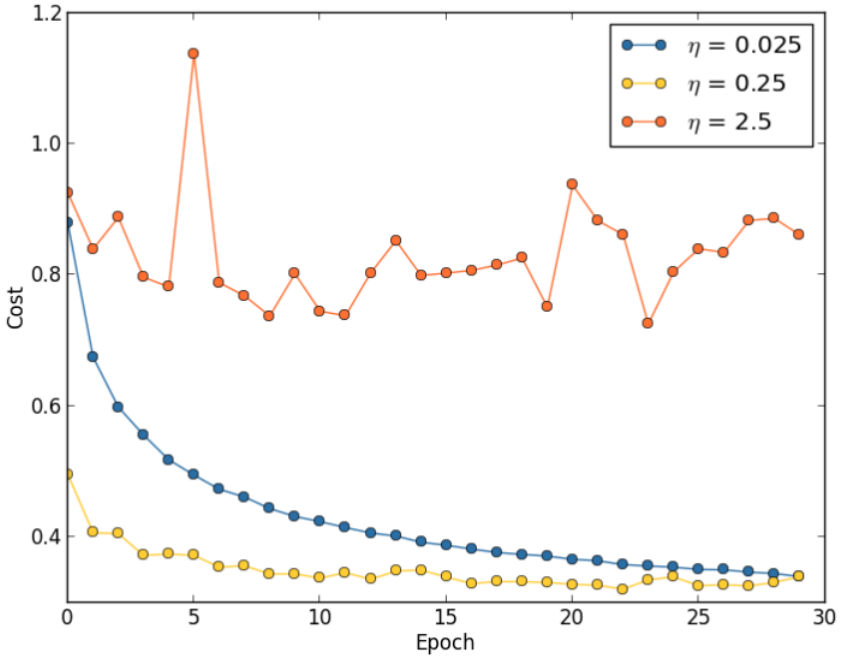

Figura 3.44: Evolución según tasa de aprendizaje

Para entender estos datos, tenemos que volver a imaginarnos la bola que baja por el valle [3.14]. Si η es demasiado grande (2.5), los pasos son tan grandes que, pese a que tome bien la dirección hacia la que se hace mínimo, vuelve a subir por el valle, y puede acabar incluso en una posición peor que en la que se encontraba inicialmente. Cuando toma el valor 0.25, llega a una zona cercana al mínimo, y ahí llegamos al mismo problema que en el caso anterior. Por último, cuando $\eta = 0.025$, estamos en una situación similar al del 0.25, pero tarda más tiempo en llegar a la solución.

Por esta razón, para calcular un buen valor para esta variable en un problema

concreto, se coge un valor muy pequeño de η, y se ve si ese valor decrece el coste durante unas cuantas épocas, se aumenta ligeramente, y así sucesivamente hasta que encontremos un valor que disminuya el coste durante unas cuantas (pocas) épocas, y luego se quede oscilando, y una vez hemos obtenido ese valor, reducirlo a la mitad, y de esta manera, conseguiremos un valor ligeramente mejor, pero sin utilizar demasiadas epoch.

Hay otra manera que se ha estado desarrollando durante los últimos años. La idea es empezar con un valor relativamente grande, para que en pocas épocas, se decremente bastante el coste, y después ir reduciendo paulatinamente la tasa de aprendizaje para ir ajustando de una manera más precisa todos los pesos.

- Número de épocas: Como hemos dicho antes, la manera que tenemos de encontrar el número de épocas es utilizando el dataset de validación. Cuando la tasa de acierto de este dataset disminuye, paramos y ese es el número de épocas que le van bien a la red. A esto se le llama "early stopping" que significa parada temprana en inglés, y no es muy precisa porque casi siempre acaba mejorando pese a que disminuya ligeramente en una época, y por esta razón, también se puede esperar a que la no mejora sea continuada es unas cuantas épocas (10 o así), e incluso de esta manera a veces se pierde algo de mejora, por lo que estas 10 épocas pueden a aumentar a 20 e incluso 50 épocas si buscamos el mejor resultado.
- Parámetro de regularización (λ): Este parámetro se utiliza en una función de coste que mejora el sobreentrenamiento. Dicha fórmula es la siguiente:

$$C = -\frac{1}{n}\sum_{xj}\left[y_j \ln a_j^L + (1-y_j)\ln(1-a_j^L)\right] + \frac{\lambda}{2n}\sum_w w^2$$

Figura 3.45: Función cuadrática de coste mejora sobreentrenamiento

No explicaré el porqué, aunque podréis encontrarlo en el link que dejo en la bibliografía. El caso es que hace que la red "prefiera" pesos pequeños que pesos grandes. Cuando λ es pequeña, se prefiere minimizar la función de coste original, pero cuando λ es grande, se prefieren los pesos pequeños.

Para buscar un buen valor para esta variable, es recomendable empezar igualándolo a 0.0, obtener un buen valor de η como hemos explicado, y liego ir variando λ, empezando con 1.0, t ir incrementándolo o decrementándolo mediante factores de 10. Una vez se ha obtenido un buen valor de λ, se vuelve a calcular η.

- Tamaño de los mini-lotes: Recordemos, lo primero de todo, que el tamaño de los mini-lotes es el número de ejemplos aleatorios que se toman para calcular una estimación del gradiente, pero sin coger todos los ejemplos, ya que sería demasiado lento.

  Para entender esto mejor, es como si quisiésemos ir al Polo Norte y nuestra brújula no apuntase exactamente al norte, sino que tuviera un margen de error de 10-20 grados. Incluso en este caso, acabaríamos llegando al Polo Norte.

  El caso es que si elegimos un tamaño de mini-lotes demasiado pequeño, no explotamos los beneficios que nos ofrecen las librerías optimizadas del hardware. Y su es muy grande, no actualizas los pesos lo suficiente.

  Por suerte, este parámetro no depende de los demás, es independiente, así que se pueden optimizar el resto de parámetros y luego optimizar este. Simplemente se va alterando el valor hasta que le brinde la mejora más rápida en el rendimiento. Una vez se ha obtenido este valor, se pueden optimizar el resto de hiperparámetros.

Como era de esperar, se han desarrollado técnicas automáticas que encuentran hiperparámetros bastante buenos. La más famosa es el "grid search", que significa búsqueda en cuadrícula. En esta técnica, se busca sistemáticamente a través de una cuadrícula en el espacio de hiperparámetros.

No se obtienen unos hiperparámetros perfectos de ninguna de estas maneras, pero de estas maneras se puede empezar de una manera bastante buena, y mejorable. Es bastante común que se dé la situación de que un hiperparámetro esté optimizado (por ejemplo, η), y pasar a optimizar otro (como λ) y una vez se ha optimizado este segundo, vemos que el primero ya no es bueno. Por eso, se suele saltar bastante de parámetro en parámetro hasta que se obtiene una combinación decente.

3.9. Un acercamiento a las redes neuronales profundas

Hasta ahora solo hemos utilizado redes neuronales de una sola capa oculta, pero... ¿Qué ocurre cuando introducimos más?

Utilizando los datos que mejor resultado nos ha dado hasta ahora: una red de 784 neuronas sigmoides en la capa entrada (una por cada píxel de la imagen), 30 neuronas sigmoides en la capa oculta y 10 neuronas sigmoides en la capa de salida (una por cada posible dígito que representa la imagen), utilizando mini-lotes de 10 datos,

una tasa de aprendizaje (η) de 0.1 y un parámetro de regularización (λ) de 5.0, obtenemos una tasa de acierto del 96,48 %.

Si utilizamos los mismos datos con otra capa oculta de 30 neuronas sigmoides, la tasa de acierto aumenta a un 96,90 %. Nada mal. El problema llega cuando añadimos una tercera capa oculta. Lo más normal es que vuelva a aumentar la tasa de acierto, pero sin embargo, esta se reduce a un 96,57 %. Si introducimos una más, la cuarta, este valor vuelve a caer hasta el 96,53 %.

¿Por qué ocurre esto? ¿No debería estar aumentando la tasa de acierto?

Para explicar esto, voy a representar en una imagen las primeras 6 neuronas sigmoides de las capas ocultas de la red con dos capas ocultas. Se va a representar una barrita dentro de cada neurona sigmoide que representa cómo de rápido ha cambiado una neurona mientras aprende la red.

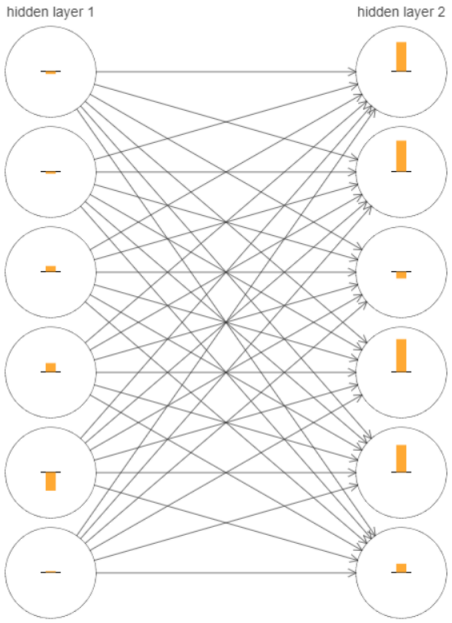

Figura 3.46: Problema con varias capas ocultas

Como puede observarse, las neuronas de la segunda capa aprenden más rápido que las de la primera capa.

Si añadiésemos una tercera capa, las velocidades de entrenamiento serían las siguientes: 0.012, 0.060, y 0.283, y si añadiésemos una cuarta capa, estas velocidades serían: 0.003, 0.017, 0.070, y 0.285. Como puede verse, las últimas capas aprenden mucho más rápido que las primeras.

Y estos datos son solo de la primera época, pero si obtuviésemos ese valor de cada época, se obtendrían los siguientes datos:

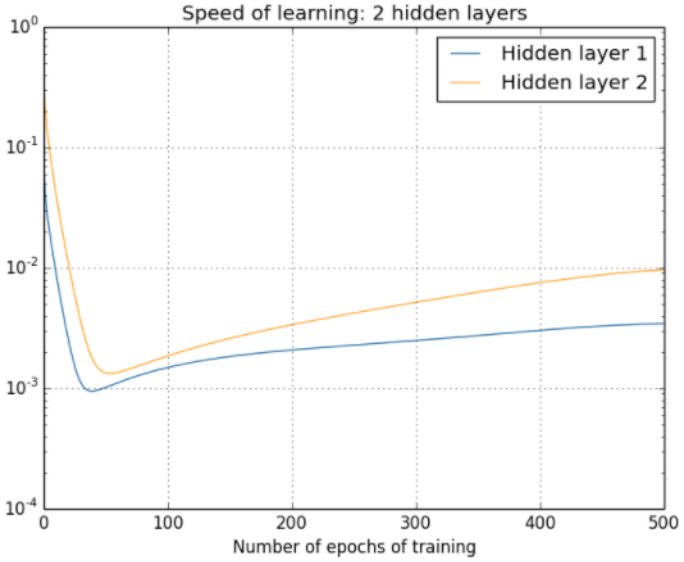

Figura 3.47: Problema con varias capas ocultas a lo largo de las épocas

Y con tres y cuatro capas ocultas, se reproduce la misma situación.

Para este problema existen formas de paliarlo, y dejaré en la bibliografía información para el que quiera ampliar información.

3.10. Redes neuronales convolucionales

Las redes neuronales convolucionales utilizan la estructura espacial a su favor, por lo que son las redes más utilizadas para el reconocimiento de imágenes.

Estas utilizan 3 ideas básicas:

1. Campos receptivos locales: Con las redes explicadas previamente, los valores de las neuronas de la capa de entrada eran un array unidimensional en la que todas las neuronas se interconectaban con todas las neuronas de la primera capa de la capa oculta. En las redes convolucionales, cada neurona de la primera capa se conecta con, por ejemplo, 25 neuronas de la primera capa de la capa oculta, formando un 5x5. A esta región de 5x5 se la llama campo receptivo local. De esta manera, la relación de las neuronas entre las dos primeras capas se vería así:

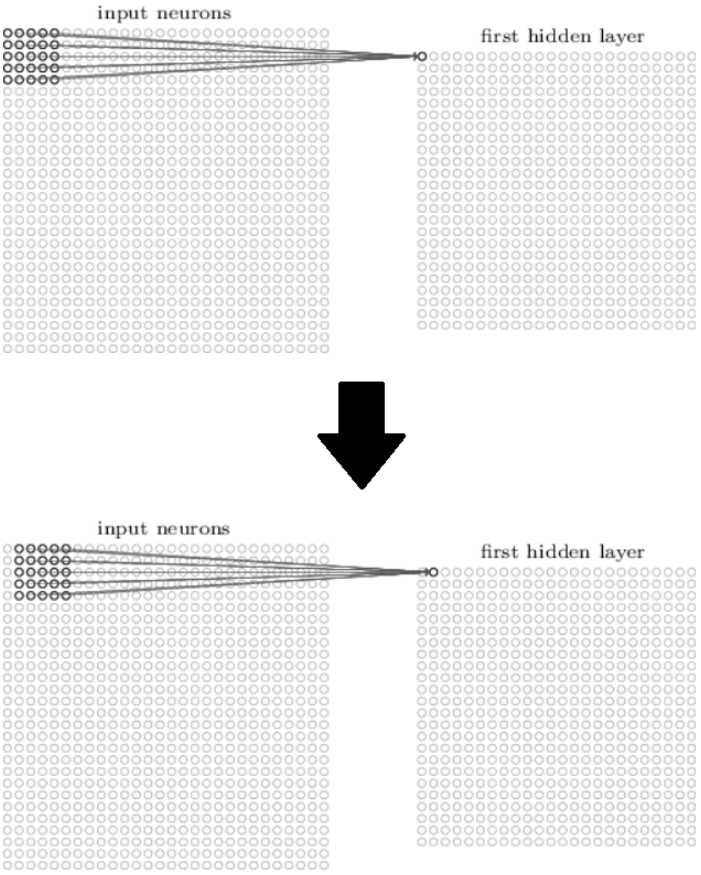

Figura 3.48: Campos receptivos locales red convolucional

De esta manera, la pasamos de tener 28x28 neuronas en la capa de entrada a tener 24x24 neuronas en la primera capa de la capa oculta, y están relacionadas por su estructura espacial.

2. Pesos y bias compartidos: Todos los pesos y bias son los mismos para cada neurona. Es decir, que el peso de la primera neurona de la capa de entrada para la primera neurona de la primera capa de la capa oculta es el mismo que el peso de la segunda neurona de la capa de entrada para la segunda neurona de la primera capa de la capa oculta, y así sucesivamente con todos. De esta manera, todas las neuronas de la misma capa oculta buscan la misma característica, y así, aunque la figura (por ejemplo, un gato), no esté en el centro, la red se adapta.

   Por esta razón, a la primera capa de la capa oculta se la llama "mapa de características". Y no tiene por qué ser siempre 1, puede haber más.

3. Agrupación de capas: Este proceso se realiza a continuación de la creación del mapa de características. Se realiza esto para condensar la información y reducir el tamaño de la matriz que produce.

   La forma de llevar esto a cabo es seleccionar fragmentos de la matriz de 2x2 y escoger el valor más alto, o el valor medio de la matriz, dependiendo de si se realiza "max pooling" o "average pooling" respectivamente.

Un ejemplo de esquema de una red convolucional sería el siguiente:

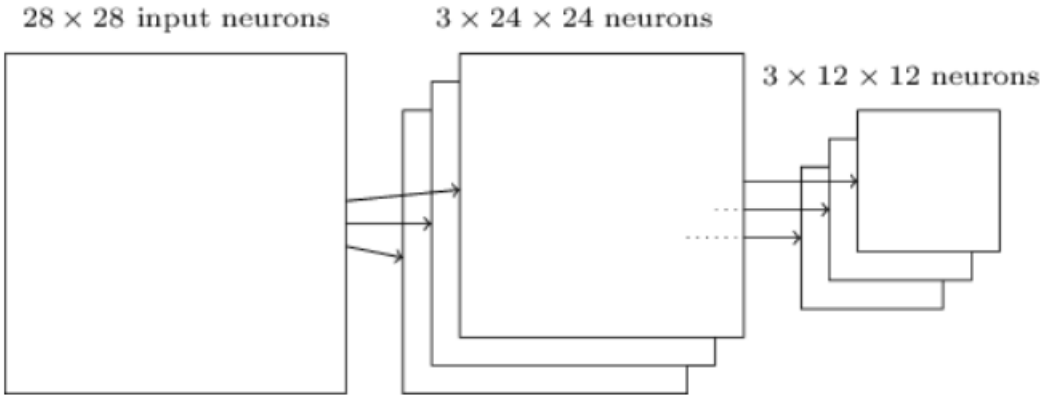

Figura 3.49: Ejemplo esquema red convolucional

Como puede observarse, inicialmente tenemos una imagen de 28x28 píxeles, desde la cual se obtienen 3 mapas de características con diferentes patrones, cada una de 24x24 neuronas, y a continuación se realiza un "max pooling" o un "average pooling" y acaban quedando 3 matrices de 12x12 neuronas, con la información seleccionada y comprimida.

Capítulo 4

Contexto tecnológico: pytorch y fastai

4.1. Introducción

Para la realización de este estudio, se va a utilizar fast.ai (versión 2), que es la librería de deep learning que más rápido ha crecido desde su creación, en 2017, y hoy en día es usada por la mayoría de los trabajos de investigación de las principales conferencias. Además, es la librería más flexible y más expresiva de la biblioteca de deep learning. No cambia velocidad por simplicidad, sino que provee ambas.

El manual del que en el que me voy a basar para explicar la librería ha sido creada por Sylvain y Jeremy, que son educadores, matemáticos, y expertos en aprendizaje automático.

