Universidad de Valladolid

Escuela de Ingeniería Informática

Trabajo Fin De Grado

Grado en Ingeniería Informática
(Mención en Ingeniería de Software)

Deep Learning para Análisis de Forma en Fabricación Automática

Autor:
D. Óscar Fernando Ibáñez Garrido

Universidad de Valladolid

Escuela de Ingeniería Informática

Trabajo Fin De Grado

Grado en Ingeniería Informática
(Mención en Ingeniería de Software)

Deep Learning para Análisis de Forma en Fabricación Automática

Autor:
D. Óscar Fernando Ibáñez Garrido
Tutor :
D. Benjamín Sahelices Fernández

Agradecimientos

A aquellos que han estado conmigo durante la elaboración de este Trabajo de Fin de Grado, en especial a mi tutor Benjamín que me introdujo en el mundo del Deep Learning y al personal de la empresa WIP que me dio la oportunidad de aprender más sobre las posibilidades de un proyecto basado en Deep Learning.

Resumen

El presente Trabajo Fin de Grado es un proyecto teórico-práctico que trata de realizar una introducción a los conceptos asociados a la tecnología "Deep Learning" y posteriormente su aplicación a un entorno industrial destinado al clasificado de soldaduras para poder encontrar posibles defectos en ellas, es decir, su aplicación a un control de calidad en un ámbito industrial. Es por ello que primero se adquirirán unos conocimientos básicos acerca de las redes neuronales, después se hará un estudio comparativos de los principales frameworks que permiten su desarrollo a un alto nivel de abstracción y por último se utilizará uno de ellos, Fastai, para resolver el problema de clasificación mencionado.

Índice general

Capítulo 1

Introducción y objetivos

1.1. Motivación

El presente proyecto surge como resultado de resolver un problema real que ocurre en el control de calidad de soldaduras, en el que normalmente debe haber un operario que compruebe si dichas soldaduras poseen algún defecto o por el contrario son válidas.

Mediante el estudio de este problema se plantea una aproximación mediante redes neuronales que permita analizar las diferentes soldaduras de forma que el proceso pueda ser automatizado.

1.2. Objetivos

Los objetivos a tratar en este trabajo son:

- Estudiar y comprender el funcionamiento de una red neuronal, así como el de las redes neuronales convolucionales y obtener conocimientos acerca de distintos frameworks con los que trabajar para su realización.
- Conocer los distintos campos de aplicación de la tecnología deep learning.
- Aplicar lo aprendido sobre deep learning para la resolución de problemas acerca de clasificación en frameworks de alto nivel de abstracción.
- Aplicar los conocimientos adquiridos sobre Machine Learning para obtener resultados aceptables para el análisis de diferentes imágenes de soldaduras que permitan automatizar su análisis e integrarlo en un proceso de producción industrial.

1.3. Estructura del trabajo

El trabajo realizo presenta una parte teórica y una práctica. En la parte teórica se ha realizado el aprendizaje del funcionamiento de una red neuronal así como la investigación de diferentes frameworks para su realización. En la parte práctica se ha elegido uno de esos frameworks y se ha desarrollado diferente experimentación con el objetivo de analizar resultados y conseguir una red neuronal capaz de realizar un clasificado de imágenes aceptable. Los apartados del proyecto son los siguientes:

- **Capítulo 2: Planificación del proyecto**
  Presenta como se ha gestionado la temporalización de las actividades así como el uso y coste de recursos para abarcar el proyecto.

- **Capítulo 3: Introducción a las redes neuronales: conceptos básicos**
  Narra el funcionamiento de las redes neuronales, estructura y algoritmia que utilizan, así como una pequeña exploración de sus aplicaciones.

- **Capítulo 4: Análisis tecnológico**
  Enfrenta dos tecnologías de desarrollo de redes neuronales Keras y Fastai, con una conclusión final del porque de la utilización del segundo.

- **Capítulo 5: Introducción tecnológica de los clasificadores**
  Realiza una primera aproximación a la clasificación de imágenes mediante fastai permitiendo el análisis de resultados.

- **Capítulo 6: Control de calidad en procesos de fabricación**
  Se detalla el proceso de experimentación mediante clasificadores sobre un conjunto de datos de soldaduras así como la comparación de diferentes resultados.

- **Capítulo 7: Desarrollo de técnicas para la mejora dela exactitud**
  Capítulo de exploración de nuevas líneas de investigación para intentar mejorar los resultados iniciales detallados en el capítulo anterior.

- **Capítulo 8: Conclusiones**
  Se presentan las conclusiones a las que se ha llegado al realizar el trabajo fin de grado y posibles líneas de investigación futura.

Capítulo 2

Análisis del proyecto

2.1. Aplicación del proyecto en el ámbito industrial

El presente Trabajo de Fin de Grado se trata de un proyecto en el que se pretende hacer una aplicación de la tecnología Deep Learning, área de conocimiento en el que se encuentran las llamadas redes neuronales, a un entorno industrial, más concretamente su incorporación a un proceso de análisis de calidad, permitiendo que se automatice este proceso sin la presencia de una persona que lo supervise.

Para ello se realizará una fase de aprendizaje ya que en un principio se desconoce esta tecnología, por lo que habrá que adquirir conocimiento sobre el campo del Deep Learning dentro de la Inteligencia Artificial. Posteriormente se deberá conseguir un conjunto de datos real sobre el que poder aplicar dichos conocimientos, estos datos típicamente son tomados mediante una capturadora, en forma de nubes de puntos que deben ser procesados para poder ser usados como imágenes bidimensionales más fácil de analizar en una red neuronal. Además en el presente proyecto existe un paso previo adicional que es obtener muestras de un subconjunto de datos dentro del conjunto total de datos, ya que no se dispone de muestras de una categoría por lo que han de ser creadas sintéticamente mediante cambios en la imagen. Por último debemos experimentar con la tecnología con el objetivo de conseguir los mejores resultados intentando obtener una red que tenga el menor error a la hora de clasificar imágenes nuevas, esto lo haremos iterando sobre el conjunto de datos disponible.

2.2. Metodología

El presente TFG se compone dos partes, por una parte el estudio y aprendizaje del campo de la inteligencia artificial relacionado con el deep learning y las redes neuronales a través de tres libros [1] [2] [3] y por otro lado el desarrollo de redes neuronales que obtengan los mejores resultados posibles para un problema de clasificación en un entorno del tipo industrial para el que se siguió una metodología ágil.

2.3. Planificación

El desarrollo del proyecto se realiza desde finales del mes de octubre de 2020 (26/10/2020) hasta junio de 2021 (7/6/2020) y consta de las siguientes fases:

- **Estudio de redes neuronales**: Adquisición de conocimientos básicos sobre redes neuronales.
- **Estudio de frameworks para el desarrollo de redes neuronales**: Adquisición de conocimientos de frameworks para el desarrollo del proyecto.
- **Experimentación y evaluación de resultados del problema**: Realización de la experimentación que obtenga los mejores resultados posibles.

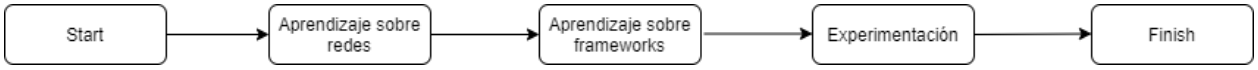

Figura 2.1: Diagrama Pert sobre la planificación del proyecto

Debido a esto los plazos seguidos en el desarrollo del proyecto han sido los siguientes:

- **Aprendizaje sobre redes**: Semanas 0-6 (26/10/2020-6/12/2020).
- **Aprendizaje sobre frameworks**: Semanas 7-13 (7/12/2020-24/1/2021).
- **Experimentación**: Semanas 14-32 (25/1/2021-7/6/2021)

2.4. Riesgos

El presente trabajo está expuesto a una serie de riesgos que se detallan a continuación bajo los siguientes criterios:

- **Descripción**: Detalla en que consiste el presente riesgo.
- **Probabilidad de ocurrencia**: Dividiremos este criterio en tres categorías: baja, media y alta, siendo baja la posibilidad de que ocurra el riesgo muy poca, y alta la posibilidad de que ocurra el riesgo casi segura.
- **Impacto**: Al igual que el caso anterior dividiremos este criterio en tres categorías: bajo, medio y alto, siendo bajo un riesgo sin mucho peligro, y alto un riesgo crítico.
- **Acciones de mitigación**: Se trata de medidas que se deben tomar para minimizar la ocurrencia del riesgo.
- **Acciones de contingencia**: Se trata de medidas que se deben tomar para que si el riesgo ocurre el daño ocasionado se minimice.

Riesgo 1 - Enfermedad	
Descripción	Debido a la crisis sanitaria a la cual estamos sometidos, existe la posibilidad de contraer la COVID-19 en cuyo caso no se podrá realizar el proyecto el tiempo que el alumno esté contagiado.
Probabilidad	Baja
Impacto	Medio
Acciones de mitigación	- Uso de mascarilla en la medida de lo posible.
Acciones de contingencia	- Replanificación del proyecto.

Cuadro 2.1: Riesgo de enfermedad

Riesgo 2 - Fallos en máquinas	
Descripción	Para la realización del proyecto se requieren de distintas máquinas, en el caso de que sufran una avería tendrá un impacto que dependerá del grado de gravedad de la misma.
Probabilidad	Baja
Impacto	Medio-Alto
Acciones de mitigación	- Realización de un correcto mantenimiento de la máquina.
Acciones de contingencia	- Reparación de la máquina. - Sustitución de la máquina por una nueva.

Cuadro 2.2: Riesgo de fallo en máquina

Riesgo 3 - Pérdida del trabajo elaborado	
Descripción	Debido a distintas circunstancias como el fallo en el hardware o software es posible que se pierdan los datos.
Probabilidad	Baja
Impacto	Alto
Acciones de mitigación	- Crear copias de respaldo cada cierto tiempo.
Acciones de contingencia	- Recuperación de copia de respaldo.

Cuadro 2.3: Riesgo de pérdida de trabajo

Riesgo 4 - Planificación no realista	
Descripción	Debido a una planificación inicial muy optimista no es posible llevar el desarrollo en los plazos previstos.
Probabilidad	Alta
Impacto	Bajo
Acciones de mitigación	- Construcción de un plan en el que se tengan en cuenta posibles retrasos.
Acciones de contingencia	- Posponer fecha de fin de realización del proyecto. - Aumentar ritmo de realización del proyecto.

Cuadro 2.4: Riesgo de planificación irreal

2.5. Presupuesto

Para la realización del proyecto el material requerido se puede dividir en tres grupos:

- **Hardware para el proceso de captura**: Debido al tamaño de la pieza y a la cantidad de soldaduras que posee, para la captura de las diferentes piezas ha sido necesario la disponibilidad de un brazo robótico, que permita repetir los movimientos para así tomar las capturas de las soldaduras siempre desde la misma posición.

Figura 2.2: Brazo robótico que permite realizar una trayectoria y repetirla.

Para realizar las capturas, se utilizará una cámara que permita la captura de puntos de forma que lo que obtenga sea nubes de puntos, es decir, las coordenadas de esos puntos en el espacio.

Figura 2.3: Cámara que permite la realización de capturas en nubes tridimensionales.

- **Hardware para la clasificación de imágenes**: Debido al funcionamiento de las redes neuronales en la que se utilizan cálculos tales como convoluciones, multiplicaciones de matriz y funciones de activación, el hardware toma un papel protagonista. Es por ello que la disposición de una GPU de cierta potencia, sea necesaria para el proyecto haciendo que los tiempos de ejecución del entrenamiento de la red no se extiendan demasiado.
- **Software**: Dentro del software nos encontramos por un lado con las librerías que tratan con matrices e imágenes como Pillow, Numpy y OpenCV, y por otro lado aquellas librerías destinadas al Deep Learning como son PyTorch y Tensorflow con sus respectivos frameworks fastai y Keras.

Una vez introducido los componentes principales para el proyecto los precios de los componentes utilizados han sido los siguientes:

- **Ordenador personal**: Necesario para el desarrollo del proyecto, el ordenador utilizado sin contar con la GPU tiene un coste aproximado de 800€.
- **GPU**: "EVGA GeForce RTX 3070 XC3 ULTRA GAMING 8GB GDDR6" 829.90€[4]
- **Brazo robótico**: Coste aproximado entre 50000$ (40970€) y 80000$ (65552€) [5]. Tomaremos como referencia la media entre ambos, aproximadamente 53267€.
- **Capturadora**: Coste aproximado 20000€.
- **Librerías de tratamiento de imágenes**: Las anteriores librerías mencionadas para el tratamiento de imágenes poseen licencia de software libre lo que significa que su precio es gratuito.
- **Librerías para Deep Learning**: Las anteriores librerías mencionadas para Deep Learning poseen licencia de software libre lo que significa que su precio es gratuito.

Como el proyecto es desarrollado por una sola persona, ya que se trata de un Trabajo de Fin de Grado, el precio que se debe pagar a los trabajadores debe ser consultado en el Convenio de Consultoría, que se encuentra en el Boletín Oficial del Estado, Número 57, Martes 6 de marzo de 2018 [6]. Donde se indica que le precio a pagar a un trabajador se encuentra entre los 9€y 10€la hora, utilizaremos como estimador la media entre ambos 9.5€.

También se debe tener en cuenta la duración del proyecto, en este caso al tratarse de un Trabajo de Fin de Grado de 12 ECTS las horas que se deben utilizar para su realización son 300.

Una vez obtenidos los datos el coste del proyecto realizado es el siguiente:

$$9{,}5 * 300 + 800 + 829 + 53267 + 20000 = 73267$$

Siendo el coste total del proyecto de 73267 €, mientras que el coste al que asciende a la parte del desarrollo de redes neuronales es de:

$$9{,}5 * 300 + 800 + 829 = 4479$$

Se trata de un coste inferior, 4479 €, ya que en el no intervienen el robot ni la capturadora de nubes de puntos.

Capítulo 3

Introducción a las redes neuronales: conceptos básicos

Para la realización de este capítulo del presente documento se ha utilizado como guía el libro online "Neural Networks and Deep Learning" [2]

3.1. Introducción a las redes neuronales

Comenzaremos nuestro aprendizaje sobre las redes neuronales respondiendo a la siguiente pregunta: **¿Qué es una red neuronal?** Es un paradigma de programación, una red neuronal aprende de los datos para brindar su propia solución. En 2006 se descubren las deep neural networks, tecnicas conocidas como el deep learning, logran un rendimiento sobresaliente en muchos problemas importantes en la visión por computadora, el reconocimiento de voz y el procesamiento del lenguaje natural. Utilizan ejemplos de entrenamiento que son una gran cantidad de datos que usan las redes para aprender.

Las redes neuronales abordan un problema de una manera diferente. Por ejemplo una idea es tomar una gran cantidad de datos, conocidos como ejemplos de entrenamiento y luego desarrollar un sistema que pueda aprender de esos ejemplos de entrenamiento. En otras palabras, la red neuronal usa los ejemplos para inferir automáticamente reglas para reconocer dígitos escritos a mano.

3.1.1. El perceptrón

El perceptrón es un tipo de neurona artificial. Un perceptrón toma varias entradas binarias x_1, x_2, ... y produce una única salida binaria:

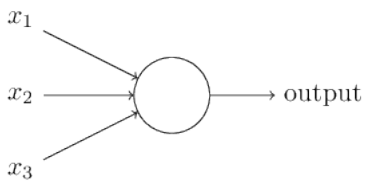

Figura 3.1: Neurona tipo perceptrón [2]

Rosenblatt propuso una regla simple para calcular la salida. Introdujo pesos, w_1, w_2, ... números

reales que expresan la importancia de las respectivas entradas a la salida. La salida de la neurona, 0 o 1, se determina por si la suma ponderada $\sum_j w_j x_j$ es menor o mayor que algún valor umbral. Al igual que los pesos, el umbral es un número real que es un parámetro de la neurona.

$$\text{salida} = \begin{cases} 0 & \text{si } \sum_j w_j x_j \leq \text{ umbral} \\ 1 & \text{si } \sum_j w_j x_j > \text{ umbral} \end{cases}$$

El perceptrón no es un modelo completo de toma de decisiones humanas. El ejemplo ilustra como el perceptrón puede sopesar diferentes tipos de evidencia para tomar decisiones. Y debería parecer plausible que una red compleja de perceptrones pueda tomar decisiones bastante sutiles.

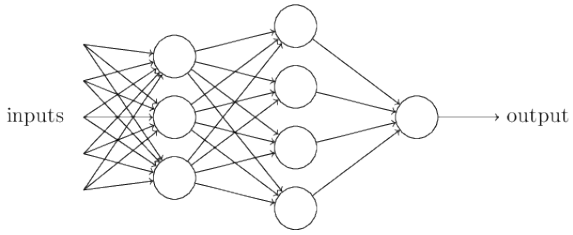

Figura 3.2: Red de perceptrones [2]

En esta red, la primera columna de perceptrones, lo que llamaremos la primera capa de perceptrones, está tomando tres decisiones, sopesando la evidencia de entrada. Los perceptrones de la segunda capa toman decisiones sopesando los resultados de la primera capa de toma de decisiones. De esta manera, un perceptrón de la segunda capa puede tomar una decisión a un nivel más complejo y abstracto que los perceptrones de la primera capa. Y el perceptrón de la tercera capa puede tomar decisiones aún más complejas. De esta manera, una red de perceptrones de muchas capas puede participar en una toma de decisiones sofisticada.

Llamaremos b al valor umbral que será el sesgo de percepción. De forma que:

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

Donde $w \cdot x \equiv \sum_j w_j x_j$

Otra forma que los perceptrones se pueden utilizar es calcular las funciones lógicas elementales, funciones tales como AND, OR, y NAND. Un perceptrón puede implementar una puerta NAND de la forma que únicamente hay que darle un peso –2 y un sesgo general 3, por ejemplo.

No debemos entender los perceptrones como simples puertas NAND. Debemos verlo de otro modo ya que resulta que podemos idear algoritmos de aprendizaje que puedan sintonizar automáticamente los pesos y sesgos de una red de neuronas artificiales. Esto ocurre en respuesta a estímulos externos, sin la intervención directa de un programador. Estos algoritmos de aprendizaje nos permiten utilizar neuronas artificiales de una manera radicalmente diferente a las puertas lógicas convencionales. En lugar de diseñar explícitamente un circuito NAND y otras puertas, nuestras redes neuronales pueden simplemente aprender a resolver problemas, a veces problemas en los que sería extremadamente difícil diseñar directamente un circuito convencional.

Neuronas sigmoideas

¿Como funcionan los algoritmos de aprendizaje? Supongamos que tenemos una red de perceptrones que nos gustaría usar para aprender a resolver algún problema, supongamos que hacemos

un pequeño cambio en algún peso (o sesgo) en la red, este pequeño cambio de peso debería provocar solo un pequeño cambio correspondiente en la salida de la red, entonces podríamos modificar los pesos y sesgos para que nuestra red se comporte más de la manera que queremos, cambiando los pesos y los sesgos una y otra vez para producir un resultado cada vez mejor, la red estaría aprendiendo. El problema es que esto no es lo que sucede cuando nuestra red contiene perceptrones. De hecho, un pequeño cambio en los pesos o el sesgo de cualquier perceptrón en la red a veces puede hacer que la salida de ese perceptrón cambie completamente, de 0 a 1. Ese cambio puede causar que el comportamiento del resto de la red cambie por completo de una manera muy complicada.

Podemos superar este problema introduciendo un nuevo tipo de neurona artificial llamada neurona sigmoide. Las neuronas sigmoides son similares a los perceptrones, pero modificadas de modo que pequeños cambios en sus pesos y sesgos provocan solo un pequeño cambio en su producción. Ese es el hecho crucial que permitirá que una red de neuronas sigmoides aprenda.

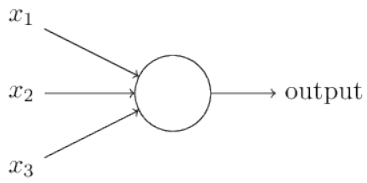

Figura 3.3: Neurona tipo sigmoide [2]

Al igual que un perceptrón, la neurona sigmoidea tiene entradas, $X_1, X_2, \ldots$. Pero en lugar de ser solo 0 o 1, estas entradas también pueden tomar cualquier valor entre 0 y 1. Entonces, por ejemplo, 0,638 ,es una entrada válida para una neurona sigmoidea. También al igual que un perceptrón, la neurona sigmoidea tiene pesos para cada entrada, w1, w2, ..., y un sesgo general, b. Pero la salida no es 0 o 1. En cambio, es $\sigma(w * x + b)$, dónde σ es lo que se llama función sigmoidea y está definido por:

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

A primera vista, las neuronas sigmoides parecen muy diferentes a los perceptrones. pero existen muchas similitudes entre los perceptrones y las neuronas sigmoides que se pueden observar al graficar su función.

La forma de σ es:

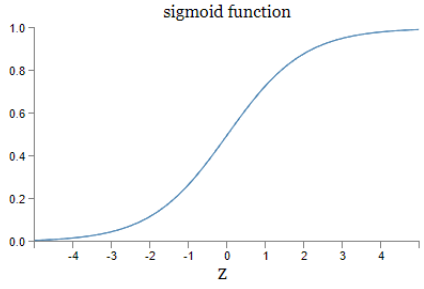

Figura 3.4: Gráfica función sigmoide [2]

Y la forma de la función de un perceptrón es:

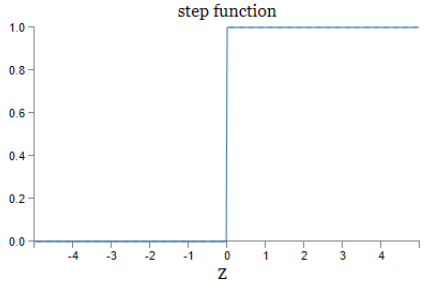

Figura 3.5: Gráfica función perceptrón [2]

¿Cómo deberíamos interpretar la salida de una neurona sigmoidea? Las neuronas sigmoides pueden tener como salida cualquier número real entre 0 y 1, por lo que se pueden obtener valores como 0,173 y 0,689. Esto puede resultar útil, por ejemplo, si queremos utilizar el valor de salida para representar la intensidad media de los píxeles en una imagen de entrada a una red neuronal. Pero otras no. Supongamos que queremos que la salida de la red indique la imagen de entrada es un 9 o la imagen de entrada no es un 9. Obviamente, sería más fácil hacer esto si el resultado fuera un 0 o un 1, como en un perceptrón. En la práctica podemos establecer una convención, por ejemplo, interpretar cualquier salida de al menos 0,5 como indicando un "9", y cualquier salida menor que 0,5 como indicando "no un 9".

La arquitectura de las redes neuronales

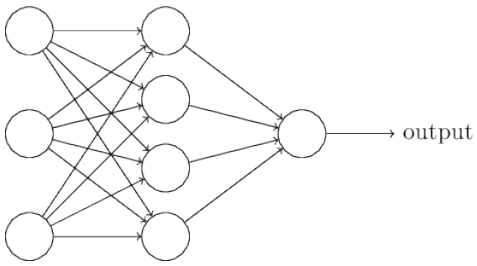

Figura 3.6: Arquitectura de red neuronal [2]

La capa más a la izquierda se denomina neuronas de entrada y la más a la derecha neuronas de salida, la capa intermedia se llama capa oculta.

Existen numerosas heurísticas de diseño para las capas ocultas, que ayudan a las personas a obtener el comportamiento que desean de sus redes, por ejemplo, determinar como el número de capas ocultas afecta al tiempo necesario para entrenar la red.

Es importante no permitir bucles sino trabajar con las denominadas redes neuronales de retroalimentación donde la salida de una capa se usa para la siguiente. Pero estos bucles si se permiten en algunos modelos denominados redes neuronales recurrentes, la idea es tener neuronas que se activen durante un tiempo limitado, antes de volverse inactivas, este disparo puede estimular otras neuronas, que se disparan más tarde. Los bucles no causan problemas en un modelo de este tipo, ya que la salida de una neurona solo afecta su entrada en algún momento posterior, no instantáneamente.

Aprendizaje con gradiente descendente

Lo primero que se necesita es un conjunto de datos de entrenamiento, los primeros datos se usan para entrenar y los restantes serán datos de prueba. Se usará la notación X para denotar los datos entrada, es conveniente considerar cada entrada X como un vector dimensional n x n. Denotaremos la salida deseada correspondiente por $y = y(x)$, dónde y es un vector dimensional.

Lo que se necesita es encontrar un algoritmo que nos permita encontrar pesos y sesgos para que la red se aproxime al resultado deseado $y(x)$ para todas las entradas de X. Para cuantificar si vamos por el buen camino se define una función de costos:

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

Donde w es la colección de todos los pesos de la red, b son los sesgos, n es el número de las entradas de entrenamiento, a es el vector de salidas de la red que depende de la salida. La notación $\|v\|$ indica el tamaño habitual de el vector v. Llamaremos a C función de costo, también se la conoce como error cuadrático medio o MSE. Se observa que $C(w, b)$ no es negativo, además la función se acerca a 0 cuando $y(x)$ se aproxima a la salida a, para todas las entradas de x. Significa que cuanto más pequeño sea $C(w, b)$ mejor será el trabajo hecho. Debemos buscar el conjunto de pesos y sesgos que hagan el coste lo menor posible haremos esto usando el algoritmo del gradiente descendiente.

¿Por qué se introduce el coste cuadrático y no se maximiza otro criterio? Porque se necesita una función uniforme de los pesos y sesgos en la red. Si usamos la cuadrática resulta fácil averiguar cómo hacer pequeños cambios para obtener mejora en el costo. Primero intentamos minimizar el costo y después examinaremos la precisión.

La meta es hallar pesos y sesgos que minimicen C. Vamos a imaginar que se nos ha dado una función de muchas variables y la queremos minimizar, en este caso una de dos variables $C(v)$ con v_1 y v_2.n El objetivo será hallar el mínimo global. Una forma de solucionar el problema sería haciendo el cálculo mediante derivadas, pero sería una pesadilla con muchas variables. Entonces empecemos pensando en nuestra función como una especie de valle, e imaginamos una pelota rodando hasta el fondo, utilizaremos un punto de partida aleatorio para la bola y luego simularíamos su movimiento mientras rueda hasta el fondo del valle. Podemos haces esta simulación calculando un par de derivadas que nos dirán todo sobre la forma del valle y como rodará la bola. Para hacer más preciso la pregunta preguntémonos que pasa cuando movemos v_1 y v_2 una pequeña cantidad Δv_1 y Δv_2 respectivamente. El cálculo nos dice que C cambiará así:

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

Ahora vamos a encontrar un valor para Δv_1 y Δv_2 para hacer ΔC negativo, haremos esto para que la bola ruede hacia abajo en la colina. Para hacernos a la idea debemos definir Δv que es el vector de cambios en v, $\Delta v = (\Delta v1, \Delta v2)^T$, donde T indica la trasposición, cambiando las filas por columnas. También definimos el gradiente de C como el vector de derivadas parciales:

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

De esta expresión sacamos que $\Delta C \approx \nabla C \cdot \Delta v$ y de aquí podemos observar como elegir Δv para hacer ΔC negativo:

$$\Delta v = -\eta \nabla C$$

Donde η es un número pequeño y positivo conocido como el ratio de aprendizaje. De aquí también sacaremos el valor para Δv, ahora moveremos la posición de la bola esa cantidad mediante la regla de actualización:

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

Usaremos esta regla, para seguir haciendo movimientos una y otra vez decrementando el gradiente de C hasta que alcancemos un mínimo global. Resumiendo, la forma en que funciona el algoritmo de descenso de gradiente es calcular repetidamente el gradiente, y luego moverse en la dirección opuesta, çayendo"por la pendiente del valle. Podemos visualizarlo así:

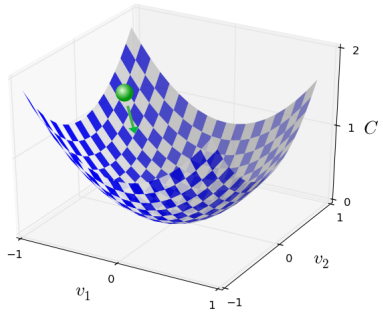

Figura 3.7: Explicación visual del gradiente descendente [2]

Para que esto funcione se debe elegir una tasa de aprendizaje lo suficientemente pequeña como para que la ecuación anterior sea lo suficientemente buena, pero no debe ser muy pequeña porque si no hará que el algoritmo de descenso de gradiente funcione muy lentamente.

La idea es usar el algoritmo para encontrar los pesos y los sesgos que minimizan el costo de la ecuación de costes C, para ello reemplazaremos los valores v por valores w y b y el gradiente de C en sus correspondientes componentes, reescribiéndolo obtenemos:

$$w_k \to w'_k = w_k - \eta \frac{\partial C}{\partial w_k}$$

$$b_l \to b'_l = b_l - \eta \frac{\partial C}{\partial b_l}$$

Aplicando repetidamente esta regla podremos encontrar el mínimo de la función, es decir, es una regla que sirve para que aprenda la red neuronal. En la práctica hay que realizar el sumatorio de C_x y luego dividirlo entre n para hacer la media, esto provoca que la red aprenda despacio. Existe una idea llamada el gradiente estocástico que puede usarse para acelerar el entrenamiento, la idea es estimar el gradiente de C mediante el gradiente de C_x en una muestra pequeña y aleatoria elegida por las entradas de entrenamiento. Con esto podemos obtener un buen y rápido estimador del gradiente de C.

Precisando la idea el gradiente estocástico funciona eligiendo aleatoriamente un pequeño número de m entradas de entrenamiento, las pondremos la etiqueta $X_1, X_2, ...X_m$ y nos referiremos a ellas como mini-batch. Con ello obtendremos:

$$\nabla C \approx \frac{1}{m} \sum_{j=1}^{m} \nabla C_{X_j}$$

Esto cambiara nuestra regla de actualización a:

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

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

Se irán eligiendo mini-batchs y entrenando con ellos hasta que agotemos las entradas de entrenamiento, a esto lo llamaremos época de entrenamiento. Cuando agotemos un mini-batch, empezaremos con una nueva época de entrenamiento.

Podemos pensar en el gradiente estocástico como una encuesta política es mejor tener una pequeña muestra que hacerla a toda la población. Por ejemplo con un n=60000 y un m=10 conseguiremos un factor de speedup de 6000 en obtener el gradiente.

Debemos tener en cuenta que un algoritmo sofisticado será en la mayoría de casos peor que un algoritmo de aprendizaje simple con unos buenos datos de entrenamiento.

Hacia el aprendizaje profundo

La heurística sugiere que, si podemos resolver los subproblemas usando redes neuronales, entonces quizás podamos construir una red neuronal, combinando las redes para los subproblemas, y no solo eso esas preguntas pueden también desglosarse a su vez en otros subsubproblemas y así sucesivamente.

3.1.2. Algoritmo de propagación hacia atrás

Enfoque rápido basado en matrices para calcular la salida de una red neuronal

Definición de conceptos: Peso w^l_{jk} donde k indica la capa destino j la neurona de salida y k el número de neurona de la capa

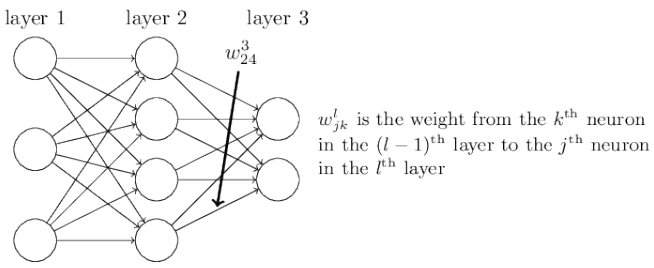

Figura 3.8: Arquitectura de red de tres capas [2]

Usamos una notación similar para los sesgos y activaciones de la red, usaremos b^l_j para el sesgo de la neurona j de la capa l y a^l_j para la activación de la neurona j de la capa l:

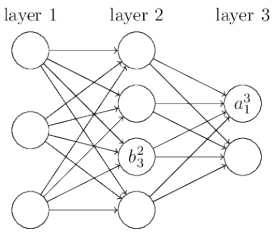

Figura 3.9: Arquitectura de red de tres capas notación [2]

Con esta notación denotaremos a la activación a_j^l con la siguiente ecuación:

$$a_j^l = \sigma\left(\sum_k w_{jk}^l a_k^{l-1} + b_j^l\right)$$

Definimos la matriz de pesos w^l para cada capa l. Las entradas para la matriz de pesos son los pesos que conectan con la capa l, que son la entrada de la fila j y la columna k, w_{jk}^l. Definiremos el vector de sesgos b^l y el vector de activaciones a^l. Con esta notación reescribimos la ecuación anterior como:

$$a^l = \sigma(w^l a^{l-1} + b^l)$$

Las dos suposiciones necesarias sobre la función de costes

La meta de la propagación hacia atrás es computar las derivadas parciales $\partial C/\partial w$,$\partial C/\partial b$ y el coste de C con respecto cualquier peso w o sesgo b. Para el algoritmo de propagación hacia atrás necesitaremos dos suposiciones principales más.