Ellos creen que el aprendizaje profundo tiene poder, flexibilidad y simplicidad, y que debería aplicarse en muchas disciplinas (ciencias sociales y físicas, las artes, la medicina, las finanzas, la investigación científica...).

De hecho, explican que a pesar de no tener experiencia en medicina, Jeremy fundó Enlitic, una empresa que utiliza algoritmos de aprendizaje profundo para diagnosticar enfermedades y dolencias. A los pocos meses de iniciar la empresa, se anunció que su algoritmo podía identificar tumores malignos con mayor precisión que los radiólogos.

De hecho, el deep learning se puede utilizar para multitud de campos diferentes. Aquí hay una lista de algunas de las miles de tareas en diferentes áreas en las que el deep learning, o los métodos que utilizan en gran medida el aprendizaje profundo,

son ahora los mejores del mundo:

1. Procesamiento del Lenguaje Natural: Responder preguntas, reconocimiento de voz, resumir documentos, clasificación de documentos, buscar artículos que mencionen un concepto...

2. Visión por computadora: interpretación de imágenes de satélites y drones , reconocimiento facial, leer señales de tráfico, localizar peatones y vehículos en vehículos autónomos...

3. Medicina: Detección de anomalías en imágenes de radiología, , contar características en diapositivas de patología, características de medición en ultrasonidos, diagnosticar la retinopatía diabética...

4. Videojuegos: Ajedrez, Go, la mayoría de los videojuegos de Atari, muchos juegos de estrategia en tiempo real...

Han completado cientos de proyectos de aprendizaje automático utilizando docenas de paquetes diferentes y muchos lenguajes de programación diferentes. En fast.ai, han escrito cursos que utilizan la mayoría de los principales paquetes de aprendizaje profundo y aprendizaje automático que se utilizan en la actualidad.

Después de que PyTorch saliera a la luz en 2017, pasaron más de mil horas probándolo antes de decidir que lo usarían para futuros cursos, desarrollo de software e investigación. Desde entonces, PyTorch se ha convertido en la biblioteca de aprendizaje profundo de más rápido crecimiento en el mundo y ya se utiliza para la mayoría de los trabajos de investigación en las principales conferencias. Este es generalmente un indicador adelantado de uso en la industria, porque estos son los papeles que terminan usándose comercialmente en productos y servicios. Hemos descubierto que PyTorch es la biblioteca más flexible y expresiva para el aprendizaje profundo. No intercambia velocidad por simplicidad, sino que proporciona ambas.

PyTorch funciona mejor como una biblioteca básica de bajo nivel, proporcionando las operaciones básicas para una funcionalidad de nivel superior. La biblioteca fastai es la biblioteca más popular para agregar esta funcionalidad de nivel superior a PyTorch. También se adapta particularmente bien a los propósitos de este proyecto, porque es único en proporcionar una arquitectura de software profundamente estratificada (incluso hay un artículo académico revisado por pares sobre esta API estratificada).

La plataforma de experimentación que vamos a utilizar se llama Colaboratory. Ya que es una herramienta popular para hacer ciencia de datos en Python, potente, flexible, fácil de usar y gratuita.

4.2. Una primera aproximación

El código de una red neuronal utilizando esta herramienta tiene el siguiente aspecto:

Figura 4.1: Ejemplo código red neuronal fastai

La primera sección importa las librerías.

La segunda sección inicializa la variable path para indicar la ruta de la que se van a tomar los datos.

La tercera sección crea un datablock llamado "soldaduras", que es un sistema extremadamente flexible llamado "API de datablock". Con esta API, se puede personalizar completamente cada etapa de la creación de sus DataLoaders.

En este caso, proporcionamos una tupla donde especificamos qué tipos queremos para las variables independientes y dependientes, en este caso, "ImageBlock" y "CategoryBlock" respectivamente. La variable independiente es lo que estamos usando para hacer predicciones, y la variable dependiente es nuestro objetivo. En este caso, nuestras variables independientes son imágenes y nuestras variables dependientes son las categorías (tipo de soldadura) para cada imagen.

A continuación, tenemos que decirle a fastai cómo obtener una lista de esos archivos. La función get_image_files toma una ruta y devuelve una lista de todas las imágenes en esa ruta.

Después, dividimos nuestros conjuntos de entrenamiento y validación al azar (en este caso, un 20 % de las imágenes no serán utilizadas para el conjunto de entrenamiento y se reservarán para el conjunto de prueba). Sin embargo, nos gustaría tener la misma división de entrenamiento / validación cada vez que ejecutamos este cuadro, por lo que utilizamos la semilla aleatoria (las computadoras realmente no saben cómo crear números aleatorios, simplemente crean listas de números que parecen aleatorios ; si proporciona el mismo punto de partida para esa lista cada vez, llamado semilla, obtendrá exactamente la misma lista cada vez). En este caso utilizarmos la semilla 42.

Por último, le decimos a fastai cómo crear las etiquetas en nuestro conjunto de datos. En este caso, utilizamos la función "parent_label", que es una función proporcionada por fastai que simplemente obtiene el nombre de la carpeta en la que se encuentra un archivo. Debido a que colocamos cada una de nuestras imágenes de soldaduras en carpetas según el tipo de soldadura, esto nos dará las etiquetas que necesitamos.

La cuarta sección creamos una instancia del objeto "soldaduras" que hemos creado antes. Después, este comando nos proporciona un objeto DataBlock, utilizando la ruta que habiamos asignado como variable previamente.

En la quinta sección, se le indica a fastai que cree una red neuronal convolucional (CNN) y especifica qué arquitectura usar (es decir, qué tipo de modelo crear), en qué datos queremos entrenarlo y qué métrica usar.

CNN es el enfoque actual de vanguardia para crear modelos de visión por compu-

tadora.

Se le indica el datablock que se quier utilziar, en este caso, “dls”.

Hay algunas arquitecturas estándar que funcionan la mayor parte del tiempo, y en este caso estamos usando una llamada ResNet. El 18 en resnet18 se refiere al número de capas en esta variante de la arquitectura (otras opciones son 34, 50, 101 y 152). Los modelos que usan arquitecturas con más capas tardan más en entrenarse y son más propensos a sobreajustarse (es decir, no puede entrenarlos durante tantas épocas antes de que la precisión en el conjunto de validación comience a empeorar), aunque, cuando se utilizan más datos, pueden ser un poco más precisos.

Por último, se indica la métrica. Una métrica es una función que mide la calidad de las predicciones del modelo utilizando el conjunto de validación y se imprimirá al final de cada época. En este caso, estamos usando error_rate, que es una función proporcionada por fastai que hace exactamente lo que dice: qué porcentaje de imágenes en el conjunto de validación se clasifican incorrectamente.

“cnn_learner” también tiene un parámetro preentrenado, que por defecto es Verdadero (por lo que se usa en este caso, aunque no se ha especificado), que establece los pesos en su modelo en valores que ya han sido entrenados por expertos para reconocer mil diferentes categorías en 1,3 millones de fotos (utilizando el famoso conjunto de datos ImageNet). Un modelo que tiene ponderaciones que ya se han entrenado en algún otro conjunto de datos se denomina modelo preentrenado. Casi siempre debe usarse un modelo previamente entrenado, porque significa que su modelo, incluso antes de que le haya mostrado alguno de sus datos, ya es muy capaz. En un modelo de aprendizaje profundo, muchas de estas capacidades son cosas que se necesitarán, casi independientemente de los detalles de su proyecto. Por ejemplo, partes de modelos previamente entrenados manejarán la detección de bordes, degradados y color, que son necesarios para muchas tareas.

Además, se le dice a fastai cómo ajustar el modelo. Esta es la clave del aprendizaje profundo: determinar cómo ajustar los parámetros de un modelo para que resuelva el problema. Para ajustar un modelo, tenemos que proporcionar al menos una pieza de información: cuántas veces mirar cada imagen (conocido como número de épocas). El número de épocas que se seleccionen dependerá en gran medida de la cantidad de tiempo que se tenga disponible y del tiempo que se considere que se necesita en la práctica para adaptarse a su modelo. Si se selecciona un número que es demasiado pequeño, siempre se pueden entrenar más épocas más adelante.

Pero, ¿por qué el método se llama “fine_tune”? Como en este caso, comenzamos con un modelo previamente entrenado y no queremos deshacernos de todas las capacidades que ya tiene, este método establece algunos parámetros, como que se use

una época para ajustar solo aquellas partes del modelo necesarias para que el nuevo cabezal aleatorio funcione correctamente con su conjunto de datos y que se utilicen el número de épocas solicitado al llamar al método para ajustar el modelo completo, actualizando los pesos de las capas posteriores (especialmente la cabeza), más rápido que las capas anteriores (que, como veremos, generalmente no requieren muchos cambios de los pesos preentrenados).

De esta manera, se imprimen los resultados después de cada época, en la que se muestran el número de época, las pérdidas del conjunto de entrenamiento y validación y cualquier métrica que haya solicitado (tasa de error, en este caso).

Una vez se han ejecutado todas esas instrucciones en ese orden concreto, se genera un cuadro como el siguiente:

epoch	train_loss	valid_loss	error_rate	time
0	0.291742	0.056248	0.024648	21:55
epoch	train_loss	valid_loss	error_rate	time
0	0.023452	0.004710	0.001761	00:22
1	0.007441	0.001221	0.000880	00:22
2	0.002747	0.000242	0.000000	00:22
3	0.000886	0.000031	0.000000	00:22
4	0.013533	0.000005	0.000000	00:22
5	0.004263	0.000002	0.000000	00:22
6	0.001134	0.000000	0.000000	00:22
7	0.000290	0.000000	0.000000	00:22
8	0.000090	0.000001	0.000000	00:22
9	0.001857	0.000031	0.000000	00:21
10	0.003377	0.000002	0.000000	00:22
11	0.000901	0.000000	0.000000	00:22
12	0.000300	0.000000	0.000000	00:22
13	0.000204	0.000000	0.000000	00:22
14	0.000052	0.000000	0.000000	00:22
15	0.000014	0.000000	0.000000	00:22
16	0.000524	0.000000	0.000000	00:22
17	0.000142	0.000000	0.000000	00:22
18	0.000042	0.000000	0.000000	00:21
19	0.000016	0.000000	0.000000	00:22

Figura 4.2: Ejemplo salida tabla con datos red neuronal fastai

En el que, como he explicado antes, se muestra el número de la época, las pérdidas del conjunto de entrenamiento y validación y la métrica (tasa de error).

4.3. Algunas funcionalidades útiles

Durante la realización del proyecto, ha habido varias funcionalidades de las que disponemos cuando utilizamos fastai que han sido de vital importancia. En esta sección se van a explicar dichas funciones, como funcionan y por qué han sido útiles.

La primera función que se ha utilizado ha sido la función ".predict". Esta función tiene el siguiente aspecto:

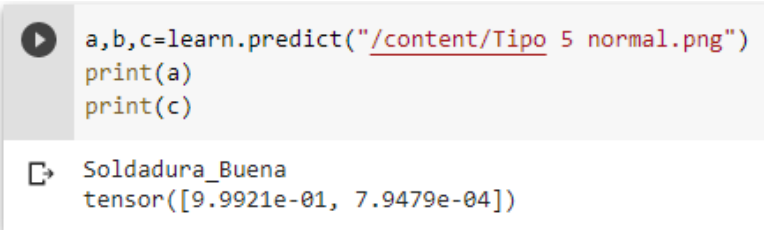

Figura 4.3: Ejemplo código fastai función ".predict()"

Esta función devuelve la predicción del item que se le pase por parámetro, utilizando la red que se ha creado previamente. Esta función devuelve, además del nombre de la clase que predice la red (en este caso, "Soldadura_Buena"), un tensor en el que indica la probabilidad de cada una de las posibles clases según la red que se haya utilizado. En este caso, hay un 99,921 % de posibilidades de ser una "Soldadura_Buena"), y un 0,079 % de posibilidades de ser la clase del otro tipo.

Esto es especialmente útil sobre todo para observar la predicción de una imagen concreta con la red preentrenada.

Otra función que se ha utilizado bastante ha sido la función ".recorder.plot_loss()", que tiene el siguiente aspecto:

```
In [6]: learn.recorder.plot_loss()
```

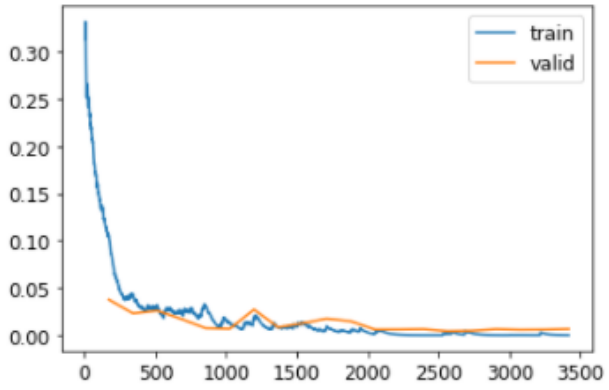

Figura 4.4: Ejemplo código fastai función ".recorder.plot_loss()"

Este método muestra el train_loss y el valid_loss a lo largo de las épocas de entrenamiento, y sirve para mostrar cuando se está produciendo sobreajuste en la red (se produce en el momento en el que el valid_loss sube y el train_loss baja),

El siguiente método que se ha utilizado ha sido el método ".plot_confusion_matrix()":

```
In [7]: interp = ClassificationInterpretation.from_learner(learn)
        interp.plot_confusion_matrix()
```

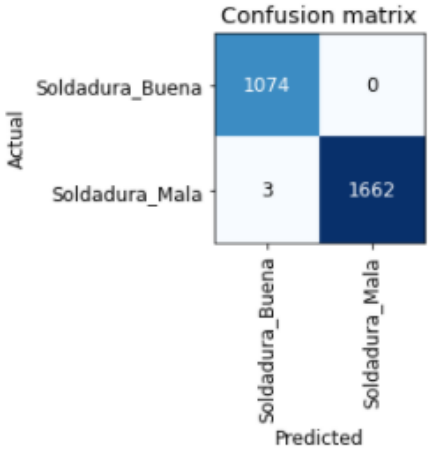

Figura 4.5: Ejemplo código fastai función ".plot_confusion_matrix()"

Las filas representan todas las imágenes de cada tipo de nuestro conjunto de datos. Las columnas representan las imágenes que el modelo predijo como cada uno de los tipos. Por lo tanto, la diagonal de la matriz muestra las imágenes que se clasificaron correctamente y las celdas fuera de la diagonal representan las que se clasificaron incorrectamente. Esta es una de las muchas formas en que fastai le permite ver los resultados de su modelo.

Es útil ver dónde ocurren exactamente nuestros errores, para ver si se deben a un problema de conjunto de datos, o un problema de modelo. Para hacer esto, podemos ordenar nuestras imágenes por su pérdida.

Por último, el último método que se ha utilizado, y que ha sido especialmente útil ha sido el método "plot_top_losses".

Este nos muestra las imágenes con la mayor pérdida en nuestro conjunto de datos. Es decir, las imágenes que más nos han hecho a la red empeorar la misma, es decir, imágenes que es posible que estén mal clasificadas o que sean especialmente diferentes, y que por lo tanto, se necesiten más de ese tipo para clasificarlas correctamente.

En este caso, fue útil porque había imágenes mal clasificadas y con este método se pudieron identificar rápidamente.

Tiene el siguiente aspecto:

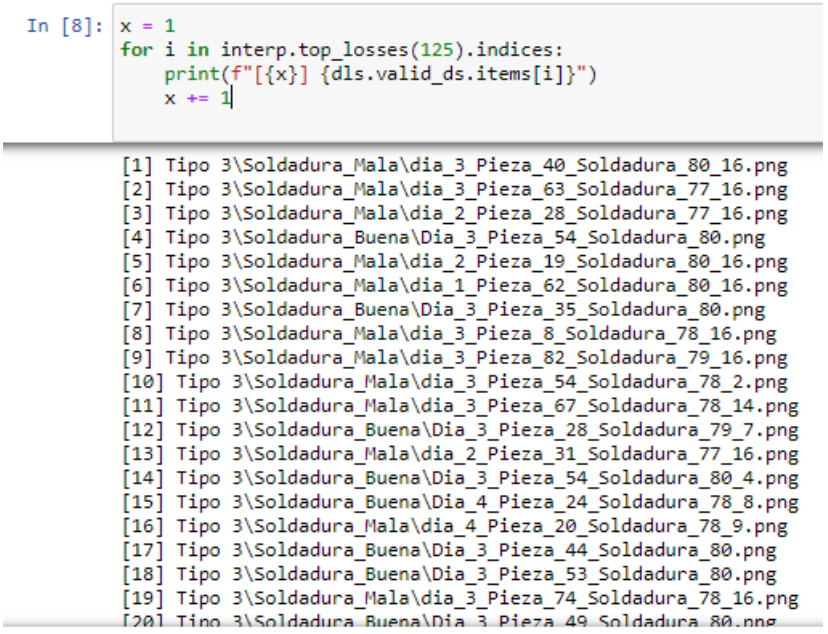

Figura 4.6: Ejemplo código fastai función ".plot_top_losses()"

En este caso, se muestran las rutas de las 125 imágenes que más hicieron empeorar la red.

Por supuesto, esta librería tiene muchos métodos más que nos pueden ayudar en momentos puntuales, pero como este no es un manual de uso de la librería, esta sección ha sido centrada en las funciones más útiles para el desarrollo del proyecto.

Capítulo 5

Análisis del clasificador

5.1. Identificación y descripción del problema

Las soldaduras son el proceso de fijación en el cual se realiza una unión de dos o más piezas de un material (generalmente metales), logrando a través de la fusión de un material de aporte (generalmente metal), formar un charco de material fundido entre las piezas, que al enfriarse se convierte en una unión fija llamada cordón.[Wikf]

En un proceso de fabricación automática, estas soldaduras deben ser lo más regulares posible, y deben cumplir una serie de requisitos de cara a garantizar una mínima calidad y seguridad para el cliente.

Imaginemos por ejemplo, una mínima burbuja de aire en una soldadura de una nave espacial. Tan solo un pequeño error puede cobrar la vida de personas inocentes.

En una gran cantidad de ocasiones, estos defectos se pueden detectar a simple vista. Por esta razón, es viable generar una red neuronal que sea capaz de detectar la mayoría de defectos de una soldadura, ahorrando así una gran cantidad de tiempo y energía.

Los principales defectos que puede llegar a detectar una red neuronal son los siguientes:

- Caída de cordón: Se produce cuando el material se calienta demasiado, tarda más en enfriarse y, por la fuerza de gravedad, cae levemente y no llega a soldar correctamente las piezas que se pretenda soldar.
- Descompensaciones norte/sur, sur/norte: Se producen cuando, de nuevo por la fuerza de gravedad, la soldadura está correctamente situada, pero una zona

de la soldadura contiene más material que la otra, generando un punto en el que hay poco material, y facilitando la ruptura por esa zona.

- Cortes: Se producen cuando la herramienta que se está utilizando para soldar se queda sin material momentáneamente. Dependiendo de la finalidad de la soldadura, esto puede llegar a ser admisible si es pequeña en comparación al resto de la soldadura, aunque también puede ser inadmisible si hay riesgo de un fallo de estanqueidad.
- Agujeros en la soldadura: Se producen cuando se generan burbujas por dentro del material fundido de la soldadura, que explotan y generan agujeros en el material. De nuevo es potencialmente admisible, dependiendo del tipo de soldadura que sea.
- Agujeros en las chapas que se pretenden soldar: El material que se utiliza para soldar puede ser punzante, o estar muy caliente, lo que puede generar un contacto con las chapas y agujerearlas o deformarlas.
- Bolas: Se puede dar la situación que se produzca una acumulación de material en algún punto concreto de la soldadura, generando así una bola de material que, pese a que parece que no perjudica de ninguna manera, empeora la calidad global de la soldadura y la fragiliza.
- Cualquier movimiento en la soldadura que no cubra totalmente la zona que debe cubrir en las placas para soldarlas, ya sea un defecto de forma o de posición del cordón.

Todos estos problemas se pueden identificar fácilmente utilizando el ojo humano, y por lo tanto, se puede entrenar una red para realizar esta diferenciación entre soldaduras buenas y malas.

5.2. Dataset

Durante este estudio, se han ido puliendo los datasets poco a poco. Inicialmente, obteníamos una tasa de error medio de en torno a un 1 %, pero a base de eliminar imágenes que estaban mal clasificadas con diferentes herramientas, alterar el código y pulir defectos en el preprocessing de las imágenes, esta tasa de error fue reducida considerablemente.

En este estudio solo nos fijaremos solo en los últimos datasets, los que menor tasa de error han obtenido.

Las imágenes de las soldaduras se dividen en 9 datasets diferentes, cada uno representando una soldadura diferente.

Antes de nada, para llevar este proyecto a cabo, se ha realizado un pequeño preprocessing a las imágenes:

1. Utilizamos un robot con una cámara que obtiene la nube de puntos en tres dimensiones de la soldadura.
2. Creamos un programa que utiliza la altura de los puntos de la nube de puntos para crear una imagen en dos dimensiones con una escala de grises. De esta manera, cuanto más alto esté un punto en la nube, más blanco se imprime, y cuanto más bajo, más negro.
3. Las soldaduras malas están creadas digitalmente con un programa en python, simulando los posibles defectos previamente mencionados.
4. Después de crear las imágenes malas había un desbalanceo entre el número de imágenes buenas y malas, así que se tuvo que hacer augmentation de las imágenes buenas introduciendo ruido gausiano y otras técnicas para balancear el número de imágenes buenas y malas de cada dataset.

Las características de los datasets son las siguientes:

- Tipo 1: contiene 2193 imágenes de soldaduras buenas y 3904 imágenes de soldaduras malas. El aspecto de las soldaduras que forman este dataset es el siguiente (en el ejemplo se marca con un círculo rojo el defecto, corte en la zona media del cordón):

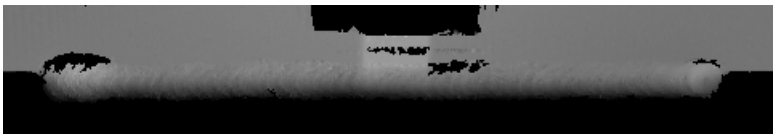

Figura 5.1: Ejemplo imagen soldadura tipo 1

- Tipo 2: contiene 5363 imágenes de soldaduras buenas y 8925 imágenes de soldaduras malas. El aspecto de las soldaduras que forman este dataset es el siguiente (en el ejemplo se marca con un círculo rojo el defecto, caída en la parte derecha del cordón):

Figura 5.2: Ejemplo imagen soldadura tipo 2

- Tipo 3: contiene 5245 imágenes de soldaduras buenas y 8299 imágenes de soldaduras malas. El aspecto de las soldaduras que forman este dataset es el siguiente (en el ejemplo se marca con un círculo rojo el defecto, bola en la zona izquierda del cordón):

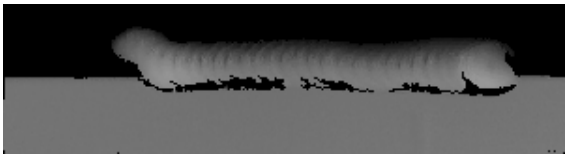

Figura 5.3: Ejemplo imagen soldadura tipo 3

- Tipo 4: contiene 10674 imágenes de soldaduras buenas y 21454 imágenes de soldaduras malas. El aspecto de las soldaduras que forman este dataset es el siguiente (en el ejemplo se marca con un círculo rojo el defecto, corte en la zona media del cordón):

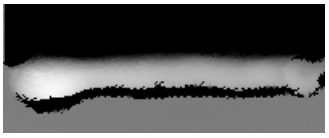

Figura 5.4: Ejemplo imagen soldadura tipo 4

- Tipo 5: contiene 4140 imágenes de soldaduras buenas y 10656 imágenes de soldaduras malas. El aspecto de las soldaduras que forman este dataset es el siguiente (en el ejemplo se marca con un círculo rojo el defecto, agujero en la zona izquierda de la soldadura):

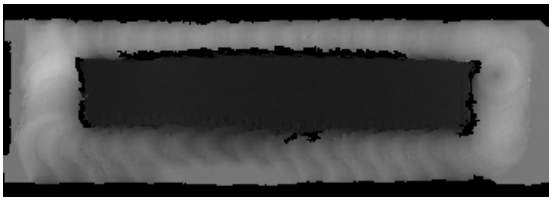

Figura 5.5: Ejemplo imagen soldadura tipo 5

- Tipo 6: contiene 2642 imágenes de soldaduras buenas y 4151 imágenes de soldaduras malas. El aspecto de las soldaduras que forman este dataset es el siguiente (en el ejemplo se marca con un círculo rojo el defecto, desplazamiento total del cordón):

Figura 5.6: Ejemplo imagen soldadura tipo 6

- Tipo 7: contiene 4144 imágenes de soldaduras buenas y 9482 imágenes de soldaduras malas. El aspecto de las soldaduras que forman este dataset es el siguiente (en el ejemplo se marca con un círculo rojo el defecto, agujero en el cordón):

Figura 5.7: Ejemplo imagen soldadura tipo 7

- Tipo 8: contiene 2097 imágenes de soldaduras buenas y 3587 imágenes de soldaduras malas. El aspecto de las soldaduras que forman este dataset es el siguiente (en el ejemplo se marca con un círculo rojo el defecto, más material en la parte superior que en la inferior):

Figura 5.8: Ejemplo imagen soldadura tipo 8

- Tipo 9: contiene 5328 imágenes de soldaduras buenas y 8882 imágenes de soldaduras malas. El aspecto de las soldaduras que forman este dataset es el siguiente (en el ejemplo se marca con un círculo rojo el defecto, bola en la zona central del cordón):

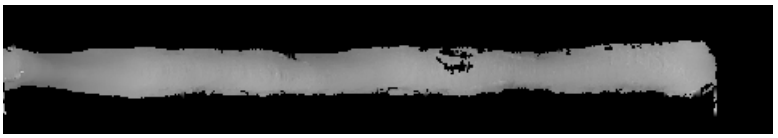

Figura 5.9: Ejemplo imagen soldadura tipo 9

5.3. Transformaciones

Inicialmente el dataset solo contenía imágenes de soldaduras buenas, por lo que se han tenido que realizar programas en python para simular defectos en las imágenes de las soldaduras buenas.

Para la creación de estas imágenes, se ha utilizado la librería de python "opencv" para la detección de bordes y el cálculo de gradientes, lo que fue de vital importancia para obtener una máscara del cordón y poder aislar la soldadura, así como numpy, pillow y scipy, que tienen métodos que aceleran mucho las tareas que había que llevar a cabo.

Las principales deformaciones que se han llevado a cabo en casi todos los tipos de soldaduras fueron los siguientes:

- Girar derecha arriba: Consiste en deformar la parte derecha del cordón hacia arriba, simulando una caída de cordón.

Figura 5.10: Ejemplo imagen defecto derecha arriba

- Girar derecha abajo: Consiste en deformar la parte derecha del cordón hacia abajo, simulando una caída de cordón.

Figura 5.11: Ejemplo imagen defecto derecha abajo

- Girar izquierda arriba: Consiste en deformar la parte izquierda del cordón hacia arriba, simulando una caída de cordón.

Figura 5.12: Ejemplo imagen defecto izquierda arriba

- Girar izquierda abajo: Consiste en deformar la parte izquierda del cordón hacia abajo, simulando una caída de cordón.

Figura 5.13: Ejemplo imagen defecto izquierda abajo

- Abombar abajo: Consiste en deformar la parte central del cordón hacia abajo, simulando una caída de cordón.

Figura 5.14: Ejemplo imagen defecto abombar abajo

- Abombar arriba: Consiste en deformar la parte central del cordón hacia arriba, simulando una caída de cordón.

Figura 5.15: Ejemplo imagen defecto abombar arriba

- Descompensación norte sur: Consiste en simular más cantidad de material en la zona superior que la inferior.

Figura 5.16: Ejemplo imagen defecto descompensación N/S

- Descompensación sur norte: Consiste en simular más cantidad de material en la zona inferior que la superior.

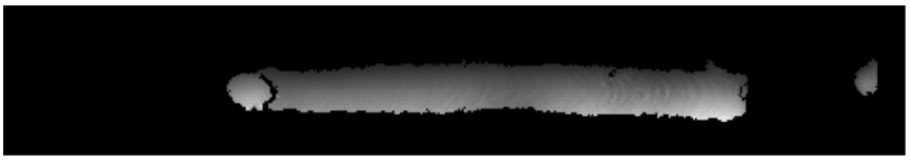

Figura 5.17: Ejemplo imagen defecto descompensación S/N

- Bolas: Consiste en simular una acumulación de material en un punto concreto, formando una bola en algún punto del cordón.

Figura 5.18: Ejemplo imagen defecto bola

- Corte: Consiste en simular ausencia de material en alguna zona del cordón.

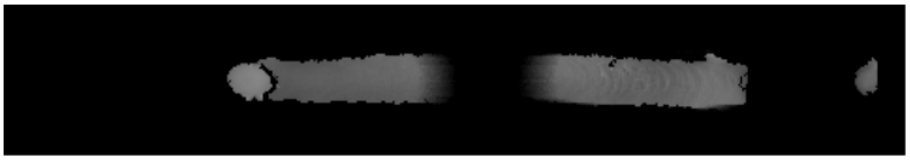

Figura 5.19: Ejemplo imagen defecto corte

- Agujero: Consiste en simular ausencia de material en algún punto del cordón.

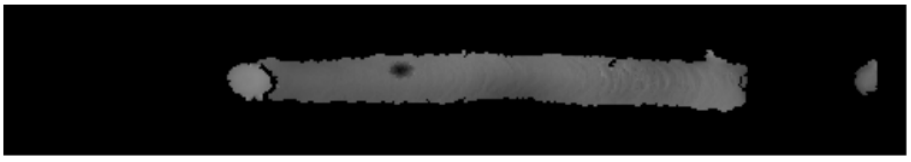

Figura 5.20: Ejemplo imagen defecto agujero

Las transformaciones que he mencionado previamente se han realizado a casi todos los datasets, sin embargo ha habido algunas excepciones, como por ejemplo en el tipo 8, que la soldadura es tan pequeña que no merece la pena simular una caída de cordón solo por la parte central.

Además, el tipo 5 tiene otros defectos porque la forma es totalmente diferente. También tiene agujeros, bolas, cortes y caídas de cordón, pero evidentemente, al tener otra forma son ligeramente diferentes.

Todas estas transformaciones que hemos mencionado son para entrenar a la red con imágenes de soldaduras malas. Cuando se entrenaron las primeras redes, se dieron algunas situaciones en las que había más de diez imágenes de soldaduras malas por cada imagen de una soldadura buena. Por esta razón, se llegó a dar el caso de que algunas redes clasificaban todas las imágenes como malas para garantizar una buena tasa de acierto. Como no era esto lo que se pretendía, se tuvo que hacer augmentation de las imágenes buenas para que el número de imágenes de soldaduras buenas y soldaduras malas de cada dataset fueran más similares.

Para crear más imágenes de soldaduras buenas, se llevaron a cabo las siguientes transformaciones:

- Ruido gausiano: Consiste en sustituir el valor de píxeles aleatorios por 0 para simular ruido en la imagen. Además, como los píxeles son aleatorios, se puede realizar esta acción varias veces sobre la imagen y se obtienen imágenes diferentes.

Figura 5.21: Ejemplo imagen ruido gausiano

- Ruido aleatorio: Consiste en alterar el valor de los píxeles, y aumentar o reducir su valor un valor aleatorio. De nuevo, se puede realizar esta acción varias veces sobre la misma imagen. En la imagen es posible que no se note demasiado la diferencia, pero la intensidad de los píxeles ha variado de manera aleatoria.

Figura 5.22: Ejemplo imagen ruido aleatorio

5.4. Diseño del modelo

Para la creación del modelo, se ha utilizado una herramienta llamada "anaconda".

Anaconda es una distribución libre y abierta de los lenguajes de programación python (que es el lenguaje que utilizaremos nosotros), y R. Es muy utilizada en ciencia de datos y aprendizaje automático. Vamos a utilizarlo porque está orientado a simplificar el despliegue y la administración de los paquetes de software.[Wika]

Como es evidente por el apartado 4 de este proyecto, vamos a utilizar la librería orientada al deep learning "fastai", que nos permitirá, con unas pocas líneas de código, obtener un clasificador que es capaz de clasificar las soldaduras con un porcentaje de acierto de más de un 99 %.

Como el código es tan breve, iré explicando línea por línea lo que se va haciendo.

1. Lo primero, como siempre, importamos las librerías de fastai que nos van a hacer falta. El warning no tiene ningún impacto en el código, y cuando arreglo el warning, no ejecuta el programa, así que vamos a obviarlo.

   Cabe destacar que es necesario haberse descargado las librerías que vamos a utilizar en el ordenador antes de ejecutarlas. Para hacer esto basta con abrir el cmd y utilizar pip para instalar fastai y fastbook.

```
In [1]: import fastai
        import fastbook
        from fastbook import *
        from fastai.vision.widgets import *
        fastbook.setup_book()
        from PIL import Image

        C:\Users\Tottem\.conda\envs\fastai\lib\site-packages\fastbook\__init__.py:18: UserWarning: Missing `graphviz` - please run `con
        da install fastbook`
          except ModuleNotFoundError: warn("Missing `graphviz` - please run `conda install fastbook`")
```

Figura 5.23: Código red neuronal 1

2. Introducimos la ruta del directorio donde están las imágenes. En este caso es simplemente el nombre de la carpeta. Dentro de esta carpeta, hay dos carpetas, con los nombres: "Soldadura_Buena" y "Soldadura_Mala", y en cada una de ellas, se encuentran las imágenes de las soldaduras con el preprocessing ya aplicado. Evidentemente, en Soldadura_Buena, están todas las imágenes de soldaduras sin defectos, y en Soldadura_Mala, las imágenes de soldaduras con defectos.

```
In [2]: path ='Tipo 2'
```

Figura 5.24: Código red neuronal 2

3. A continuación, creamos el datablock (llamado "soldaduras").

   Especificamos que nuestra variable independiente serán las imágenes, que es lo que vamos a utilizar para hacer nuestras predicciones, y nuestra variable dependiente serán las categorías en las que las vamos a dividir, en este caso, soldaduras buenas y malas.

   Decimos a fastai que queremos obtener una lista de todos los ficheros de la ruta que hemos introducido con el get_image_files.

   Vamos a utilizar holdout para el cálculo de la tasa de error, así que dividimos el dataset de manera aleatoria en un 20 % de imágenes formando el conjunto de prueba y un 80 % formando el conjunto de entrenamiento, y la semilla que vamos a utilizar (en este caso, 42, aunque esto da absolutamente lo mismo).