Primera: escribir la función de costes como una media $C = \frac{1}{2n}\sum_x \|y(x) - a^L(x)\|^2$ en vez de funciones de coste individuales para cada entrenamiento. Esto se aplicará para todas las funciones de costo posteriores. Se usa para cuando un ejemplo de entrenamiento x sale mal con la media puede recuperarse.

La segunda es que se puede escribir el costo en función de las salidas de la red neuronal. La función de costes cuadrática satisface este requisito: $C = \frac{1}{2}\|y - a^L\|^2 = \frac{1}{2}\sum_j (y_j - a_j^L)^2$.

Producto de Hadamart

El algoritmo de propagación hacia atrás se basa en operaciones algebraicas lineales comunes, como la suma de vectores, la multiplicación de un vector por una matriz, etc.Pero una de las operaciones se utiliza con más frecuencia.En particular, suponga syt son dos vectores de la misma dimensión.Entonces usamos s⊙t para denotar el producto elemental de vectores como en el ejemplo:

$$\begin{bmatrix} 1 \\ 2 \end{bmatrix} \odot \begin{bmatrix} 3 \\ 4 \end{bmatrix} = \begin{bmatrix} 1*3 \\ 2*4 \end{bmatrix} = \begin{bmatrix} 3 \\ 8 \end{bmatrix}$$

A esto se denomina producto de Hadamart, implementado en muchas librerías.

Las cuatro ecuaciones fundamentales detrás de la propagación hacia atrás

La propagación hacia atrás va sobre comprender como cambios en los pesos y en los sesgos en una red provoca cambios en los costes de la función. Pero para comprender esto debemos primero introducir el error, δ_j^l, que será el error de la neurona j de la capa l. Para entender como se define el error podemos imaginar que hay un demonio en la red neuronal que se encuentra en la neurona j de la capa l, en el momento que entra algo a la neurona el demonio altera las operaciones de la neurona. Esto lo hace añadiendo un pequeño cambio Δz_j^l al peso de entrada de la neurona haciendo que la salida sea $\sigma(z_j^l + \Delta z_j^l)$, este error se va propagando causando que el coste total cambie la cantidad $\frac{\partial C}{\partial z_j^l}\Delta z_j^l$. El objetivo del demonio es conseguir que el coste sea 0 con lo que intentara ajustar Δz_j^l para obtener dicho resultado.

Definiremos el error δ_j^l como

$$\delta_j^l \equiv \frac{\partial C}{\partial z_j^l}$$

Y usaremos δ^l para denotar el vector de errores de la capa l.

El algoritmo de propagación hacia atrás se basa en cuatro ecuaciones, que combinadas nos proporcionan el error de la capa y el gradiente de costes de la función. Estas son:

- Ecuación del error en la salida de la capa BP1
- Ecuación del error en términos de la capa siguiente BP2
- Ecuación para la tasa de cambio en el coste con respecto a cualquier sesgo en la red BP3
- Ecuación para la tasa de cambio en el coste con respecto a cualquier peso en la red BP4

Cuyas ecuaciones son las siguiente:

$$\delta^L = \nabla_a C \odot \sigma'(z^L) \qquad \text{(BP1)}$$

$$\delta^l = ((w^{l+1})^T \delta^{l+1}) \odot \sigma'(z^l) \qquad \text{(BP2)}$$

$$\frac{\partial C}{\partial b_j^l} = \delta_j^l \qquad \text{(BP3)}$$

$$\frac{\partial C}{\partial w_{j\,k}^l} = a_k^{l-1}\delta_j^l \qquad \text{(BP4)}$$

Figura 3.10: Tabla de ecuaciones de propagación hacia atrás [2]

El algoritmo de propagación hacia atrás

El algoritmo es el siguiente:

1. Entrada x: Estable la correspondiente activación a la entrada de la primera capa
2. Pre alimentación: Para cada capa l= 2,3,...,L hacer $z^l = w^l a^{l-1} + b^l$ y $a^l = \sigma(z^l)$
3. Calcular el error: $\delta^L = \nabla_a C \odot \sigma'(z^L)$
4. Propagar hacia atrás el error: Para cada capa l= L-1,L-2,...,2 calcular $\delta^l = ((w^{l+1})^T \delta^{l+1}) \odot \sigma'(z^l)$

5. Salida mediante las funciones de coste: $\frac{\partial C}{\partial w^l_{jk}} = a^{l-1}_k \delta^l_j$ y $\frac{\partial C}{\partial b^l_j} = \delta^l_j$

Si lo combinamos con algoritmos de aprendizaje como el gradiente descendente estocástico con una cantidad m de mini-batch como ejemplos de entrenamientos obtenemos el algoritmo modificado de la siguiente forma:

1. Entrada: Conjunto de ejemplos de entrenamiento

2. Para cada ejemplo de entrenamiento x: Establecer la correspondiente activación $a^{x,1}$ y hacer:

   - Pre alimentación: Para cada capa l= 2,3,...,L hacer $z^{x,l} = w^l a^{x,l-1} + b^l$ y $a^{x,l} = \sigma(z^{x,l})$
   - Calcular el error: $\delta^{x,L} = \nabla_a C_x \odot \sigma'(z^{x,L})$
   - Propagar hacia atrás el error: Para cada capa l= L-1,L-2,...,2 calcular $\delta^{x,l} = ((w^{l+1})^T \delta^{x,l+1}) \odot \sigma'(z^{x,l})$

3. Gradiente descendiente: Para cada l= L-1,L-2,...,2 actualizar los pesos acorde con la regla $w^l \to w^l - \frac{\eta}{m} \sum_x \delta^{x,l} (a^{x,l-1})^T$, y los sesgos acorde con la regla $b^l \to b^l - \frac{\eta}{m} \sum_x \delta^{x,l}$

Por supuesto es necesario un bucle exterior que genere los mini-batchs de entrenamiento y otro que de pasos a través de las múltiples épocas de entrenamiento.

Se trata de un algoritmo rápido porque permite calcular simultáneamente todas las derivadas parciales $\partial C/\partial w_j$ usando un solo paso hacia delante a través de la red, seguido de un paso atrás.

Propagación hacia atrás la gran imagen

El algoritmo presenta un misterio: ¿Qué es lo que realmente hace el algoritmo?

Para responder a la cuestión vamos a imaginar que tenemos un pequeño cambio Δw^l_{jk} a algún peso en la red, w^l_{jk}, Este cambio en el peso causará un cambio en la salida de activación de la neurona correspondiente, después, provocará un cambio en todas las activaciones de la neurona de la siguiente capa. Estos cambios provocarán cambios en la siguiente capa, y así en las sucesivas causando cambios en la capa final y así en la función de costes.

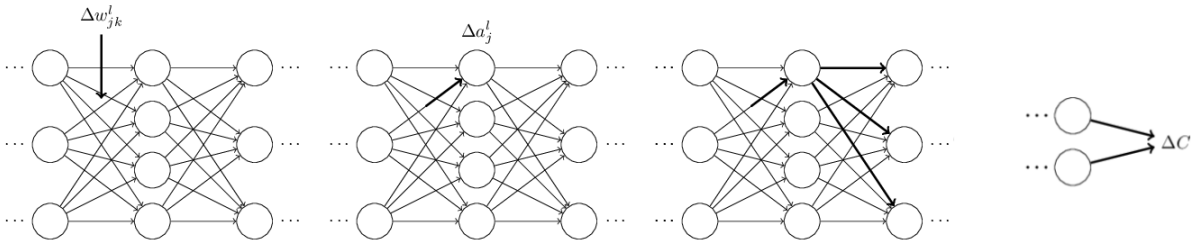

Figura 3.11: Primer paso y sucesivos de la propagación hacia atrás [2]

3.1.3. Mejorando la forma en que las neuronas aprenden

La función de coste de entropía cruzada

Si comparamos el aprendizaje humano con el de una red neuronal utilizando el gradiente descendiente nos damos cuenta que cuando una red neurona está equivocada por mucho en algo tarda mucho en darse cuenta del error, cosa que no ocurre en las personas. Para comprenderlo, debemos

entender que esto ocurre si las derivadas parciales $\partial C/\partial w$ y $\partial C/\partial b$ son pequeñas, para entender esto recordamos que la ecuación de coste cuadrática es $C = \frac{(y-a)^2}{2}$, para escribir esto de forma más explicita en términos de ponderación y sesgo, recordamos que $a = \sigma(z)$ donde $z = wx + b$ aplicando la regla de la cadena para diferenciar al peso y al sesgo obtenemos:

$$\frac{\partial C}{\partial w} = (a - y)\sigma'(z)x = a\sigma'(z)$$

$$\frac{\partial C}{\partial b} = (a - y)\sigma'(z) = a\sigma'(z)$$

Antes de seguir recordemos conviene la forma de la funcion sigmoide 3.1.1.Las ecuaciones anteriores nos dicen que las derivadas parciales son muy pequeñas cuando la salida de la neurona se acerca a 1, ese es el origen del problema. Entonces, ¿cómo podemos abordar la ralentización del aprendizaje? Reemplazando la función de coste por la ecuación de entropía cruzada. Definiremos esta función como:

$$C = -\frac{1}{n}\sum_x [y \ln a + (1 - y)\ln(1 - a)]$$

Donde n es el número total de entradas de entrenamiento, x es el numero de entrada, y y es la salida deseada.

Dos propiedades en particular hacen que sea razonable interpretar la entropía cruzada como una función de costes. Primero, no da un resultado negativo, es decir, C¿0. La otra propiedad, es que si la salida real de la neurona está cerca de la salida deseada para todas las entradas de entrenamiento, X, entonces la entropía cruzada será cercana a 0. Es decir la entropía cruzada es positiva y tiende a cero a medida que la neurona mejora el cálculo de la salida deseada para todas las entradas de entrenamiento. Esto ya lo cumplía la función de coste cuadrática pero a diferencia de esta, la de entropía cruzada tiene el beneficio que evita la desaceleración de aprendizaje.

Deberemos usar esta función de coste siempre que las neuronas sean neuronas sigmoideas. Esta función de costes es una medida de sorpresa, en realidad podríamos decir que mide cuán sorprendidos estamos, en promedio cuando conocemos el verdadero valor de y. Tenemos una sorpresa baja si la salida es la que esperamos y una sorpresa alta si la salida es inesperada.

Softmax

La idea es definir un nuevo tipo de capa de salida para nuestras redes neuronales. Comienza de la misma manera que con una capa sigmoidea, formando las entradas ponderadas, $z_j^L = \sum_k w_{jk}^L a_k^{L-1} + b_j^L$. Sin embargo no se aplica la función sigmoidea para obtener el resultado, en cambio, en una capa softmax aplicamos la función softmax. Según esta función la activación de la neurona es:

$$a_j^L = \frac{e^{z_j^L}}{\sum_k e^{z_k^L}}$$

Esta función garantiza que las activaciones de salida siempre sumen 1 y sean siempre positivas ya que la función exponencial es positiva, en otras palabras, la salida de la capa softmax es una distribución de probabilidad. Esto tiene implicaciones como que sen muchos problemas es conveniente interpretas la activación de salida a_j^L como la estimación de la red como la probabilidad de

que la salida correcta sea j. Esta solución también permite abordar el problema de la ralentización del aprendizaje. Para ello definimos la función de coste logarítmica de probabilidad:

$$C \equiv -\ln a_y^L$$

Calculando en función de las derivadas parciales y poniéndolas en función del peso o del sesgo obtenemos las mismas ecuaciones que en la entropía cruzada, y esto nos asegura que no hay una ralentización del aprendizaje. Esto se suele utilizar para problemas de clasificación.

Sobreajuste y regularización

Cuando una red deja de aprender después de una determinada época diremos que la red está sobreajustada o sobreentrenada.Esto se puede ver en la precisión en los datos de prueba donde llegará un momento que, con picos, se mantienen estables, en el coste de los datos de prueba que no paran de aumentar, o en la precisión en los datos de entrenamiento, que puede llegar a un 100 % y mantenerse estable ahí mientras que en los datos que se pretenden clasificar no supera el 90 % u 80 %. El sobreajuste es un problema importante en las redes neuronales, por lo que es necesaria una forma de detectar cuando se esta produciendo un sobreajuste, para no entrenar demasiado. Una de ellas es la **parada temprana** en la que en una entrada le digamos cual debe ser la tasa de precisión de clasificación, aunque en la práctica no sabremos de inmediato cuando ocurre esto. Otra de las formas sería **aumentar el tamaño de los datos de entrenamiento**. Con suficientes datos de entrenamiento, es difícil sobreajustar incluso una red muy grande. Desafortunadamente, los datos de entrenamiento pueden ser costosos o difíciles de adquirir, por lo que esta no siempre es una opción práctica.

Existen otras técnicas que pueden reducir el sobreajuste, incluso cuando tenemos una red fija y datos de entrenamiento fijos. Estas se conocen como **técnicas de regularización**. Describiremos la técnica conocida como d**isminución de peso o regularización L2**, la idea es agregar un término adicional a la función de costo, un término llamado término de regularización. Aplicándolo a la entropía cruzada obtendríamos la entropía cruzada regularizada:

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

Aquí lo que se ha hecho es agregar un segundo termino que es la suma de los cuadrados de toda la red escalado por un factor $\lambda/2n$ donde $\lambda > 0$, a este término se le conoce como el parámetro regularizador. Esto se puede reescribir como:

$$C = C_0 + \frac{\lambda}{2n}\sum_w w^2$$

donde C_0 es la función de costes original, no regularizada. El efecto de la regularización es hacer que la red prefiera aprender pesos pequeños, en igualdad de condiciones. Solo se permitirán grandes pesos si mejoran considerablemente la primera parte de la función de costes. Dicho de otra forma, la regularización puede verse como una forma de comprometer entre encontrar pesos pequeños y minimizar la función de costo original. La importancia relativa de los dos elementos del compromiso depende del valor de λ: cuando λ es pequeño, preferimos minimizar la función de costo original, pero cuando λ es grande, preferimos pesos pequeños. Pongamos un ejemplo de como la regularización reduce el sobreajuste.

Para ello aplicaremos el gradiente estocástico descendiente en una red neuronal regularizada, en concreto debemos saber como calcular las derivadas parciales $\partial C/\partial w$ y $\partial C/\partial b$ para todos los pesos y sesgos de la red usando las ecuaciones anteriores, los términos $\partial C_0/\partial w$ y $\partial C_0/\partial b$ se pueden calcular aplicando el algoritmo de propagación hacia atrás. Las derivadas parciales con respecto a los sesgos no cambian:

$$b \to b - \eta \frac{\partial C_0}{\partial b}$$

Aunque si la regla de aprendizaje para los pesos se convierte en:

$$w \to w - \eta \frac{\partial C_0}{\partial w} - \frac{\eta\lambda}{n} w = \left(1 - \frac{\eta\lambda}{n}\right) w - \eta \frac{\partial C_0}{\partial w}.$$

Esto sería para el gradiente descendiente, para el gradiente descendiente estocástico obtendríamos:

$$w \to \left(1 - \frac{\eta\lambda}{n}\right) w - \frac{\eta}{m} \sum_x \frac{\partial C_x}{\partial w}$$

$$b \to b - \frac{\eta}{m} \sum_x \frac{\partial C_x}{\partial b}$$

Una red no regularizada puede utilizar grandes pesos para aprender un modelo complejo que transporta mucha información sobre el ruido en los datos de entrenamiento. En pocas palabras, las redes regularizadas están limitadas a construir modelos relativamente simples basadosen patrones que se ven a menudo en los datos de entrenamiento y son resistentes a las peculiaridades de aprendizaje del ruido en los datos de entrenamiento. La esperanza es que esto obligue a nuestras redes a hacer un aprendizaje real sobre el fenómeno en cuestión y a generalizar mejor a partir de lo que aprenden.

¿Por qué la regularización L2 no regulariza sesgos? tener un sesgo grande no hace que una neurona sea sensible a sus entradas de la misma manera que tiene grandes pesos. Por lo tanto, no tenemos que preocuparnos por grandes sesgos que permitan a nuestra red conocer el ruido en nuestros datos de entrenamiento. Al mismo tiempo, permitir grandes sesgos da a nuestras redes más flexibilidad de comportamiento; en particular, los grandes sesgos facilitan la saturación de las neuronas, lo que a veces es deseable.

Otras técnicas de regularización:

1. **Regularización L1**: modificamos la función de costo no regularizado sumando la suma de los valores absolutos de los pesos, similar a la regularización L2, ambos tipos de regularización penalizan a los grandes pesos. Pero la forma en que se encogen los pesos es diferente. En la regularización L1, los pesos se reducen en una cantidad constante hacia 0. En la regularización L2, los pesos se reducen en una cantidad que es proporcional a w. Entonces, cuando un peso particular tiene una gran magnitud, $|w|$, La regularización L1 reduce el peso mucho menos que la regularización L2. El resultado neto es que la regularización L1 tiende a concentrar el peso de la red en un número relativamente pequeño de conexiones de alta importancia, mientras que los otros pesos se dirigen hacia cero.

2. **Abandono**: no se basa en modificar la función de costos. En cambio, en el abandono modificamos la propia red. Comenzamos eliminando aleatoriamente (y temporalmente) la mitad

de las neuronas ocultas en la red, mientras dejamos intactas las neuronas de entrada y salida. Reenviamos y propagamos la entrada x a través de la red modificada, y propagamos el resultado hacia atrás. Después de hacer esto sobre mini-batches actualizamos los pesos y sesgos. Luego repetimos el proceso, primero restaurando las neuronas abandonadas, luego eligiendo un nuevo subconjunto aleatorio de neuronas ocultas para eliminar, estimando el gradiente para un mini-batch diferente y actualizamos los pesos y sesgos en la red.Al repetir este proceso una y otra vez, nuestra red aprenderá un conjunto de pesos y sesgos. Por supuesto, esos pesos y sesgos se habrán aprendido en condiciones en las que se eliminó la mitad de las neuronas ocultas. Cuando realmente ejecutamos la red completa, eso significa que el doble de neuronas ocultas estarán activas. Para compensar eso, reducimos a la mitad los pesos que salen de las neuronas ocultas. Heurísticamente, cuando abandonamos diferentes conjuntos de neuronas, es como si estuviéramos entrenando diferentes redes neuronales. Entonces, el procedimiento de abandono es como promediar los efectos de una gran cantidad de redes diferentes. Las diferentes redes se sobreajustarán de diferentes maneras, por lo que, con suerte, el efecto neto de la deserción será reducir el sobreajuste. Este método ha sido especialmente útil en el entrenamiento de redes grandes y profundas, donde el problema del sobreajuste suele ser agudo.

3. **Expansión artificial de los datos de entrenamiento**:la precisión de la clasificación mejora considerablemente a medida que usamos más datos de entrenamiento. Es de suponer que esta mejora continuaría aún más si se dispusiera de más datos. Obtener más datos de entrenamiento es una gran idea. Desafortunadamente, puede resultar caro, por lo que no siempre es posible en la práctica. Sin embargo, hay otra idea que puede funcionar casi igual de bien: expandir artificialmente los datos de entrenamiento. Podemos expandir nuestros datos de entrenamiento haciendo muchas rotaciones pequeñas de todas las imágenes de entrenamiento y luego usando los datos de entrenamiento expandidos para mejorar el rendimiento de nuestra red.Esta idea es muy poderosa y ha sido ampliamente utilizada. El principio general es expandir los datos de entrenamiento aplicando operaciones que reflejan la variación del mundo real.

4. **Un aparte sobre big data y lo que significa comparar las precisiones de clasificación** más datos de entrenamiento a veces pueden compensar las diferencias en el algoritmo de aprendizaje automático utilizado. La respuesta correcta a la pregunta "¿Es el algoritmo A mejor que el algoritmo B?.[es] realmente: "¿Qué conjunto de datos de entrenamiento estás usando?"

Inicialización del peso

Cuando creamos nuestras redes neuronales, tenemos que elegir los pesos y sesgos iniciales. Hasta ahora, se elegían tanto los pesos como los sesgos utilizando variables aleatorias gaussianas independientes, normalizadas para tener una media 0 y desviación estándar 1. Resulta que podemos hacerlo bastante mejor que inicializar con gaussianos normalizados. Con estos datos es probable que $\|z\|$ sea bastante grande lo que implica $\sigma(z)$ de una neurona oculta este muy cerca de 1 o 0, eso significa que nuestra neurona oculta se habrá saturado y cuando eso suceda, hacer pequeños cambios en los pesos solo producirá cambios absolutamente minúsculos en la activación de nuestra

neurona oculta. Como resultado, esos pesos solo aprenderán muy lentamente cuando usemos el algoritmo de descenso de gradiente. Luego, inicializaremos esos pesos como variables aleatorias gaussianas con media0 y desviación estándar $\frac{1}{\sqrt{n_{in}}}$ Es decir, aplastaremos a la gaussiana, haciendo menos probable que nuestra neurona se sature.Podemos ver la comparativa en las siguientes gráficas

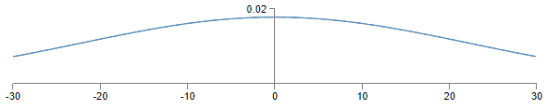

Figura 3.12: Distribución gaussiana de media 0 y desviación estandar 1 [2]

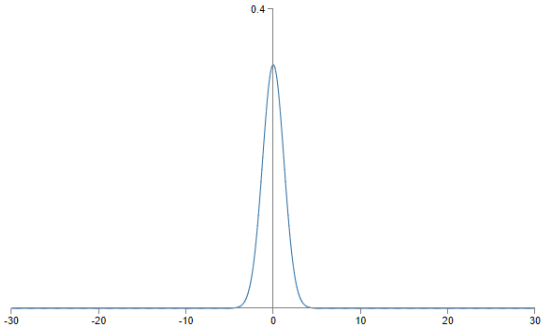

Figura 3.13: Distribución gaussiana de media 0 y desviación estandar 1 $\frac{1}{\sqrt{n_{in}}}$ [2]

Continuaremos inicializando los sesgos como antes, como variables aleatorias gaussianas con una media de 0 y una desviación estándar de 1. Esto está bien, porque no aumenta la probabilidad de que nuestras neuronas se saturen. De hecho, no importa mucho cómo inicializamos los sesgos, siempre que evitemos el problema de la saturación.

¿Cómo elegir los hiperparámetros de una red neuronal?

Cuando se utilizan redes neuronales para atacar un nuevo problema, el primer desafío es conseguir cualquier aprendizaje no trivial, es decir, que la red logre resultados mejores que el azar. Esto puede resultar sorprendentemente difícil, especialmente cuando se enfrenta a una nueva clase de problema. Estrategias:

- Estrategia amplia: Consiste en reducir los ejemplos de prueba por ejemplo entrenar una red que reconozca dígitos a entrenar una red que reconozca el 1 y el 0 con eso simplificamos la red, disminuimos el numero de neuronas de ocultas... Después monitorizaremos la exactitud en la evaluación Y así podemos continuar, ajustando individualmente cada hiperparámetro, mejorando gradualmente el rendimiento. Una vez que hayamos explorado para encontrar un valor mejorado para η, luego avanzamos para encontrar un buen valor para λ. Luego experimente con una arquitectura más compleja y luego reajuste los valores... Y así sucesivamente, en cada etapa evaluando el desempeño usando nuestros datos de validación retenidos y usando esas evaluaciones para encontrar mejores y mejores hiperparámetros.

- Tasa de aprendizaje: Puede estimar el orden de magnitud comenzando con η=0.01. Si el costo disminuye durante las primeras épocas, debe intentar sucesivamenteη=0.1,1.0,... hasta que encuentre un valor para ηdonde el costo oscila o aumenta durante las primeras épocas. Alternativamente, si el costo oscila o aumenta durante las primeras épocas cuandoη=0.01, entonces intenta η=0.001,0.0001,... hasta que encuentre un valor para ηdonde el costo disminuye durante las primeras épocas. Seguir este procedimiento nos dará una estimación de orden de magnitud para el valor umbral de η. Opcionalmente, puede refinar su estimación para seleccionar el mayor valor de η en el que el costo disminuye durante las primeras épocas, digamos η=0.5 o η=0.2(no es necesario que esto sea superpreciso). Esto nos da una estimación del valor umbral de η.

- Utilizar la parada temprana para determinar el número de épocas de entrenamiento: al final de cada época debemos calcular la precisión de la clasificación en los datos de validación. Cuando eso deje de mejorar, termine. Esto hace que configurar el número de épocas sea muy sencillo. En particular, significa que no tenemos que preocuparnos por averiguar explícitamente cómo el número de épocas depende de los otros hiperparámetros. Para implementar la detención anticipada, necesitamos decir con mayor precisión qué significa que la precisión de la clasificación ha dejado de mejorar. Una buena regla es terminar si la mejor precisión de clasificación no mejora durante bastante tiempo.

- Calendario de la tasa de aprendizaje: a menudo es ventajoso variar la tasa de aprendizaje. Al principio del proceso de aprendizaje, es probable que los pesos estén muy mal. Por lo tanto, es mejor usar una tasa de aprendizaje grande que haga que los pesos cambien rápidamente. Más tarde, podemos reducir la tasa de aprendizaje a medida que realizamos ajustes más precisos en nuestros pesos. Un enfoque natural es utilizar la misma idea básica que la parada anticipada. La idea es mantener constante la tasa de aprendizaje hasta que la precisión de la validación comience a empeorar. Luego, disminuya la tasa de aprendizaje en cierta cantidad, digamos un factor de dos o diez. Repetimos esto muchas veces, hasta que, digamos, la tasa de aprendizaje es un factor de 1.024 (o 1.000) veces menor que el valor inicial. Entonces terminamos.

- El parámetro de regularización, λ:Comenzar inicialmente sin regularización (λ = 0.0), y determinando un valor para η, como anteriormente. Usando esa elección de η, podemos utilizar los datos de validación para seleccionar un buen valor para λ . Empezamos probandoλ = 1.0 y luego aumentamos o disminuimos por factores de 10, según sea necesario para mejorar el rendimiento de los datos de validación. Una vez que haya encontrado un buen orden de magnitud, pasamos a ajustar el valor deλ. Hecho esto, se debe volver a optimizar η de nuevo

- Tamaño de mini-batch: necesitamos es una estimación lo suficientemente precisa como para que nuestra función de costos tienda a seguir disminuyendo. Si es demasiado pequeño, no podrá aprovechar al máximo los beneficios de las buenas bibliotecas matriciales optimizadas para hardware rápido. Demasiado grande y simplemente no está actualizando sus pesos con la suficiente frecuencia. Lo que necesita es elegir un valor de compromiso que maximice la velocidad de aprendizaje. La elección del tamaño de mini-batch en el que se maximiza la velocidad es relativamente independiente de los otros hiperparámetros, por lo que no

es necesario haber optimizado esos hiperparámetros para encontrar un buen tamaño de mini-batch. Por lo tanto, el camino a seguir es usar algunos valores aceptables para los otros hiperparámetros, y luego probar varios tamaños de mini-batch diferentes. Luego graficaremos la precisión de la validación en función del tiempo y eligiremos el tamaño de mini-batch que le brinde la mejora más rápida en el rendimiento. Con el tamaño de mini-batch elegido, procedemos a optimizar los otros hiperparámetros.

- Técnicas automatizadas: Una técnica común es la búsqueda en cuadrícula , que busca sistemáticamente a través de una cuadrícula en el espacio de hiperparámetros.

Variaciones en el descenso de gradiente estocástico

Existen muchos otros enfoques para optimizar la función de costos y, a veces, esos otros enfoques ofrecen un rendimiento superior al descenso de gradiente estocástico de mini-batches

- Técnica de Hesse: Sigue la fórmula:

$$\Delta w = -H^{-1}\nabla C.$$

donde H es la matriz de Hesse cuya entrada j k es $\partial^2 C/\partial w_j \partial w_k$. Este enfoque para minimizar una función de costo se conoce como técnica de Hesse o optimización de Hesse . Hay resultados teóricos y empíricos que muestran que los métodos de Hesse convergen en un mínimo en menos pasos que el descenso de gradiente estándar. Pero tiene una gran desventaja, es muy difícil de aplicar en la práctica. Parte del problema es el gran tamaño de la matriz de Hesse. Sin embargo, eso no significa que no sea útil de entender. De hecho, hay muchas variaciones en el descenso de gradientes que se inspiran en la optimización hessiana, pero que evitan el problema con matrices demasiado grandes.

- Descenso de gradiente basado en el momento: la ventaja que tiene la optimización hessiana es que incorpora no solo información sobre el gradiente, sino también información sobre cómo está cambiando el gradiente. El descenso de gradiente basado en la cantidad de movimiento se basa en una intuición similar, pero evita grandes matrices de segundas derivadas. Para entenderlo pensamos en la imagen en la que consideramos una bola rodando hacia un valle. La técnica del impulso modifica el descenso del gradiente de dos formas que lo hacen más similar a la imagen física. Primero, introduce una noción de "velocidad"para los parámetros que estamos tratando de optimizar. El gradiente actúa para cambiar la velocidad, no la "posición", de la misma manera que las fuerzas físicas cambian la velocidad y solo afectan indirectamente a la posición. En segundo lugar, el método del momento introduce una especie de término de fricción que tiende a reducir gradualmente la velocidad. Para entenderlo piensa en lo que sucede si nos movemos en línea recta por una pendiente. Por cada paso, la velocidad aumenta en la pendiente, por lo que nos movemos cada vez más rápidamente hacia el fondo del valle. Esto puede permitir que la técnica del impulso funcione mucho más rápido que el descenso de gradiente estándar. Por supuesto, un problema es que una vez que lleguemos al fondo del valle nos sobrepasaremos, por eso es necesario que haya un parámetro de fricción.Lo bueno de la técnica de impulso es que casi no se necesita trabajo para modificar una implementación de descenso de gradiente para incorporar el impulso. En la práctica, la técnica del impulso se usa comúnmente y, a menudo, acelera el aprendizaje.

- Otros enfoques: Es interesante buscar otros enfoques como BFGS y gradiente acelerado de Nesterov.

Otros modelos de neurona artificial

En la práctica, las redes construidas con otros modelos de neuronas a veces superan a las redes sigmoides. Dependiendo de la aplicación, las redes basadas en dichos modelos alternativos pueden aprender más rápido, generalizar mejor para probar datos o quizás hacer ambas cosas. La variación más simple es la neurona **tanh** que reemplaza la función sigmoidea por la función tangente hiperbólica. La salida de la neurona viene dada por:

$$\tanh(w \cdot x + b)$$

. Se puede observar gráficamente que la función sigmoidea y la función tanh tienen la misma forma:

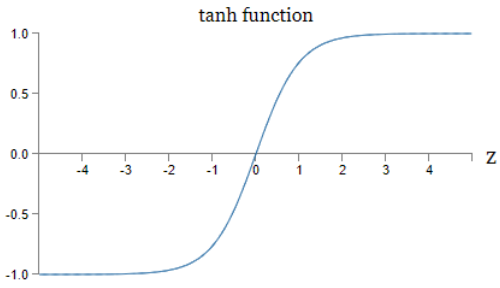

Figura 3.14: Función neurona tipo tanh [2]

Una diferencia entre las neuronas tanh y las neuronas sigmoides es que la salida de las neuronas tanh varía de -1 a 1, no de 0 a 1. Esto significa que si vas a construir una red basada en neuronas tanh, es posible que debas normalizar tus salidas, ideas como la propagacion hacia atrás y el descenso de gradiente estocástico se aplican tan fácilmente a una red de neuronas tanh como a una red de neuronas sigmoides. tanh proporciona solo una pequeña o ninguna mejora en el rendimiento sobre las neuronas sigmoides. Desafortunadamente, todavía no tenemos reglas estrictas para saber qué tipos de neuronas aprenderán más rápido o darán el mejor rendimiento de generalización para una aplicación en particular.