   Por último, especificamos que la variable dependiente (y), tomará los nombres de los ficheros padre ("Soldadura_Buena" y "Soldadura_Mala").

```
In [3]: soldaduras = DataBlock(
            blocks=(ImageBlock, CategoryBlock),
            get_items=get_image_files,
            splitter=RandomSplitter(valid_pct=0.2, seed=42),
            get_y=parent_label)
```

Figura 5.25: Código red neuronal 3

4. Después, se llama a la clase "dataloaders", que provee mini-lotes de unos cuantos elementos a la vez a la GPU. Si no se especifica nada, los mini-lotes son de 64 elementos, que van a parar al mismo tensor a la vez.

```
In [4]: soldaduras = soldaduras.new()
        dls = soldaduras.dataloaders(path,num_workers=0)
```

Figura 5.26: Código red neuronal 4

5. Por último, se crea un learner, utilizando una red preentrenada (lo que se hace aquí es transfer learning, es decir, que reentrenamos una red ya entrenada, preparada para clasificar imágenes), llamada resnet18 (porque tiene 18 capas ocultas), y vamos a mostrar la tasa de error (el error_rate).

   Con el fine_tune mostramos el número de épocas de entrenamiento, en este caso, 20.

```
In [5]: learn = cnn_learner(dls, resnet18, metrics=error_rate)
        learn.fine_tune(20)
```

epoch	train_loss	valid_loss	error_rate	time
0	0.467392	0.304623	0.130899	01:44

epoch	train_loss	valid_loss	error_rate	time
0	0.155077	0.101144	0.035619	00:18
1	0.060253	0.056015	0.020481	00:17
2	0.031194	0.036077	0.012467	00:18
3	0.016301	0.036205	0.012467	00:17
4	0.033772	0.052788	0.016919	00:16
5	0.017880	0.035406	0.009795	00:17
6	0.012217	0.008251	0.003562	00:17
7	0.014682	0.009548	0.001781	00:16
8	0.005324	0.002117	0.000890	00:17
9	0.004421	0.003551	0.000890	00:17
10	0.011184	0.005460	0.002671	00:16
11	0.005335	0.000731	0.000000	00:16
12	0.002355	0.002147	0.000890	00:16
13	0.003171	0.025594	0.003562	00:16
14	0.002566	0.002163	0.000890	00:16
15	0.001946	0.000479	0.000000	00:17
16	0.000948	0.000413	0.000000	00:17
17	0.001410	0.000420	0.000000	00:17
18	0.000959	0.000390	0.000000	00:17
19	0.000627	0.000392	0.000000	00:17

Figura 5.27: Código red neuronal 5

6. A estas alturas, la red neuronal ya está creada, y como puede comprobarse, acierta el 100 % de las imágenes (tiene un 0 % de tasa de error). A partir de aquí, vamos a obtener datos de nuestro interés.

   Mostramos en una matriz el número de imágenes "Soldadura_Buena" que el clasificador ha sido capaz de clasificar como "Soldadura_Buena" y el número de imágenes "Soldadura_Buena" que el clasificador ha clasificado erróneamente como "Soldadura_Mala", y lo mismo con las imágenes que eran "Soldadura_Mala". A esta matriz se la llama matriz de confusión.

   El resultado que obtenemos, en este caso, es el siguiente:

```
In [7]: interp = ClassificationInterpretation.from_learner(learn)
        interp.plot_confusion_matrix()
```

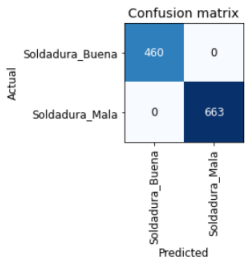

Figura 5.28: Código red neuronal 6

7. Por último, un dato muy importante a la hora de pulir una red neuronal es saber en qué imágenes, la red ha tenido más dudas a la hora de clasificarlas.

   El resultado de la clasificación de una imagen utilizando la red neuronal es un tensor de dos valores, cuya suma es siempre 1. Cada número expresa la probabilidad de que la imagen sea de un tipo u otro. De esta manera, por ejemplo, si el primer número representa la categoría "Soldadura_Buena" y el segundo representa la categoría "Soldadura_Mala", un tensor [0.996353, 0.003647], indica que con una probabilidad del 99,63 % según la red, la imagen está representando una Soldadura_Buena.

   Existe una función en fastai llamada top_loses, que te devuelve una lista con el nombre de las imágenes que menor tensor han tenido, es decir, que más cercanas están del 50 %, y más lejanas del 100 %.

   Es muy sencillo de entender el código que se ha utilizado para mostrar esta información para las 125 imágenes que menor tensor tienen.

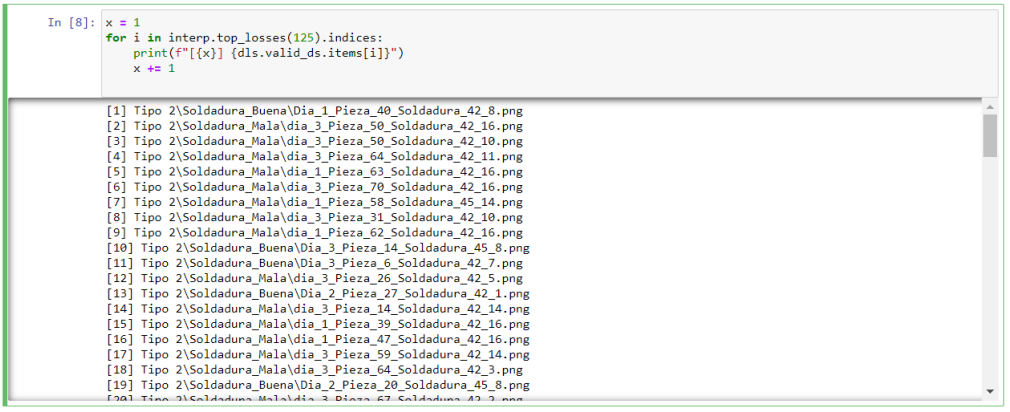

Figura 5.29: Código red neuronal 7

5.5. Experimentación

Una vez hemos entendido el código que se va a utilizar para generar las redes neuronales (en todas va a ser el mismo código cambiando la ruta de las imágenes), vamos a exponer los diferentes experimentos que hemos realizado para ir puliendo el dataset.

En esta tabla, se han introducido las tasas de error de cada tipo con el primer dataset:

DataSet 1				
22/03/2021	ResNet18	ResNet34	ResNet50	ResNet101
Tipo 1	0,11%	0,00%	0,22%	0,00%
Tipo 2	0,61%	0,28%	0,28%	0,19%
Tipo 3	0,92%	0,97%	0,56%	0,61%
Tipo 4	2,88%	2,80%	2,33%	2,11%
Tipo 5	3,65%	3,78%	2,93%	3,11%
Tipo 6	2,29%	1,79%	1,19%	1,59%
Tipo 7	0,15%	0,00%	0,15%	0,15%
Tipo 8	0,00%	0,00%	0,00%	0,00%
Tipo 9	0,19%	0,09%	0,00%	0,05%
MEDIA	1,20%	1,08%	0,85%	0,87%

Figura 5.30: Tasas de error primer dataset

Como puede observarse, hay 9 tipos de soldaduras, y para cada una, se crean cuatro redes diferentes, ResNet18, ResNet34, ResNet50 y ResNet101.

En todas estas redes, se utiliza una red previamente entrenada con millones de imágenes, que desde el principio tiene los parámetros ajustados para reconocer figuras, lo que agiliza mucho el proceso. La diferencia principal entre los diferentes tipos de ResNet es el número de capas ocultas que tiene la red. Este número viene indicado por el número que hay al final de nombre de cada ResNet, de tal manera que la ResNet34, tiene 34 capas ocultas, y así sucesivamente. Evidentemente, cuantas más capas ocultas tenga una red, más tardará en entrenarse, por eso hay que evaluar qué tipo de red es la mejor para nuestro proyecto concreto.

De esta manera, la red que menos tasa de error media tiene es la ResNet50, con un 0,86 % de tasa de error media. A continuación va la ResNet101, con un 0,87 % de tasa de error media, después la ResNet34, con un 1,08 % y por último, y como era de esperar, la ResNet18, que es la más rápida y la que menos capas ocultas tiene, con un 1,2 % de tasa de error media.

Como en nuestro caso, al principio solo estamos realizando el estudio, nos da igual el tiempo, así que elegiremos simplemente la que menor tasa de error medio tenga, pero en el último dataset, buscaremos un buen resultado para la ResNet18, de tal manera que sea relativamente rápida de entrenar.

Estos datasets tenían una serie de defectos que eran bastante fáciles de mejorar:

- En este dataset, observando las matrices de confusión de cualquier tipo, por ejemplo esta del tipo 1 con ResNet18: Podemos observar que existen muchas

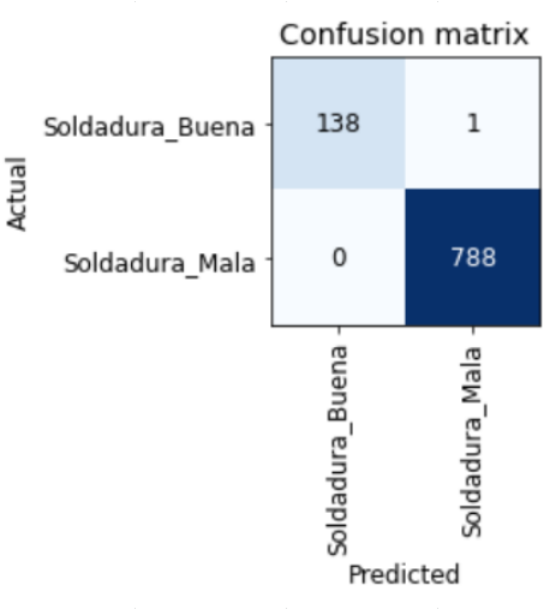

Figura 5.31: Ejemplo matriz de confusión tipo 1 ResNet18

más imágenes en Soldadura_Mala que en Soldadura_Buena. Para arreglar esto, se hizo augmentation de las imágenes de Soldadura_Buena, para equilibrar el número de imágenes de cada tipo, ya que en algunos (más exagerados que el de la imagen), la mejor solución era clasificar todas las imágenes como "Soldadura_Mala", ya que obtenía una tasa de fallo muy pequeña debido a este desequilibrio.

- Introducimos imágenes enteras de color negro y blanco para poder hacer frente a pequeños lapsus del robot, e imágenes de la pieza sin la soldadura por si se daba el caso de que se quedaba el soldador sin material, no soldaba nada y se quedaba la pieza sin soldar. En ambos casos el resultado debía ser "Soldadura_Mala".

Tras realizar esos pequeños cambios, se completó el segundo dataset, gracias al cual se obtuvieron los siguientes resultados:

DataSet 2				
29/03/2021	ResNet18	ResNet34	ResNet50	ResNet101
Tipo 1	0,08%	0,00%	0,00%	0,08%
Tipo 2	0,28%	0,11%	0,18%	0,19%
Tipo 3	0,86%	0,59%	0,47%	0,43%
Tipo 4	1,61%	1,45%	1,07%	1,04%
Tipo 5	2,87%	2,74%	2,25%	2,33%
Tipo 6	0,66%	0,51%	0,44%	0,59%
Tipo 7	0,00%	0,00%	0,00%	0,00%
Tipo 8	0,00%	0,00%	0,00%	0,00%
Tipo 9	0,09%	0,04%	0,04%	0,13%
MEDIA	0,72%	0,60%	0,49%	0,53%

Figura 5.32: Tasas de error segundo dataset

Lo que produce una mejora con respecto al anterior dataset de lo siguiente:

Mejora D2 respecto a D1				
22/03/2021	ResNet18	ResNet34	ResNet50	ResNet101
Tipo 1	0,03%	0,00%	0,22%	-0,08%
Tipo 2	0,33%	0,18%	0,11%	0,00%
Tipo 3	0,06%	0,38%	0,09%	0,18%
Tipo 4	1,27%	1,34%	1,26%	1,07%
Tipo 5	0,77%	1,04%	0,69%	0,78%
Tipo 6	1,63%	1,28%	0,75%	1,00%
Tipo 7	0,15%	0,00%	0,15%	0,15%
Tipo 8	0,00%	0,00%	0,00%	0,00%
Tipo 9	0,10%	0,05%	-0,04%	-0,08%
MEDIA	0,48%	0,47%	0,36%	0,34%

Figura 5.33: Mejora tasas de error segundo dataset respecto al primer dataset

Como se puede observar, la mejora es bastante notable, pero todavía se podía mejorar más, especialmente los tipos 4 y 5.

El problema que había con este segundo dataset era un fallo humano. Se habían introducido imágenes que deberían estar en la carpeta "Soldadura_Mala" en la carpeta "Soldadura_Buena", y viceversa. Esto fue relativamente fácil de identificar gracias a la función "top_losses" [5.29].

Una vez se hubo revisado y depurado las imágenes del dataset, se obtuvieron los siguientes resultados (solo de la 18, buscando una mayor eficiencia en el ritmo de entrenamiento de la red):

DataSet 3				
05/04/2021	ResNet18	ResNet34	ResNet50	ResNet101
Tipo 1	N/A	N/A	N/A	N/A
Tipo 2	0,00%	N/A	N/A	N/A
Tipo 3	0,11%	N/A	N/A	N/A
Tipo 4	0,40%	N/A	N/A	N/A
Tipo 5	1,04%	N/A	N/A	N/A
Tipo 6	N/A	N/A	N/A	N/A
Tipo 7	1,46%	N/A	N/A	N/A
Tipo 8	0,48%	N/A	N/A	N/A
Tipo 9	0,00%	N/A	N/A	N/A
Tipo 10	0,00%	N/A	N/A	N/A
Tipo 11	0,00%	N/A	N/A	N/A
Tipo 12	N/A	N/A	N/A	N/A
Tipo 13				
MEDIA	0,39%			

Figura 5.34: Tasas de error tercer dataset

Así, obtenemos una tasa de error media de un 0,39 %, lo que significa que, de media, solo clasifica mal una imagen por cada 256.

5.6. Conclusiones

Se ha obtenido una red que llega a clasificar con una tasa de error media de un 0,39 % soldaduras de 9 tipos diferentes, siendo capaz de identificar defectos de unos pocos píxeles, y que suponen una falta de calidad importante en el plano físico.

Se ha demostrado la facilidad de uso de la librería de python "fastai", con la que con tan solo unas pocas líneas de código se han obtenido resultados excelentes.

Capítulo 6

Evaluación del aprendizaje

6.1. Descripción del problema

Hay una situación que se repite de una manera relativamente constante a la hora de crear redes que clasifiquen imágenes, en la cual, una imagen que es de un tipo claramente, la red no es capaz de identificarla como tal.

Imaginemos una situación en la que se pretenda entrenar a la red a largo plazo, de tal manera que cada día se la introduzcan unos cuantas imágenes que no ha sido capaz de clasificar correctamente a lo largo de ese día. ¿Cómo se debería proceder para intentar que la red aprenda a clasificar correctamente las nuevas imágenes sin que eso afecte al resto de imágenes que ya clasifica correctamente?

La respuesta corta es: no se puede. Siempre que se reentrena una red, los bias y los pesos de los perceptrones sufren modificaciones, y esto puede hacer que alguna de las imágenes que antes clasificaba correctamente, aunque tuviera dudas, ahora no sea capaz de clasificarla correctamente.

Sin embargo, hay maneras mejores y peores de llevar esto a cabo, y esto es lo que se pretende investigar.

6.2. Descripción del dataset

El dataset que se va a utilizar para llevar a cabo los experimentos que intentarán responder a la pregunta previamente planteada será el mismo dataset que se ha explicado en el anterior apartado.

Como he mencionado anteriormente, el dataset está formado por imágenes de soldaduras a las que se las ha llevado a cabo un preprocessing en el que se obtiene una nube de puntos de la soldadura, que se utilizará para crear una imagen en dos dimensiones utilizando una escala de grises en las que cada píxel contiene un valor entre el 0 y el 255, indicando la altura de esa región del plano físico, y representándose de un color más claro, llegando al blanco en el caso del 255, suponiendo la máxima altura, y un color más oscuro, llegando al negro, en el caso del 0, suponiendo la mínima altura, y todos los intermedios formando una escala de grises.

6.3. Diseño de los experimentos

Lo primero de todo, cabe destacar que para realizar todos los experimentos se van a utilizar los mismos parámetros:

- Se utilizará la última versión de fastai (2.2.7).
- Se utilizará siempre la misma lectura de datos, que estarán ubicadas en lugares similares.
- Se utilizará el mismo porcentaje de imágenes para formar el dataset de prueba, y la selección será aleatoria utilizando la misma semilla.
- Todas las redes serán reentrenadas desde la misma red, la 18.
- Siempre se mostrarán los mismos datos: "epoch", que indica el número de la época, "train_loss" y "valid_loss", que indican valores de la función de pérdida y nos ayudan a evitar el sobreentrenamiento, "error_rate", que nos indica la tasa de error de la red en cada época de entrenamiento, y "time", que indica el tiempo que ha tardado en entrenarse la época.
- Siempre que haga falta reentrenar una red, se utilizará el método fine_tune, con 20 épocas de entrenamiento.

Para llevar a cabo esta investigación, se realizará un experimento base, en el que se entrenará una red neuronal con un dataset y se clasificarán una serie de imágenes, de las que se obtendrán un conjunto de datos (tasa de error, clasificación...).

A continuación, se realizarán una serie de experimentos que ahora se describirán, en los que se obtendrán los mismos datos, que se compararán con los datos obtenidos en el experimento base, y de esta manera, se sacarán conclusiones.

Las estrategias de experimentación que se van a realizar son las siguientes:

A. Se entrenará desde cero el mismo dataset, incluyendo las imágenes de la nueva soldadura, a la que se la aplicarán algunas transformaciones para que tenga más peso sobre el resto de imágenes del dataset. En total, se introducirán 16 imágenes similares.

B. Se reentrenará la red generada en el experimento base, utilizando tan solo imágenes de la nueva soldadura, a la que también se la aplicarán algunas transformaciones. En total, se introducirán 64 imágenes similares.

C. Se reentrenará la red generada en el experimento base, incluyendo en el dataset las imágenes de la nueva soldadura, a la que también se la aplicarán algunas transformaciones para que tenga más peso sobre el resto de las imágenes del dataset. En total, se introducirán 64 imágenes similares.