Otra variación de la neurona sigmoidea es la neurona lineal rectificada o unidad lineal rectificada . La salida de una unidad lineal rectificada es:

$$\text{máx}(0, w \cdot x + b)$$

Gráficamente, la función rectificadora) se ve como esto:

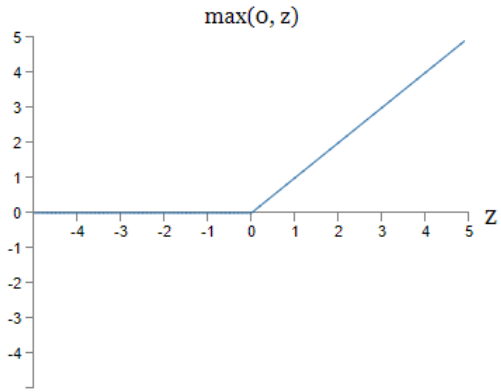

Figura 3.15: Función neurona tipo lineal rectificada [2]

Al igual que las neuronas sigmoidea y tanh, las unidades lineales rectificadas se pueden usar para calcular cualquier función, y se pueden entrenar usando ideas como la propagación hacia atrás y el descenso de gradiente estocástico. Al igual que con las neuronas tanh, todavía no tenemos una comprensión realmente profunda de cuándo, exactamente, las unidades lineales rectificadas son preferibles, ni por qué, pero aumentar la entrada ponderada a una unidad lineal rectificada nunca causará que se sature, por lo que no hay una desaceleración del aprendizaje correspondiente. Por otro lado, cuando la entrada ponderada a una unidad lineal rectificada es negativa, el gradiente desaparece y, por lo tanto, la neurona deja de aprender por completo.

3.1.4. Universalidad de las redes neuronales

Uno de los hechos más llamativos sobre las redes neuronales es que pueden calcular cualquier función. No importa cuál sea la función, se garantiza que habrá una red neuronal de modo que para cada entrada posible, X, el valor F(x) (o alguna aproximación cercana) es la salida de la red. Este resultado se mantiene incluso si la función tiene muchas entradas, $F = f(X_1, X_2, ..., X_M)$ y muchas salidas. Este resultado nos dice que las redes neuronales tienen una especie de universalidad . Independientemente de la función que queramos calcular, sabemos que existe una red neuronal que puede hacer el trabajo. El teorema de la universalidad es bien conocido por las personas que utilizan redes neuronales. Pero no se entiende tan ampliamente por qué es verdad. La mayoría de las explicaciones disponibles son bastante técnicas.

Advertencias

Primero, esto no significa que una red pueda usarse para calcular exactamente cualquier función. Más bien, podemos obtener una aproximación que sea tan buena como queramos. Al aumentar el número de neuronas ocultas podemos mejorar la aproximación.Para hacer esta declaración más precisa, suponga que se nos da una función f(x) que nos gustaría calcular con la precisión deseada $\epsilon > 0$ La garantía es que utilizando suficientes neuronas ocultas siempre podemos encontrar una red neuronal cuya salida g(x) cumpla que $|g(x) - f(x)| < \epsilon$ para todas las entradas de x.

La segunda advertencia es que la clase de funciones que se pueden aproximar de la forma descrita son las funciones continuas . Si una función es discontinua, es decir, realiza saltos bruscos y bruscos, en general no será posible realizar una aproximación utilizando una red neuronal. Sin

embargo, incluso si la función que realmente nos gustaría calcular es discontinua, a menudo ocurre que una aproximación continua es suficientemente buena. Si es así, entonces podemos usar una red neuronal.

Universalidad con una entrada y una salida

Aproximar función por red neuronal con una sola capa oculta el objetivo es que la salida ponderada de la capa oculta sea igual a $\sigma^{-1} \circ f(x)$ que tendrá la misma forma que f(x) (función original). Para ello lo haremos mediante funciones escalonadas realizaremos lo siguiente:

- La primera capa de pesos tiene un valor constante grande, digamos w = 1000.
- Los sesgos en las neuronas ocultas son solo b = - w x s. Entonces, por ejemplo, para la segunda neurona ocultas = 0,2 se convierte en b = - 1000 × 0,2 = - 200.
- La capa final de pesos está determinada por el h valores. Entonces, por ejemplo, el valor que eligió arriba para el primer h, h = -1,3 , significa que los pesos de salida de las dos neuronas ocultas superiores son -1,3 y 1.3 , respectivamente. Y así sucesivamente, para toda la capa de pesos de salida.
- Finalmente, el sesgo en la neurona de salida es 0

Con esto se conseguirá una burda aproximación mediante la unión de neuronas en una sola capa obteniendo un resultado como el siguiente:

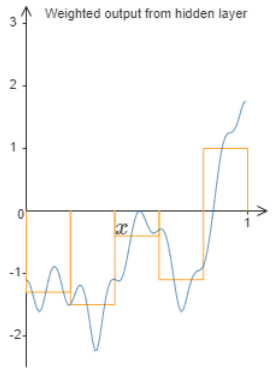

Figura 3.16: Aproximación de funciones a través de red neuronal [2]

Universalidad con una entrada y una salida

Si introducimos una neurona de entrada más lo que veremos es una nueva función escalonada la diferencia con lo anterior es que ahora la función es en tres dimensiones y cuantas más neuronas de entrada introduzcamos más complejo será ya que tendrá más dimensiones. Esto será aplicable para todo tipo de neuronas cuyas funciones de activación $s(z)$ estén bien definidas es decir z este en $(-\infty, +\infty)$.

Arreglando las funciones de paso

Hasta ahora, hemos asumido que nuestras neuronas pueden producir funciones escalonadas exactamente. Esa es una aproximación bastante buena, pero es solo una aproximación. De hecho, habrá una ventana estrecha de fallo. En estas ventanas de fracaso, la explicación que he dado de la universalidad fracasará. Ahora bien, no es un terrible fracaso. Al hacer que la entrada de pesos a las neuronas sea lo suficientemente grande, podemos hacer que estas ventanas de fallo sean tan pequeñas como queramos.

El problema resulta fácil de solucionar. En particular, supongamos que queremos que nuestra red calcule alguna función, F. Si hiciéramos esto usando la técnica descrita anteriormente, usaríamos las neuronas ocultas para producir una secuencia de funciones de golpe. Si multiplicamos la función f(x) por el factor 1/2 obtendríamos que nuestra aproximación sería: $\sigma^{-1} \circ f(x)/2$ y veríamos que los puntos de la ventana de fallo se habrían movido, esto ocurrirá siempre que modifiquemos la función. Siempre que las ventanas de fallo sean lo suficientemente estrechas, un punto solo estará en una ventana de falla. Y siempre que usemos un número suficientemente grande M de aproximaciones superpuestas, el resultado será una excelente aproximación general.

3.1.5. ¿Por qué las redes neuronales profundas son difíciles de entrenar?

Hasta ahora casi todas las redes con las que hemos trabajado tienen una sola capa oculta de neuronas (más las capas de entrada y salida), estas redes simples han sido muy útiles, no obstante, intuitivamente esperaríamos que las redes con muchas más capas ocultas fueran más poderosas. Tales redes podrían usar las capas intermedias para construir múltiples capas de abstracción. Por ejemplo, si estamos haciendo un reconocimiento de patrones visual, entonces las neuronas de la primera capa podrían aprender a reconocer los bordes, las neuronas de la segunda capa podrían aprender a reconocer formas más complejas, digamos triángulos o rectángulos, construidos a partir de los bordes. La tercera capa reconocería formas aún más complejas. Y así. Es probable que estas múltiples capas de abstracción otorguen a las redes profundas una ventaja convincente para aprender a resolver problemas complejos de reconocimiento de patrones. La pregunta es: ¿Cómo podemos entrenar redes tan profundas? Cuando miramos de cerca, descubriremos que las diferentes capas de nuestra red profunda están aprendiendo a velocidades muy diferentes.

El problema del gradiente que desaparece

El gradiente de desaparición se produce cuando las primeras capas aprenden más lento que las siguientes, debemos preguntarnos ¿Por qué ocurre el problema del gradiente de desaparición? ¿Hay formas de evitarlo? ¿Y cómo debemos lidiar con esto en el entrenamiento de redes neuronales profundas? No es inevitable, aunque la alternativa tampoco es muy atractiva, a veces el gradiente se hace mucho más grande en capas anteriores, este es el problema del gradiente explosivo, el gradiente en las redes neuronales profundas es inestable , y tiende a explotar o desaparecer en capas anteriores.Esta inestabilidad es un problema fundamental para el aprendizaje basado en gradientes en redes neuronales profundas. Es algo que debemos comprender y, si es posible, tomar medidas para abordarlo.

¿Qué está causando el problema del gradiente que desaparece? Gradientes inestables en redes neuronales profundas

Por qué ocurre el problema del gradiente de desaparición: para comprender por qué ocurre el problema del gradiente de desaparición, escribamos la expresión para el gradiente:

$$\frac{\partial C}{\partial b_1} = \sigma'(z_1)\, w_2\sigma'(z_2)\, w_3\sigma'(z_3)\, w_4\sigma'(z_4)\, \frac{\partial C}{\partial a_4}$$

Excepto el último término, esta expresión es un producto de términos de la forma $w_j\sigma'(z_j)$. Para entender cómo se comporta cada uno de esos términos, veamos una gráfica de la función σ':

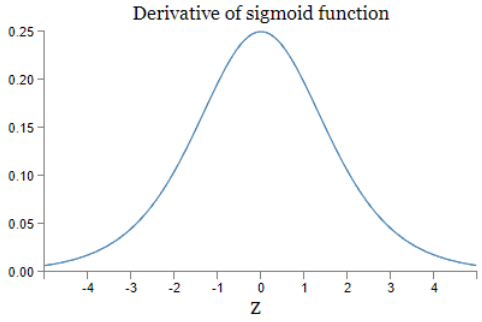

Figura 3.17: Derivada de la función sigmoide [2]

La derivada alcanza el máximo cuando Z=0 y σ'=1/4 . Ahora, si usamos nuestro enfoque estándar para inicializar los pesos en la red, elegiremos los pesos usando una gaussiana con media 0 y desviación estándar 1. Entonces, los pesos generalmente satisfarán $|w_j| < 1$. Juntando estas observaciones, vemos que los términos $w_j\sigma'(z_j)$ normalmente serán menor que 1/4. Y cuando tomamos un producto de muchos de esos términos, el producto tenderá a disminuir exponencialmente: cuantos más términos, menor será el producto.

El gradiente explosivo es otro de los problemas a los que debemos enfrentarnos, veamos un ejemplo. Hay dos pasos para obtener un gradiente explosivo. Primero, elegimos todos los pesos de la red para que sean grandes, ($w_1 = w_2 = w_3 = w_4 = 100$. En segundo lugar, elegiremos los sesgos para que $\sigma'(z_j)$ no sea demasiado pequeño, lo que tenemos que hacer es elegir los sesgos para asegurarnos de que la entrada ponderada de cada neurona sea $z_j = 0$ de tal forma que $\sigma'(z_j) = 1/4$. Así que queremos que $z_1 = w_1a_0 + b_1 = 0$, esto lo podemos conseguir estableciendo $b_1 = -100 * a_0$. Podemos usar la misma idea para seleccionar otros sesgos, cuando hacemos esto vemos que todos los términos $w_j\sigma'(z_j)$ son $100 * \frac{1}{4} = 25$. Y hemos obtenido un gradiente explosivo

En el caso del problema con **el gradiente inestable**, el gradiente en las primeras capas es el producto de términos de todas las capas posteriores. Cuando hay muchas capas, es una situación intrínsecamente inestable. La única forma en que todas las capas pueden aprender casi a la misma velocidad es si todos esos productos de términos se acercan a equilibrarse. Sin algún mecanismo o razón subyacente para que ocurra ese equilibrio, es muy poco probable que ocurra simplemente por casualidad. Suelen aparecer en las redes complejas.

La prevalencia del problema del gradiente que desaparece. Cuando se utilizan neuronas sigmoideas, el gradiente suele desaparecer. Para evitar el problema necesitamos $|w\sigma'(z)| \geq 1$. Se podría pensar que esto podría suceder fácilmente si w es muy grande. Sin embargo, es más difícil de lo que parece. La razón es que $\sigma'(z)$ también depende de w: $\sigma'(z) = \sigma'(wa + b)$. Entonces cuando

hacemos w grande, debemos tener cuidado de no hacer simultáneamente $\sigma'(wa+b)$ pequeño. Esto además nos pone en valores de la función muy pequeños. La única forma de evitar esto es si la activación de entrada cae dentro de un rango de valores bastante estrecho, a veces es posible que eso suceda. Sin embargo, más a menudo no sucede

3.1.6. Deep Learning

Vamos a desarrollar técnicas que puedan usarse para entrenar redes profundas y las aplicaremos a la práctica. Hablaremos de las redes convolucionales profundas, también examinaremos otros modelos de redes neuronales como las redes neuronales recurrentes y las unidades de memoria a corto plazo, asó como también como estos modelos pueden aplicarse a problemas de reconocimiento de voz, procesamiento del lenguaje natural y otras áreas.

Introduciendo redes convolucionales

Es extraño usar redes con capas completamente conectadas para clasificar imágenes. La razón es que dicha arquitectura de red no tiene en cuenta la estructura espacial de las imágenes. Por ejemplo, trata los píxeles de entrada que están muy separados y juntos exactamente en la misma base. En cambio, estos conceptos de estructura espacial deben inferirse de los datos de entrenamiento. Pero, ¿y si, en lugar de comenzar con una arquitectura de red que es tabula rasa, usamos una arquitectura que intenta aprovechar la estructura espacial? Las redes convolucionales utilizan una arquitectura especial que está particularmente bien adaptada para clasificar imágenes haciendo que sean más rápidas de entrenar. Esto, a su vez, nos ayuda a entrenar redes profundas de muchas capas, que son muy buenas para clasificar imágenes. Estas redes utilizan tres ideas:

Campos receptivos locales En las capas completamente conectadas anteriores, las entradas se representaron como una línea vertical de neuronas. En una red convolucional, será útil pensar en lugar de las entradas como un cuadrado n x n. Como es habitual, conectaremos los píxeles de entrada a una capa de neuronas ocultas. Pero no conectaremos cada píxel de entrada a cada neurona oculta. En cambio, solo hacemos conexiones en regiones pequeñas y localizadas de la imagen de entrada, para ser más precisos, cada neurona de la primera capa oculta se conectará a una pequeña región de las neuronas de entrada. Entonces, para una neurona oculta en particular, podríamos tener conexiones que se ven así:

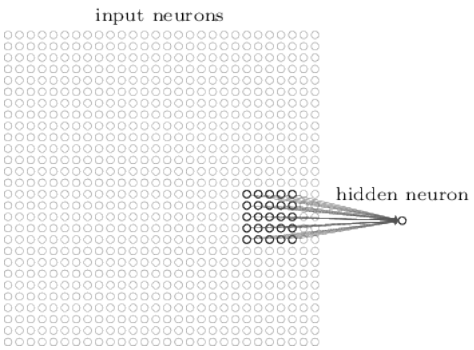

Figura 3.18: Conexiones de la red de entrada con las neuronas ocultas [2]

Esa región en la imagen de entrada se llama campo receptivo local para la neurona oculta. Es una pequeña ventana en los píxeles de entrada. Cada conexión aprende un peso. Y la neurona

oculta también aprende un sesgo general. Puede pensar que esa neurona oculta en particular está aprendiendo a analizar su campo receptivo local particular. Luego deslizamos el campo receptivo local a través de toda la imagen de entrada. Para cada campo receptivo local, hay una neurona oculta diferente en la primera capa oculta. Luego, deslizamos el campo receptivo local un píxel hacia la derecha (es decir, una neurona), para conectarlo a una segunda neurona oculta y así sucesivamente, construyendo la primera capa oculta.

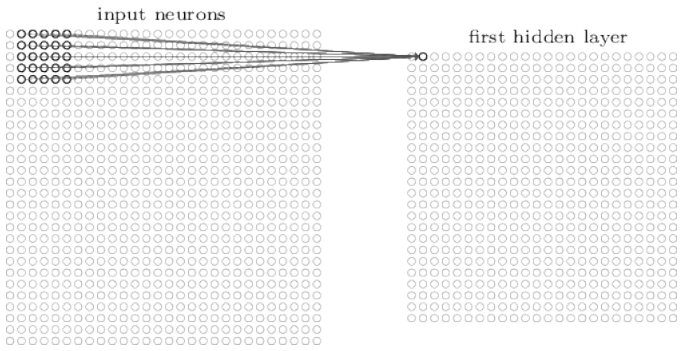

Figura 3.19: Ejemplo de campos receptivos [2]

A veces se utiliza una zancada diferente . Por ejemplo, podríamos mover el campo receptivo local2 píxeles a la derecha (o hacia abajo), en cuyo caso diríamos que se utiliza una longitud de zancada de 2.

Pesos y sesgos compartidos Se usan los mismos sesgos y pesos para cada una de las n x n neuronas de la capa oculta, en otras palabras, para la neurona oculta j,k la salida es:

$$\sigma\left(b+\sum_{l=0}^{4}\sum_{m=0}^{4}w_{l,m}a_{j+l,k+m}\right)$$

donde σ es la función de activación de la neurona, b es el sesgo compartido, $w_{l,m}$ es un vector m x m de pesos compartidos y $a_{x,y}$ es la entrada de activación en la posición x,y. A veces a esta ecuación se la denomina convolución. Esto supone una gran ventaja ya que reduce en gran medida la cantidad de parámetros involucrados en una red convolucional, intuitivamente, parece probable que reducirá la cantidad de parámetros que necesita para obtener el mismo rendimiento que el modelo completamente conectado. Eso, a su vez, dará como resultado un entrenamiento más rápido para el modelo convolucional y, en última instancia, nos ayudará a construir redes profundas utilizando capas convolucionales.

Esto significa que todas las neuronas de la primera capa oculta detectan exactamente la misma característica, solo en diferentes ubicaciones en la imagen de entrada. A veces llamamos el mapa de la capa de entrada a la capa oculta, el mapa de características. Llamamos a los pesos que definen el mapa de características, los pesos compartidos. Y llamamos al sesgo que define el mapa de características de esta manera, el sesgo compartido. A menudo se dice que los pesos y el sesgo compartidos definen un núcleo o filtro.

Para realizar el reconocimiento de imágenes, necesitaremos más de un mapa de características. Y así, una capa convolucional completa consta de varios mapas de características diferentes:

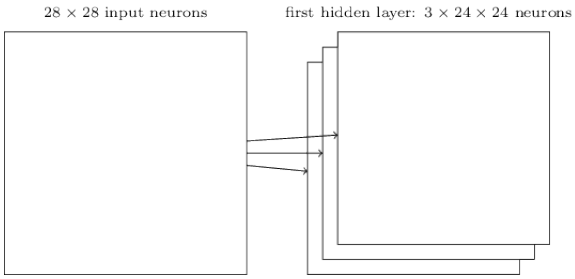

Figura 3.20: Mapas de características [2]

Capas de agrupación Se tarta de una capa adicional característica de estas redes. Las capas de agrupación se utilizan generalmente inmediatamente después de las capas convolucionales. Lo que hacen las capas de agrupación es simplificar la información en la salida de la capa convolucional. En detalle, una capa de agrupación toma cada mapa de características, que es la salida de la capa convolucional y prepara un mapa de características condensado. Como ejemplo concreto, un procedimiento común para la agrupación se conoce como agrupación máxima . La capa convolucional generalmente involucra más de un mapa de características. Aplicamos la agrupación máxima a cada mapa de características por separado. Entonces, si hubiera tres mapas de características, las capas combinadas convolucional y de agrupación máxima se verían así:

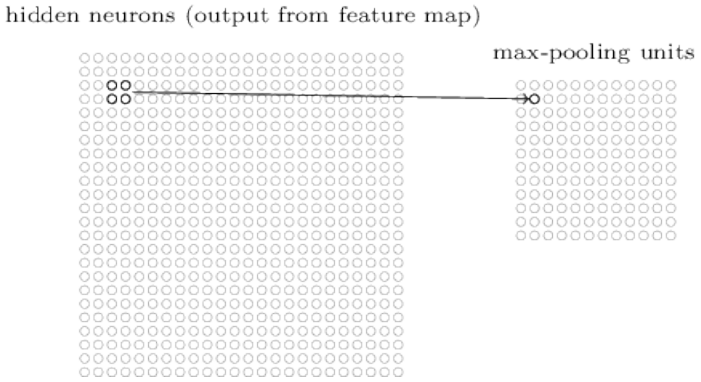

Figura 3.21: Ejemplo de matriz de neuronas [2]

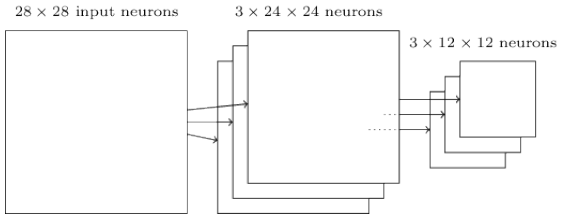

Figura 3.22: Ejemplo de arquitectura de red convulcional [2]

Podemos pensar en la agrupación máxima como una forma de que la red pregunte si una característica determinada se encuentra en alguna parte de una región de la imagen. Luego desecha la información de posición exacta. Un beneficio es que hay muchas menos entidades agrupadas, por lo que esto ayuda a reducir la cantidad de parámetros necesarios en capas posteriores. Existen otros enfoques como la agrupación L2.

Viendo todo en conjunto veríamos una arquitectura de este tipo:

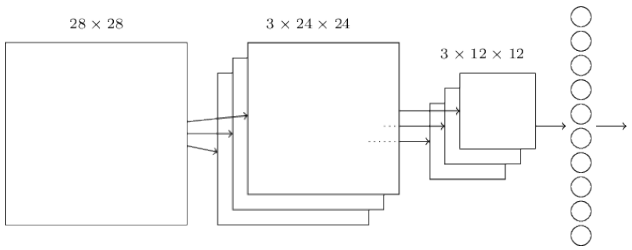

Figura 3.23: Arquitectura con agrupación L2 [2]

Donde la ultima capa es una capa completamente conectada. Esta arquitectura convolucional es bastante diferente a la de anteriores secciones pero en lineas generales es similar: una red formada por muchas unidades simples, cuyos comportamientos están determinados por sus pesos y sesgos, que utilizan datos de entrenamiento para entrenarse para que la red haga un buen trabajo clasificando la entrada.

Otros enfoques para las redes neuronales profundas

Existen muchas ideas de las que no se han hablado: redes neuronales recurrentes, máquinas de Boltzmann, modelos generativos, aprendizaje por transferencia, aprendizaje por refuerzo, etc. A continuación se muestran algunas de ellas:

- **Redes neuronales recurrentes**: Parten de la suposición que permitimos que los elementos de la red sigan cambiando de forma dinámica. Por ejemplo, ocurrirían cosas como que el comportamiento de las neuronas ocultas podría estar determinado no solo por las activaciones en capas ocultas anteriores, sino también por las activaciones en momentos anteriores. De hecho, la activación de una neurona podría estar determinada en parte por su propia activación en un momento anterior o quizás las activaciones de neuronas ocultas y de salida no estarán determinadas solo por la entrada actual a la red, sino también por entradas anteriores. La idea general es que las RNN son redes neuronales en las que existe alguna noción de cambio dinámico a lo largo del tiempo y son especialmente útiles para analizar datos o procesos que cambian con el tiempo, por ejemplo han resulta particularmente útiles en el reconocimiento de voz.

- **Unidades de memoria a largo plazo a corto plazo (LSTM)**: El problema de las RNN es que son difíciles de entrenar esto se debe al problema del gradiente inestable. el problema empeora en las RNN, ya que los gradientes no solo se propagan hacia atrás a través de capas, sino que se propagan hacia atrás en el tiempo. Si la red funciona durante mucho tiempo, puede hacer que el gradiente sea extremadamente inestable y difícil de aprender. Afortunadamente, es posible incorporar una idea conocida como unidades de memoria a corto plazo (LSTM) a las RNN. Los LSTM hacen que sea mucho más fácil obtener buenos resultados al entrenar RNN

- **Redes de creencias profundas, modelos generativos y máquinas de Boltzmann:** Las redes de creencias profundas (DBN) fueron influyentes durante varios años, pero desde entonces su popularidad ha disminuido, mientras que modelos como las redes de alimentación directa y las redes neuronales recurrentes se han puesto de moda. A pesar de esto, los DBN tienen varias propiedades que los hacen interesantes. Una de ellas es que son un ejemplo

de lo que se llama modelo generativo, esto implica que es posible especificar los valores de algunas de las neuronas de características y luego .ejecutar la red hacia atrás", generando valores para las activaciones de entrada. Un DBN entrenado en imágenes de dígitos escritos a mano también puede usarse para generar imágenes que parezcan dígitos escritos a mano. En esto, un modelo generativo es muy parecido al cerebro humano: no solo puede leer dígitos, también puede escribirlos. Una segunda razón por la que los DBN son interesantes es que pueden realizar un aprendizaje semi-supervisado y sin supervisión. Por ejemplo, cuando se entrena con datos de imágenes, los DBN pueden aprender funciones útiles para comprender otras imágenes, incluso si las imágenes de entrenamiento no están etiquetadas.

- **Otras:** Las áreas activas de investigación incluyen el uso de redes neuronales para realizar el procesamiento del lenguaje natural, la traducción automática , así como aplicaciones quizás más sorprendentes como la informática musical.

Sobre el futuro de las redes neuronales

Interfaces de usuario impulsadas por la intención: En esta visión, en lugar de responder a las consultas literales de los usuarios, la búsqueda utilizará el aprendizaje automático para tomar información vaga del usuario, discernir con precisión lo que se quería decir y tomar medidas sobre la base de esos conocimientos. La idea se puede aplicar de manera mucho más amplia que la búsqueda. Durante las próximas décadas, miles de empresas crearán productos que utilizan el aprendizaje automático para crear interfaces de usuario que puedan tolerar la imprecisión, al mismo tiempo que disciernen y actúen sobre la verdadera intención del usuario.

Aprendizaje automático, ciencia de datos y el círculo virtuoso de la innovación: Una aplicación notable del aprendizaje automático es la ciencia de datos, donde el aprendizaje automático se utiliza para encontrar las ïncógnitas conocidas.ºcultas en los datos. Una consecuencia de esta moda que no se comenta tan a menudo es que el aprendizaje automático es un motor que impulsa la creación de varios mercados nuevos e importantes y áreas de crecimiento en tecnología. El resultado serán grandes equipos de personas con amplia experiencia en el tema y con acceso a recursos extraordinarios. Eso impulsará el aprendizaje automático más adelante, creando más mercados y oportunidades, un círculo virtuoso de innovación.

¿Las redes neuronales y el aprendizaje profundo conducirán pronto a la inteligencia artificial? El papel de las redes neuronales y el aprendizaje profundo

Allá por la década de 1980, había mucho entusiasmo y optimismo sobre las redes neuronales, especialmente después de que la propagación hacia atrás se hizo ampliamente conocida. Ese entusiasmo se desvaneció y, en la década de 1990, el testigo del aprendizaje automático pasó a otras técnicas, como las máquinas vectoriales de soporte. Hoy en día, las redes neuronales están nuevamente en lo alto, estableciendo todo tipo de récords, derrotando a todos los interesados en muchos problemas. Pero, ¿quién puede decir que mañana no se desarrollará un nuevo enfoque que vuelva a barrer las redes neuronales? ¿O quizás el progreso con las redes neuronales se estancará y nada surgirá de inmediato para ocupar su lugar?

3.1.7. Campos de aplicación del Deep Learning

Por último cabe destacar algunos de los campos donde el Deep Learning tiene especial interes:

- Procesamiento del lenguaje natural(NLP) : Respondiendo preguntas; reconocimiento de voz; resumir documentos;clasificación de documentos; encontrar nombres, fechas, etc. en documentos; buscando artículos que mencionan un concepto
- Visión por ordenador : Interpretación de imágenes de satélite y drones, reconocimiento facial, subtítulos de imágenes, lectura de señales de tráfico, localización de peatones y vehículos en vehículos autónomos
- Medicina : Encontrar anomalías en imágenes de radiología, incluidas tomografía computarizada, resonancia magnética y rayos X; contar características en diapositivas de patología; características de medición en ultrasonidos para diagnosticar la retinopatía diabética.
- Biología : Proteínas plegables; clasificar proteínas; muchas tareas genómicas, como tla clasificación de mutaciones genéticas clínicamente accionables;clasificación celular; analizar las interacciones proteína / proteína
- Genaración de imágenes: Colorear imagenes, aumentar la resolucion de las imagenes, eliminar el ruido de las imagenes, convertir imagenes en arte al estilo de los famosos.
- Sistemas de recomendación: Búsqueda web, recomendaciones de productos, diseño de la página de inicio
- Videojuegos: Ajedrez, juegos de estrategia a tiempo real, juegos de Atari...
- Robotica : Manipular objetos que son difíciles de localizar (p. Ej., Transparentes, brillantes, sin textura) o difícil de recoger
- Otras: Previsión financiera y logística, texto a voz y mucho, mucho más ...

Capítulo 4

Análisis tecnológico

Para la realización de este capítulo se han utilizado como guía los libros "Practical Deep Learning for Cloud, Mobile and Edge: Real-World AI and Computer Vision Projects Using Python, Keras and TensorFlow" [3] y "Deep Learning for Coders with Fastai and Pytorch: AI Applications Without a PhD" [1]

4.1. Explorando el paisaje de la Inteligencia Artificial

En esta sección es útil recapitular algunos términos que dejamos atrás al principio del documento como:

- **Inteligencia Artificial**: Da a las máquinas la capacidad de imitar el comportamiento humano
- **Machine Learning**: Es la rama de la Inteligencia Artificial en las que las máquinas usan técnicas estadísticas para aprender de información y experiencias previas. La meta es que las máquinas hagan acciones en el futuro basadas en experiencias pasadas.
- **Deep Learning**: Es un subcampo del Machine Learning donde se utilizan redes neuronales profundas y multicapas para hacer predicciones, especialmente para el reconocimiento de imágenes, del habla...

4.1.1. Breve historia de la Inteligencia Artificial centrada en las redes neuronales

El principio de la IA nace en los años 50 en el documento de Alan Turing "Computing Machinery and Intelligence", donde se pregunta por primera vez "¿Pueden las máquinas pensar?", también propuso el conocido test de Turing, en el que se dice que una IA que puede imitar a un humano es en esencia un humano.

El termino IA fue acuñado por John McCarthy en 1956 en el "Dartmouth Summer Research Project". Siendo la primera vez que la IA se convierte en un campo de investigación en vez de un proyecto único. En el documento "Perceptron: A Perceiving and Recognizing Automaton" de 1957 Frank Rosenblatt asentó las bases de las redes neuronales profundas, en el postula la posibilidad de crear un sistema electrónico o electromecánico capaz de aprender a reconocer la similitud entre

patrones de información óptica, eléctrica o de tonos, que funcione de forma similar a un cerebro humano. Y que en vez de usar un modelo basado en reglas, se use un modelo estadístico. En 1965, Ivakhnenko y Lapa publican la primera red neuronal que funciona en el documento "Group Method of Data Handling - A Rival Methds of Stochastic Approximation". Ivakhnenko es considerado para algunos el padre del Deep Learning.