D. Se reentrenará la red generada en el experimento base, incluyendo en el dataset las imágenes de la nueva soldadura, a la que también se la aplicarán algunas transformaciones para que tenga más peso sobre el resto de las imágenes del dataset. En este caso, sin embargo, se dará mucho más peso a la nueva imagen, y se realizará un augmentation lo suficientemente grande como para que en torno al 10 % del dataset contenga imágenes relacionadas con esta. En total, se introducirán 600 imágenes similares.

E. Se entrenará desde cero el dataset utilizado para el apartado D.

Las imágenes similares que se introducirán en los datasets, estarán creadas a partir de la nueva imagen que no es capaz de clasificar correctamente, a la que se la modificarán en la mitad de estas nuevas imágenes un 2 % de los píxeles, los cuales se "pintarán" de negro, y a la otra mitad, se modificará su intensidad (que recordemos que estaba formada por números comprendidos entre el 0 y el 255), en un número aleatorio entre el -5 y el +5.

Además, para cada uno de los casos se realizará lo siguiente:

- Se generarán los resultados y se compararán con los resultados del experimento base.
- Se generarán gráficos en los que se mostrará la tasa de error y la exactitud de la red.
- Se enunciarán las conclusiones.

En resumen, lo que se va a realizar para cada una de las imágenes que se pretenden estudiar es lo siguiente:

Experimentos	Dataset	Tipo de entrenamiento
Caso Base	Original	Desde cero
Caso A	Original + 16 imágenes nuevas	Desde cero
Caso B	Original + 64 imágenes nuevas	Reentrenamiento
Caso C	Original + (original + 16 imágenes nuevas)	Reentrenamiento
Caso D	Original + (original + 600 imágenes nuevas)	Reentrenamiento
Caso E	Original + 600 imágenes nuevas	Desde cero

Figura 6.1: Resumen de los experimentos

6.4. Experimentación. Análisis de aprendizaje sobre soldaduras sencillas

Para la realización de estos experimentos, se va a utilizar una red obtenida de un dataset del que se obtiene una tasa de error de un 0 %, por lo que se esperan buenos resultados en poco tiempo.

6.4.1. Experimento 1

Para el primer experimento se va a utilizar la siguiente imagen, que representa una soldadura mala, y esta imagen no estaba incluida en el dataset que ha entrenado la red:

Figura 6.2: Ejemplo soldadura experimento 1

6.4.1.1. Caso base

En el caso base utilizamos solo la base de datos original.

Los datos obtenidos se pueden ver resumidos en la siguiente gráfica:

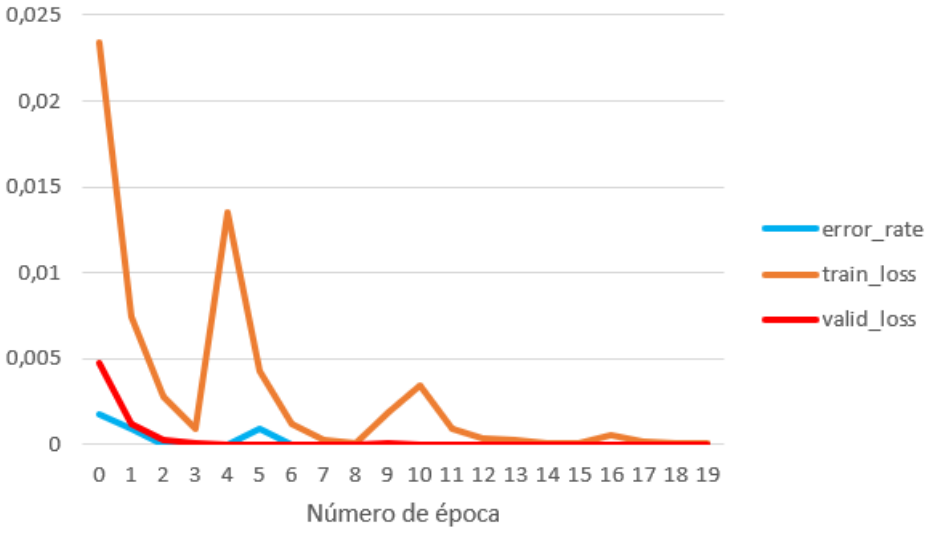

Figura 6.3: Gráfica de datos, experimento 1, caso base

Y la salida que obtenemos al predecir la clasificación de esta imagen con esta red es la siguiente:

```
a,b,c=learn.predict("/content/Defecto exagerado 1 tipo 8.png")
print(a)
print(c)

Soldadura_Buena
tensor([0.9712, 0.0288])
```

Figura 6.4: Resultado experimento 1 caso base

Lo que significa que la red predice que esta imagen representa una soldadura buena (sin defectos).

Esto nos indica que la red tiene que ser reentrenada.

6.4.1.2. Caso A

En el caso A se utilizará la base de datos del caso base, a la que se le añadirán 16 imágenes derivadas de la imagen que queremos clasificar, y esto se entrenará desde cero.

El gráfico que muestra la tasa de error, la función de optimización y función de pérdida en función de la época de entrenamiento es el siguiente:

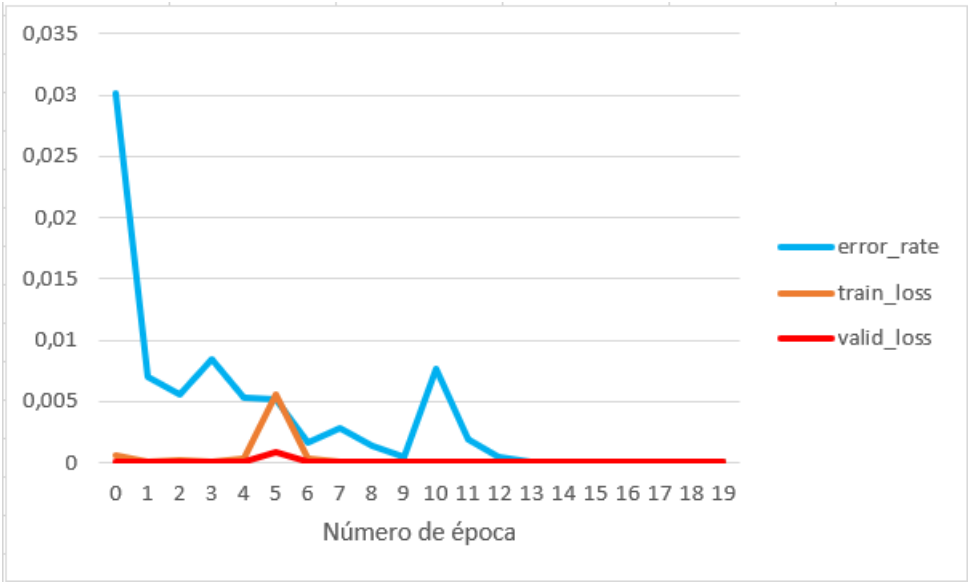

Figura 6.5: Gráfica de datos, experimento 1, caso A

Lo que no supone una bajada en el rendimiento con respecto al caso base, ya que mantiene la tasa de error.

En este caso, el resultado de la predicción de la imagen es el siguiente:

```
a,b,c=learn.predict("/content/Defecto exagerado 1 tipo 8.png")
print(a)
print(c)

Soldadura_Mala
tensor([5.2308e-10, 1.0000e+00])
```

Figura 6.6: Resultado experimento 1, caso A

Como puede observarse, la predicción en este caso es que la imagen es una soldadura mala que se clasifica correctamente, con bastante seguridad (más de un 99.99 %).

6.4.1.3. Caso B

En el caso B se utilizarán 64 imágenes derivadas de la imagen que queremos clasificar. Este dataset se utilizará para reentrenar la red del caso base.

El gráfico que muestra la tasa de error, la función de optimización y función de pérdida en función de la época de entrenamiento es el siguiente:

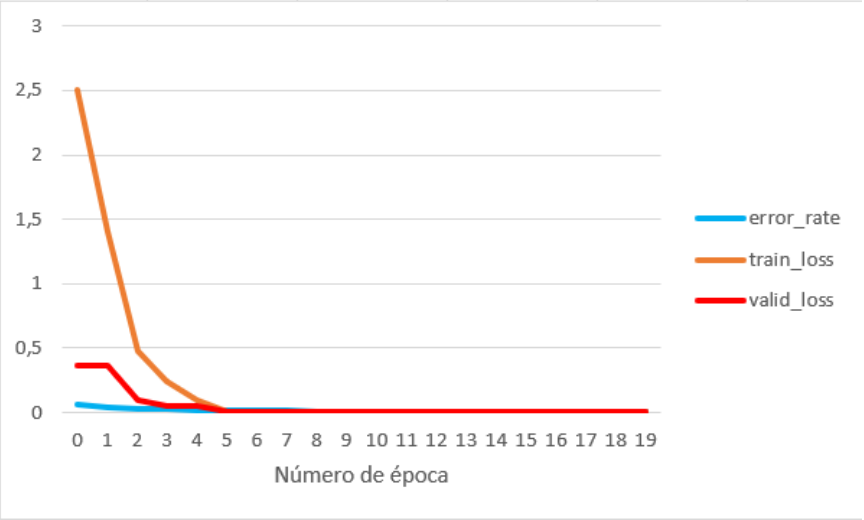

Figura 6.7: Gráfica de datos, experimento 1, caso B

Lo que no supone una bajada en el rendimiento con respecto al caso base, ya que mantiene la tasa de error.

Después de reentrenar la red del caso base, el resultado de la predicción de la imagen es el siguiente:

```
[36] a,b,c=learn2.predict("/content/Defecto exagerado 1 tipo 8.png")
     print(a)
     print(c)

     Soldadura_Mala
     tensor([4.5586e-16, 1.0000e+00])
```

Figura 6.8: Resultado experimento 1 caso B

Como puede observarse, la predicción en este caso es que la imagen es una soldadura mala, por lo que se clasifica correctamente, con bastante seguridad (más de un 99.99 %).

6.4.1.4. Caso C

En el caso C se utilizará la base de datos del caso base, a la que se le añadirán 16 imágenes derivadas de la imagen que queremos clasificar. Este dataset se utilizará para reentrenar la red del caso base.

El gráfico que muestra la tasa de error, la función de optimización y función de pérdida en función de la época de entrenamiento es el siguiente:

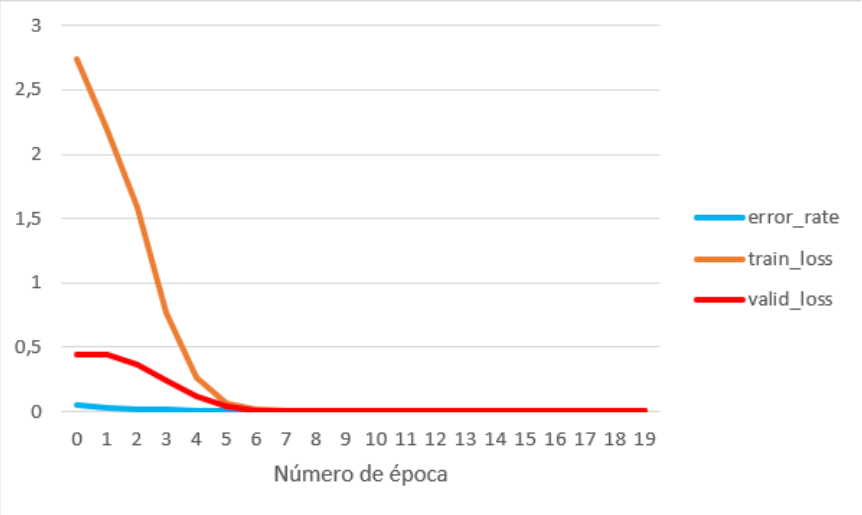

Figura 6.9: Gráfica de datos, experimento 1, caso C

Lo que no supone una bajada en el rendimiento con respecto al caso base, ya que mantiene la tasa de error.

Después de reentrenar la red del caso base, el resultado de la predicción de la imagen es el siguiente:

```
a,b,c=learn3.predict("/content/Defecto exagerado 1 tipo 8.png")
print(a)
print(c)

Soldadura_Mala
tensor([7.7604e-11, 1.0000e+00])
```

Figura 6.10: Resultado experimento 1 caso C

Como puede observarse, la predicción en este caso es que la imagen es una soldadura mala, es decir, que la clasifica correctamente. Además, con bastante seguridad (más de un 99.99 %).

6.4.1.5. Caso D

En el caso D se utilizará la base de datos del caso base, a la que se le añadirán 600 imágenes derivadas de la imagen que queremos clasificar. Este dataset se utilizará para reentrenar la red del caso base.

El gráfico que muestra la tasa de error, la función de optimización y función de pérdida en función de la época de entrenamiento es el siguiente:

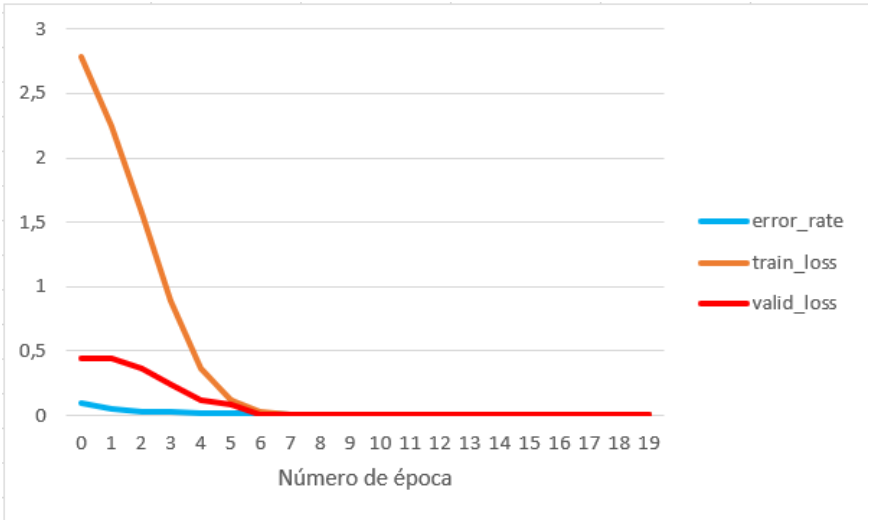

Figura 6.11: Gráfica de datos, experimento 1, caso D

Lo que no supone una bajada en el rendimiento con respecto al caso base, ya que mantiene la tasa de error.

Después de reentrenar la red del caso base, el resultado de la predicción de la imagen es el siguiente:

```
[47] a,b,c=learn4.predict("/content/Defecto exagerado 1 tipo 8.png")
     print(a)
     print(c)

     Soldadura_Mala
     tensor([7.0368e-11, 1.0000e+00])
```

Figura 6.12: Resultado experimento 1 caso D

Como puede observarse, la predicción en este caso es que la imagen es una soldadura mala, es decir, que la clasifica correctamente. Además, con bastante seguridad (más de un 99.99 %).

6.4.1.6. Caso E

En el caso E se utilizará la base de datos del caso base, a la que se le añadirán 600 imágenes derivadas de la imagen que queremos clasificar, y esto se entrenará desde cero.

El gráfico que muestra la tasa de error, la función de optimización y función de pérdida en función de la época de entrenamiento es el siguiente:

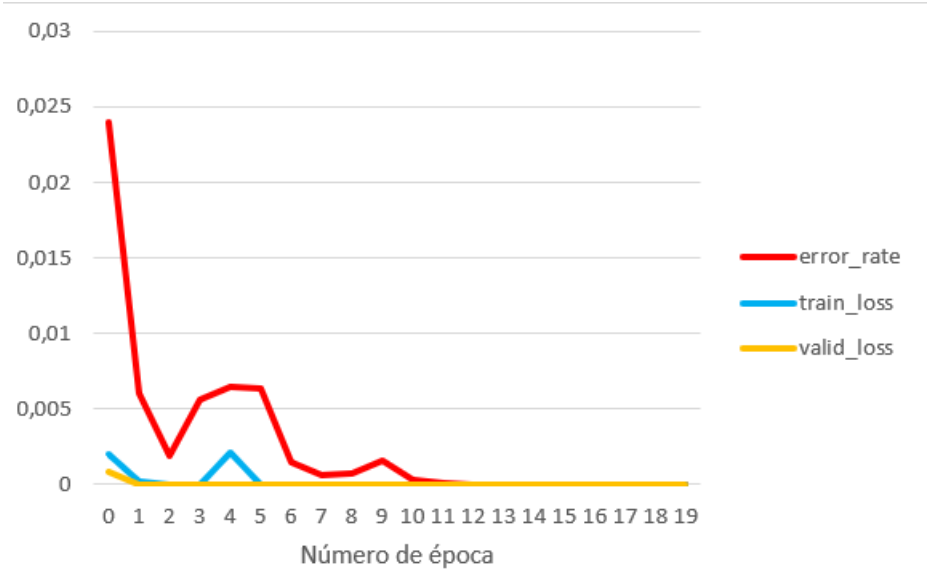

Figura 6.13: Gráfica de datos, experimento 1, caso E

Lo que no supone una bajada en el rendimiento con respecto al caso base, ya que mantiene la tasa de error.

Después de reentrenar la red del caso base, el resultado de la predicción de la imagen es el siguiente:

```
a,b,c=learn.predict("/content/Defecto exagerado 1 tipo 8.png")
print(a)
print(c)
```

```
Soldadura_Mala
tensor([8.1309e-10, 1.0000e+00])
```

Figura 6.14: Resultado experimento 1 caso E

Como puede observarse, la predicción en este caso es que la imagen es una soldadura mala, es decir, que la clasifica correctamente. Además, con bastante seguridad (más de un 99.99 %).