Alrededor de 1974, se produce una época llamada "IA winter" en la que se plantea si la IA es una pérdida de dinero, haciendo que la investigación en esta rama se diezmase .Pero durante esa época también hubo trabajo de fondo como fue el desarrollado por Geoffrey Hinton y su equipo quienes publicaron "Learning reprasentations by back-propagation errors" donde sale a la luz el algoritmo de propagación hacia atrás. En 1989, George Cybenko provee la primera prueba del llamado Teorema de Aproximación Universal que dice que cualquier red neuronal de una sola capa oculta es capaz de modelar cualquier problema, también en esta época se hicieron grandes avances surgiendo las LSTM(Long Short-Term Memory) y la técnica basada en Machine Learning SVM(Support Vector Machines). En años más recientes destaca la creación de numerosas redes neuronales que han intentado desafiar la fuente de imágenes ImageNet consiguiendo mejores resultados de forma casi anual.

4.1.2. Prerrequisitos para el deep learning

Para poder proveer una buena solución de deep learning es necesario:

- **Conjunto de datos**: Para entrenar cualquier sistema basado en deep learning necesitamos datos de entrenamiento. Por ejemplo para clasificar reviews de Amazon será necesario, coger un conjunto y etiquetarlas de forma que sepamos si la review es positiva o negativa. Estos datos pueden ser costosos no obstante se disponen de muchos en la cedidos por universidades o compañías, así como la técnica llamada "transferencia de aprendizaje".

- **Arquitectura del modelo**: A alto nivel lo podemos imaginar como una función a la que tu añades una entrada y te devolverá una salida. Un buen modelo será aquel que para cada entrada te devuelva la salida esperada. Pero es cuando profundizamos en el modelo cuando se vuelve interesante, dentro de un modelo podemos observar un grafo, cuyos nodos representar operaciones matemáticas y sus aristas como se mueven los datos de un nodo a otro. La estructura de este grafo determinará la velocidad, la precisión y los recursos que va a consumir. Este grafo es conocido como la arquitectura de la red, pero es solo un plano. Para entrenar este modelo debemos seguir los siguientes pasos:

  1. Dar datos de entrada
  2. Obtener datos de la salida
  3. Determinar como de lejos se encuentran de los datos reales
  4. Propagar la magnitud de error a través de la red para permitir que esta aprenda

  Este proceso de entrenamiento se aplicará iterativamente hasta que obtengamos las predicciones esperadas El resultado de este proceso serán los pesos, que son los parámetros necesarios para que cada nodo pueda operar con la entrada dada, en la primera iteración

estos pesos serán asignados aleatoriamente. La meta del entrenamiento es ajustar estos pesos para obtener la predicción deseada. Dependiendo del tipo de problema que queramos resolver existen diferentes modelos de arquitectura. Por ejemplo en el caso de imágenes y audio se suelen utilizar las redes neuronales convulcionales, mientras que para el análisis de texto se suelen utilizar redes neuronales recurrentes.

- **Framework**: Actualmente existen muchas librerías que nos ayudan a entrenar nuestro modelo, así como, existen numerosos frameworks especializados en usar esos modelos para hacer predicciones optimizando el sitio donde la aplicación reside. Algunos de los más populares son:

  - **TensorFlow**: Se trata de un framework desarrollado por Google que vio a la luz en noviembre de 2015. Su problema es que no es lo suficientemente sencillo de usar
  - **Keras**: Como respuesta a los problemas de los desarrolladores de deep learning, Fran"cois Chollet desarrolló Keras en marzo de 2015. Esta solución hizo el deep learning accesible para todo el mundo, proveyendo una interfaz intuitiva y sencilla para programar, que usara después otras librerias como un framework computacional de backend
  - **PyTorch**: En paralelo, PyTorch empieza en Facebook a principios de 2016, cuando se empiezan a observar las limitaciones de TensorFlow. PyTorch conserva las estructuras de Python y su capacidad de debugging, haciéndolo flexible y fácil de usar convirtiéndose rápidamente en uno de los favoritos de los desarrolladores de IA.

- **Hardware**: Entrenar una red lleva mucho tiempo, desde minutos a horas o incluso días. Un mejor hardware hará que este proceso sea más rápido. Habitualmente las GPUs van a acelerar el orden de 10 a 15 veces lo hecho por una CPU y con un mejor rendimiento

4.1.3. IA responsable

Por mucho que la IA puede ayudar a la humanidad también la puede dañar, y la culpa sería del diseñador de IA. Algunos ejemplos es en la creación de sesgos (de selección, implicitos, de comunicación, in-group/out-group...) por lo que es necesaria una responsabilidad a la hora de usar este tipo de algoritmos ya que pueden afectar a temas como a la privacidad de las personas.

4.2. Clasificación de imágenes con Keras

Para esta sección se utilizará un modelo preentrenado, reusando el famoso modelo ResNet-50.

Keras es un framework que empezó en 2015 como una capa de abstracción fácil de usar de otras librerías, haciendo posible un rápido prototipado. Al mismo tiempo que ayuda a los desarrolladores de deep learning a iterar rápidamente en experimentos. En el año 2017 toda la implementación de Keras fue añadida en TensorFlow. En internet normalmente vemos una versión de Keras de TensorFlow llamada tf.keras.

4.2.1. Predecir la categoria de una imagen

La clasificación de una imagen seguiría una serie de pasos como los siguientes:

1. Cargar una imagen
2. Redimensionarla a un tamaño predefinido como puede ser 224x224 píxeles
3. Normalizar el valor de un píxel normalmente entre [0,1]
4. Seleccionar un modelo preentrenado
5. Ejecutar el modelo preentrenado en la imagen para conseguir una lista de categorías predichas y sus probabilidades respectivas
6. Mostrar algunas de las categorías más probables

Antes de profundizar acerca de como se procesan las imágenes, sería bueno echar un vistazo como estas almacenan la información. En el nivel más básico, una imagen es una colección de píxeles que constan de 1 a 4 partes conocidas como componentes o canales.

Para el análisis de imágenes se suele utilizar las llamadas redes neuronales convulcionales como por ejemplo Res-Net50 un modelo entrenado con la base de datos ImageNet. Estos modelos se pueden encontrar en los llamados "zoos de modelos" donde organizaciones o personas pueden subir modelos que han para que otros los reutilicen. Esta tradición comenza con "Caffe" uno de los primeros frameworks de deep learning desarrollado por la Universidad de California, Berkely. Cuando se comienza a trabajar con un nuevo proyecto de deep learning, es buena idea explorar si ya hay algún modelo que realiza una tarea similar y ha sido entrenado con un conjunto de datos parecido. También es interesante conocer los llamados "mapas de calor" que colorean de una tonalidad roja aquellos píxeles que son responsables de que una imagen sea categorizada de una manera u otra, estos pueden ser muy útiles para detectar visualmente sesgos en los datos. Esto es importante porque la calidad del modelo dependerá de los datos con los que ha sido entrenado, si estos datos son sesgados, esto se reflejará en las predicciones.

4.2.2. Transferir conocimiento con Keras

Empezar un proyecto de deep learninng puede ser relativamente rápido cuando usamos un modelo preentrenado, que reutilice el conocimiento adquirido durante su entrenamiento y lo adapte a la tarea que estamos haciendo, este proceso es conocido como transferencia de aprendizaje.

Si queremos transferir conocimiento de un modelo a otro, lo que queremos es reutilizar mas de las capas generales y menos de las especificas, es decir, borrar las últimas capas (normalmente las capas totalmente conectadas) y así usar las más genéricas, y añadir capas orientadas a nuestra a nuestra clasificación específica. Una vez que el entrenamiento comience las capas genéricas se mantienen y no son modificadas, mientras que las capas nuevas para tareas específicas se permite que sean modificadas. Así es como la transferencia de aprendizaje permite entrenar rápidamente nuevos modelos.

Una transferencia básica de aprendizaje puede llevarnos lejos, normalmente añadiremos dos o tres capas totalmente conectadas después de las capas genéricas para hacer el nuevo modelo de clasificación. Pero si queremos mayor precisión debemos permitir que se entrenen más capas. Esto significa descongelar algunas de las capas que habiamos congelado previamente en la transferencia de aprendizaje, a este proceso es lo que llamamos "fine tunning". Es obvio que utilizando esta

técnica conseguiremos una mayor precisión para nuestra tarea ya que un mayor numero de capas ha sido adaptado a nuestra tarea comparado con la transferencia de conocimiento.

Pero esta técnica debe usarse en su justa medida lo cual nos lleva a preguntarnos ¿Cuánto hacer fine tunning? Bueno esto puede ser dado por dos factores. Por un lado dependerá de la cantidad de datos que disponemos, ya que si tenemos pocos será dificil hacer un modelo, es decir necesitaremos muchos más datos. El peligro aquí es que corremos el riesgo de que nuestras redes puedan memorizar, dando lugar al indeseado sobreajuste. En vez de eso, podemos tomar prestado un modelo preentrenado y ajustar unas pocas capas evitando esos problemas. En cambio si tenemos una alta cantidad de imágenes es posible hacer un modelo de la nada . Esto significa que la cantidad de datos determinarán si es posible hacer fine tunning o no. Por otro lado dependerá de como de parecidos son los datos, si los datos son muy parecidos será posible hacer un fine tunning en las ultimas capas, en cambio si fueran datos muy dispares haría falta preentrenar todo el modelo

Pasos para hacer una transferencia de conocimiento en Keras:

1. Organizar los datos.

2. Construir como va a ser el tratamiento de los datos

3. A falta de datos suficientes alterar los datos: rotarlos, aumentarlos...

4. Definir el modelo, coger uno preentrenado y eliminar las ultimas capas añadiendo unas nuevas capas clasificadoras.

5. Entrenar y probar.

Es importante distinguir entre entrenar, validar y probar. Nuestros datos se podrán distribuir en estos tres sectores, una distribución típica sería un 80 % para entrenamiento, un 10 % para validación y un 10 % para pruebas, es importante dividir estos datos aleatoriamente para evitar sesgos. La precisión final del modelo vendrá determinada por el conjunto de pruebas, el modelo aprenderá de los datos de prueba y usará los datos de validación para comprobar y mejorar su precisión. Aquí habrá muchas formas de las que podemos mejorar el modelo como cambiando el numero de capas de entrenamiento.

Existen dos formas de clasificar: clasificación binaria y clasificación multiclase.

La clasificación binaria consistirá en “es o no es”. Poniendo de ejemplo una clasificación de gatos, todos los gatos deberán ser clasificados como ”gatos” mientras que si aparece una imagen de un perro o un escritorio da igual lo que sea que será clasificado como “no gato”. Para ello podemos establecer un límite, aquellas probabilidades que sean menores al valor umbral del limite irán a una sección en cambio las mayores irán a otra(normalmente 0.5). En cambio en una clasificación multiclases no solo clasificaremos si algo “es o no es” si no que iremos más allá y diremos que es lo que estamos clasificando, así un perro irá a la sección de perros, un escritorio a la de escritorios...

Otra cosa que hay que tener en cuenta es el tamaño del lote que define cuantas imágenes son vistas por el modelo al mismo tiempo, siendo importante que cada lote tenga una buena variedad de imágenes de diferentes clases para evitar fluctuaciones en la precisión de las métricas en las distintas iteraciones. También es importante destacar que no debe ser muy grande ya que si excede el tamaño corremos el riesgo de que por ejemplo no quepa en la memoria de la GPU.

Una técnica importante es el aumento de los datos ya que nos da una forma de aumentar la cantidad de datos que tenemos utilizando distintas formas de crear estos nuevos datos como son la rotación, poner en espejo la imagen aleatoriamente o hacer zoom, combinando estas tres herramientas el programa puede generar un numero infinito de imágenes únicas.

Para transferir el aprendizaje debemos congelar los pesos del modelo original, esto es conservar las capas como no modificables, así solo las nuevas capas clasificadoras puedan ser modificables.

4.2.3. Entrenar el modelo

Para entrenar el modelo debemos seleccionar y modificar algunos parámetros de entrenamiento

Función de perdida Es el castigo que imponemos al modelo para las predicciones incorrectas durante el proceso de entrenamiento, Este será el valor que querremos minimizar. Función de costes.

Optimizador Es un algoritmo que ayuda a minimizar la función de perdida. Un ejemplo es el algoritmo Adam.

Tasa de aprendizaje Ayuda al optimizador a ver cuando la función de perdida es mínima. Depende de su valor esto lo hará en pasos pequeños o grandes.

Métrica Juzga la actuación del modelo entrenado. La precisión suele ser una buena métrica.

4.3. Construyendo un motor de búsqueda inversa de imágenes

La búsqueda inversa de imágenes (o recuperación de instancias) permite a los desarrolladores e investigadores construir escenarios a través de una simple palabra (como recomendaciones de Spotify o imágenes similares en Pinterest).

4.3.1. Similitud entre imágenes

Intenta dar respuesta a dadas dos imágenes ¿son similares?

Hay muchas aproximaciones a este problema, una sería comparar pequeñas áreas de las imágenes, y aunque esto pueda ayudar a encontrar en imágenes exactas o casi exactas, una ligera rotación puede dar lugar a pensar que son distintas. Guardando los hashes de las áreas, se pueden encontrar duplicados de la imagen. Una aplicación de esto podría ser determinar si hay plagio en una fotografía.

Otra aproximación es calcular el histograma de valores RGB y comparar sus similitudes. Esto puede ayudar a encontrar imágenes casi similares cogidas en el mismo entorno sin que tengan mucho cambio en los contenidos. Lo que pasa que esto puede dar lugar a falsos positivos o que pequeños cambios en el color, matices o en el balance de blancos, hacen el reconocimiento más complicado.

Un enfoque más tradicional es encontrar características visuales cerca de los bordes usando algoritmos como SIFT, SURF y ORB y luego comparar el número de características que son comunes en las dos fotos. Aunque esta técnica funciona bien con los objetos rígidos con poca variación como una caja de cereales, no ayuda mucho con objetos que puedan variar su forma como es el caso de personas o animales.

Yendo más profundo otra aproximación es encontrar la categoría de la imagen usando deep learning y después buscar en imágenes de la misma categoría, es equivalente a extraer los metadatos de una imagen para que luego pueda ser usada e indexada en una consulta típica en una búsqueda de texto. Esto es fácilmente escalable usando los metadatos en un motor de búsqueda libres. Como es de esperar, extrayendo esta categorización, perdemos una cierta información como color, posición, relaciones entre objetos en la misma imagen... Otra desventaja es que requiere de muchos datos etiquetados para entrenar a los clasificadores para etiquetar nuevas imágenes, y cada vez que se introduzca una nueva categoría necesita que el modelo se reentrene.

Debido a nuestro objetivo es buscar entre millones de imágenes, idealmente lo que necesitaremos el resumir la información de millones de píxeles de una imagen en una representación más pequeña, que nos ayude a distinguir si dos objetos son similares o no. Afortunadamente, las redes neuronales convolucionales pueden usar una imagen de entrada y convertirla en un vector de características, que luego actúe como entrada de un clasificador cuya salida identifique lo que hay en la imagen. Estos vectores de características son esencialmente una colección de valores en punto flotante. Este proceso es un acto de reducción, para filtrar la información en una constitución saliente que a su vez forma el vector de características también llamado característica de cuello de botella. Aquellos objetos que se clasifiquen en la misma clase a través de un modelo, tendrán una distancia Euclidea más pequeña que aquellos que pertenezcan a diferentes clases. Esta característica es muy importante para solucionar problemas cuando no se puede usar un clasificador.

Podemos adoptar dos estrategias para mejorar la búsqueda entre dos imágenes similares: reducir el tamaño del vector de características o usar un algoritmo mejor para buscar en el vector de características.

4.3.2. Longitud del vector de características

Casi todos los modelos generan un gran número de características. El objetivo es reducir el tamaño del vector para mejorar velocidad sin comprometer la calidad, a menos características, mayor velocidad. Otra gran mejora para escenarios de big data, si los datos son capaces de entrar en la RAM al mismo tiempo en vez de cargas periodicas de partes de ellos, nos daría una mejora en la velocidad. PCA nos ayuda a que esto pase.

PCA es un procedimiento estadístico que se pregunta cuales de las características que representan los datos son igual de importantes, preguntándose cuales de las características son redundantes. No elimina las características redundantes, en vez de eso genera un nuevo conjunto de características que son combinaciones de las características de entrada. Estas son ortogonales a otras que es por lo que las características redundantes se abstienen. Este nuevo vector es conocido como componentes principales. PCA también puede decirnos la importancia relativa de cada característica. El número de dimensiones de PCA es un parametro aportante que deberemos ajustar, encontrando un buen balance entre el numero de características y su efector entre la precisión contra la velocidad.

4.3.3. Escalando la búsqueda entre similares mediante la aproximación de algoritmos de vecinos más próximos

Partiremos de la línea base que usamos el algoritmo de búsqueda de fuerza bruta, ya que en la mayoría de aproximaciones algorítmicas suele ser el más lento. Una vez establecida esta linea base vamos a explorar todas las aproximaciones de algoritmos de vecinos más cercanos. La mayoría de estos algoritmos ofrecen alguna forma de ajustar el balance entre calidad y velocidad. Y siempre es posible comparar la calidad de los resultados con el resultado obtenido por fuerza bruta.

La librería se va a usar depende fuertemente del escenario. Cada librería presenta una compensación entre velocidad de búsqueda, precisión, tamaño e los índices, consumo de memoria, uso de hardware (CPU/GPU) y facilidad de configuración Dependiendo del escenario se recomienda:

- **Annoy o NMSLIB**: Si se quiere experimentar rápidamente en Python sin mucha configuración y también importa la velocidad.
- **NGT**: Si se tiene una gran base de datos y la velocidad importa.
- **Faiss**: Si la base de datos es muy grande y se tiene un cluster de GPUs
- **Fuerza bruta**: Si se quiere una precisión del 100

4.3.4. Mejorando la precisión mediante Fine Tuning

La mayoría de los modelos preeentrenados han sido entrenado en la base de datos de ImageNet. Esto les proporciona un punto de comienzo para cómputos similares en la mayoría de ocasiones. Si somos capaces de ajustar estos modelos para adaptarnos a problemas específicos conseguiremos aumentar la precisión de encontrar imágenes parecidas.

Una buena métrica para comprobar si se están obteniendo buenos resultados es el calculo de la precisión, basándose en que si una imagen pertenece a la categoría X, ¿las imágenes similares pertenecen a la misma categoría? Llamaremos a esto la precisión de similitud.

4.3.5. Fine Tuning sin capas completamente conectadas

Como sabemos las redes neuronales se componen de tres partes: las capas convolucionales que acaban generando los vectores de características, las capas completamente conectadas y las capas finales clasificatorias. El fine tuning se encarga de modificar un poco la red neuronal para adaptarla a un nuevo conjunto de datos, normalmente desmontando las capas totalmente conectadas sustituyéndolas con otras nuevas y después entrenar la red con este nuevo conjunto de datos. El entrenamiento de esta manera va a provocar que los pesos en todas las nuevas capas añadidas completamente conectadas sean afectados de forma significativa mientras que en las capas convolucionales solo se verán ligeramente modificados. Como resultado de estola red que generaba el vector de características tendrá cambios insignificantes, implicando que el vector apenas cambie.

Nuestro objetivo es que los objetos parecidos tengan un vector de características parecido. Forzando a que todas las tareas de entrenamiento ocurran en las capas convolucionales, veremos mejores resultados, esto lo podremos conseguir, borrando todas las capas completamente conectadas y poniendo una capa de clasificación directamente después de las capas convolucionales. Así el modelo estará optimizado para la búsqueda de imágenes similares en vez de para la clasificación.

4.4. Herramientas para maximizar la precisión de las redes neuronales convolucionales

Existen multitud de herramientas que nos ayudarán a reducir el código y el esfuerzo involucrado en la fase de experimentación mientras tratan de ganar visión intentando alcanzar una alta precisión. Algunas de ellas son:

- **TensorFlow Datasets** Proporciona una acceso rápido y sencillo a alrededor de 100 conjuntos de datos de manera eficiente obteniendo una forma de entrenar de alto rendimiento modelos basados en TensorFlow. Estos conjuntos de datos van desde las más pequeñas como MNIST a otras más grandes como ImagenNet o OpenImages. En vez de descargar y manipular estos conjuntos de datos manualmente y después figurarnos como leer sus etiquetas, esta herramienta estandariza el formato de los datos de esta forma es fácil intercambiar un conjunto de datos con otro, a menudo con un cambio en una sola línea de código. Hacer cosas como romper el conjunto de datos en datos de entrenamiento, validación y test es solo cuestión de una línea de código.

- **TensorBoard** Es una herramienta con cerca de 20 métodos fáciles de usar para visualizar diferentes aspectos del entrenamiento, incluyendo la visualización de gráficos, rastreo de experimentos y inspección de imágenes, texto y datos de audio a través de la red en su entrenamiento. Tradicionalmente, para seguir el progreso del experimento, debíamos guardar los valores de perdida y precisión por época y después hacer el gráfico utilizando matplotlib. La desventaja de esto es que no es en tiempo real. Nuestra opción habitual era ver el progreso de entrenamiento en el texto y después de que el entrenamiento hubiera acabado, debíamos escribir código para que se hiciera la gráfica. TensorBoard soluciona esto y más ofreciendo un tablero a tiempo real que nos ayuda a visualizar todos los logs asistiendo a entender la progresión del entrenamiento. otro beneficio que ofrece es la habilidad de comparar el progreso nuestros experimentos en curso con el de experimentos anteriores, de forma que podamos ver como un cambio en los parámetros afecta a la precisión global. Se debe dar cuenta que TensorBoard no es TensorFlow expreso, y puede ser usado con otros frameworks como Pythorch, scikit-learn y más dependiendo del plugin que utilicemos. Para hacer que un plugin funcione necesitamos escribir el metadato específico que queremos visualizar.

- **What-if tool** Ejecuta experimentos en paralelo de modelos distintos y muestra las diferencias en ambos comparando su rendimiento en puntos de datos específicos. Se pueden editar estos puntos para ver como afecta al entrenamiento del modelo. Esta herramienta desarrollada por Google People + AI Research (PAIR), nos ayuda a abrir la "caja negra" de los modelos de inteligencia artificial para permitir la explicación del modelos y los datos. Para usarla necesitaremos un conjunto de datos y el modelo.

- **tf-explain** Los modelos de deep learning tradicionalmente han sido cajas negras, y hasta ahora aprendíamos de su rendimiento viendo la probabilidad de las clases y validando su precisión. Para hacer estos modelos más interpretables y explicativos utilizamos los mapas de calor, que muestran el área de la imagen que conduce la predicción a través de las áreas de más intensidad, es decir los mapas de calor nos ayudan a visualizar el aprendizaje.

Los mapas de calor pueden ser especialmente útiles para explorar sesgos y filtrar falsas correlaciones del conjunto de datos si no es cuidado con atencion. tf-explain nos ayuda a entender estos resultados y trabaja inherentemente con las redes neuronales con la ayuda de estas visualizaciones, ayudándonos a intentar eliminar los sesgos del conjunto de datos. Se pueden utilizar diferentes aproximaciones visuales disponibles en tf.explain:

  - Grad CAM: Ayuda a visualizar como las partes de la imagen afectan a la salida de la red neuronal prestando atención a los mapas de activación. Los mapas de calor son generados basados en el gradiente del ID de objetos de la ultima capa convolucional. Grad CAM es un generados de mapas de calor de amplio espectro, robusto frente al ruido y puede ser usado en vectores de modelos de redes neuronales convolucionales.
  - Occlusion Sensitivity: Encapsula una parte de la imagen para establecer como de robusta es una red. Si la predicción es correcta, de media, la red es robusta. El área más cálida en la imagen tiene el mayor efecto de predicción cuando es ocluida.
  - Activations: Visualiza la activación para las capas convolucionales

- **Keras Tuner** Librería construida para tf.keras que activa automáticamente el ajuste de hiperparámetros en TensorFlow 2.0. Con muchas combinaciones potenciales de hiperparámetros de ajuste, dar con el modelo adecuado puede ser un proceso tedioso. Para ello Keras Tuner automatiza el proceso, definimos un algoritmo de búsqueda, los valores que cada parámetro pueda tener, lo que queremos maximizar y dejamos el programa ejecutándose. Keras Tuner hará múltiples experimentos cambiando los parámetros a nuestro favor y guardando los metadatos del mejor modelo.

- **AutoKeras** Automatiza la búsqueda de arquitectura neuronal (NAS) a través de diferentes tareas como imágenes, texto y clasificación de audio y detección de imágenes. Utiliza aprendizaje reforzado para escoger los mejores bloques de miniarquitecturas antes de ser capaz de maximizar el objetivo. Después del entrenamiento, podemos ver como es el nuevo modelo de entrenamiento.

- **AutoAugment** Utiliza aprendizaje reforzado para mejorar la cantidad y diversidad de datos de un conjunto de datos de entrenamiento existente y de este modo aumentar la precisión. Se basa en una inteligencia artificial que aprendiendo de las mejores combinaciones de parámetros de aumentos es capaz de aplicarlas a nuestros problemas.

4.4.1. Técnicas comunes de experimentación de Machine Learning

El mayor obstáculo a la hora de inspeccionar los datos es determinar la estructura de los mismos. Los conjuntos de datos de Tensor-Flow hacen este paso relativamente fácil por la cantidad de conjuntos y la estructura que usan en cuanto a rendimiento. Todo lo que tenemos que hacer es cargar el conjunto en la herramienta What-if Tool y usar varias opciones para inspeccionar los datos. Gracias a esto podemos detectar sesgos y solucionarlo modificando pesos de las métricas acordes a través de la herramienta.

Dividir el conjunto de datos en datos de entrenamiento, validación y test es muy importante porque lo que queremos es obtener los resultados en un conjunto de datos que no hemos visto por

medio de un un clasificador. Esto lo permite hacer de una manera sencilla TensorFlow Datasets permitiendo descargar, cargar y dividir los datos en esos tres conjunto, incluso algunos de ellos ya vienen con tres divisiones por defecto. Aunque algunos de ellos pueden venir sin el conjunto de validación, se puede escoger una muestra de los datos y usarlos para este fin.

La parada temprana nos ayuda a evitar el sobreentrenamiento de una red fijándonos en el numero de épocas que muestran una mejora limitada. En el momento que la red dee de mejorar durante una cantidad determinada de épocas debemos pararla para evitar que el gasto de recursos de entrenamiento vaya a más. Si el numero de épocas excede un limite predefinido llamado paciencia, el entrenamiento se para si hay más épocas de entrenamiento. En otras palabras la parada temprana decide el punto en el cual el entrenamiento deja de ser útil y lo acaba.

Al entrenar en diferentes veces con los mismo parámetros una red neuronal se observa que los resultados son ligeramente diferentes. Esto se debe a las variables aleatorias. Para hacer el experimento reproducible a través de las ejecuciones, debemos controlar esa aleatoriedad. Inicializar los pesos del modelo, aleatorizar la mezcla de datos, y continuando por algoritmos de aleatoriedad. Sabemos que los generadores de números aleatorios pueden ser reproducibles inicializándoles con una semilla por lo que es lo que haremos exactamente.

4.5. Clasificación de imágenes con fastai

PyTorch se tarta de la librería más flexible y expresiva para deep learning hoy en día, contando con la ventaja que no cambia velocidad por simplicidad, sino que proporciona ambas. Trabaja mejor como una librería de bajo nivel proporcionando las operaciones básicas para funcionalidad de alto nivel. La librería fastai es la librería más popular para añadir este alto nivel de funcionalidad encima de PyTorch.

Una vez adquirido los conocimiento básicos con Keras, el uso de fastai es más sencillo por ello vamos a ir viendo que hace cada línea en un programa de ejemplo donde se usa esta tecnología.

Lo primero que haremos será añadir todas las librerías necesarias esto lo haremos con:

```
from fastai.vision.all import *
```

Lo siguiente sera añadir un conjunto de datos para poder usarlo en la red y posteriormente definiremos una función para que etiquete, en nuestro caso gatos, basándose en la regla prevista por el creador del conjunto de datos

```
path = untar_data(URLs.PETS)/'images'
def is_cat(x): return x[0].isupper()
```

Esta función la usaremos en esta línea para decir a fastai que tipo de conjunto de datos tenemos y como está estructurado:

```
dls = ImageDataLoaders.from_name_func(
path, get_image_files(path), valid_pct=0.2, seed=42,
label_func=is_cat, item_tfms=Resize(224))
```

Donde lo que hacemos es usar la clase ImageDataLoaders, apta para imágenes, mediante la función la indicamos el path donde se encuentran estas imágenes, el tamaño en porcentaje que debe usar para validación que en este caso es un 20 por ciento de las imagenes, y la función de clasificación de estas imágenes. El ultimo parametro define la cantidad de transformaciones que estos elementos reciben, redimensionarlo, antes de meterlos al batch, en este caso un cuadrado de 224 píxeles.

Fastai siempre mostrará la precisión de tu modelo usando los datos de validación, nunca los de entrenamiento. Esto es imprescindible para que la red no memorice.

La siguiente línea de código, indicará que queremos entrenar a nuestra red creando una red neuronal convolucional y indicará la arquitectura que queremos, que datos queremos que entrene y que métricas vamos a usar.

```
learn = cnn_learner(dls, resnet34, metrics=error_rate)
```

En ella vemos que utilizamos la arquitectura resnet34 que indica que usamos ResNet de 34 capas, otras opciones son 18,50,101,152. Los modelos que utilizan arquitecturas con más capas tardan más tiempo en entrenarse, y son más propensos al sobreentrenamiento. Por otro lado, cuando se usan más datos, pueden ser bastante más precisos. También vemos que usamos la métrica de tasa de errores, función que provee fastai, y indica que porcentaje de imágenes del conjunto de validación no ha clasificado bien. Otra métrica comun para clasificar es la precisión (accuracy).

La función cnn_learner también tiene un parámetro pretrained que esta declarado a True por defecto, que establece los pesos en su modelo a valores que ya han sido entrenados por expertos para reconocer mil categorías diferentes a través de 1,3 millones de fotos (usando el famoso conjunto de datos ImageNet). Un modelo que tiene pesos que ya han sido entrenados en algún otro conjunto de datos se llama un modelo pre-entrenado. Casi siempre se debe usar un modelo preentrenado, porque significa que el modelo, antes de que le haya mostrado siquiera alguno de los datos, ya es muy capaz. Esto permitirá utilizar la técnica conocida como transferencia de conocimiento.

La última línea indica a fastai como ajustar nuestro modelo:

```
learn.fine_tune(1)
```

Estas líneas de código son solo una pequeña parte del proceso de usar deep learning. Sin embargo las mismas seis líneas de código no funcionarán para todos los problemas.

4.5.1. Como afrontar un problema mediante deep learning

Es importante tener en cuenta la forma en la que debemos enfocar un problema de este tipo, una de ellas puede ser la aproximación **"Drivetrain"** La idea básica es comenzar considerando su objetivo, luego pensar en qué acciones puede tomar para alcanzar ese objetivo y qué datos tiene (o puede adquirir) que puedan ayudar, y luego construir un modelo que pueda usar para determinar las mejores acciones a tomar para obtener los mejores resultados en términos de su objetivo.

Figura 4.1: Aproximación "Drivetrain" [1]

En este caso se va a usar una aproximación end-to-end, con una base de datos obtenida de la plataforma Kaggle en la que existen tres tipos de elementos basados en el juego "Piedra, papel o tijera", donde las imágenes están clasificadas por la forma de la mano en estos tres subgrupos, clasificados en tres carpetas diferentes.

Usaremos la clase **DataLoaders** que es una pequeña clase que almacena cualquier objeto **DataLoader** que le pasemos y le convertirá en un objeto de entrenamiento o de validación. Pese a que sea una clase muy simple se trata de una clase muy importante en fastai, ya que proporciona los datos al modelo.

A continuación a crear un DataLoaders para el conjunto de datos que vamos a utilizar

```
hand_types = 'scissors','paper','rock'
path = Path('Prueba')

hands = DataBlock(
blocks=(ImageBlock, CategoryBlock),
get_items=get_image_files,
splitter=RandomSplitter(valid_pct=0.2, seed=42),
get_y=parent_label,
item_tfms=Resize(128))
```

Para ello lo que hemos hecho es, primero indicar los tipos de manos en las que se pueden clasificar y dónde se localiza. Posteriormente hemos creado un **DataBlock** con lo los siguientes argumentos:

- blocks: Especifica que tipos queremos para las variables tanto dependientes como independientes. La variable independiente es aquella que estamos usando para hacer predicciones de algo mientras que la dependiente es nuestro objetivo. En este caso la variable independiente son las imágenes y la dependiente son las categorías de cada imagen (tipos de manos).
- get_items: En este caso nuestros objetos serán rutas de ficheros, así que tenemos que comunicar a fastai como obtener una lista de estos ficheros. La función get_image_files coge una ruta y devuelve una lista de imágenes que se encuentran en dicha ruta.
- splitter: A veces los conjuntos de datos se encuentran ya divididos y existe el conjunto de validación, esto se puede hacer poniendo los conjuntos de validación y entrenamiento en diferentes carpetas. fastai provee una aproximación general que permite usar una de sus calases predefinidas para tal fin, o que escribas la tuya propia. En este caso lo dividimos de manera aleatoria mediante RandomSplitter.

- get_y: normalmente a la variable independiente se la llama **x** y a la variable dependiente **y**. Con este parametro estamos indicando a fastai que función debe llamar para etiquetar nuestro conjunto de datos. parent_label es la función que se encarga que una vez dividamos las imagenes según su grupo en carpetas las etiquete según la carpeta en la que se encuentren.

- item_tfms: por último, esto se encarga que todas las imágenes estén dimensionadas de una forma uniforme.

Un **DataBlock** es una especie de plantilla para crear **DataLoaders**, por lo que aun necesitaremos decir a fastai cual es la fuente de nuestros datos, en este caso la ruta en la que se encuentran:

```
dls = hands.dataloaders(path)
```

También debemos tener en cuenta el uso de **Data Augmentation** tecnica que nos permitirá crear variaciones aleatorias para los datos aumentando así el número de datos disponibles mediante rotaciones, giros, distorsiones... Una forma de uso sería:

```
hands = hands.new(item_tfms=Resize(128),batch_tfms=aug_transforms(mult=2))
dls = hands.dataloaders(path)
```

Procederemos entonces a entrenar el modelo usando líneas similares a las seis que vimos al principio de la sección:

```
hands = hands.new(
    item_tfms=Resize(128),
    batch_tfms=aug_transforms())
    dls = hands.dataloaders(path, num_workers=0,batch_size=64)
```

Usaremos un Learner y lo ajustaremos mediante:

```
learn = cnn_learner(dls, resnet18, metrics=error_rate)
learn.fine_tune(3)
```

Una forma de la que podemos visualizar los errores del modelo, es mediante una matriz de confusión

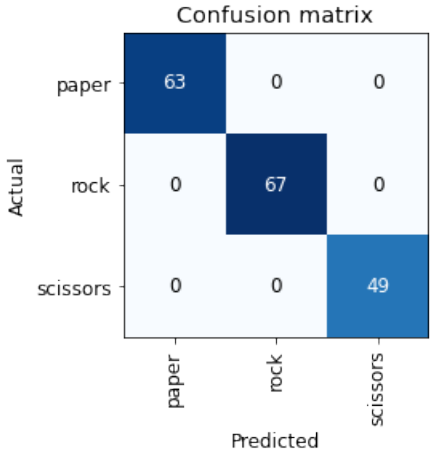

Figura 4.2: Matriz de confusión para ejemplo de piedra papel o tijera

En ella la diagonal de la matriz muestra las imagenes que se han clasificado correctamente y el resto de celdas las que han tenido error. Esta es una de las formas en las que fastai permite ver los resultados de tu modelo. Para ello debemos usar las siguientes líneas:

```
interp = ClassificationInterpretation.from_learner(learn)
interp.plot_confusion_matrix()
```

También es útil ver donde exactamente han ocurrido los errores para ver si es un problema del conjunto de datos o del modelo, por ello podemos clasificar las imágenes en función del valor de perdida. Usando la función "plot_top_losses" podemos mostrar las imagenes con el mayor valor de perdida en nuestro conjunto de datos.

```
interp.plot_top_losses(5, nrows=1)
```

La aproximación más intuitiva nos dirá que debemos limpiar los datos antes de entrenar al modelo, pero a veces, un modelo puede ayudar a encontrar problemas más rápido y de forma más fácil. Así que preferiremos entrenar un modelo rápido y sencillo y posteriormente usarlo para que nos ayude a limpiar datos. fastai dispone de una interfaz gráfica para limpiar datos llamada ImageClassifierCleaner que permite escoger una categoría y el conjunto y ver las imágenes con más perdida permitiendo reetiquetarlas o eliminarlas.

Por último debemos tomar una serie de precauciones para evitar el desastre en nuestro proyecto ya que entender y probar el comportamiento de un modelo de aprendizaje profundo es mucho más difícil que con la mayoría de los otros códigos que escribes ya que con el desarrollo normal de software puedes analizar los pasos exactos que el software está dando, y estudiar cuidadosamente cuál de estos pasos se ajusta al comportamiento deseado que estás tratando de crea, cosa que no ocurre con una red neuronal donde su comportamiento emerge del intento del modelo de emparejar los datos de entrenamiento, en lugar de estar exactamente definido.

4.5.2. Clasificación de imágenes

Una vez visto como crear un modelo con fastai, es momento de empezar a profundizar para que un modelo funcione de forma fiable hay detalles que tienes que comprobar. Este proceso requiere ser capaz de mirar dentro de tu red neuronal mientras se entrena, y mientras hace predicciones, encontrar posibles problemas, y saber cómo solucionarlos.

Para el entrenamiento lo primero que vamos a necesitar es el conjunto de datos que normalmente se dispondrá de la siguiente manera:

- Archivos individuales que representan elementos de datos, como documentos de texto o imágenes, posiblemente organizados en carpetas o con nombres de archivo que representan información sobre esos elementos.
- Un cuadro de datos, por ejemplo en formato CSV, en el que cada fila es un elemento que puede incluir nombres de archivo que proporcionan una conexión entre los datos del cuadro y los datos en otros formatos, como documentos de texto e imágenes.

Aunque hay excepciones a estas reglas, por ejemplo en dominios como la genómica, donde puede haber formatos de bases de datos binarios, pero en general la gran mayoría de los conjuntos de datos trabajan con combinaciones de los dos anteriores.

En muchos casos necesitaremos etiquetar las imágenes del conjunto de datos en el nombre del archivo. Para hacer esto deberemos usar las **expresiones regulares** que nos permitirán obtener parte de la cadena que forma el nombre, fastai viene con muchas clases para ayudar con el etiquetado. Para etiquetar con expresiones regulares, podemos usar la clase RegexLabeller como en el ejemplo que se muestra a continuación:

```
pets = DataBlock(blocks = (ImageBlock, CategoryBlock),
             get_items=get_image_files,
             splitter=RandomSplitter(seed=42),
             get_y=using_attr(RegexLabeller(r'(.+)_\d+.jpg$'), 'name'),
             item_tfms=Resize(460),
             batch_tfms=aug_transforms(size=224, min_scale=0.75))
dls = pets.dataloaders(path/''images'', num_workers=0)
```

Es importante fijarse en las dos ultimas lineas que componen el DataBlock en ellas se usa una estrategia de aumento llamada presizing, que es una forma de hacer "data augmentation" minimizando la destruccion de datos a la vez que manteniendo un buen rendimiento

Presizing

Necesitamos que nuestras imágenes tengan las mismas dimensiones, para que puedan cotejarse en tensores para ser pasadas a la GPU. También queremos minimizar el número de cálculos de aumento distintos que realizamos. El requisito de rendimiento sugiere que deberíamos componer nuestras transformaciones de aumento en menos transformaciones (para reducir el número de cálculos y el número de operaciones con pérdidas) y transformar las imágenes en tamaños uniformes (para un procesamiento más eficiente en la GPU).

El problema es que, si se realizan después de reducir el tamaño al tamaño aumentado, varias transformaciones comunes de aumento de datos podrían introducir zonas vacías, degradado los datos o ambas cosas. Muchas operaciones de rotación y zoom requerirán interpolar para crear píxeles. Estos píxeles interpolados se derivan de los datos de la imagen original pero siguen siendo de menor calidad.

Presizing adopta os estrategias para solventar esto:

1. Redimensionar las imágenes a dimensiones relativamente "grandes" para que tengan margen de sobra para permitir transformaciones de aumento en sus regiones internas sin crear zonas vacías. En el ejemplo anterior esto se ve en **item_tfms**, y nos asegura que todas las imágenes tienen el mismo tamaño. En el conjunto de entrenamiento las regiones internas se eligen al azar, en el set de validación en cambio se elige siempre la central

2. Componer todas las operaciones de aumento comunes (incluido el cambio de tamaño al tamaño final del objetivo) en una sola, y realizar la operación combinada en la GPU sólo una vez al final del procesamiento, en lugar de realizar las operaciones de forma individual e interpolando varias veces. Con esto se consigue una sola interpolación minimizando la destrucción de datos. Esto se aplica en el ejemplo en **batch_tfms**. En el caso de las del conjunto de validación el cambio de tamaño final necesario del modelo se hace aquí, en el caso del conjunto de entrenamiento cualquier cambio o recorte se hace primero.

Para implementar este proceso en fastai se usa "Resize" como una transformación de ítem con un tamaño grande, y "RandomResizedCrop" como una transformación de lote con un tamaño más pequeño. "RandomResizedCrop" se añadirá si incluyes el parámetro "min_scale" en tu función "aug_transforms", como se hizo en la llamada a "DataBlock" en la sección anterior.

Comprobando y depurando un DataBlock

No se puede asumir que nuestro código está funcionando como queremos, no existe la garantía de que vaya a funcionar como pretendes. Así que antes de entrenar un modelo se deben comprobar los datos. Esto se puede hacer usando el método "show_batch":

```
dls.show_batch(nrows=1, ncols=3)
```

Debemos echar un vistazo a estas imágenes y comprobar que la etiqueta que vemos es la correcta. En este paso veremos si hemos cometido un error al crear el DataBlock. Para depurar esto podemos usar el método "summary", que intentará crear un lote a partir de una fuente dado con muchos detalles. En el caso de fallar aparecerá exactamente en que punto está ocurriendo el error y mostrará alguna ayuda.

Una vez que creemos que los datos están bien, deberemos usarlos para entrenar un modelo simple. No es recomendable posponer el entrenamiento de un modelo real durante demasiado tiempo, ya que no podremos cotejarlo con resultados de referencia.

Lo siguiente que debemos hacer es interpretar las funciones de perdida lo cual es complicado ya que están diseñadas para que los ordenadores puedan diferenciarlas y optimizarlas, pero resultan dificiles de entender para una persona. Es por ello que tenemos métricas, para ayudar a interpretar

estas funciones. En la matriz de confusión podemos ver en que caso el modelo lo esta haciendo bien y en que caso mal. El problema reside cuando el número de categorías es muy alto que hará que nuestra matriz sea enorme. En esos casos podemos usar el método "most_confused" que mostrará que celdas de la matriz son las que más predicciones incorrectas tienen:

```
interp.most_confused(min_val=5)
```

Con esto tenemos una buena base, pero tendremos un modelo que puede ser mejorable veamos a continuación algunas técnicas que nos permitan que nuestro modelo mejore. Lo primero que tenemos que establecer a la hora de entrenar un modelo es la tasa de aprendizaje, fastai proporciona una herramienta para encontrar la más eficiente. La función "learn.lr_find()" implementa lo que se denomina un buscador de tasas de aprendizaje, la idea es empezar con un ritmo de aprendizaje muy pequeño y lo usamos para un mini lote, encontramos cuáles son las pérdidas, y aumentamos la tasa de aprendizaje en algún porcentaje (por ejemplo, duplicándola cada vez). Luego hacemos otro mini lote, rastreamos la pérdida, y duplicamos la tasa de aprendizaje de nuevo. Seguimos haciendo esto hasta que la pérdida empeora, en lugar de mejorar. Este es el punto en el que sabemos que estamos en torno a un valor equivocado. Entonces seleccionamos una tasa de aprendizaje un poco más baja que este punto. Podemos obtener dos valores:

- Un orden de magnitud menor que el lugar donde se alcanzó la pérdida mínima
- El último punto en el que la pérdida estaba claramente disminuyendo

Aplicaremos esta tasa en la función "fine_tune" mediante el atributo base_lr:

```
learn = cnn_learner(dls, resnet34, metrics=error_rate)
learn.fine_tune(2, base_lr=3e-3)
```

Ahora que tenemos una buena tasa de aprendizaje para entrenar nuestro modelo, veamos como podemos ajustar los pesos de un modelo preentrenado. Sabemos que una red neural convolucional consiste en muchas capas lineales con una función de activación no lineal entre cada par, seguida de una o más capas lineales finales con una función de activación como softmax en el extremo. La capa lineal final utiliza una matriz con suficientes columnas de tal manera que el tamaño de salida es el mismo que el número de clases en nuestro modelo (suponiendo que estamos haciendo una clasificación). Es poco probable que esta capa lineal final nos sea de utilidad cuando estemos haciendo un ajuste fino en un entorno de aprendizaje por transferencia, porque está diseñada específicamente para clasificar las categorías en el conjunto de datos original de preformación. Por lo tanto, cuando realizamos el aprendizaje por transferencia lo eliminamos, lo tiramos y lo reemplazamos por una nueva capa lineal con el número correcto de salidas para nuestra tarea deseada.

Nuestro problema a la hora de hacer los ajustes es sustituir los pesos aleatorios en nuestras capas lineales añadidas por pesos que logren correctamente nuestra tarea deseada sin cambiar los pesos preentrenados de las otras capas. Para ello debemos decirle al optimizador que sólo actualice los pesos en esas capas finales añadidas al azar y no cambie los pesos del resto de la red neuronal.

A esto se le llama congelar esas capas pre-entrenadas. Cuando creamos un modelo de una red pre-entrenada, Fastai automáticamente congela todas las capas pre-entrenadas para nosotros. Cuando llamamos al método fine_tune, fastai hace dos cosas:

- Entrena las capas añadidas aleatoriamente para una época, con todas las demás capas congeladas
- Descongela todas las capas, y las entrena todas para el número de épocas solicitadas

Aunque se trata de un enfoque razonable, es probable que para un conjunto de datos particular se pueda obtener mejores resultados haciendo las cosas de forma diferente. El método fine_tune tiene un número de parámetros que se puede usar para cambiar su comportamiento. Por ejemplo podemos usar alternativamente fit_one_cycle que lo que hace es empezar a entrenar a una tasa de aprendizaje baja, aumentarla gradualmente para la primera sección de entrenamiento, y luego volverla a disminuir gradualmente para la última sección de entrenamiento. Después descongelaríamos el modelo y volveríamos a utilizar el buscador de tasa de aprendizaje porque tener más capas que entrenar, y pesos que ya han sido entrenados durante tres épocas, significa que nuestro ritmo de aprendizaje previamente encontrado ya no es apropiado, el gráfico obtenido será diferente al anterior, deberemos escoger u punto anterior al aumento brusco y entrenar con una tasa de entrenamiento adecuada

```
learn = cnn_learner(dls, resnet34, metrics=error_rate)
learn.fit_one_cycle(3, 3e-3)
learn.unfreeze()
learn.lr_find()
learn.fit_one_cycle(6, lr_max=1e-5)
```

Otra forma de mejorar el modelo es modificando las capas mas profundas de nuestro modelo preentrenado ya que no necesitan una tasa de aprendizaje tan alta como las últimas, por lo que probablemente deberíamos usar diferentes tasas de aprendizaje para estos - esto se conoce como usar tasas de aprendizaje discriminatorias, enfoque que usa fastai por defecto. fastai permite pasar un objeto slice de Python en cualquier lugar donde se espere una tasa de aprendizaje. El primer valor que se pase será la tasa de aprendizaje en la capa más temprana de la red neural, y el segundo valor será la tasa de aprendizaje en la capa final. Las capas intermedias tendrán tasas de aprendizaje que serán equidistantes a lo largo de ese rango. Para ello siguiendo nuestro ejemplo cambiaremos el argumento **lr_max** por **lr_max=slice(1e-6,1e-4)**. Con esto obsermaos que mejoraremos la perdida de entrenamiento, pero eventualmente la mejora de la perdida de validación es más lenta e incluso puede ser peor. En ese momento el modelo está empezando a sobreentrenarse, el modelo se está volviendo demasiado confiado en sus predicciones. Pero esto no tiene porque significar que se esté volviendo menos preciso. Al final lo que importa es su precisión, o más generalmente su métrica elegida, no la pérdida. La pérdida es sólo la función que le hemos dado al ordenador para ayudarnos a optimizar.

Otra decisión importante que hay que tomar es sobre el número de épocas de entrenamiento. Un primer enfoque para entrenar debería ser simplemente elegir un número de épocas que entrenarán

una cantidad de tiempo que estés dispuesto a esperar. A continuación, observaremos los gráficos de pérdidas de entrenamiento y validación, y en particular las métricas, si se ve que siguen mejorando incluso en las últimas épocas, entonces sabremos que no hemos entrenado durante demasiado tiempo.

Una ultima aproximación para la mejora del modelo es escoger arquitecturas más profundas. En general, un modelo con más parámetros puede modelar sus datos con más precisión, en la práctica, las arquitecturas tienden a venir en un pequeño número de variantes, por ejemplo, la arquitectura ResNet que estamos usando en este capítulo viene en variantes con 18, 34, 50, 101, y 152 capas, pre-entrenadas en ImageNet, aunque debemos tener en cuenta que una versión más grande de una ResNet siempre podrá darnos una menor pérdida de entrenamiento, pero puede sufrir más de sobreajuste, porque tiene más parámetros con los que sobreajustarse.En general, un modelo más grande tiene la capacidad de captar mejor las relaciones reales subyacentes en sus datos, y también de captar y memorizar los detalles específicos de sus imágenes individuales.

4.5.3. Clasificación de multietiqueta y regresión

Clasificación de multietiqueta

Cuando hablamos de la clasificación de multietiqueta nos referimos al problema de la identificación de las categorías de objetos en las imágenes que no contienen un solo tipo de objeto. Puede haber un tipo de objeto o no haber objeto en absoluto en las clases que está buscando. Es decir hablamos de que pasaría si introducimos una imagen de un lápiz en un clasificador de osos, tal y como hemos visto hasta ahora nos lo clasificaría como un tipo de oso.

Para ello construiremos el DataBlock cambiando el parametro "blocks" en el que cambiamos el CategoryBlock que devuelve un solo entero, por MultiCategoryBlock, que nos permitirá tener múltiples etiquetas para cada objeto:

```
dblock = DataBlock(blocks=(ImageBlock, MultiCategoryBlock),
                   get_x = get_x, get_y = get_y)
```

De esta forma la lista de categorías no está codificada de la misma manera que la del CategoryBlock, en ella había un solo número entero que representaba la categoría que estaba presente, basado en su ubicación en nuestras categorías, ahora tendremos una lista de ceros, con un uno en cualquier posición donde esa categoría esté presente. Esto se conoce como "one-hot encoding".

Una vez creado este DataBlock, estaremos listos para entrenar al modelo, nada va a cambiar cuando creemos nuestro Learner, pero por detras fastai elegirá una nueva función de pérdida para nosotros, la entropía cruzada binaria.

También es posible que queramos cambiar las métricas a la hora de entrenar nuestro modelo para por ejemplo usar métricas acordes a nuestro problema, ese es el caso de **accuracy_multi**, ya que la métrica **accuracy** no funcionará ya que podemos tener más de una predicción en una sola imagen. Esta métrica requiere de un valor umbral para que por debajo al aplicar a las activaciones la función sigmoide sea capaz de decidir cuales son 0s y cuales 1s. Podemos ver u ejemplo e como se usa a continuación:

```
learn = cnn_learner(dls, resnet50, metrics=partial(accuracy_multi, thresh=0.2))
```

Elegir un umbral es importante, ya que si eliges un umbral demasiado bajo, no podrás seleccionar correctamente los objetos etiquetados y si eliges un umbral demasiado alto, sólo seleccionarás los objetos para los que tu modelo tiene mucha confianza. Podemos encontrar el mejor umbral probando algunos niveles y ver qué es lo que mejor funciona. Esto es mucho más rápido si tomamos las predicciones una vez, y luego podemos llamar directamente a la métrica:

```
preds,targs = learn.get_preds()
accuracy_multi(preds, targs, thresh=0.9, sigmoid=False)
```

Podemos realizar la gráfica mediante esta aproximación para encontrar el mejor valor del umbral.

Regresión

Un modelo se define por sus variables independientes y dependientes, junto con su función de pérdida. Podemos tener una variable independiente que es una imagen, y una dependiente que es un texto (por ejemplo, la generación de un pie de foto a partir de una imagen); o tal vez tenemos una variable independiente que es un texto y una dependiente que es una imagen (por ejemplo, la generación de una imagen a partir de un pie de foto) o tal vez tenemos imágenes, textos y datos tabulares como variables independientes, y estamos tratando de predecir las compras de productos ... las posibilidades realmente son infinitas. consideremos el problema de la regresión de imágenes. Se trata de aprender de un conjunto de datos en el que la variable independiente es una imagen, y la variable dependiente es uno o más floats.

Vamos a tratar esto desde una perspectiva de modelo de puntos clave. Un punto clave se refiere a una ubicación específica representada en una imagen. Por ejemplo buscar el centro de la cara de la persona en cada imagen, esto quiere decir que en realidad predeciremos dos valores para cada imagen: la fila y la columna del centro de la cara.

4.5.4. Técnicas avanzadas para entrenar un modelo de clasificación de imágenes

Normalización

Cuando se entrena un modelo, es útil que los datos de entrada estén normalizados, es decir, que tengan una media de 0 y una desviación estándar de 1. Pero la mayoría de las bibliotecas de imágenes utilizan valores entre 0 y 255 para los píxeles, o entre 0 y 1; en cualquier caso, los datos no van a tener una media de 0 y una desviación estándar de 1. Afortunadamente, normalizar los datos es fácil de hacer en fastai añadiendo la transformación "Normalize". Esto actúa en un mini lote cada vez, así que puedes añadirlo a la sección "batch_tfms" del bloque de datos, es necesario pasar a esta transformación la media y la desviación estándar que se desea utilizar. A continuación se ve un ejemplo:

```
dblock = DataBlock(blocks=(ImageBlock, CategoryBlock),
                   get_items=get_image_files,
                   get_y=parent_label,
                   item_tfms=Resize(460),
```

```
                batch_tfms=[*aug_transforms(size=size, min_scale=0.75),
                            Normalize.from_stats(*imagenet_stats)])
```

La normalización se vuelve especialmente importante a la hora de usar modelos preeentrenados, ya que estos solo saben trabajar con datos del tipo que ha visto antes. Si el valor medio de los píxeles era 0 en los datos con los que fue entrenado, pero sus datos tienen 0 como el mínimo valor posible de un píxel, el modelo va a ver algo muy diferente a lo que se pretende. Esto significa que cuando distribuyes un modelo, necesitas distribuir también las estadísticas usadas para la normalización ya que cualquiera que lo use para inferir, o transferir el aprendizaje, necesitará usar las mismas estadísticas. De la misma manera, si estás usando un modelo que alguien más ha entrenado, asegúrate de averiguar qué estadísticas de normalización usaron, y hazlas coincidir.

Redimensionamiento progresivo

El enfoque de redimensionamiento progresivo consiste en pasar la mayor parte de las épocas entrenando con imágenes pequeñas haciendo que el entrenamiento se complete mucho más rápido y luego completarle utilizando imágenes grandes haciendo que la precisión final sea mucho mayor. Debemos tener en cuenta que existen diferencias entre las imágenes pequeñas y las grandes, lo que hará que nuestro modelo no funcione de la misma manera, al igual que pasaba con la transferencia de aprendizaje, por tanto usaremos el método fine_tune después de cambiar el tamaño de nuestras imágenes. Existe una nueva ventaja del uso de este método, la utilización de una nueva forma de aumento de datos.

Para implementar el redimensionamiento progresivo primero usaremos un tamaño de imágenes más pequeños y entrenaremos la red, una vez que ya haya sido entrenada reemplazaremos el DataLoader por uno con un tamaño de imagen más grande y usaremos la función fine_tune.

Hay que tener en cuenta que para el aprendizaje por transferencia, el cambio de tamaño progresivo puede perjudicar el rendimiento. Esto es más probable que ocurra si el modelo preentrenado era bastante similar a su tarea de aprendizaje por transferencia y al conjunto de datos y fue entrenado en imágenes de tamaño similar, por lo que los pesos no necesitan ser cambiados mucho. Por otro lado, si la tarea de aprendizaje de transferencia va a utilizar imágenes de tamaños, formas o estilos diferentes a los utilizados en la tarea de preentrenamiento, el cambio de tamaño progresivo probablemente ayudará.

Aumento del tiempo de prueba

Se trata de durante la validación, la creación de múltiples versiones de cada imagen, utilizando el aumento de datos, y luego tomando el promedio o el máximo de las predicciones para cada versión aumentada de la imagen. Dependiendo del conjunto de datos, el aumento del tiempo de prueba puede suponer una mejora drástica de la precisión. No cambia en absoluto el tiempo necesario para el entrenamiento, pero aumentará la cantidad de tiempo necesario para la validación. Para implementarlo usaremos el método “tta”:

```
preds,targs = learn.tta()
```

Mixup

El Mixup se trata de una técnica de aumento de datos muy potente que puede proporcionar una precisión drásticamente mayor, especialmente cuando no tienes muchos datos y no tienes un modelo preentrenado que fue entrenado en datos similares a tu conjunto de datos. Mixup funciona de la siguiente manera, para cada imagen:

1. Selecciona otra imagen aleatoria del conjunto de datos
2. Selecciona un peso aleatorio
3. Toma una media ponderada de la imagen seleccionada con la imagen original ésta será la variable independiente.
4. Toma una media ponderada de las etiquetas de la imagen seleccionada con la imagen original ésta será la variable dependiente.

Es decir realiza una combinación linear de imágenes.

Para que realice esto debemos introducir al Learner el siguiente parámetro "cbs=MixUp()".

Mixup requiere muchas más épocas de entrenamiento para obtener una mayor precisión, evidentemente, va a ser más difícil de entrenar, porque es más difícil ver lo que hay en cada imagen. Además, el modelo tiene que predecir dos etiquetas por imagen, en lugar de sólo una, y tiene que calcular la ponderación de cada una. Sin embargo, parece menos probable que el sobreajuste sea un problema, porque no mostramos la misma imagen en cada época, sino que mostramos una combinación aleatoria de dos imágenes.

Un problema con esto, es que Mixup está haciendo las etiquetas más grandes que 0, o más pequeñas que 1. Es decir, no estamos diciendo explícitamente a nuestro modelo que queremos cambiar las etiquetas de esta manera. Por lo tanto, si queremos que las etiquetas estén más cerca o más lejos de 0 y 1, tenemos que cambiar la cantidad de Mixup, lo que también cambia la cantidad de aumento de datos.Hay una manera de manejar esto más directamente, que es utilizar el suavizado de etiquetas.

Suavizado de etiquetas

El modelo está entrenado para devolver 0 para todas las categorías excepto una, para la que está entrenado para devolver 1. Incluso 0,999 no es "suficientemente bueno", el modelo obtendrá gradientes y aprenderá a predecir activaciones con una confianza aún mayor. Esto fomenta el sobreajuste y le da en el momento de la inferencia un modelo que no va a dar probabilidades significativas: siempre dirá 1 para la categoría predicha incluso si no está muy seguro, sólo porque fue entrenado de esta manera. En su lugar, podríamos sustituir todos nuestros 1s por un número un poco menor que 1, y nuestros 0s por un número un poco mayor que 0, y luego entrenar. Esto se llama suavizar la etiqueta. Al animar a tu modelo a ser menos confiado, el suavizado de etiquetas hará que tu entrenamiento sea más robusto, incluso si hay datos mal etiquetados. El resultado será un modelo que generaliza mejor.

Para utilizar esto en la práctica, sólo tenemos que cambiar la función de pérdida en nuestra llamada a Learner por "loss_func=LabelSmoothingCrossEntropy()".

4.5.5. ResNets

En esta sección se hablará de la arquitectura de "red residual" ResNet una de las más utilizadas actualmente, introducida en 2015 en el artículo "Deep Residual Learning for Image Recognition".

Los autores de este articulo observaron que incluso después de normalizar el batch, las redes que utilizaban más capas lo hacían peor que las que utilizaban menos capas, sin haber diferencia entre los modelos. Esta diferencia no se daba solo en el conjunto de validación sino también en el de entrenamiento, por tanto no era solo un problema de generalización sino también de entrenamiento. Esto se observa en el siguiente gráfico:

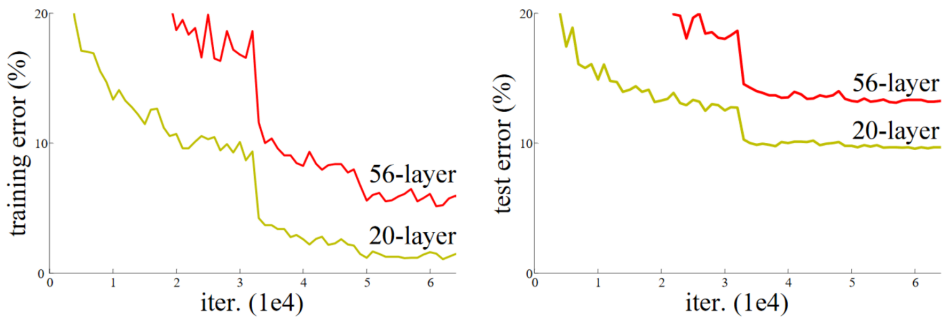

Figura 4.3: Error con arquitecturas más profundas [1]

Para la solución de este problema propusieron una solución para la construcción del modelo más profundo, que las capas añadidas fueran mapeos de identidad, devolver la entrada sin cambiar, esto se realiza por medio de la función identidad, y las capas antiguas se copian del modelo anterior. Una explicación más sencilla del concepto es la siguiente: se parte de una red neuronal de n capas bien entrenada y se añaden m capas que no hacen nada en absoluto, por ejemplo capas lineales con peso igual a 1 y un sesgo igual a 0, con ello tendremos una red de n+m capas que hace lo mismo que una red de n capas, demostrando así que existen redes más profundas que son al menos tan buenas que cualquier red menos profunda, pero que por algún motivo el algoritmo de descenso de gradiente estocástico no es capaz de encontrarla.

Existen otras formas de crear estas m capas, que resultan más interesantes. Podemos reemplazar cada ocurrencia de conv(x) con x + conv(x) siendo conv la función que añade la segunda convolución, luego un ReLU y por último una capa de normalización del batch. La normalización del batch es: gamma*y + beta, si inicializamos gamma a cero par cada una de estas capas finales, entonces conv(x) para estas m capas siempre será igual a cero, lo que significa que conv(x)+x siempre será igual a x. Esto hace que las m capas extras sean un mapeo de identidad, pero tiene parámetros luego es entrenable. Así podemos empezar con un modelo de n capas añadir m capas que no hacen nada, y luego afinar el modelo completo de n+m capas. Estas m capas adicionales pueden entonces aprender los parámetros que los hacen más útiles. En el documento de ResNets se propone una variante de esto que consiste en saltarse cada segunda convolución por lo que obtenemos x+conv2(conv1(x)). Como se muestra en la siguiente imagen:

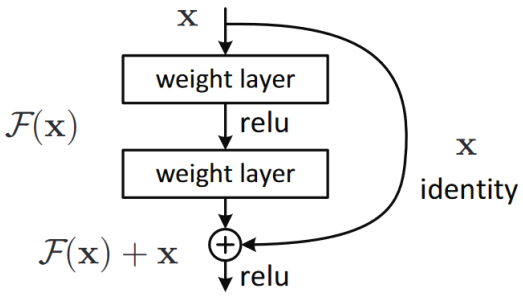

Figura 4.4: [1]

La flecha de la derecha es sólo la parte de x de x+conv2(conv1(x)) y se conoce como la identidad o conexión de salto.