El resumen del experimento 1 ha sido el siguiente:

	Resultado	Tensor	Tiempo
Caso Base	Soldadura_Buena	[1.0000e+00, 3.3391e-10]	30 min 05 seg
Caso A	Soldadura_Mala	[4.6516e-15, 1.0000e+00]	41 min 41 seg
Caso B	Soldadura_Mala	[3.7714e-09, 1.0000e+00]	00 min 19 seg
Caso C	Soldadura_Mala	[1.4042e-08, 1.0000e+00]	12 min 10 seg
Caso D	Soldadura_Mala	[6.9172e-09, 1.0000e+00]	11 min 25 seg
Caso E	Soldadura_Mala	[1.0420e-11, 1.0000e+00]	33 min 06 seg

Figura 6.15: Resumen experimento 1

6.4.2. Experimento 2

Para el segundo experimento se va a utilizar la siguiente imagen, que representa una soldadura mala, y esta imagen no estaba incluida en el dataset que ha entrenado la red:

Figura 6.16: Ejemplo soldadura experimento 2

6.4.2.1. Caso base

En el caso base utilizamos solo la base de datos original.

Los datos obtenidos se pueden ver resumidos en la siguiente gráfica:

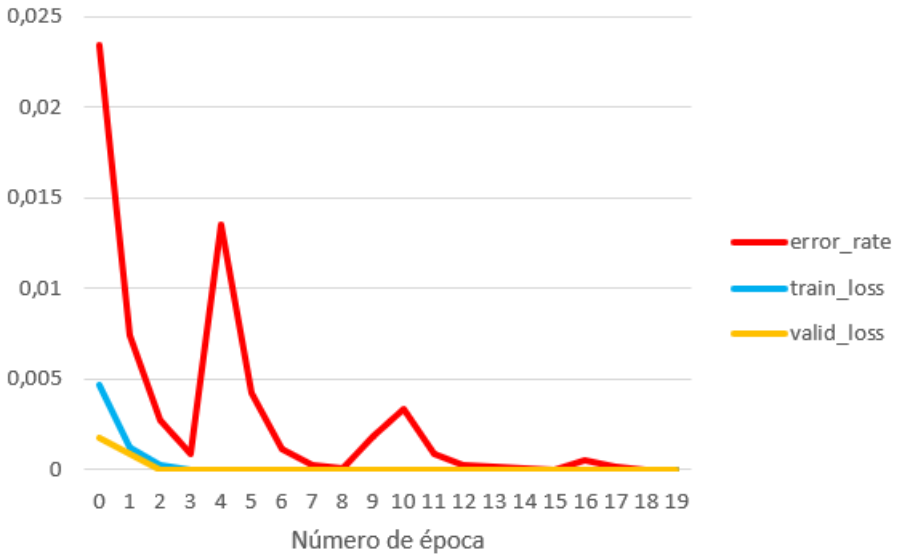

Figura 6.17: Gráfica de datos, experimento 2, caso base

Lo que no supone una bajada en el rendimiento con respecto al caso base, ya que mantiene la tasa de error.

Y la salida que obtenemos al predecir la clasificación de esta imagen con esta red es la siguiente:

```
a,b,c=learn.predict("/content/Defecto exagerado 1 tipo 8.png")
print(a)
print(c)

Soldadura_Buena
tensor([0.9712, 0.0288])
```

Figura 6.18: Resultado experimento 2 caso base

Lo que significa que la red predice que esta imagen representa una soldadura buena (sin defectos), y no es así.

Esto nos indica que la red tiene que ser reentrenada.

6.4.2.2. Caso A

En el caso A se utilizará la base de datos del caso base, a la que se le añadirán 16 imágenes derivadas de la imagen que queremos clasificar, y esto se entrenará desde cero.

El gráfico que muestra la tasa de error, la función de optimización y función de pérdida en función de la época de entrenamiento es el siguiente:

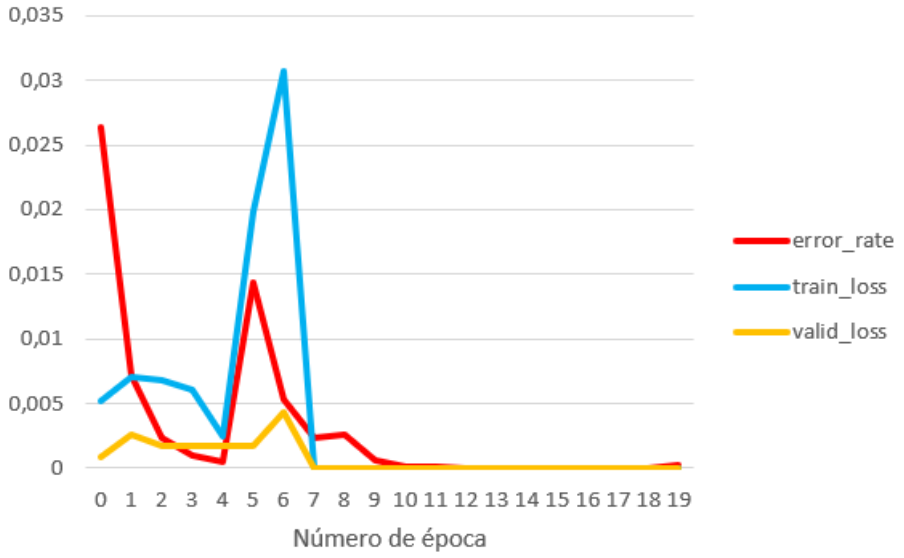

Figura 6.19: Gráfica de datos, experimento 2, caso A

Lo que no supone una bajada en el rendimiento con respecto al caso base, ya que mantiene la tasa de error.

En este caso, el resultado de la predicción de la imagen es el siguiente:

```
a,b,c=learn.predict("/content/Defecto exagerado 2 tipo 8.png")
print(a)
print(c)

Soldadura_Mala
tensor([4.6516e-15, 1.0000e+00])
```

Figura 6.20: Resultado experimento 2, caso A

Como puede observarse, la predicción en este caso es que la imagen es una soldadura mala que se clasifica correctamente, con bastante seguridad (más de un 99.99 %).

6.4.2.3. Caso B

En el caso B se utilizarán 64 imágenes derivadas de la imagen que queremos clasificar. Este dataset se utilizará para reentrenar la red del caso base.

El gráfico que muestra la tasa de error, la función de optimización y función de pérdida en función de la época de entrenamiento es el siguiente:

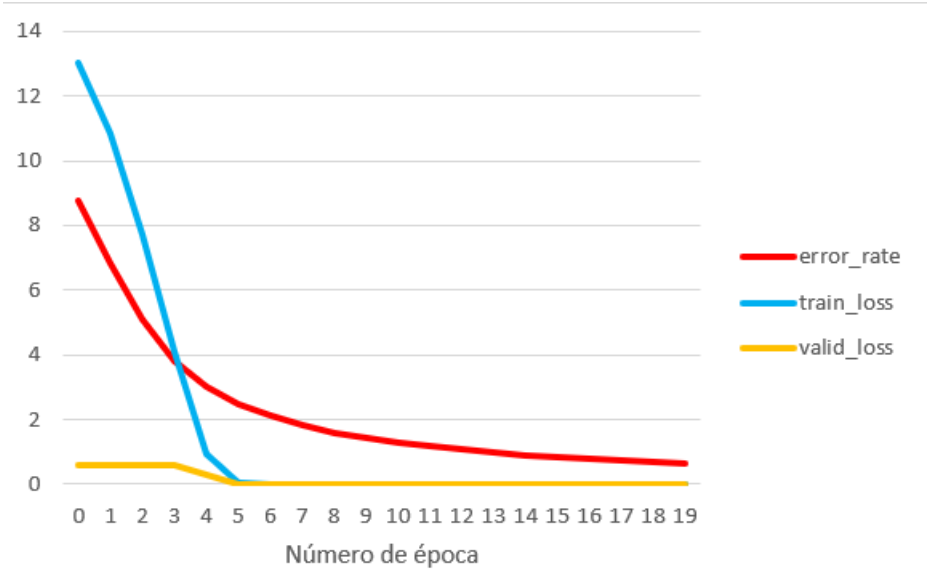

Figura 6.21: Gráfica de datos, experimento 2, caso B

Lo que no supone una bajada en el rendimiento con respecto al caso base, ya que mantiene la tasa de error.

Después de reentrenar la red del caso base, el resultado de la predicción de la imagen es el siguiente:

```
a,b,c=learn2.predict("/content/Defecto exagerado 2 tipo 8.png")
print(a)
print(c)

Soldadura_Mala
tensor([3.7714e-09, 1.0000e+00])
```

Figura 6.22: Resultado experimento 2 caso B

Como puede observarse, la predicción en este caso es que la imagen es una soldadura mala, por lo que se clasifica correctamente, con bastante seguridad (más de un 99.99 %).

6.4.2.4. Caso C

En el caso C se utilizará la base de datos del caso base, a la que se le añadirán 16 imágenes derivadas de la imagen que queremos clasificar. Este dataset se utilizará para reentrenar la red del caso base.

El gráfico que muestra la tasa de error, la función de optimización y función de pérdida en función de la época de entrenamiento es el siguiente:

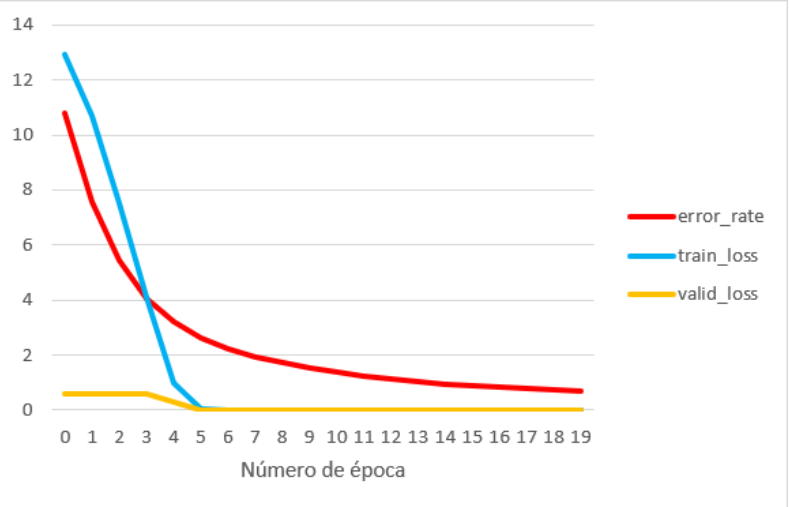

Figura 6.23: Gráfica de datos, experimento 2, caso C

Lo que no supone una bajada en el rendimiento con respecto al caso base, ya que mantiene la tasa de error.

Después de reentrenar la red del caso base, el resultado de la predicción de la imagen es el siguiente:

```
a,b,c=learn3.predict("/content/Defecto exagerado 2 tipo 8.png")
print(a)
print(c)
```

```
Soldadura_Mala
tensor([1.4042e-08, 1.0000e+00])
```

Figura 6.24: Resultado experimento 2 caso C

Como puede observarse, la predicción en este caso es que la imagen es una soldadura mala, es decir, que la clasifica correctamente. Además, con bastante seguridad (más de un 99.99 %).

6.4.2.5. Caso D

En el caso D se utilizará la base de datos del caso base, a la que se le añadirán 600 imágenes derivadas de la imagen que queremos clasificar. Este dataset se utilizará para reentrenar la red del caso base.

El gráfico que muestra la tasa de error, la función de optimización y función de pérdida en función de la época de entrenamiento es el siguiente:

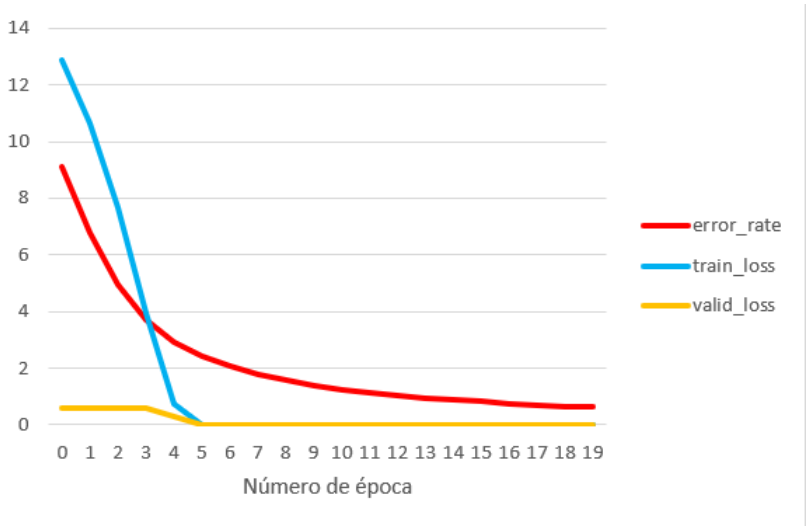

Figura 6.25: Gráfica de datos, experimento 2, caso D

Lo que no supone una bajada en el rendimiento con respecto al caso base, ya que mantiene la tasa de error.

Después de reentrenar la red del caso base, el resultado de la predicción de la imagen es el siguiente:

```
a,b,c=learn4.predict("/content/Defecto exagerado 2 tipo 8.png")
print(a)
print(c)

Soldadura_Mala
tensor([6.9172e-09, 1.0000e+00])
```

Figura 6.26: Resultado experimento 2 caso D

Como puede observarse, la predicción en este caso es que la imagen es una soldadura mala, es decir, que la clasifica correctamente. Además, con bastante seguridad (más de un 99.99 %).

6.4.2.6. Caso E

En el caso E se utilizará la base de datos del caso base, a la que se le añadirán 600 imágenes derivadas de la imagen que queremos clasificar, y esto se entrenará desde cero.

El gráfico que muestra la tasa de error, la función de optimización y función de pérdida en función de la época de entrenamiento es el siguiente:

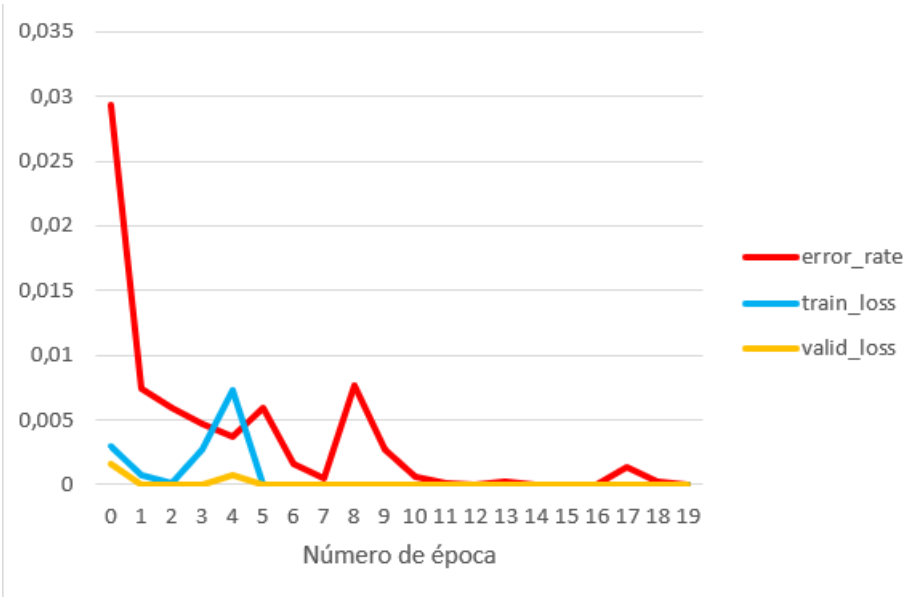

Figura 6.27: Gráfica de datos, experimento 2 caso E

Lo que no supone una bajada en el rendimiento con respecto al caso base.

Después de reentrenar la red del caso base, el resultado de la predicción de la imagen es el siguiente:

```
a,b,c=learn.predict("/content/Defecto exagerado 2 tipo 8.png")
print(a)
print(c)

Soldadura_Mala
tensor([1.0420e-11, 1.0000e+00])
```

Figura 6.28: Resultado experimento 2 caso E

Como puede observarse, la predicción en este caso es que la imagen es una soldadura mala, es decir, que la clasifica correctamente. Además, con bastante seguridad (más de un 99.99 %).

El resumen del experimento 2 ha sido el siguiente:

	Resultado	Tensor	Tiempo
Caso Base	Soldadura_Buena	[1.0000e+00, 3.3391e-10]	29 min 15 seg
Caso A	Soldadura_Mala	[4.6516e-15, 1.0000e+00]	99 min 50 seg
Caso B	Soldadura_Mala	[3.7714e-09, 1.0000e+00]	00 min 07 seg
Caso C	Soldadura_Mala	[1.4042e-08, 1.0000e+00]	11 min 39 seg
Caso D	Soldadura_Mala	[6.9172e-09, 1.0000e+00]	10 min 37 seg
Caso E	Soldadura_Mala	[1.0420e-11, 1.0000e+00]	33 min 05 seg

Figura 6.29: Resumen experimento 2

6.5. Experimentación. Análisis de aprendizaje sobre soldaduras complejas

Para la realización de estos experimentos, se va a utilizar una red obtenida de un dataset del que se obtiene una tasa de error de más de un 1 %, por lo que se esperan unos resultados intermedios.

6.5.1. Experimento 1

Para el primer experimento se va a utilizar la siguiente imagen, que representa una soldadura mala, y esta imagen no estaba incluida en el dataset que ha entrenado la red:

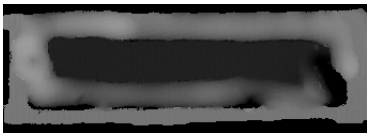

Figura 6.30: Ejemplo soldadura experimento 3

6.5.1.1. Caso base

En el caso base utilizamos solo la base de datos original.