En una ResNet, no entrenamos un número menor de capa y luego añadimos nuevas al final y afinamos. Sino que utilizamos bloques ResNet en toda la red neuronal convolucional, inicializamos desde cero y entrenamos mediante gradiente descendente estocástico y nos basamos en las conexiones de salto para que la red sea más fácil de entrenar.

Hay otra forma de pensar sobre estos bloques ResNet que se describen de la siguiente forma: en lugar de esperar que cada una de las capas apiladas se ajuste directamente a una cartografía subyacente deseada, dejamos explícitamente que estas capas se ajusten a una cartografía residual. Formalmente, denotando la cartografía subyacente deseada como H(x), dejamos que las capas no lineales apiladas se ajusten a otra cartografía de F(x) := H(x)-x. El mapeo original se refunde en F(x)+x. Nuestra hipótesis es que es más fácil optimizar el mapeo residual que optimizar el mapeo original no referenciado. En el extremo, si un mapeo de identidad fuera óptimo, sería más fácil empujar el residual a cero que ajustar un mapeo de identidad por una pila de capas no lineales.

Dicho de otro modo si el resultado de una capa determinada es x, cuando se utiliza un bloque de ResNet que devuelve y = x+bloque(x) no le estamos pidiendo que prediga y, sino que prediga la diferencia entre y,x. Así pues, el trabajo de esos bloques no es predecir ciertas características, sino minimizar el error entre x y el y deseado. De ahí el nombre ya que son residuales de predicción, donde residual es la predicción menos el objetivo.

Un concepto clave de esto es la facilidad de aprendizaje ya que según el teorema de aproximación universal una red suficientemente grande puede aprender cualquier cosa. Esto sigue siendo cierto pero hay una diferencia muy importante entre lo que una red puede aprender en principio y lo que es fácil que aprenda con datos y regímenes de entrenamiento realistas.

Una definición de un simple blocque Resnet podría ser:

```
class ResBlock(Module):
def __init__(self, ni, nf):
    self.convs = nn.Sequential(
        ConvLayer(ni,nf),
        ConvLayer(nf,nf, norm_type=NormType.BatchZero))

def forward(self, x): return x + self.convs(x)
```

Donde norm_type=NormType.BatchZero haciendo que fastai inicie los pesos gamma de la ultima capa de normalización de batch a cero.

No obstante hay dos problemas con esto, no puede manejar una zancada que no sea 1, y requiere que ni==nf.

Un ResBlock que soluciona estos problemas es el siguiente:

```
def _conv_block(ni,nf,stride):
    return nn.Sequential(
        ConvLayer(ni, nf, stride=stride),
        ConvLayer(nf, nf, act_cls=None, norm_type=NormType.BatchZero))

class ResBlock(Module):
 def __init__(self, ni, nf, stride=1):
    self.convs = _conv_block(ni,nf,stride)
    self.idconv = noop if ni==nf else ConvLayer(ni, nf, 1, act_cls=None)
    self.pool = noop if stride==1 else nn.AvgPool2d(2, ceil_mode=True)

 def forward(self, x):
    return F.relu(self.convs(x) + self.idconv(self.pool(x)))
```

Gracias a estos descubrimiento los autores del artículo sobre ResNet ganaron el desafío ImageNet de 2015. Desde que se introdujo la ResNets, se ha estudiado y se ha utilizado en muchos sitios. Uno de los trabajos más interesantes es el de Hao Li "Visualizing the Loss Landscape of Neural Nets". En el se muestra que el uso de conexiones de salto ayuda a suavizar la función de pérdida facilitando el entramiento evitando caer en una zona muy marcada. Nuestro modelo ya e slo suficientemente bueno pero gracias a algunos estudios posteriores se han descubierto más trucos que se pueden aplicar para hacerlo mejor.

4.5.6. Aplicando arquitecturas en Deep Learning

Para la vision por ordenador usamos las funciones cnn_learner y unet_learner aunque para diferentes propósitos, la primera para clasificación de imágenes mientras que la segunda para problemas de segmentación. Veamos como se construye un objeto Learner utilizado en otras secciones

cnn_learner

Cuando utilizamos esta función la pasamos una arquitectura para usar en la red, en su mayoría ResNets, los pesos preentrenados se descargan y se cargan en ella. Después para el transfer learning, hay que dividir la red, es decir eliminar las últimas capas, todas aquellas posteriores a la capa de agrupación media adaptativa, que incluye la última, encargada de categorizar ImageNet. Dado que no todas las arquitecturas son iguales esto no siempre ocurre así, en su lugar, existe una información que se utiliza para en cada modelo determinar dónde termina su cuerpo, y comienza su cabeza. Esto se puede obtener, en este caso de una resnet50, mediante su nombre model_meta:

```
model_meta[resnet50]
```

Cabe aclarar que lo que conocemos como cabeza de una red neuronal es la parte especializada en una tarea concreta. En el caso de una CNN, suele ser la parte que se encuentra después de

la capa de agrupación de promedios adaptativos. El "cuerpo" es todo lo demás. La cabeza de la arquitectura se puede obtener utilizando la función create_head. Con esta función se puede elegir cuántas capas lineales adicionales se añaden al final, cuánto dropout utilizar después de cada una, y qué tipo de pooling utilizar. Por defecto, fastai aplicará tanto el pooling promedio, como el pooling máximo, y concatenará los dos juntos.

unet_learner

La segmentación es una tarea compleja, ya que la salida requerida es una imagen, o una cuadrícula de píxeles, que contenga la etiqueta predicha para cada píxel. En ella partimos de una imagen y la convertimos en otra de las mismas dimensiones o relación de aspecto, pero con los píxeles alterados de alguna manera. Las tareas de este tipo son tareas de modelos de visión generativa.

La forma de realizar esto es utilizar el mismo enfoque que el usado para desarrollar una cabeza de una CNN, como acabamos de ver, pero sustituiremos esa cabeza con una cabeza personalizada que hará la tarea generativa. El problema al que nos enfrentamos es el de desde una imagen de, por ejemplo 224 píxeles, cuya salida es una cuadrícula de 7x7 activaciones convolucionales a una máscara de segmentación de 224píxeles, esto lo haremos mediante la red neuronal. Para ello necesitaremos algún tipo de capa que pueda aumentar el tamaño de cuadrícula de una CNN. Un método podría ser sustituir cada píxel de la cuadrícula de 7×7 por cuatro píxeles en un cuadrado de 2×2. Cada uno de esos cuatro píxeles tendrá el mismo valor, a esto se lo conoce como interpolación por vecino más cercano. PyTorch proporciona una capa que realiza esta acción, así que una opción sería crear un cabezal que contenga capas convolucionales stride-1 (junto con capas batchnorm y ReLU) intercaladas con capas de interpolación de vecinos más cercanos 2×2. Otro enfoque es reemplazar la combinación de vecino más cercano y convolución con una convolución transpuesta, también conocida como media convolución de zancada. Esto es idéntico a una convolución regular, pero primero se inserta un relleno de cero entre todos los píxeles de la entrada, como se observa en la siguiente imagen:

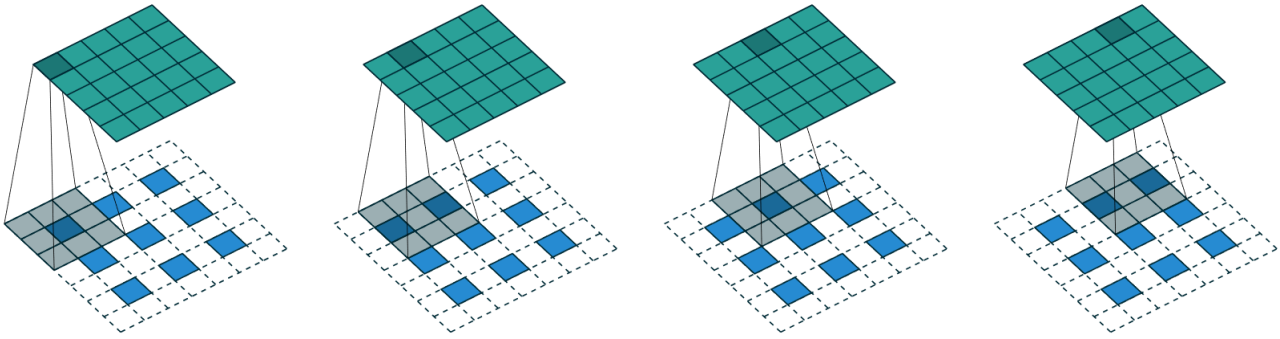

Figura 4.5: [1]

Esto se puede probar usando la clase ConvLayer de fastai, pasándole el parámetro transpose=True para crear una convolución transpuesta, en lugar de una regular.

Aunque ninguno de estos enfoques funcione realmente bien, la solución a este problema es utilizar las conexiones de salto, como las que usan las ResNet, pero saltando desde las activaciones de las ResNet a las activaciones de convolución transpuesta en el lado opuesto de la arquitectura

como ilustra la siguiente imagen, donde se observa por qué son llamadas "Red-U":

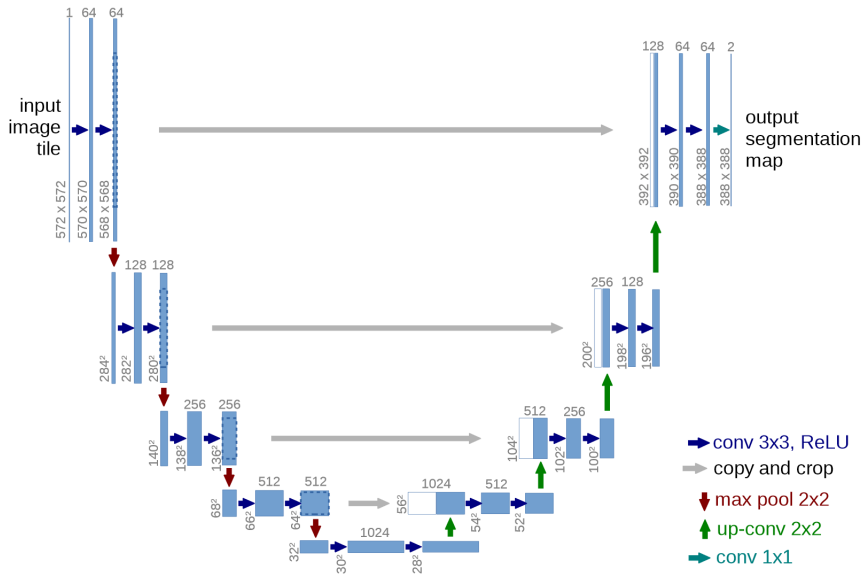

Figura 4.6: [1]

Gracias a esta arquitectura la entrada a las convoluciones transpuestas no es sólo la cuadrícula de menor resolución de la capa anterior, sino también la cuadrícula de mayor resolución de la cabeza de la ResNet. Esto permite a la U-Net utilizar toda la información de la imagen original, según sea necesario

4.5.7. Mapa de activación de clases

El mapa de activación de clases es una herramienta que nos permite hacernos a la idea de por qué una red neuronal convolucional hace las predicciones que hace. En la siguiente sección se describe como podemos mostrar estos mapas así como el concepto de hook.

El mapa de activacion de clases (CAM), utiliza la salida de la última capa convolucional junto con las predicciones para darnos una visualización del mapa de calor de por qué el modelo tomo su decisión. Esto se hace gracias a que en cada posición de nuestra última capa convolucional, tenemos tantos filtros como en la última capa lineal. Por lo tanto, podemos calcular el producto de esas activaciones con los pesos finales para obtener, para cada posición del mapa de características, la puntuación de la característica que se utilizó para tomar una decisión.

Para poder acceder a las activaciones dentro del modelo mientras este se está entrenando debemos hacer uso del concepto de hook, estos son el equivalente en PyTorch de los callbacks de fastai, solo que en lugar de permitirte inyectar código en el bucle de entrenamiento como un callback, los hooks permiten inyectar código en los propios cálculos hacia delante y hacia atrás. Dando la posibilidad de adjuntar un hook a cualquier capa del modelo, haciendo que se ejecute cuando calculemos las salidas(forward hook) o durante la retropropagación (backward hook). Un forward hook es una función que toma tres cosas -un módulo, su entrada y su salida- y puede realizar cualquier comportamiento que desee.

En nuestro hook para CAM querremos almacenar las activaciones de la ultima capa convolucional, por lo que pondremos nuestra funcion hook en una clase para que tenga estado y podamos acceder más tarde, y que simplemente guarde una copia de la salida:

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

Una vez creado el hook podemos instanciarlo y adjuntarlo a la capa que queramos, que es la última capa del cuerpo de la CNN:

```
hook_output = Hook()
hook = learn.model[0].register_forward_hook(hook_output.hook_func)
```

A continuación tomaremos un batch y lo pasaremos por nuestro modelo:

```
with torch.no_grad(): output = learn.model.eval()(x)
```

Para hacer el producto de nuestra matriz de pesos,con las activaciones utilizaremos una einsum personalizada

```
cam_map = torch.einsum('ck,kij->cij', learn.model[1][-1].weight, act)
```

Para cada imagen de nuestro batch, y para cada clase, obtenemos un mapa de características de 7×7 que nos indica dónde las activaciones fueron mayores y dónde fueron menores. Esto nos permite ver qué áreas de las imágenes influyen en la decisión del modelo. En la imagen de salida las zonas en amarillo corresponden a activaciones altas y las zonas en morado a activaciones bajas.

Es importante tener en cuenta que una vez que hayas terminado con el hook, debes eliminarlo para no perder memoria. Es por ello que hay que hacer a la clase Hook un gestor de contexto, registrando el hook cuando se entra en él y eliminándolo cuando se sale. Un gestor de contexto es una construcción de Python que llama a __enter__ cuando se crea el objeto en una cláusula with, y a __exit__ al final de la cláusula with. Esto se puede hacer de la siguiente manera:

```
class Hook():
def __init__(self, m):
    self.hook = m.register_forward_hook(self.hook_func)
def hook_func(self, m, i, o): self.stored = o.detach().clone()
def __enter__(self, *args): return self
def __exit__(self, *args): self.hook.remove()
```

Lo que nos permitirá usarlo de forma segura así:

```
with Hook(learn.model[0]) as hook:
with torch.no_grad(): output = learn.model.eval()(x.cuda())
act = hook.stored
```

Este método es útil, pero sólo funciona para la última capa. El gradiente de mapas de activacion de clase es una variante que soluciona este problema, ya que utiliza los gradientes de la activación final para la clase deseada. Los gradientes de cada capa son calculados por PyTorch durante el pase hacia atrás, pero no son almacenados (excepto para los tensores donde requires_grad es True). Pero se puede registrar un hook en el pase hacia atrás, al que PyTorch le dará los gradientes como parámetro. La clase HookBwd interceptará y almacenará gradientes en lugar de activaciones:

```
class HookBwd():
def __init__(self, m):
    self.hook = m.register_backward_hook(self.hook_func)
def hook_func(self, m, gi, go): self.stored = go[0].detach().clone()
def __enter__(self, *args): return self
def __exit__(self, *args): self.hook.remove()
```

La novedad de este sistema es que podemos utilizarlo en cualquier capa.

Aunque la interpretación de modelos es una área de investigación activa, los mapas de activación de clases nos dan una idea de por qué un modelo predijo un determinado resultado mostrando las áreas de las imágenes que fueron más responsables de una determinada predicción. Esto puede ayudarnos a analizar los falsos positivos y a averiguar qué tipo de datos faltan en nuestro entrenamiento para evitarlos.

4.6. Análisis tecnológico: Keras vs Fastai

Una vez visto como funcionan los frameworks de Fastai y Keras es el momento de tomar una decisión para el desarrollo de la solución al problema. En el caso del presente trabajo de fin de grado se ha optado por la utilización de fastai ya que se trata de un framework más reciente, más flexible y que destaca por su sencillez para resolver pequeños problemas. Además al tratarse de una tecnología nueva resulta más interesante su exploración ya que nos permitirá ver cuales son sus posibilidades.

Capítulo 5

Introducción tecnológica de los clasificadores

5.1. Descripción del problema

Una vez aprendido acerca del marco teórico sobre el uso de redes neuronales para la clasificación de imágenes, nuestra siguiente objetivo es aprender a trabajar con ellas para obtener buenos clasificadores.

Para realizar dicho aprendizaje, al no disponer de un conjunto de datos de fabricación, se ha optado por utilizar un conjunto de datos propio nuevo que permitan ver resultados. Para ello se ha optado por realizar un dataset de balones clasificados en tres categorías: fútbol, baloncesto y voleibol.

Las imágenes fueron obtenidas a través de la red mediante búsquedas en el buscador de imágenes de "Google", y no se han sometido a ningún tratamiento previo, por lo que en algunas de ellas aparecen elementos externos como personas, pistas deportivas, césped... el tamaño y resolución de cada una de las imágenes es distinto por lo que requerirá que ajustemos parámetros al introducir el conjunto de datos en fastai. En total se disponen de 300 imágenes, de forma que existen 100 en cada categoría.

Figura 5.1: Ejemplo de elementos del conjunto de datos

5.2. Descripción de la experimentación

En la línea de conseguir una configuración óptima para obtener el mejor clasificador, se ha definido un conjunto de pruebas que permitan observar que conjunto de parámetros hace que la red funcione mejor. Para que los diferentes datos seleccionados no cambien entre las ejecuciones

dentro de la creación del DataBlock se ha configurado el Splitter con una semilla con valor 42. Los parámetros que se han tenido en cuenta para la experimentación han sido los siguientes:

- **Arquitecturas**: Para las diferentes pruebas se han utilizado las arquitecturas: ResNet18, ResNet34, ResNet50, XResNet18, XResNet34, XResNet50, SqueezeNet1.0, SqueezeNet1.1, AlexNet, DenseNet121, DenseNet161, DenseNet169, DenseNet201, VGG16_bn y VGG19_bn.
- **Número de épocas**: Para este caso se han utilizado diferentes números de épocas en las diferentes arquitecturas para ver si existen mejoras con respecto al aumento de épocas, para ello se han cogido como ejemplos los numero de épocas 10,20,30,40 y 50.
- **Cambios en la tasa de aprendizaje**: Para este caso lo que haremos es utilizar las arquitecturas donde obtuvimos los mejores resultados, en ellas cambiaremos el parámetro lr_base para probar si existe una mejora significativa al cambiar el ratio de aprendizaje. Los valores del lr_base que utilizaremos serán en función en lo obtenido por lr_find queándonos con el lr_min, lr_steep y la media entre ambos.
- **Aplicando augmentation**: En este caso lo que vamos a realizar es unas transformaciones en las imágenes que solo produzcan cambios en cuando a la rotación de la misma, es decir, nada que deforme la imagen. Para ello usaremos las clase de las librerías de fastai "Rotate".

5.2.1. Resultados

Una vez definido los experimentos a realizar se procede a la realización de los mismos, en todos ellos para ajustar el tamaño de las imágenes se realizó un ajuste de tamaño a 224x224 píxeles y se añadió un padding(relleno) de ceros, añadiendo franjas negras a las imágenes menores a ese tamaño. Estas transformaciones se aplican en CPU ya que se realizan en el "item_tfms".

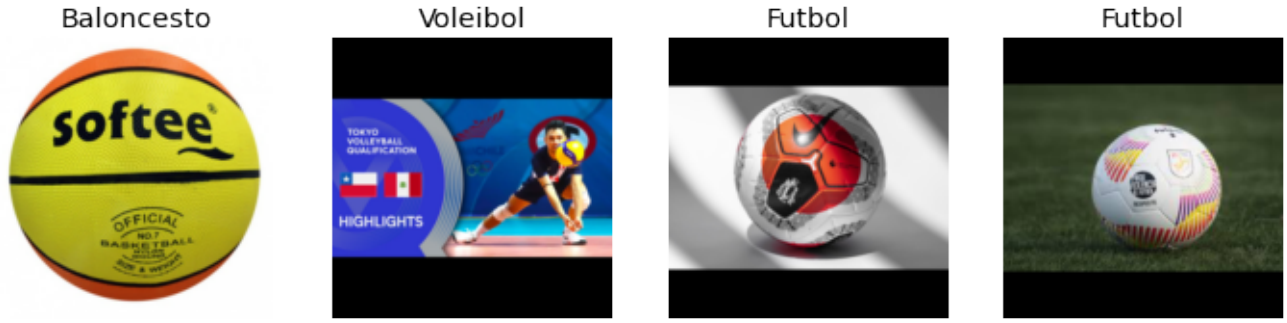

Figura 5.2: Imágenes tras el tratamiento

Tras la realización de las pruebas los resultados se detallan a continuación mediante tablas en las que aparece la tasa de error obtenida en cada experimento, así como gráficos que nos dan idea de su función de pérdida, así como la evolución del aprendizaje y la aparición de overfitting o underfitting.

Arquitecturas y número de épocas

En este primer experimento se realizó para cada una de las arquitecturas mencionadas anteriormente un entrenamiento de 10, 20, 30, 40 y 50 épocas, utilizando la función fine_tune, obteniendo

los resultados detallados en la tabla que se muestra a continuación. En ella se han marcado en verde las medias de las dos arquitecturas que han obtenido mejores resultados para las diferentes épocas, así como las épocas en las que la media a través de las diferentes arquitecturas su tasa de error ha sido menor. Utilizaremos estos resultados para probar con ellos en los siguientes experimentos.

	10 Epochs	20 Epochs	30 Epochs	40 Epochs	50 Epochs	Media
ResNet18	0.06896549	0.08620691	0.06896549	0.06896549	0.06896549	0.072413778
ResNet34	0.08620691	0.10344828	0.06896549	0.10344828	0.10344828	0.093103449
ResNet50	0.10344824	0.05172411	0.03448276	0.05172412	0.0689655	0.062068947
XResNet18	0.25862068	0.25862068	0.18965518	0.22413796	0.20689656	0.227586213
XResNet34	0.29310346	0.22413796	0.22413796	0.22413796	0.22413796	0.237931061
XResNet50	0.25862068	0.20689653	0.12068964	0.18965517	0.12068964	0.179310331
SqueezeNet1.0	0.10344828	0.13793105	0.12068963	0.10344828	0.08620691	0.11034483
SqueezeNet1.1	0.15517241	0.17241383	0.15517241	0.12068963	0.10344828	0.14137931
AlexNet	0.12068963	0.13793105	0.13793105	0.12068963	0.12068963	0.127586198
DenseNet121	0.08620688	0.12068962	0.12068962	0.05172414	0.0689655	0.089655154
DenseNet161	0.05172414	0.06896551	0.06896552	0.08620689	0.06896552	0.068965515
DenseNet169	0.06896552	0.06896552	0.06896552	0.10344826	0.12068965	0.086206892
VGG16_bn	0.10344826	0.12068962	0.12068962	0.137931	0.13793102	0.124137905
VGG19_bn	0.10344824	0.10344826	0.10344824	0.12068962	0.12068964	0.1103448
Media	0.13300492	0.13300492	0.11453201	0.12192118	0.11576354	

Figura 5.3: Resultados de la tasa de error para las diferentes arquitecturas entrenadas diferentes épocas

En la tabla observamos que por norma general que en arquitecturas iguales, aquellas con más profundidad suelen ser las que obtienen mejores resultados, siendo las más profundas las que presentan los dígitos más alto. Por lo que puede ser una idea de cara a futuros experimentos utilizar arquitecturas complejas en estos casos. Observamos también que el número de épocas influye positivamente en la disminución de la tasa de error, obteniendo mejores resultados cuántas más épocas dure el entrenamiento.

Veamos el proceso de entrenamiento, para detallar esto utilizaremos la arquitectura ResNet50 e iremos viendo que pasa en el diferente conjunto de épocas seleccionadas. Lo que observamos aquí es que fastai mediante la función fine_tune intenta hacer el mejor ajuste de parámetros para optimizar el entrenamiento durante las épocas que se hayan introducido, esto lo observamos al comparar los resultados de los dos entrenamientos:

epoch	train_loss	valid_loss	error_rate	time
0	0.23284964	0.22029749	0.0689655	0:07
1	0.15359138	0.3042289	0.0689655	0:06
2	0.11793059	0.36338338	0.0689655	0:06
3	0.0862074	0.29797715	0.05172412	0:07
4	0.06768856	0.36067793	0.0689655	0:06
5	0.0551786	0.33433458	0.0689655	0:06
6	0.04424567	0.35411698	0.10344824	0:06
7	0.04061439	0.68522799	0.13793102	0:06
8	0.05480731	0.63464093	0.18965515	0:06
9	0.05727843	0.4315283	0.10344824	0:06
10	0.05646002	0.50467449	0.08620688	0:06
11	0.05698041	0.42822286	0.10344824	0:06
12	0.05337921	0.49278253	0.12068965	0:06
13	0.04785414	0.40791327	0.10344828	0:06
14	0.05161272	0.36059874	0.0689655	0:06
15	0.04887348	0.39432907	0.06896549	0:06
16	0.04395501	0.35889432	0.05172411	0:06
17	0.03793812	0.3791329	0.05172411	0:06
18	0.03381034	0.3686474	0.05172411	0:06
19	0.03039778	0.37833983	0.05172411	0:06

epoch	train_loss	valid_loss	error_rate	time
0	0.23282407	0.22024132	0.08620686	0:06
1	0.15297981	0.29176995	0.0689655	0:06
2	0.11785793	0.34038708	0.08620688	0:06
3	0.08564302	0.33598065	0.0689655	0:06
4	0.06652095	0.3106302	0.0689655	0:06
5	0.05606711	0.29220614	0.08620688	0:06
6	0.0455248	0.28506377	0.08620688	0:06
7	0.0444341	0.54424369	0.12068964	0:06
8	0.05281182	0.4248679	0.13793102	0:06
9	0.06098886	0.41107467	0.06896549	0:06
10	0.06807984	0.63095748	0.08620688	0:06
11	0.07322285	0.96542555	0.1724138	0:06
12	0.07565121	0.45970219	0.12068962	0:06
13	0.06757528	0.50516546	0.10344826	0:06
14	0.05945402	0.52775425	0.08620688	0:06
15	0.05387912	0.51778686	0.08620688	0:06
16	0.04919968	0.51383221	0.05172414	0:06
17	0.0434669	0.48889536	0.05172414	0:06
18	0.03767521	0.54463249	0.08620686	0:06
19	0.03533906	0.5416742	0.08620686	0:06
20	0.03392159	0.52615362	0.0689655	0:07
21	0.03028852	0.52680248	0.0689655	0:06
22	0.02616776	0.53027201	0.05172414	0:07
23	0.02455059	0.50955153	0.03448276	0:06
24	0.02150874	0.49152443	0.03448276	0:06
25	0.01867069	0.48910636	0.03448276	0:06
26	0.01628505	0.48996958	0.03448276	0:06
27	0.01667188	0.48825338	0.03448276	0:06
28	0.01478437	0.48477563	0.03448276	0:06
29	0.01329167	0.47646385	0.03448276	0:06

Figura 5.4: Resultados de la tasa de error en entrenamiento de ResNet50 en 20 y 30 épocas

En ella podemos observar como los valores de mismas épocas difieren entre el entrenamiento de 20 épocas y el de 30 épocas lo que nos indica que valores como el learning_rate son ajustados automáticamente según el número de épocas.

Por último veamos como evoluciona la función de pérdida en una de las arquitecturas seleccionadas: DenseNet161 entrenada 30 épocas.

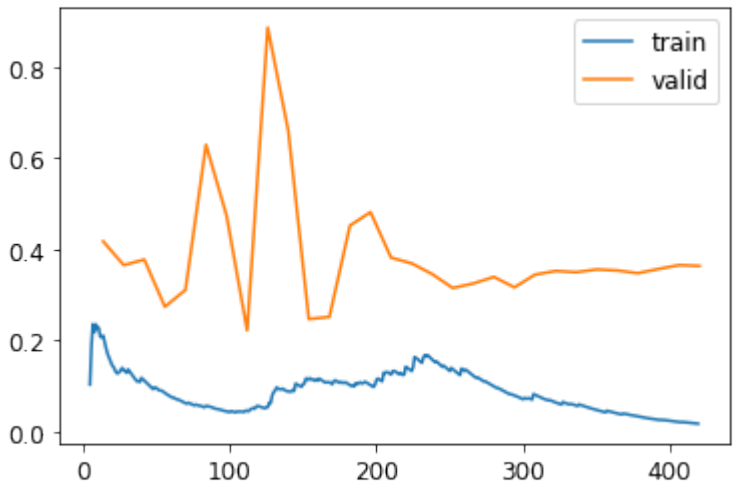

Figura 5.5: Evolución de la función de perdida en DenseNet161 entrenada 30 épocas

Observamos como la función de pérdida de validación en las últimas épocas se mantiene estable mientras que la función de perdida en entrenamiento sigue disminuyendo, esto nos hace pensar

que podemos encontrarnos ante un caso de overfitting, si la función de perdida en validación se mantuviera o aumentara su valor.

Cambios en la tasa de aprendizaje

En esta sección vamos a utilizar las arquitecturas seleccionadas anteriormente, DenseNet161 y ResNet161, y las entrenaremos durante 30 y 50 épocas, utilizando la función fine_tune en la que estableceremos el learning_rate base (parámetro base_lr) con valores obtenidos mediante la función lr_find(). Los valores que usaremos serán el lr_min, el lr_steep y la media de ambos.

Realizando el experimento hemos obtenido los siguientes valores:

	30				50			
	lr_min	lr_steep	(lr_min+lr_steep)/2	No modificado	lr_min	lr_steep	(lr_min+lr_steep)/2	No modificado
ResNet50	0.06896549	0.10344824	0.068965502	0.034482758	0.10344826	0.12068962	0.103448242	0.068965502
DenseNet161	0.0862069	0.1724138	0.086206891	0.068965517	0.05172414	0.12068965	0.05172414	0.068965517

Figura 5.6: Resultados de la tasa de error al realizar cambios en lr_base

Viendo los resultados observamos que se obtiene el mejor resultado en la mayoría de los casos al no modificar el parámetro lr_base, obteniendo los peores resultados utilizando el lr_steep. Por lo que deducimos que el propio fine_tune ajusta ese parámetro para que salga el mejor resultado en la mayoría de los casos.

Augmentation

En este experimento vamos a aplicar augmentations a los datos de entrenamiento, para ello utilizaremos las clases Rotate y FlipItem proporcionada por fastai dentro de batch_tfms para que las transformaciones las realice en cada época y así los datos con los que entrene vayan variando, intentando obtener así mejores resultados.

La clase Rotate realiza una rotación aleatoria sobre el conjunto de imágenes provocando que la imagen rote una cantidad de grados máximos (por defecto 10 grados).

Figura 5.7: Batch de entrenamiento en el que se ha aplicado Rotate

La clase FlipItem voltea las imágenes con una probabilidad del 50 %.

Figura 5.8: Batch de entrenamiento en el que se ha aplicado FlipItem

Una vez realizadas las ejecuciones necesarias los resultados obtenidos han sido los siguientes:

	30				50			
	Rotate	FlipItem	Rotate & FlipItem	Sin augmentation	Rotate	FlipItem	Rotate & FlipItem	Sin augmentation
ResNet50	0.0862069	0.086206898	0.086206898	0.034482758	0.05172414	0.0862069	0.034482758	0.068965502
DenseNet161	0.03448276	0.05172414	0.017241379	0.068965517	0.05172414	0.03448276	0.05172414	0.068965517

Figura 5.9: Resultados de la tasa de error al aplicar distintas augmentations

Por un lado observamos que al realizar las transformaciones los resultados son algo mejores en la mayoría de casos que al no aplicar ninguna transformación, esto se debe a que al aplicarlas la red puede ver más imágenes con las que entrenar y no siempre entrena con las mismas. También podemos ver que al combinar las dos transformaciones obtenemos resultados algo mejores que al realizar las transformaciones de forma individual.

5.2.2. Conclusiones

En este capítulo hemos aprendido a realizar clasificadores para un conjunto de datos de prueba de balones, se tata de un conjunto previo a disponer de uno con datos reales, en el que podemos ver como no existe un gran número de datos, por lo que el entrenamiento por lo general, tenderá al overfitting.

De él hemos aprendido como funciona por debajo la función fine_tune, viendo como para diferentes épocas seleccionadas, automáticamente ajusta los parámetros, además ajusta el parámetro lr_base para obtener el mejor resultado. Por último hemos utilizado transformaciones para observar una mejora en los resultados del entrenamiento

Capítulo 6

Control de calidad en procesos de fabricación

6.1. Descripción del problema

En esta sección del documento el problema a tratar es la resolución de un control de calidad en imágenes de soldaduras con el objetivo de indicar si estas son correctas o por el contrario son defectuosas, para ello ya existe un conjunto de datos con el que poder realizar nuestras primeras redes neuronales. El conjunto de datos que se ha utilizado en el documento proviene de una toma de imágenes de una capturadora controlada por un robot. Estas capturas son sometidas a un proceso de transformación de captura tridimensional a una imagen con extensión png que se encuentra en dos dimensiones.Las imágenes de soldadura para mostrar toda la información posible se representan en escala de grises donde el color más oscuro representa menor altura en el eje Z mientras que colores más próximos al blanco son los de mayor altura en el mismo eje. También hay que tener en cuenta que la capturadora al realizar la toma puede mostrar algo de ruido, o no capturar zonas que presenten sombra para ella, esto dependerá del ángulo de captura. El conjunto inicial solo dispone de tomas de imágenes correctas, por lo que para la clasificación de estas imágenes en subcategorías de buenas y malas, se ha tenido que hacer un proceso previo que permita transformar las imágenes del conjunto de datos, todas buenas, a imágenes malas que disponen de ciertas deformaciones que se detallan en la siguiente sección. Posteriormente se han realizado diferentes experimentos con distintas redes neuronales en busca de obtener el mejor resultado por cada tipo de soldadura disponible.

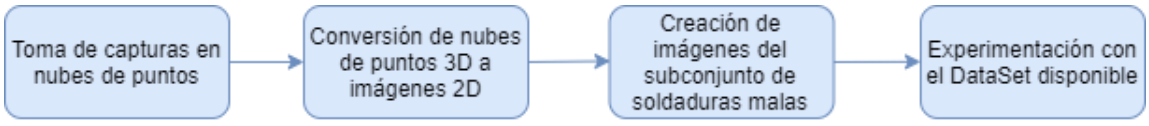

Figura 6.1: Diagrama de toma de datos

En resumen los objetivos del presente capítulo son:

- Desarrollar una metodología con la que tratar las imágenes.
- Desarrollar técnicas de augmentation con la que poder generar imágenes sintéticas a partir de unas originales.

- Obtener los mejores resultados posibles, consiguiéndolos iterando sobre el conjunto de datos.

Es importante mencionar que en el presente Trabajo de Fin de Grado no se va a realizar la transformación de las nubes de puntos a imágenes bidimensionales, ese proceso nos será facilitado de forma externa.

6.2. Tipologías

Las imágenes mencionadas anteriormente se encuentran divididas en diferentes tipologías, ya que presentan diferentes características y los defectos que aparecen en ellas pueden no ser los mismos ya que están atados a "normas de soldadura", es por ello que han sido asociadas a un número:

- **Tipo 2**: Se trata de una soldadura cuyo plano de la chapa superior no está aislado, aparece en la imagen, y presenta un plano.

Figura 6.2: Imagen de muestra de Tipo 2

- **Tipo 3**: Es una soldadura aislada en la que solo se aprecia la propia soldadura.

Figura 6.3: Imagen de muestra de Tipo 3

- **Tipo 4**: En este grupo se encuentran soldaduras con un plano abajo no aislado.

Figura 6.4: Imagen de muestra de Tipo 4

- **Tipo 5**: Las imágenes de este tipo son soldaduras que presentan un plano abajo, similar al tipo 4, pero por como se obtiene la imagen presentan muchas zonas de sombra.

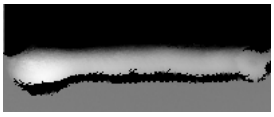

Figura 6.5: Imagen de muestra de Tipo 5

- **Tipo 7**: Son soldaduras que no presentan forma de línea si no que son cuadradas. Entre ellas se pueden diferenciar dos subtipos, aquellas cuya chapa externa es oblicua y aquellas cuya chapa externa tiene la misma altura en todos sus puntos

Figura 6.6: Imagen de muestra de Tipo 7

- **Tipo 8**: Soldaduras que presentan chapa superior no aislada.

Figura 6.7: Imagen de muestra de Tipo 8

- **Tipo 9**: Soldaduras de corta longitud con chapa superior no aislada.

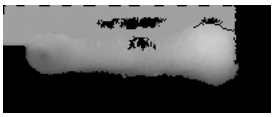

Figura 6.8: Imagen de muestra de Tipo 9

- **Tipo 10**: Similar a las de la tipología 3 pero con menor longitud.

Figura 6.9: Imagen de muestra de Tipo 10

- **Tipo 11**: Soldaduras similares a las del tipo 3 pero con mayor longitud.

Figura 6.10: Imagen de muestra de Tipo 11

Las tipologías 1 y 6 no aparecen dentro del conjunto de datos, esto se debe a que se realizo una división inicial en 11 subtipos pero se observo que las soldaduras de Tipo 1 y de Tipo 6 eran muy similares a otras tipologías, lo que llevo a anexionarlas con otros tipos.

6.3. Transformaciones del conjunto de datos

Como se ha indicado en la primera sección las imágenes originales que se han obtenido para la realización de este trabajo no disponían de una categoría adicional sobre la que clasificar si no que, solo se tenían datos de una categoría. Esto hizo necesario que mediante un programa se realizaran ajustes sobre estas imágenes pudiendo obtener un nuevo conjunto de datos de imágenes a las que podríamos clasificar en una nueva categoría de imágenes malas.

Previo a la explicación de las transformaciones es necesario hacer una aclaración sobre lo que estas imágenes representan, se tratan de nubes de puntos en tres dimensiones obtenidas por una capturadora sobre diferentes soldaduras realizadas en piezas en procesos de fabricación industrial, estas nubes de puntos han sido sometidas a un procesado que ha permitido eliminar parte del ruido así como la conversión de la nube de puntos a dos dimensiones, convirtiéndose en una imagen en escala de grises, donde el color indica la altura que alcanza la soldadura siendo las zonas blanquecinas las más blancas y las más oscuras las más bajas. Aquellas zonas con valor 0 dependiendo de su posición en la imagen pueden ser, un plano eliminado por sencillez a la hora de deformar las piezas, zonas que tienen altura mínima, o ruido, zonas que la capturadora no puede obtener debido a factores como la luminosidad o posición de captura. Esta conversión de la imagen de nube de puntos a imagen bidimensional, no es contenido de la presente memoria ya que las imágenes facilitadas ya están procesadas.

Para la realización de estas transformaciones ha sido necesario el uso de diferentes librerías que se encuentran en el lenguaje de programación Python, entre las que destacan:

- **PIL**: Se trata de una biblioteca que proporciona soporte para abrir, manipular imágenes.
- **OpenCV**: Es una biblioteca libre de visión artificial originalmente desarrollada por Intel. Actualmente es la biblioteca más popular de visión artificial.
- **Numpy**: Biblioteca que da soporte para crear vectores y matrices grandes multidimensionales, junto con una gran colección de funciones matemáticas de alto nivel para operar con ellas.

Entre las transformaciones aplicadas para esta nueva categoría se encuentran las siguientes:

- **Desviación en cordón**: Este tipo de deformaciones se da cuando un robot sufre una ligera desviación en la trayectoria, haciendo que la soldadura deje de ser válida. Giros al principio, final y en el centro de la soldadura simularán este defecto.

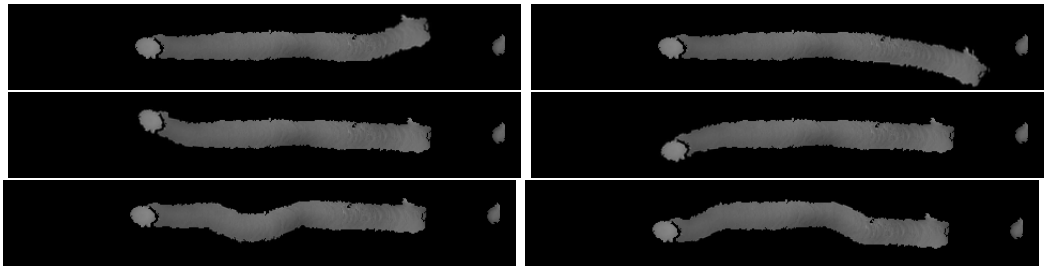

Figura 6.11: Imágenes de tipo 4 aplicando deformaciones de desviación de cordón

- **Exceso de material**: Se produce cuando existe un exceso de material en la parte norte, o sur, de la soldadura, habiendo una descompensación entre ambas. Si la parte norte, o sur, esta más blanca que la parte opuesta, que estará más oscura nos encontramos en esta situación.

Figura 6.12: Imágenes de tipo 11 aplicando deformaciones de exceso de material

Esta descompensación se puede dar solo en una sección de la soldadura como se observa a continuación:

Figura 6.13: Imágenes de tipo 11 aplicando deformaciones de exceso de material en una sección

- **Bolas**: Este tipo de defecto se trata de la situación en la que en una parte de la pieza ha aparecido una acumulación de material formándose una especie de bola. Esto se puede apreciar si en una imagen encontramos una parte circular en la que su centro está más blanquecino que el resto.

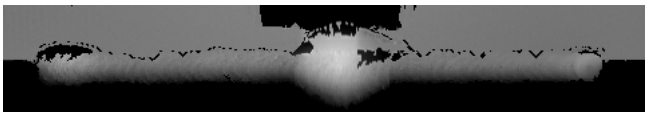

Figura 6.14: Imagen de tipo 2 con acumulación de material en forma de bola

- **Cortes**: Es la discontinuidad mayor a un cierto tamaño en una soldadura que hace que sea desechada. Zonas en la imagen negras mayores a un cierto tamaño umbral en donde se encuentra la pieza nos ayudarán a simular este defecto.

Figura 6.15: Imagen de tipo 4 con deformaciones del tipo cortes

Se puede dar el caso en el que el corte en la soldadura no haga que la pieza sea desechada, por lo que añadiremos imágenes en el conjunto de buenas cuyo corte sea menor al tamaño umbral indicado en la norma de soldadura.

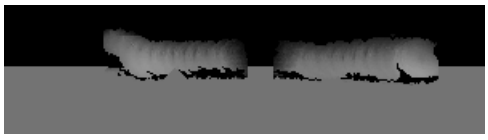

Figura 6.16: Imagen de tipo 4 con deformaciones del tipo cortes aceptables

- **Agujeros**: En el proceso de soldadura puede ocurrir que se produzca un agujero en la soldadura haciendo que deba ser desechada.

Figura 6.17: Imagen de tipo 5 con deformaciones del tipo agujeros

- **Falta de material en una chapa**: Si se produce esta situación es debido a que una de las dos chapas no ha soldado como debería.

Figura 6.18: Imágenes de tipo 8 con deformaciones del tipo escasez de material en una de las chapas

6.4. Primera iteración sobre el conjunto de datos: Clasificación inicial

6.4.1. Descripción del problema

Una vez ya creado el primer conjunto de datos es el momento de empezar a utilizarlo para obtener resultados. Para ello utilizaremos los datos tal y como se nos han ido entregados, después de aplicar las transformaciones sintéticas necesarias para obtener el subconjunto de "soldaduras_malas" para cada una de las tipologías mencionadas anteriormente.

6.4.2. Diseño del experimento

En este caso el experimento ha consistido en usar para cada tipología, cuatro redes neuronales convolucionales, utilizando transfer learning, ya que consideramos que establece un buen punto de entrada ya que partimos de un modelo entrenado, de las arquitecturas ResNet18, ResNet34, ResNet50 y ResNet101, entrenadas en 20 épocas mediante fine_tune (explicado en el capitulo 4). Obteniendo los resultados mostrados en el siguiente punto.

El código del experimento se detalla de la siguiente forma:

- En primer lugar indicamos la ruta donde se encuentran las imágenes organizadas en subcarpetas con las dos categorías en las que queremos que nuestro clasificador clasifique nuestras imágenes:

```
path ='Tipo 3'
```

- Después creamos un DataBlock en el que indicaremos como se deben dividir los conjuntos de validación (20 % de los datos) y entrenamiento (80 % de los datos) además de añadir una "semilla" para que si repetimos el experimento los resultados sean los mismos y los criterios para clasificar que son las subcarpetas de la ruta indicada.

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

- Posteriormente inicializamos el DataBlock y lo utilizamos para cargar el DataLoader con la ruta donde se encuentran nuestras imágenes

```
soldaduras = soldaduras.new()
dls = soldaduras.dataloaders(path,num_workers=0)
```

- Por último creamos el learner en el que indicamos la arquitectura de la cual queremos hacer fine_tune, la métrica que utilizaremos y además añadimos un callback mediante el parámetro cbs para que imprima los resultados en un csv, entrenaremos el learner durante 20 épocas e imprimiremos los resultados en forma de gráfico de pérdida de entrenamiento y validación y como matriz de confusión.

```
learn = cnn_learner(dls, resnet18, metrics=error_rate,
    cbs=[CSVLogger("ResNet18.csv")])
learn.fine_tune(20)
learn.recorder.plot_loss()
interp = ClassificationInterpretation.from_learner(learn)
interp.plot_confusion_matrix()
```

6.4.3. Resultados

Antes de mostrar los gráficos de resultados es importante hacer un resumen con la cantidad de datos presentes en cada subconjunto de datos, ya que será de utilidad en el momento que analicemos los resultados. Posteriormente se muestran las matrices de confusión presentes en cada experimento, además adicionalmente se muestran los gráficos de las funciones de pérdida de aquellas redes que hayan obtenido peores resultados. Por último se muestra una tabla resumen con las métricas de error obtenidas en esta iteración.

- **Tipo 2**

  - **Imágenes de soldaduras buenas**: 732
  - **Imágenes de soldaduras malas**: 3904

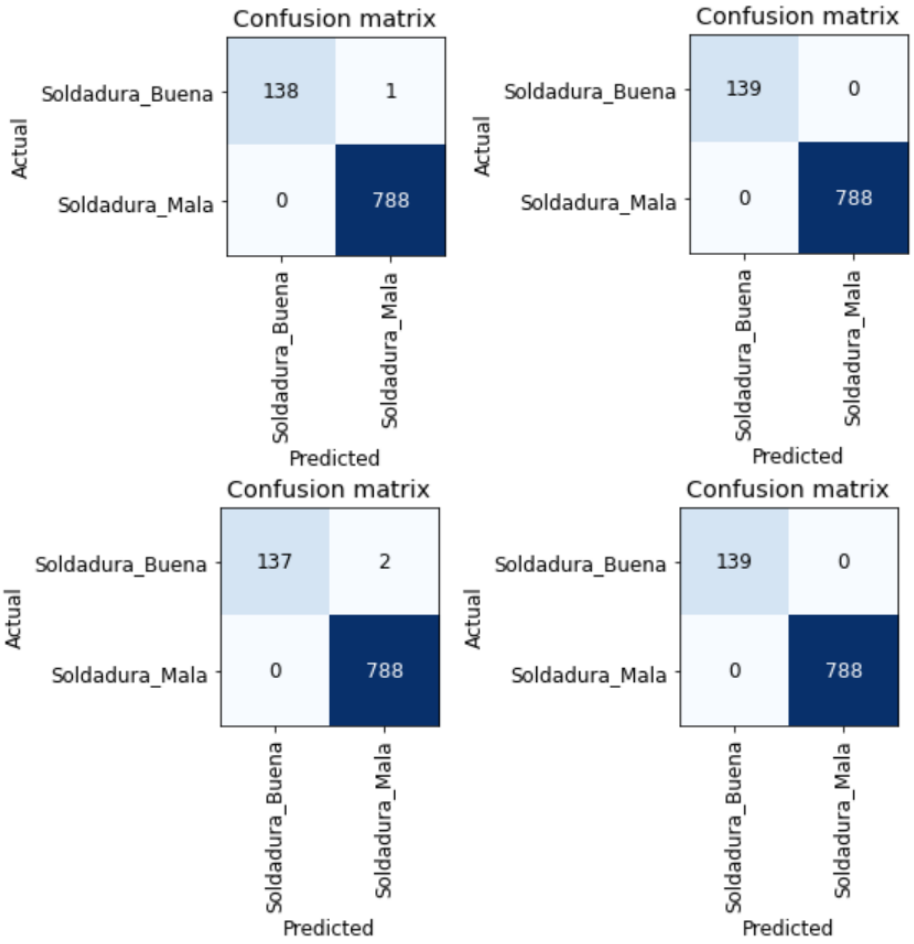

Figura 6.19: Matriz de confusión para tipo 2 para ResNet18, ResNet34, ResNet50 y ResNet101

En esta matriz observamos que la cantidad de imágenes que falla la red es mínima, falla una o dos imágenes, nos indica que no es un tipo que nos vaya a generar demasiados problemas.

- **Tipo 3**

  - **Imágenes de soldaduras buenas**: 1786

  - **Imágenes de soldaduras malas**: 8925

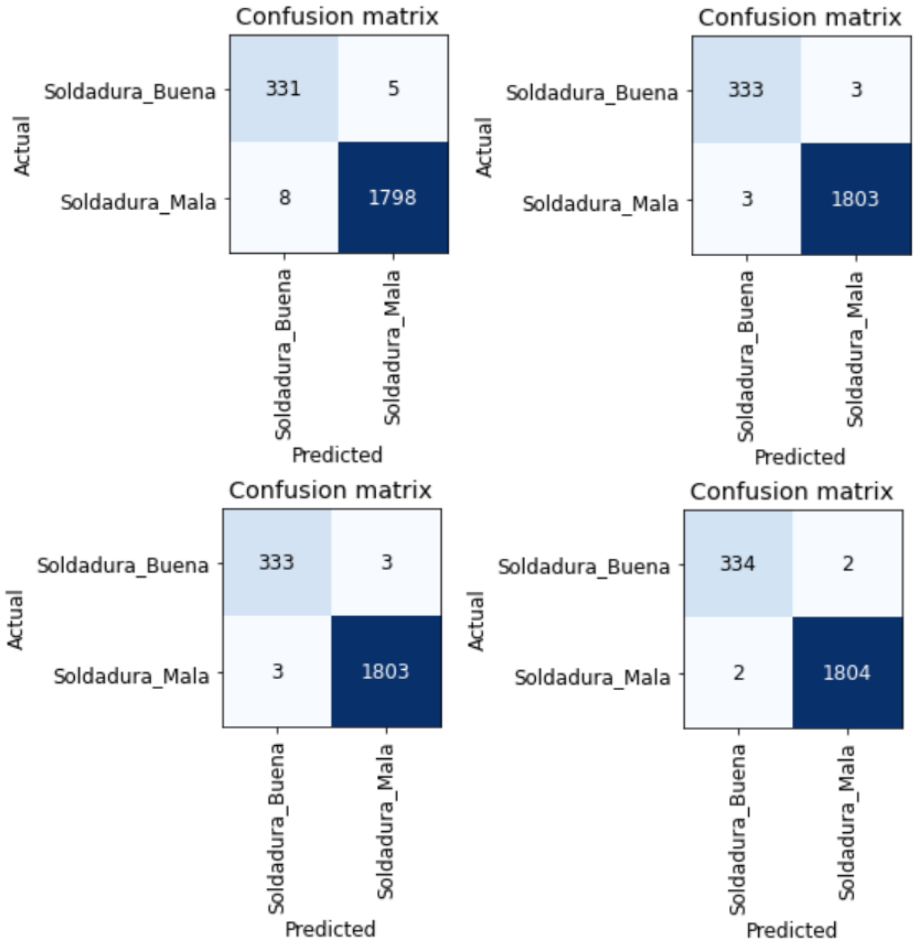

Figura 6.20: Matriz de confusión para tipo 3 para ResNet18, ResNet34, ResNet50 y ResNet101

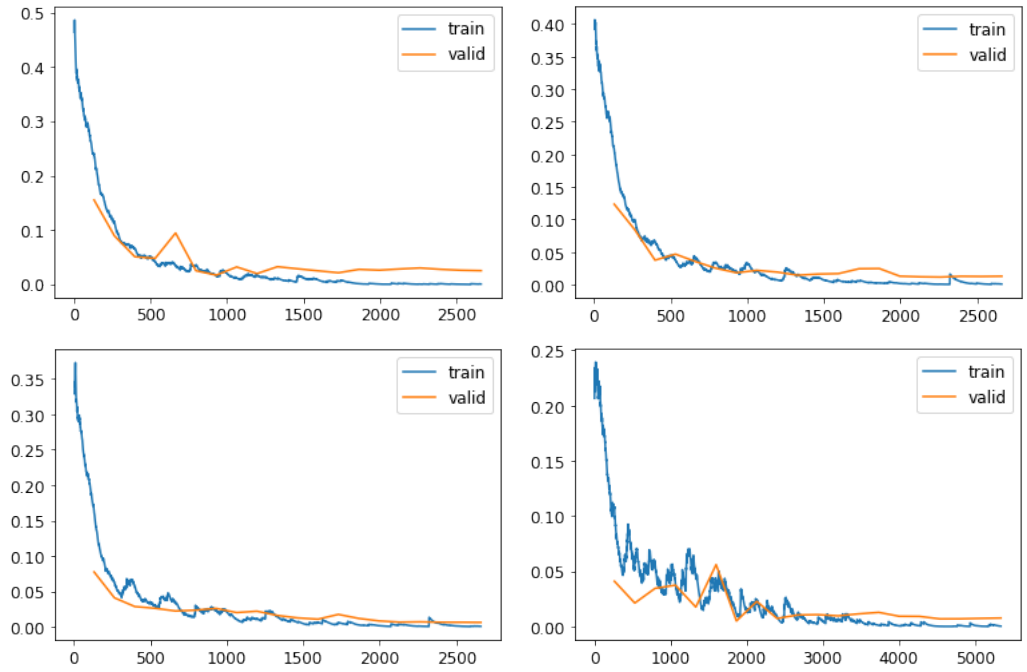

Figura 6.21: Evolución de pérdida en validación y entrenamiento para tipo 3 para ResNet18, ResNet34, ResNet50 y ResNet101

Debido a que la tasa de error es algo mayor a lo visto en el "Tipo 2", falla más imágenes en validación, resulta interesante ver si en este caso la red está aprendiendo correctamente,

esto lo podemos ver gracias a las gráficas de perdidas donde aparentemente la función de perdida en validación no se dispara lo que nos hace ver que no existe overfitting.

- **Tipo 4**

  - **Imágenes de soldaduras buenas**: 1465
  - **Imágenes de soldaduras malas**: 8299

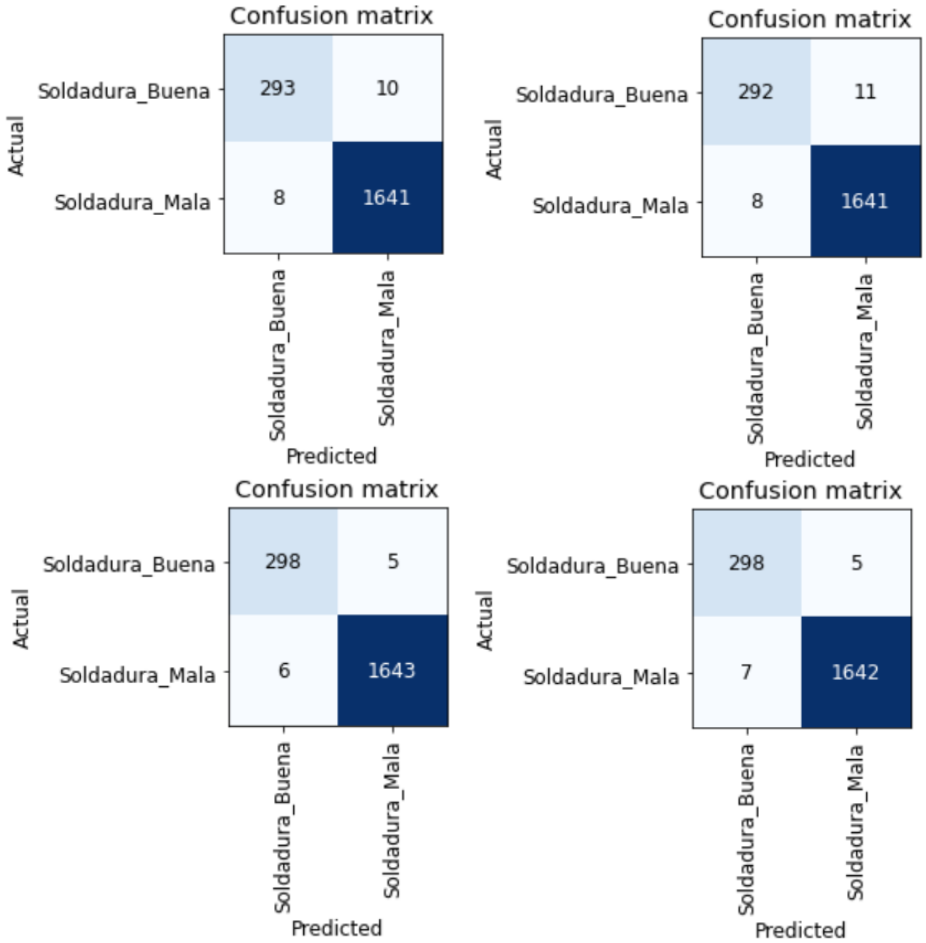

Figura 6.22: Matriz de confusión para tipo 4 para ResNet18, ResNet34, ResNet50 y ResNet101

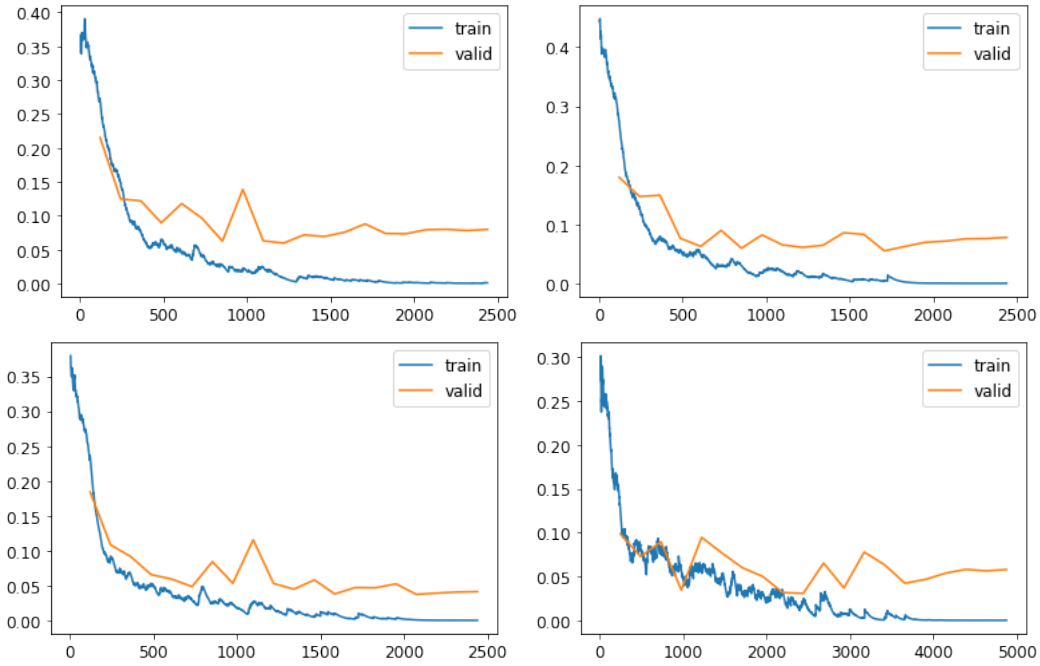

Figura 6.23: Evolución de pérdida en validación y entrenamiento para tipo 4 para ResNet18, ResNet34, ResNet50 y ResNet101

A diferencia del "Tipo 3" mencionado anteriormente, en esta ocasión los gráficos de pérdida en validación nos indican que la red no está realizando su aprendizaje de forma correcta, ya que no generaliza bien lo que ha aprendido en el conjunto de entrenamiento con el conjunto de validación, por lo que será uno de los tipos en los que deberemos fijarnos en las próximas iteraciones.

- **Tipo 5**

  - **Imágenes de soldaduras buenas**: 1960

  - **Imágenes de soldaduras malas**: 21476

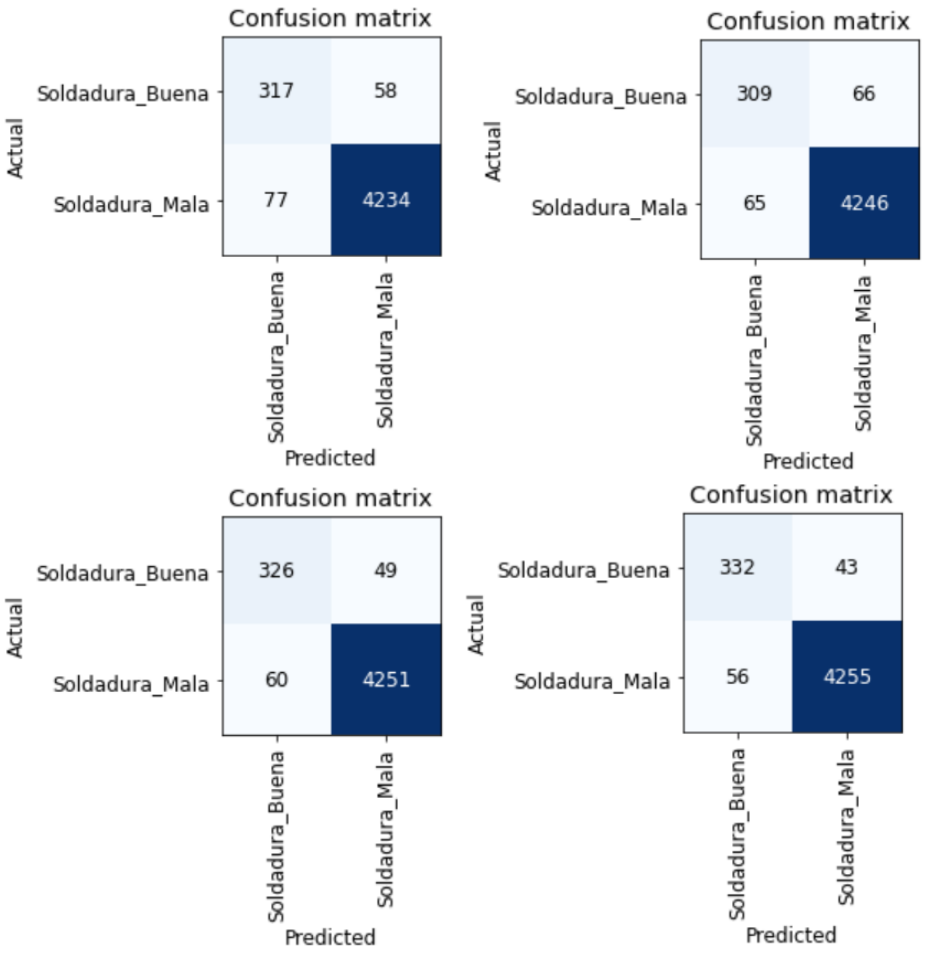

Figura 6.24: Matriz de confusión para tipo 5 para ResNet18, ResNet34, ResNet50 y ResNet101

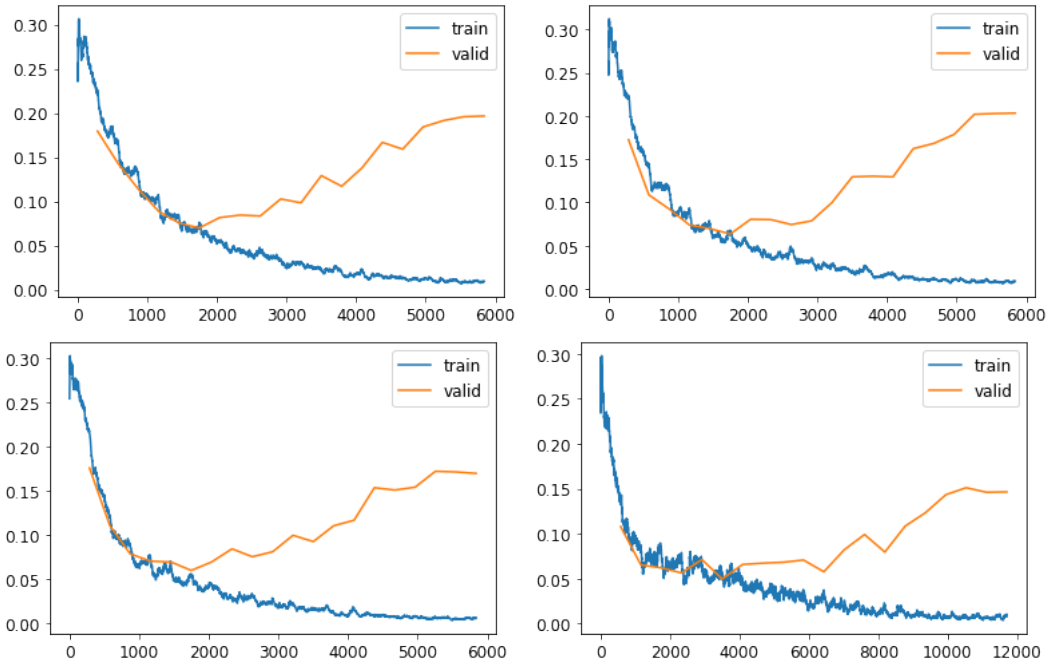

Figura 6.25: Evolución de pérdida en validación y entrenamiento para tipo 5 para ResNet18, ResNet34, ResNet50 y ResNet101

Al igual que en el "Tipo 4" la función de validación sufre un aumento que nos indica la

presencia de overfitting que deberá ser corregido en iteraciones posteriores, además se trata de uno de los conjuntos que presenta mayor tasa de error.