Los datos obtenidos se pueden ver resumidos en la siguiente gráfica:

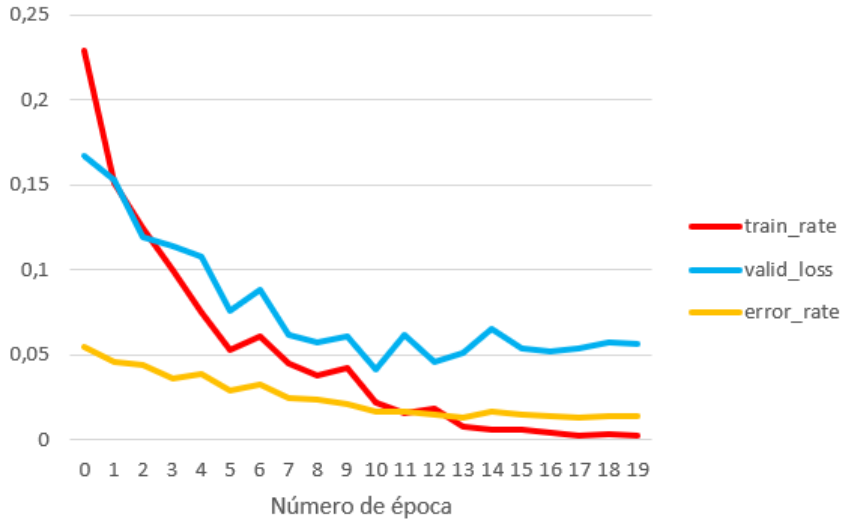

Figura 6.31: Gráfica de datos, experimento 1, caso base

Y la salida que obtenemos al predecir la clasificación de esta imagen con esta red es la siguiente:

```
a,b,c=learn.predict("/content/Esta para el tipo 5.png")
print(a)
print(c)

Soldadura_Buena
tensor([9.9909e-01, 9.1155e-04])
```

Figura 6.32: Resultado experimento 1 caso base

Lo que significa que la red predice que esta imagen representa una soldadura buena (sin defectos).

Esto nos indica que la red tiene que ser reentrenada.

6.5.1.2. Caso A

En el caso A se utilizará la base de datos del caso base, a la que se le añadirán 16 imágenes derivadas de la imagen que queremos clasificar, y esto se entrenará desde cero.

El gráfico que muestra la tasa de error, la función de optimización y función de pérdida en función de la época de entrenamiento es el siguiente:

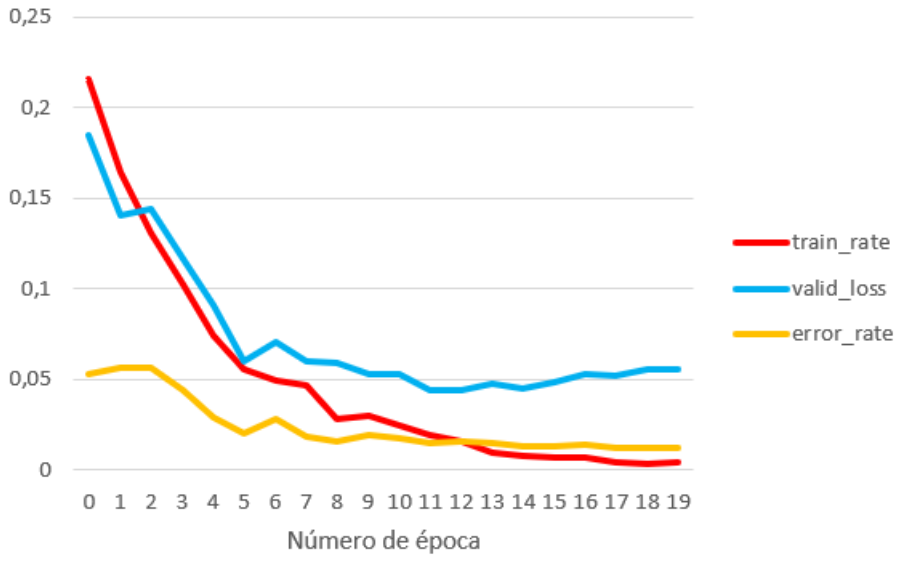

Figura 6.33: Gráfica de datos, experimento 1, caso A

Lo que no supone una bajada en el rendimiento con respecto al caso base, ya que mantiene una tasa de error similar.

En este caso, el resultado de la predicción de la imagen es el siguiente:

```
a,b,c=learn.predict("/content/Defecto exagerado 1 tipo 5.png")
print(a)
print(c)

Soldadura_Mala
tensor([8.6172e-05, 9.9991e-01])
```

Figura 6.34: Resultado experimento 1, caso A

Como puede observarse, la predicción en este caso es que la imagen es una soldadura mala que se clasifica correctamente, con bastante seguridad (más de un 99.99 %).

6.5.1.3. Caso B

En el caso B se utilizarán 64 imágenes derivadas de la imagen que queremos clasificar. Este dataset se utilizará para reentrenar la red del caso base.

El gráfico que muestra la tasa de error, la función de optimización y función de pérdida en función de la época de entrenamiento es el siguiente:

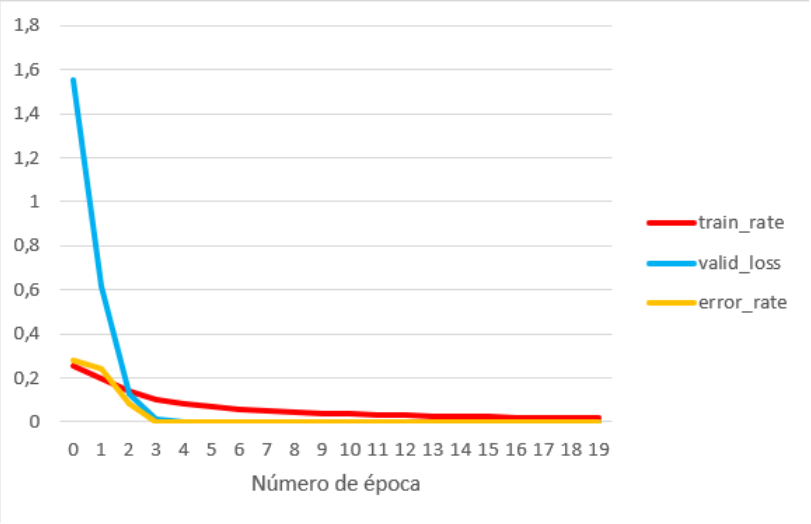

Figura 6.35: Gráfica de datos, experimento 1, caso B

En este caso no podemos saber si la red ha perdido o ganado tasa de error. Con los resultados obtenidos, la tasa de error se obtiene del dataset introducido, formado por 64 imágenes, por lo que el resultado no es comparable.

Después de reentrenar la red del caso base, el resultado de la predicción de la imagen es el siguiente:

```
a,b,c=learn2.predict("/content/Defecto exagerado 1 tipo 5.png")
print(a)
print(c)

Soldadura_Mala
tensor([1.1129e-10, 1.0000e+00])
```

Figura 6.36: Resultado experimento 1 caso B

Como puede observarse, la predicción en este caso es que la imagen es una soldadura mala, por lo que se clasifica correctamente, con bastante seguridad (más de un 99.99 %).

6.5.1.4. Caso C

En el caso C se utilizará la base de datos del caso base, a la que se le añadirán 16 imágenes derivadas de la imagen que queremos clasificar. Este dataset se utilizará para reentrenar la red del caso base.

El gráfico que muestra la tasa de error, la función de optimización y función de pérdida en función de la época de entrenamiento es el siguiente:

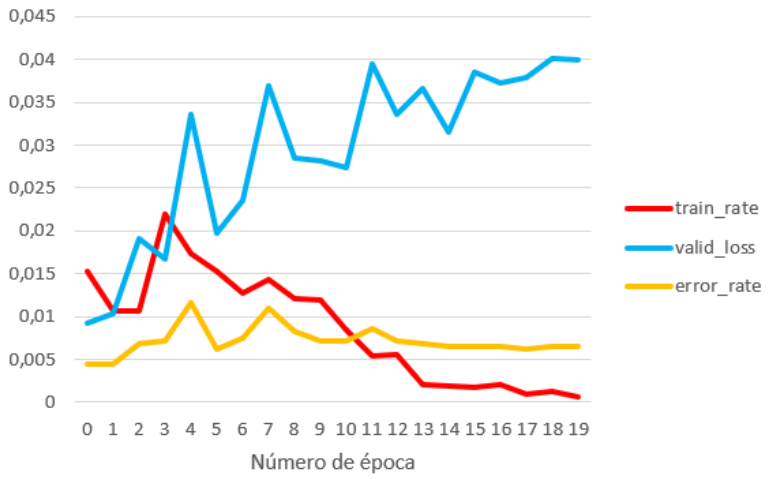

Figura 6.37: Gráfica de datos, experimento 1, caso C

Lo que nos indica que ha habido sobreentrenamiento, y pese a que la tasa de error sea menor que en el caso base, ha sido mejor en épocas anteriores. De todas maneras, las tasas de error no varían lo suficiente como para tenerlas en cuenta, ni de manera positiva ni de manera negativa.

Después de reentrenar la red del caso base, el resultado de la predicción de la imagen es el siguiente:

```
a,b,c=learn3.predict("/content/Defecto exagerado 1 tipo 5.png")
print(a)
print(c)

Soldadura_Mala
tensor([8.0981e-11, 1.0000e+00])
```

Figura 6.38: Resultado experimento 1 caso C

Como puede observarse, la predicción en este caso es que la imagen es una soldadura mala, es decir, que la clasifica correctamente. Además, con bastante seguridad (más de un 99.99 %).

6.5.1.5. Caso D

En el caso D se utilizará la base de datos del caso base, a la que se le añadirán 600 imágenes derivadas de la imagen que queremos clasificar. Este dataset se utilizará para reentrenar la red del caso base.

El gráfico que muestra la tasa de error, la función de optimización y función de pérdida en función de la época de entrenamiento es el siguiente:

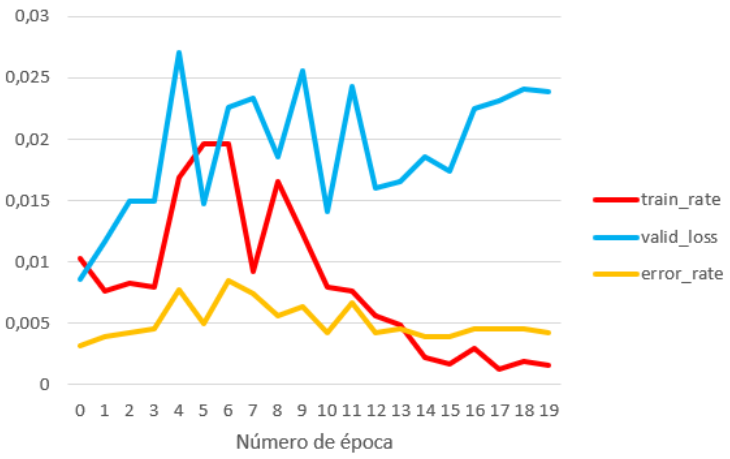

Figura 6.39: Gráfica de datos, experimento 1, caso D

Lo que nos indica que ha habido sobreentrenamiento, y pese a que la tasa de error sea menor que en el caso base, ha sido mejor en épocas anteriores. De todas maneras, las tasas de error no varían lo suficiente como para tenerlas en cuenta, ni de manera positiva ni de manera negativa.

Después de reentrenar la red del caso base, el resultado de la predicción de la imagen es el siguiente:

```
a,b,c=learn3.predict("/content/Defecto exagerado 1 tipo 5.png")
print(a)
print(c)

Soldadura_Mala
tensor([2.7003e-11, 1.0000e+00])
```

Figura 6.40: Resultado experimento 1 caso D

Como puede observarse, la predicción en este caso es que la imagen es una soldadura mala, es decir, que la clasifica correctamente. Además, con bastante seguridad (más de un 99.99 %).

6.5.1.6. Caso E

En el caso E se utilizará la base de datos del caso base, a la que se le añadirán 600 imágenes derivadas de la imagen que queremos clasificar, y esto se entrenará desde cero.

El gráfico que muestra la tasa de error, la función de optimización y función de pérdida en función de la época de entrenamiento es el siguiente:

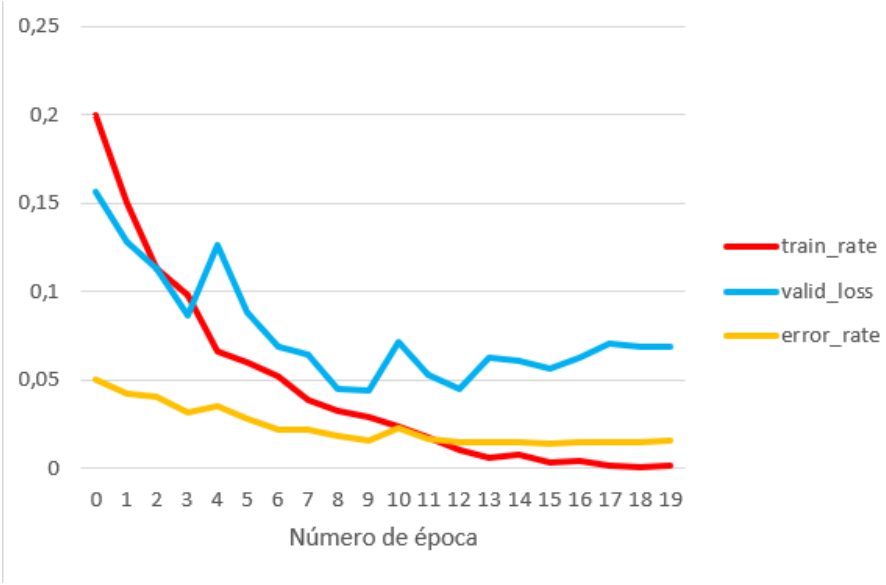

Figura 6.41: Gráfica de datos, experimento 1, caso E

Lo que no supone una bajada en el rendimiento con respecto al caso base, ya que mantiene la tasa de error.

Después de reentrenar la red del caso base, el resultado de la predicción de la imagen es el siguiente:

```
a,b,c=learn.predict("/content/Defecto exagerado 1 tipo 5.png")
print(a)
print(c)

Soldadura_Mala
tensor([4.8010e-07, 1.0000e+00])
```

Figura 6.42: Resultado experimento 1 caso E

Como puede observarse, la predicción en este caso es que la imagen es una soldadura mala, es decir, que la clasifica correctamente. Además, con bastante seguridad (más de un 99.99 %).

El resumen del experimento 1 ha sido el siguiente:

	Resultado	Tensor	Tiempo
Caso Base	Soldadura_Buena	[9.9909e-01, 9.1155e-04]	129 min 45 seg
Caso A	Soldadura_Mala	[8.6172e-05, 9.9991e-01]	73 min 46 seg
Caso B	Soldadura_Mala	[1.1129e-10, 1.0000e+00]	00 min 46 seg
Caso C	Soldadura_Mala	[8.0981e-11, 1.0000e+00]	35 min 58 seg
Caso D	Soldadura_Mala	[2.7003e-11, 1.0000e+00]	32 min 02 seg
Caso E	Soldadura_Mala	[4.8010e-07, 1.0000e+00]	92 min 20 seg

Figura 6.43: Resumen experimento 1

6.5.2. Experimento 2

Para el segundo experimento se va a utilizar la siguiente imagen, que representa una soldadura mala, y esta imagen no estaba incluida en el dataset que ha entrenado la red:

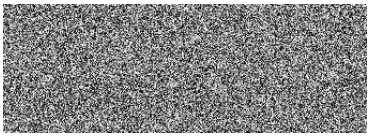

Figura 6.44: Ejemplo soldadura experimento 2

6.5.2.1. Caso base

En el caso base utilizamos solo la base de datos original.

Los datos obtenidos se pueden ver resumidos en la siguiente gráfica:

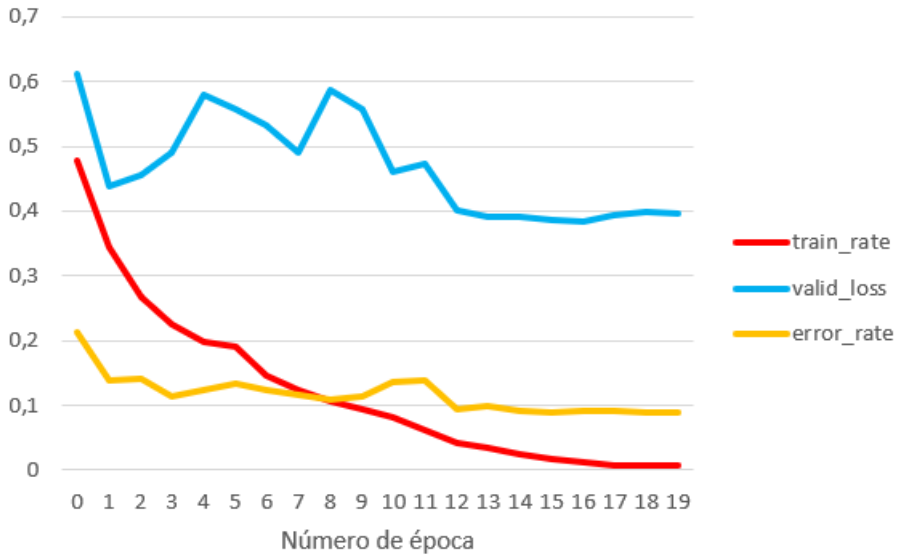

Figura 6.45: Gráfica de datos, experimento 2, caso base

Lo que no supone una bajada en el rendimiento con respecto al caso base, ya que se mantiene la tasa de error.

Y la salida que obtenemos al predecir la clasificación de esta imagen con esta red es la siguiente:

```
a,b,c=learn.predict("/content/Esta para el tipo 5.png")
print(a)
print(c)

Soldadura_Buena
tensor([0.7318, 0.2682])
```

Figura 6.46: Resultado experimento 2 caso base

Lo que significa que la red predice que esta imagen representa una soldadura buena (sin defectos), y no es así.

Esto nos indica que la red tiene que ser reentrenada.

6.5.2.2. Caso A

En el caso A se utilizará la base de datos del caso base, a la que se le añadirán 16 imágenes derivadas de la imagen que queremos clasificar, y esto se entrenará desde cero.

El gráfico que muestra la tasa de error, la función de optimización y función de pérdida en función de la época de entrenamiento es el siguiente:

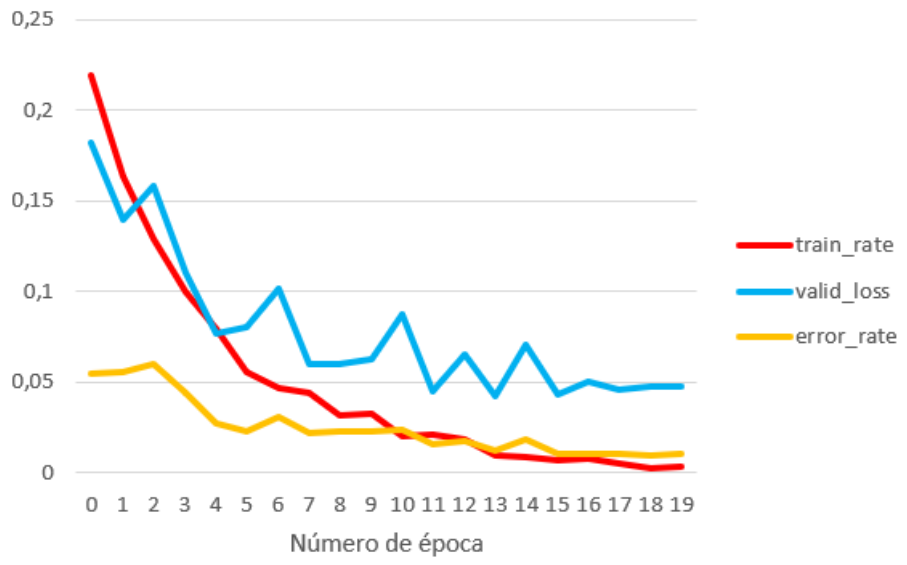

Figura 6.47: Gráfica de datos, experimento 2, caso A

Lo que no supone una bajada en el rendimiento con respecto al caso base, ya que se mantiene la tasa de error.

En este caso, el resultado de la predicción de la imagen es el siguiente:

```
a,b,c=learn.predict("/content/Esta para el tipo 5.png")
print(a)
print(c)

Soldadura_Mala
tensor([1.2424e-05, 9.9999e-01])
```

Figura 6.48: Resultado experimento 2, caso A

Como puede observarse, la predicción en este caso es que la imagen es una soldadura mala que se clasifica correctamente, con bastante seguridad (más de un 99.99 %).

6.5.2.3. Caso B

En el caso B se utilizarán 64 imágenes derivadas de la imagen que queremos clasificar. Este dataset se utilizará para reentrenar la red del caso base.

El gráfico que muestra la tasa de error, la función de optimización y función de pérdida en función de la época de entrenamiento es el siguiente:

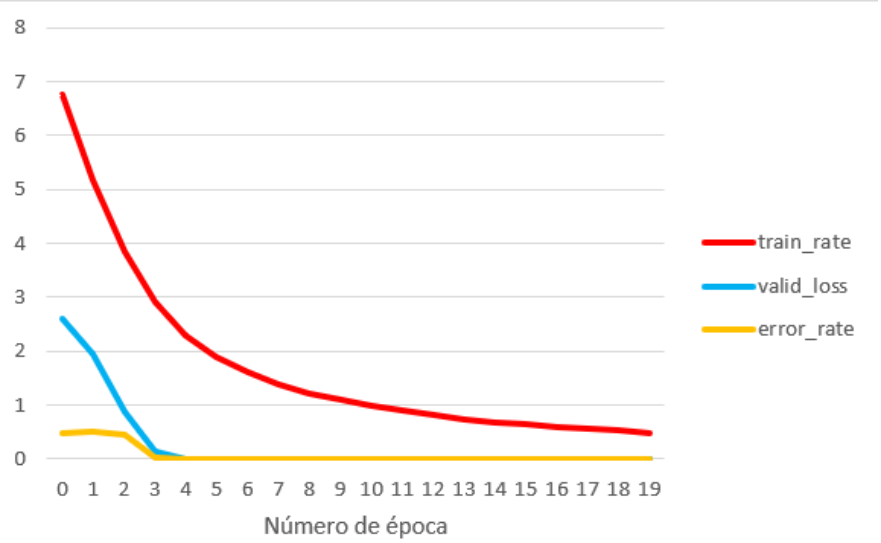

Figura 6.49: Gráfica de datos, experimento 2, caso B

En este caso no podemos saber si la red ha perdido o ganado tasa de error. Con los resultados obtenidos, la tasa de error se obtiene del dataset introducido, formado por 64 imágenes, por lo que el resultado no es comparable.

Después de reentrenar la red del caso base, el resultado de la predicción de la imagen es el siguiente:

```
a,b,c=learn2.predict("/content/Esta para el tipo 5.png")
print(a)
print(c)

Soldadura_Mala
tensor([4.0837e-12, 1.0000e+00])
```

Figura 6.50: Resultado experimento 2 caso B

Como puede observarse, la predicción en este caso es que la imagen es una soldadura mala, por lo que se clasifica correctamente, con bastante seguridad (más de un 99.99 %).

6.5.2.4. Caso C

En el caso C se utilizará la base de datos del caso base, a la que se le añadirán 16 imágenes derivadas de la imagen que queremos clasificar. Este dataset se utilizará para reentrenar la red del caso base.

El gráfico que muestra la tasa de error, la función de optimización y función de pérdida en función de la época de entrenamiento es el siguiente:

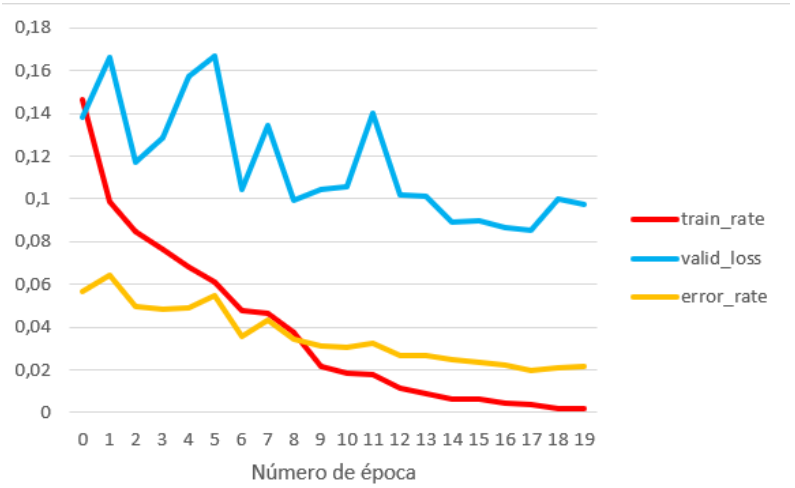

Figura 6.51: Gráfica de datos, experimento 2, caso C

Lo que no supone una bajada en el rendimiento con respecto al caso base, ya que se mantiene la tasa de error.

Después de reentrenar la red del caso base, el resultado de la predicción de la imagen es el siguiente:

```
a,b,c=learn3.predict("/content/Esta para el tipo 5.png")
print(a)
print(c)

Soldadura_Mala
tensor([2.5445e-06, 1.0000e+00])
```

Figura 6.52: Resultado experimento 2 caso C

Como puede observarse, la predicción en este caso es que la imagen es una soldadura mala, es decir, que la clasifica correctamente. Además, con bastante seguridad (más de un 99.99 %).

6.5.2.5. Caso D

En el caso D se utilizará la base de datos del caso base, a la que se le añadirán 600 imágenes derivadas de la imagen que queremos clasificar. Este dataset se utilizará para reentrenar la red del caso base.

El gráfico que muestra la tasa de error, la función de optimización y función de pérdida en función de la época de entrenamiento es el siguiente:

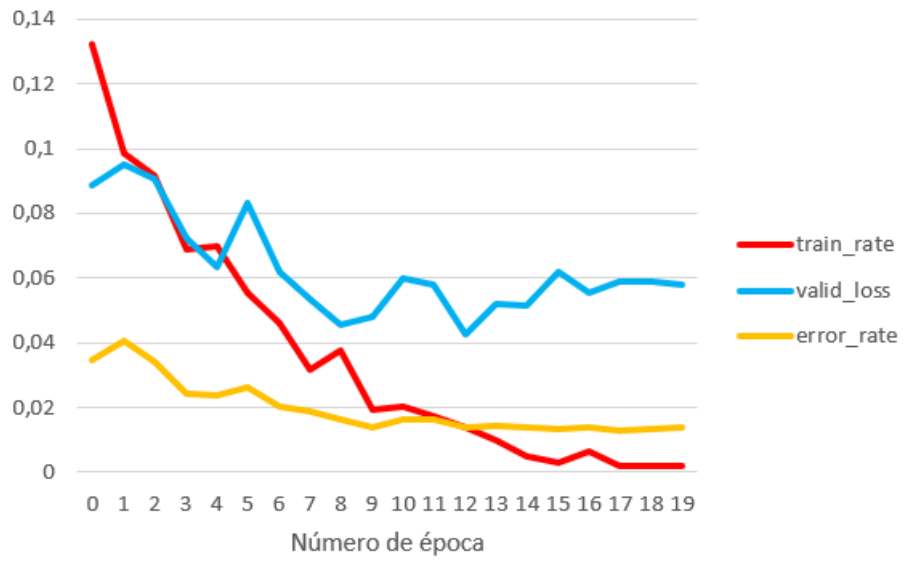

Figura 6.53: Gráfica de datos, experimento 2, caso D

Lo que no supone una bajada en el rendimiento con respecto al caso base.

Después de reentrenar la red del caso base, el resultado de la predicción de la imagen es el siguiente:

```
a,b,c=learn4.predict("/content/Esta para el tipo 5.png")
print(a)
print(c)

Soldadura_Mala
tensor([4.9648e-06, 9.9999e-01])
```

Figura 6.54: Resultado experimento 2 caso D

Como puede observarse, la predicción en este caso es que la imagen es una soldadura mala, es decir, que la clasifica correctamente. Además, con bastante seguridad (más de un 99.99 %).

6.5.2.6. Caso E

En el caso E se utilizará la base de datos del caso base, a la que se le añadirán 600 imágenes derivadas de la imagen que queremos clasificar, y esto se entrenará desde cero.

El gráfico que muestra la tasa de error, la función de optimización y función de pérdida en función de la época de entrenamiento es el siguiente:

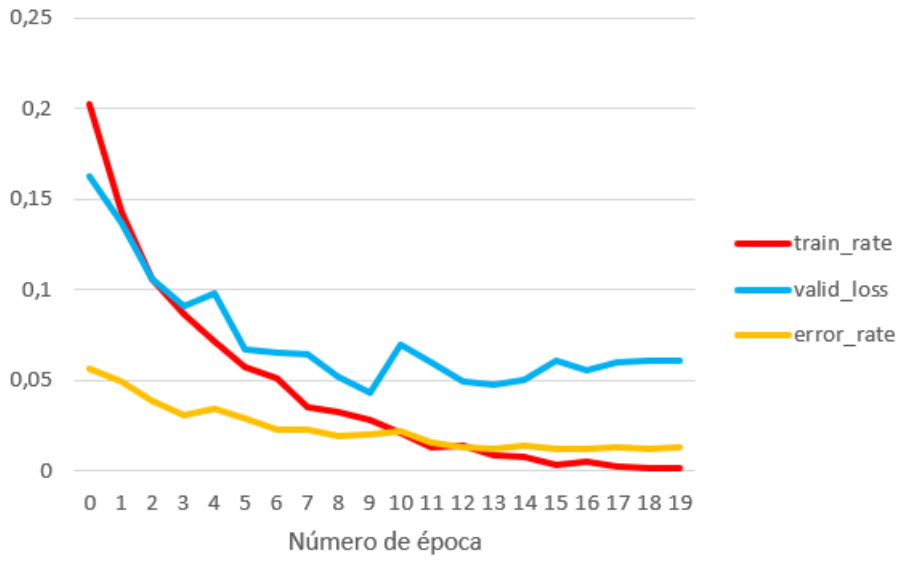

Figura 6.55: Gráfica de datos, experimento 2 caso E

Lo que no supone una bajada en el rendimiento con respecto al caso base.

Después de reentrenar la red del caso base, el resultado de la predicción de la imagen es el siguiente:

```
a,b,c=learn.predict("/content/Esta para el tipo 5.png")
print(a)
print(c)

Soldadura_Mala
tensor([1.3929e-06, 1.0000e+00])
```

Figura 6.56: Resultado experimento 2 caso E

Como puede observarse, la predicción en este caso es que la imagen es una soldadura mala, es decir, que la clasifica correctamente. Además, con bastante seguridad (más de un 99.99 %).

El resumen del experimento 2 ha sido el siguiente:

	Resultado	Tensor	Tiempo
Caso Base	Soldadura_Buena	[0.7318, 0.2682]	13 min 19 seg
Caso A	Soldadura_Mala	[1.2424e-05, 9.9999e-01]	71 min 58 seg
Caso B	Soldadura_Mala	[4.0837e-12, 1.0000e+00]	00 min 45 seg
Caso C	Soldadura_Mala	[2.5445e-06, 1.0000e+00]	50 min 29 seg
Caso D	Soldadura_Mala	[4.9648e-06, 9.9999e-01]	79 min 07 seg
Caso E	Soldadura_Mala	[1.3929e-06, 1.0000e+00]	105 min 15 seg

Figura 6.57: Resumen experimento 2

6.6. Conclusiones

El parámetro inicial que se iba a tener en cuenta para elegir el mejor caso, era el tensor medio en cada caso al clasificar la imagen, pero este parámetro es muy similar en todos los casos, y la diferencia entre ellos es despreciable.

Por esta razón, el siguiente parámetro que se iba a tener en cuenta, es la tasa de error, por si se diese el caso de que empeorase la tasa de error general de la red al introducir las imágenes nuevas, pero tampoco ha sido un parámetro a tener en cuenta, porque en ningún caso se ha visto mermado este parámetro significativamente.

Finalmente, el parámetro que se ha tenido en cuenta para la elección del mejor método para hacer que una red aprenda a clasificar una imagen que no sabe clasificar inicialmente, ha sido el tiempo.

El tiempo medio para la correcta clasificación de la imagen en el Caso B ha sido de 29,25 segundos, por lo que es el mejor con bastante diferencia.

El resto de tiempos medios se reflejan en el siguiente cuadro:

	Tiempo medio
Caso A	71 min 48 seg
Caso B	00 min 29 seg
Caso C	27 min 34 seg
Caso D	33 min 17 seg
Caso E	65 min 56 seg

Figura 6.58: Resumen tiempo medio

Capítulo 7

Conclusiones

A lo largo de la realización del proyecto, se han alcanzado todos los objetivos que se habían establecido inicialmente, y se ha seguido casi sin incidentes la planificación propuesta inicialmente.

En cuanto a los objetivos personales y del proyecto, se ha cumplido todo lo esperado:

- Se han adquirido conocimientos sobre deep learning y redes neuronales.
- Se han adquirido conocimientos sobre los frameworks "Fast.ai" y "Keras".
- Se ha comprobado que la utilización de redes neuronales, y en concreto de fastai, ha sido un acierto a la hora de clasificar un conjunto de imágenes de soldaduras en soldaduras con y sin defectos, con una tasa de error media de un 0,39 %.
- Se ha demostrado una forma rápida y eficiente de reentrenar una red que clasifica incorrectamente una imagen concreta, realizando pequeñas modificaciones en la imagen para que obtuviera más peso en la red (en total 64 imágenes similares a la original), y reentrenando la red tan solo con estas imágenes. Con un tiempo medio de 29,25 segundos se han obtenido resultados que clasifican correctamente imágenes que inicialmente se clasificaban incorrectamente con una seguridad de más de un 99,99 %.
- Se han adquirido conocimientos para la resolución de problemas utilizando diferentes frameworks en el ámbito industrial.

Bibliografía

[Nie] Michael Nielsen. Neural networks and deep learning. http://neuralnetworksanddeeplearning.com/.

[pro] proyectosagiles. Proyectos agiles scrum. https://proyectosagiles.org/que-es-scrum/.

[Syl] Jeremy y Alexis Gallaghe Sylvain. Manual fastai fastbook. https://github.com/fastai/fastbook.

[ven] venngage. Definición diagrama de gantt. https://es.venngage.com/blog/ejemplos-diagramas-gantt-plantillas/.

[Wika] Wikipedia. Anaconda. https://es.wikipedia.org/wiki/Anaconda_(distribuci%C3%B3n_de_Python).

[Wikb] Wikipedia. Michael nielsen. https://es.wikipedia.org/wiki/Michael_Nielsen.

[Wikc] Wikipedia. Perceptron. https://es.wikipedia.org/wiki/Perceptr%C3%B3n.

[Wikd] Wikipedia. Red neuronal artificial. https://es.wikipedia.org/wiki/Red_neuronal_artificial.

[Wike] Wikipedia. Sobreajuste. https://es.wikipedia.org/wiki/Sobreajuste.

[Wikf] Wikipedia. Soldadura. https://es.wikipedia.org/wiki/Soldadura.