- **Tipo 7**

  - **Imágenes de soldaduras buenas**: 592
  - **Imágenes de soldaduras malas**: 10656

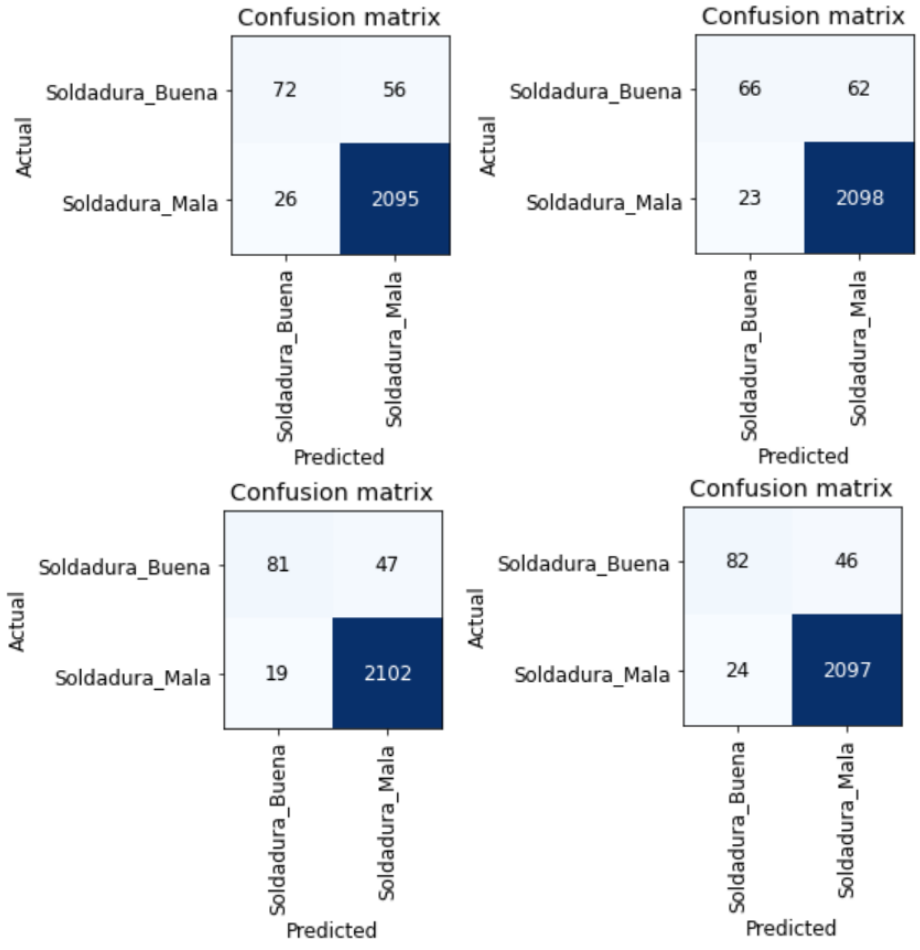

Figura 6.26: Matriz de confusión para tipo 7 para ResNet18, ResNet34, ResNet50 y ResNet101

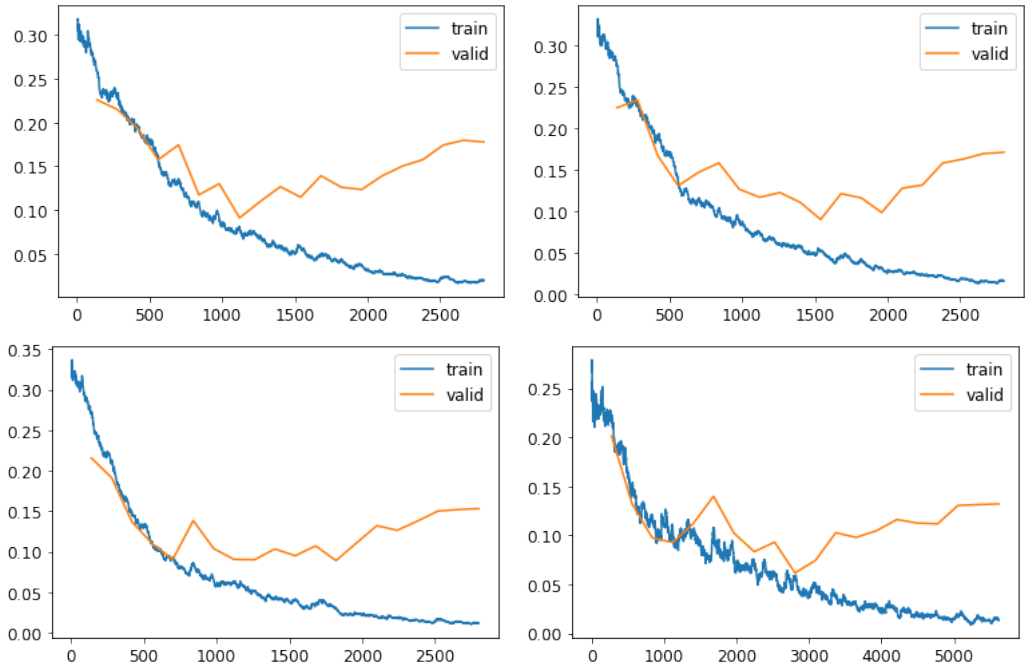

Figura 6.27: Evolución de pérdida en validación y entrenamiento para tipo 7 para ResNet18, ResNet34, ResNet50 y ResNet101

El conjunto de datos perteneciente a esta tipología junto con la tipología anterior es uno de los más problemáticos ya que presentan mayor overfitting, además de presentar una tasa de error bastante alta.

- **Tipo 8**

  - **Imágenes de soldaduras buenas**: 878

  - **Imágenes de soldaduras malas**: 4149

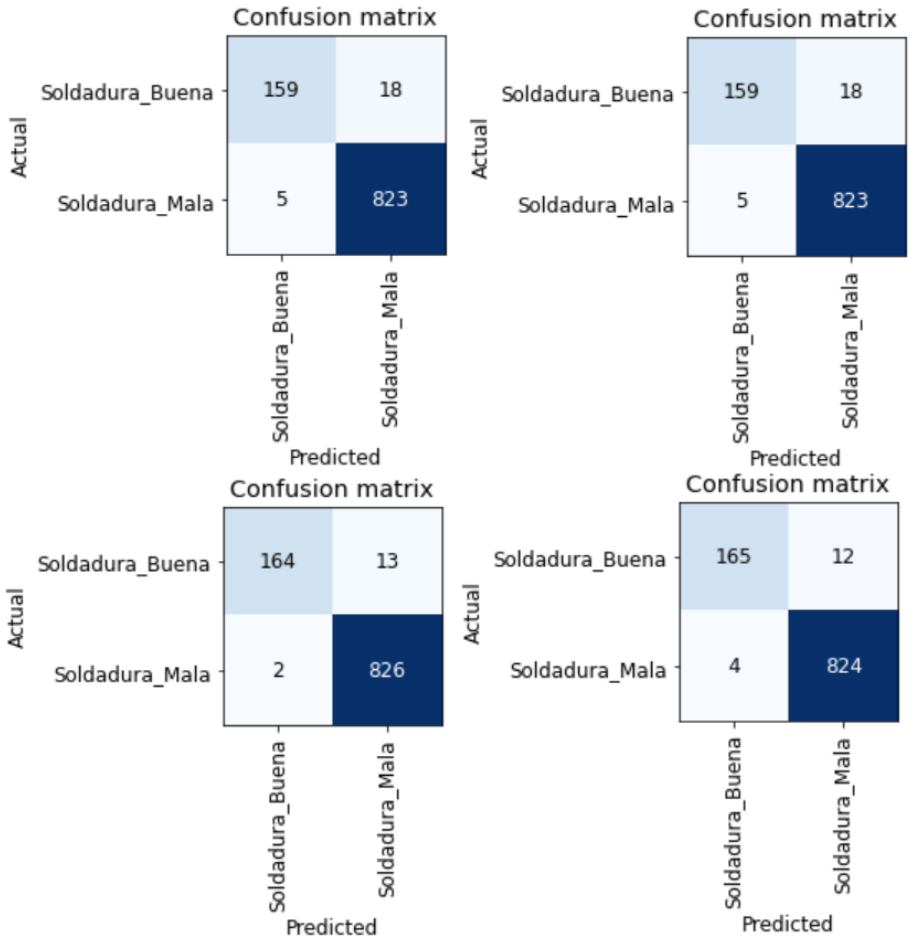

Figura 6.28: Matriz de confusión para tipo 8 para ResNet18, ResNet34, ResNet50 y ResNet101

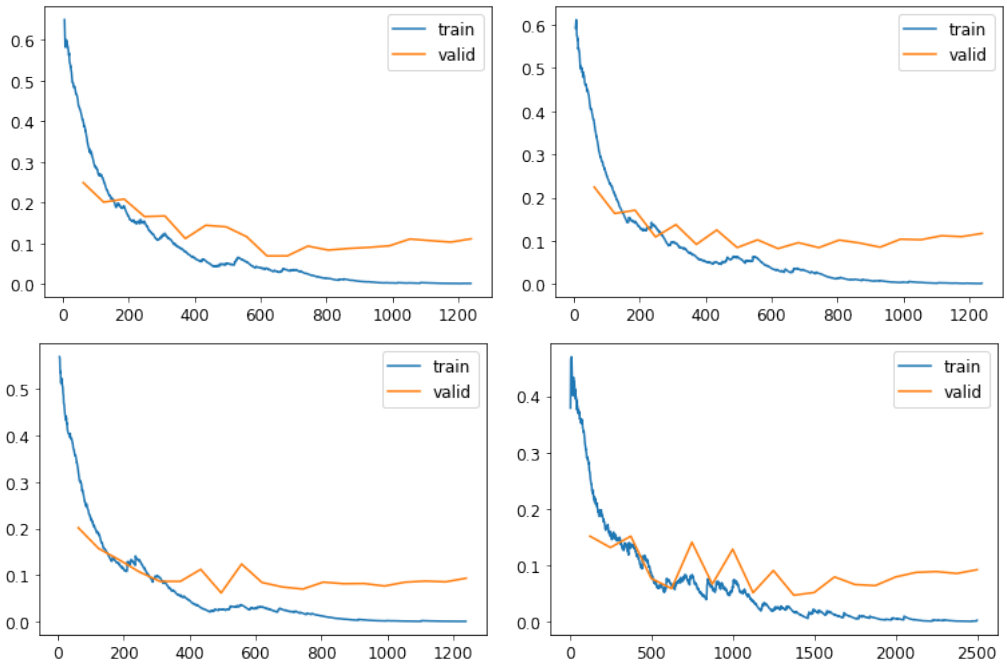

Figura 6.29: Evolución de pérdida en validación y entrenamiento para tipo 8 para ResNet18, ResNet34, ResNet50 y ResNet101

Los resultados vistos en los gráficos del "Tipo 8" hacen ver que la red no aprende de la forma que queremos, hay overfitting, por lo que intentaremos mejorarla en la siguiente iteración.

- **Tipo 9**

  - **Imágenes de soldaduras buenas**:592
  - **Imágenes de soldaduras malas**: 9472

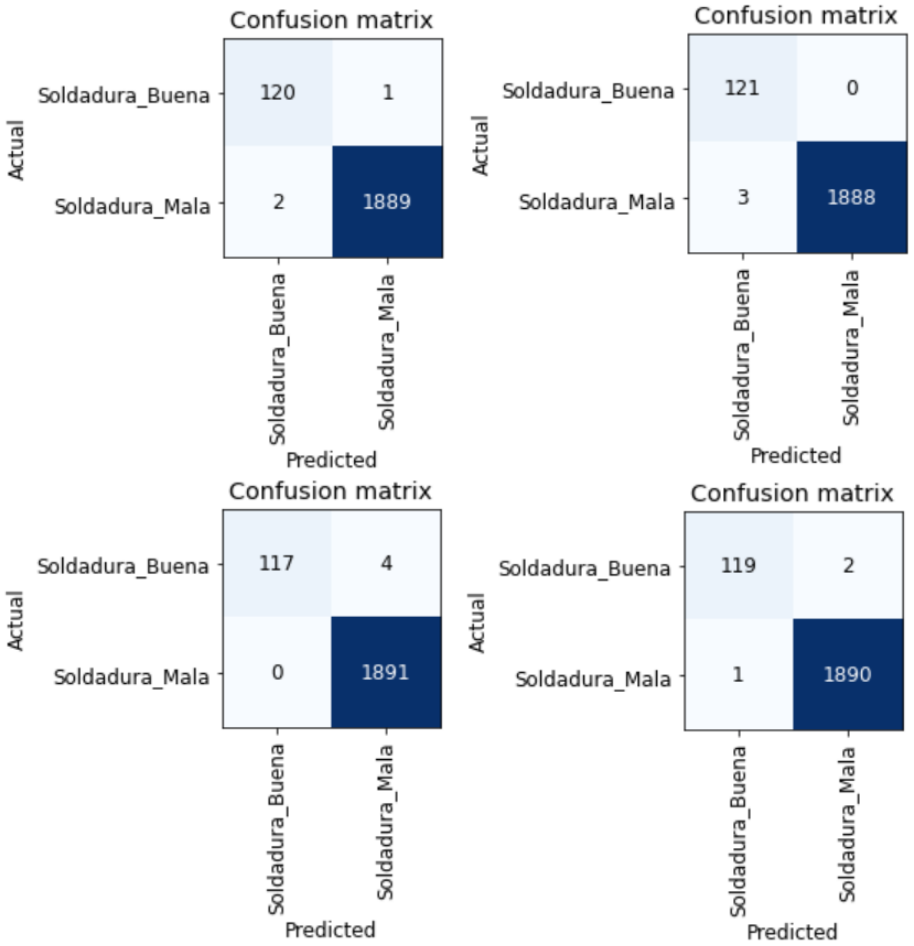

Figura 6.30: Matriz de confusión para tipo 9 para ResNet18, ResNet34, ResNet50 y ResNet101

Los resultados de esta tipología son buenos ya que el error no es muy grande al fallar entre tres y cuatro imágenes del conjunto de validación como máximo, por lo que no serán de prioridad.

- **Tipo 10**

  - **Imágenes de soldaduras buenas**: 298
  - **Imágenes de soldaduras malas**: 3576

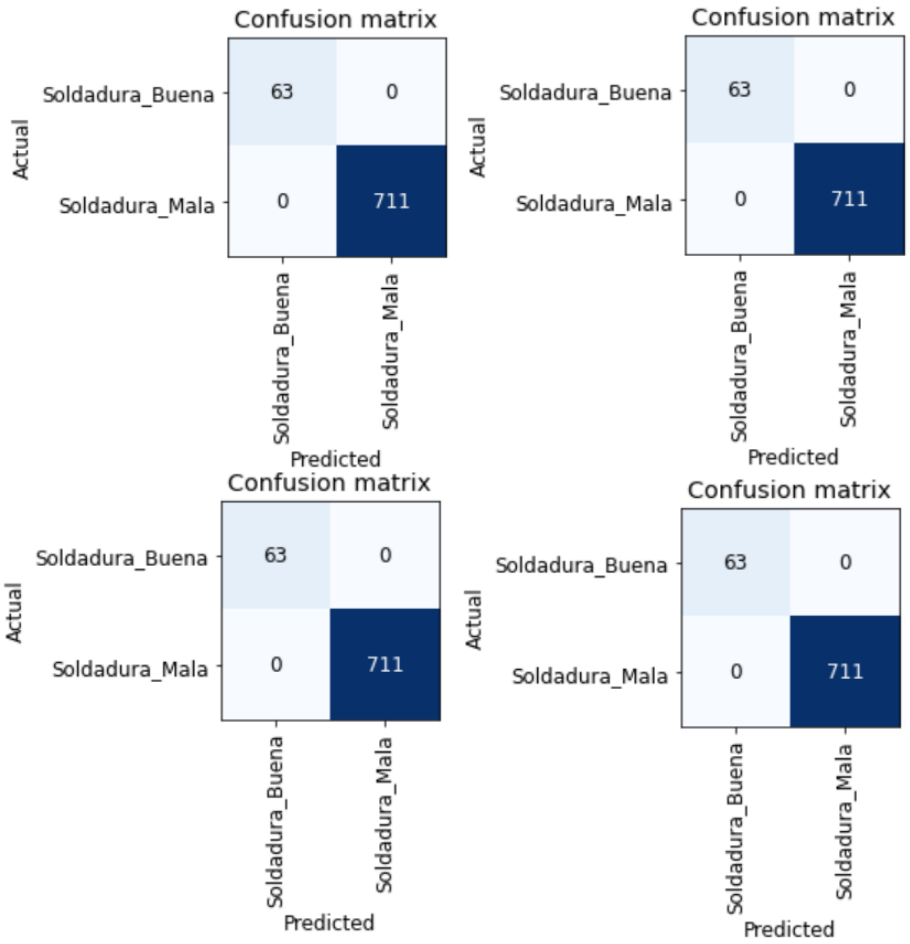

Figura 6.31: Matriz de confusión para tipo 10 para ResNet18, ResNet34, ResNet50 y ResNet101

Se trata de una tipología cuya tasa de error es 0 con lo cual nos indica que en este tipo la red diferencia muy bien las imágenes originales con las sintéticas, esto se deberá observar también ya que es posible que los defectos generados en las sintéticas disten mucho con la realidad y no tengan utilidad.

- **Tipo 11**

  - **Imágenes de soldaduras buenas**: 1776

  - **Imágenes de soldaduras malas**: 8882

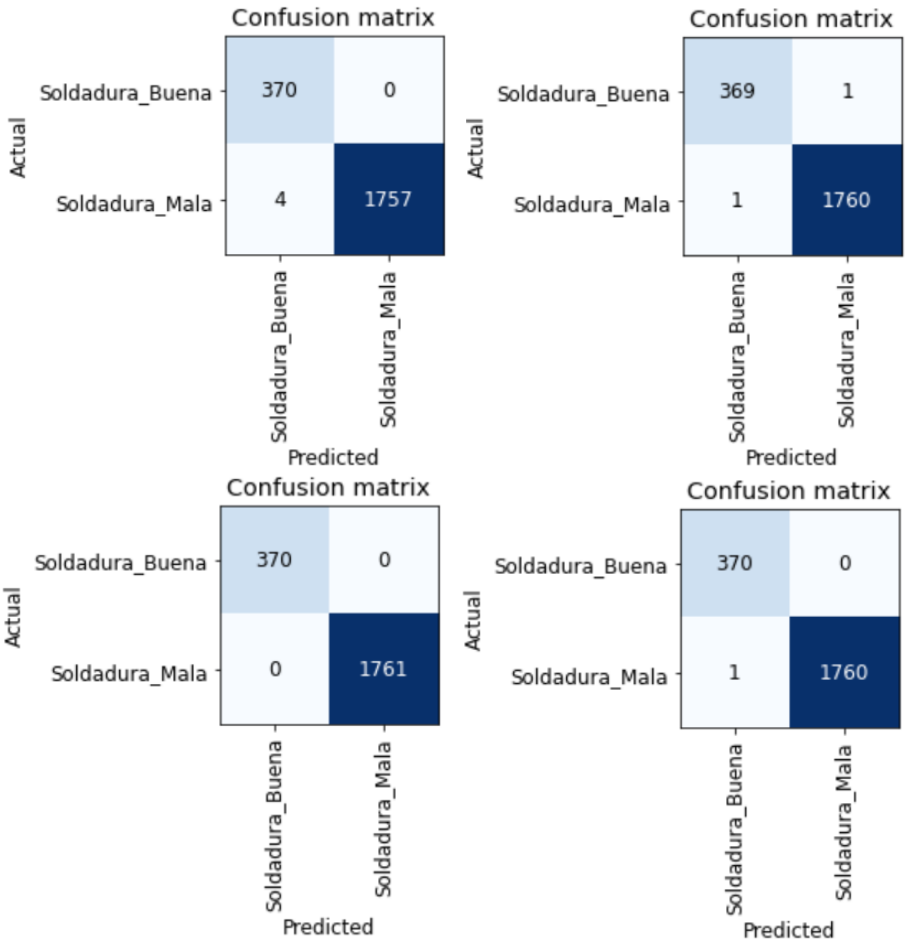

Figura 6.32: Matriz de confusión para tipo 11 para ResNet18, ResNet34, ResNet50 y ResNet101

La tipología "Tipo 11" obtiene resultados similares a los del "Tipo 2", donde casi no falla resultados en validación, al ser resultados bastante aceptables, hace que su mejora no sea prioritaria.

	DataSet1			
	ResNet18	ResNet34	ResNet50	ResNet101
Tipo 2	0.11%	0%	0.22%	0%
Tipo 3	0.61%	0.28%	0.28%	0.19%
Tipo 4	0.92%	0.97%	0.56%	0.61%
Tipo 5	2.88%	2.80%	2.33%	2.11%
Tipo 7	3.65%	3.78%	2.93%	3.11%
Tipo 8	2.29%	1.79%	1.19%	1.59%
Tipo 9	0.15%	0%	0.15%	0.15%
Tipo 10	0%	0%	0%	0%
Tipo 11	0.19%	0.09%	0%	0.05%

Figura 6.33: Tabla de resultados de la métrica tasa de error de clasificación para los diferentes tipos utilizando diferentes arquitecturas

Como podemos observar en la tabla que se encuentra justo encima, las tipologías más problemáticas son la 3, 4, 5, 7 y 8 y son las que debemos intentar tener en cuenta en futuras iteraciones.

6.4.4. Conclusiones

A la vista de los resultados obtenidos podemos sacar algunas conclusiones. En una primera instancia observamos que para los diferentes tipos de imágenes, las redes con más profundidad suelen dar mejores resultados que aquellas que tienen menor profundidad siendo las que mejores resultados obtienen normalmente las que usan transfer learning de la arquitectura ResNet101. También observamos que los tipos 3, 4, 5, 7 y 8 obtienen unos resultados peores que los de las demás tipologías, observando los conjuntos de datos vemos que existe una gran descompensación entre el conjunto de imágenes que pertenecen a la categoría de soldaduras malas frente al de soldaduras buenas, pudiendo ser uno de los motivos por los que algunos conjuntos no aprendan de forma correcta. Otra de las cosas que podemos observar gracias a los gráficos mostrados en las secciones de aquellos tipos con peores resultados es como evoluciona la función de perdida a lo largo del entrenamiento, dándose en algunos casos overfitting. Estas observaciones nos llevan a plantearnos si el problema de estos conjuntos de datos cuyos resultados no son tan buenos residen en la diferencia entre los dos subconjuntos de datos en los que clasificar.

6.5. Segunda iteración sobre el conjunto de datos: Equilibrado del conjunto de datos

6.5.1. Descripción del problema

Los resultados mostrados en la etapa anterior son correctos pero nos encontramos con tasas de error muy altas, como hemos visto en el anterior punto existe una diferencia muy elevada entre el conunto de soldaduras_buenas y el conjunto de soldaduras_malas, es por ello que realizamos la hipótesis de que existe un conjunto de entrenamiento insuficiente para uno de los conjuntos, es decir, al ser muy superior el conjunto de datos malos sobre el de datos buenos, la red está dando aquellas imágenes en las que presenta duda como malas, como ocurre con el conjunto de datos del 'Tipo 7".

6.5.2. Diseño del experimento

Debido a lo descrito anteriormente en esta sección se describe un experimento donde se han manipulado los conjuntos de soldaduras buenas multiplicando por seis su cantidad, para ello se han utilizado las siguientes transformaciones aplicadas tres veces respectivamente:

- La primera transformación consiste en sustituir el 2 % de los píxeles de la imagen de la soldadura, en caso de que aparezca parte de la chapa se la aísla, por píxeles con valor 0.
- La segunda transformación realiza el cambio de brillo de los píxeles de la imagen de forma aleatoria.

Además el tipo 7 fue dividido en dos subtipos diferentes para intentar mejorar sus resultados. También se debe mencionar que se han añadido imágenes al conjunto de malas que contienen solo la chapa e imágenes con colores sólidos.

Para el experimento hemos utilizado cuatro redes neuronales para cada tipo de soldadura, utilizando transfer learning con las arquitecturas ResNet18, ResNet34, ResNet50 y ResNet101, durante 20 épocas.

El código del experimento es igual al mostrado en la anterior iteración.

6.5.3. Resultados

Los resultados obtenidos en este experimento se detallan a continuación, mostrando las matrices de confusión y loas gráficas de pérdidas cuando resulta interesante.

- **Tipo 3**

  - **Imágenes de soldaduras buenas**: 5356
  - **Imágenes de soldaduras malas**: 8933

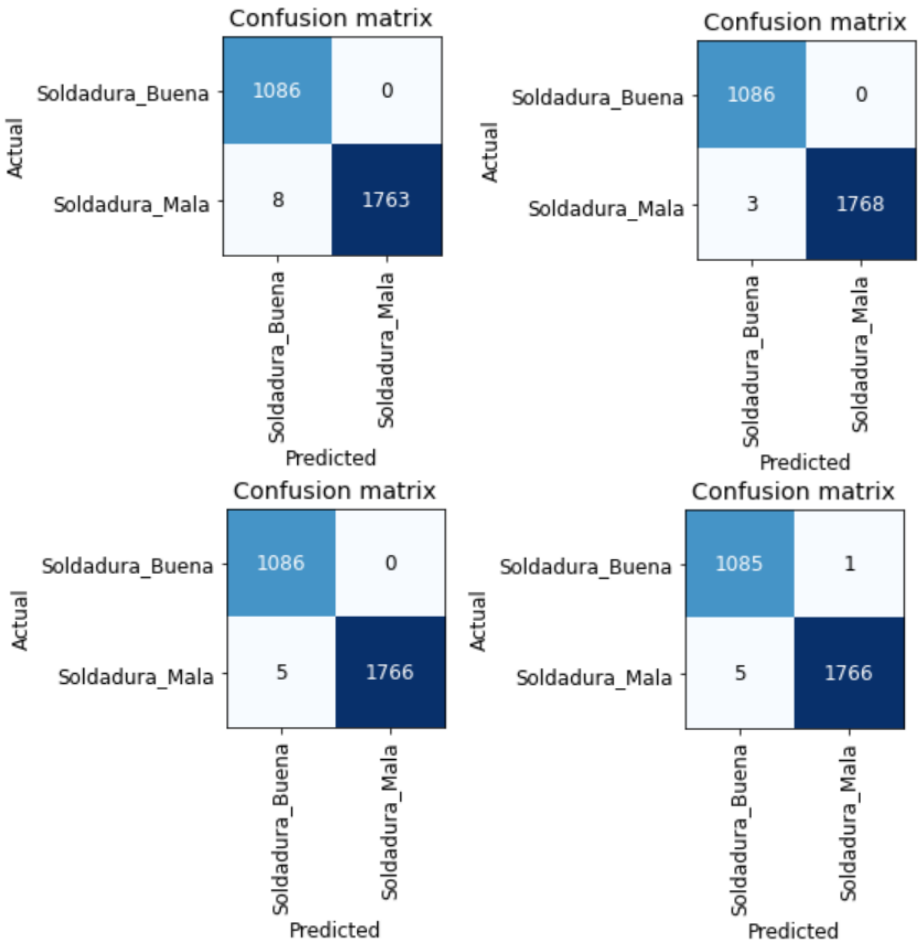

Figura 6.34: Matriz de confusión para tipo 3 para ResNet18, ResNet34, ResNet50 y ResNet101

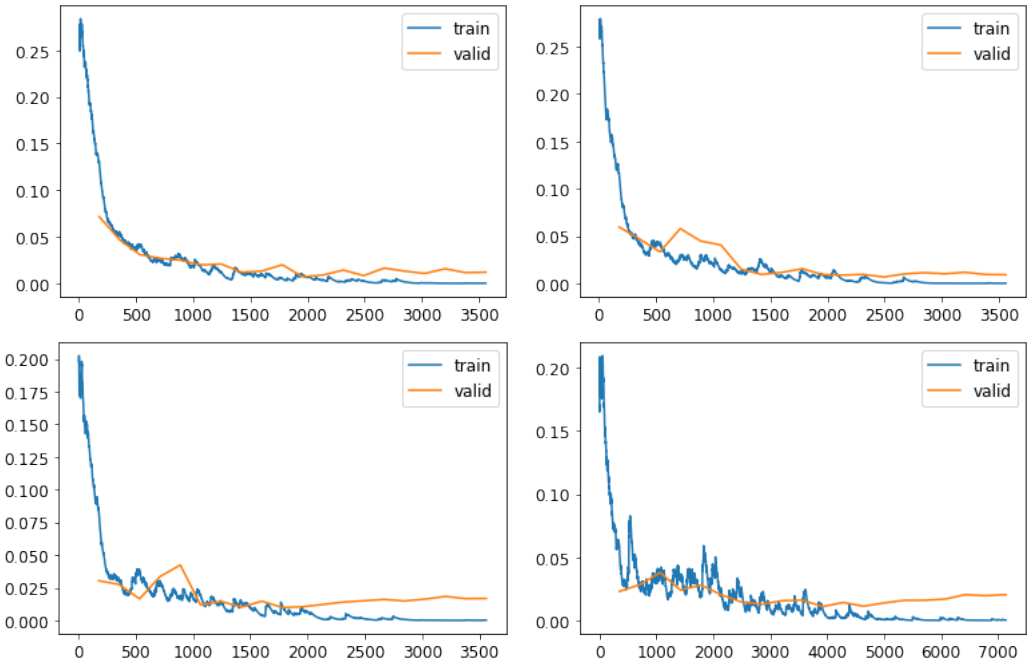

Figura 6.35: Evolución de pérdida en validación y entrenamiento para tipo 3 para ResNet18, ResNet34, ResNet50 y ResNet101

No se encuentra una gran mejoría en esta tipología en cuanto a número de fallos, pero al aumentar el número de imágenes esto hará que la tasa de error sea menor. Lo que podemos observar es que en arquitecturas complejas como ResNet50 y ResNet101 hay un comienzo de lo que podemos considerar overfitting,que no aparecía en la figura 6.4.3 y debe ser corregido en próximas iteraciones.

- **Tipo 4**

  - **Imágenes de soldaduras buenas**: 4413

  - **Imágenes de soldaduras malas**: 8310

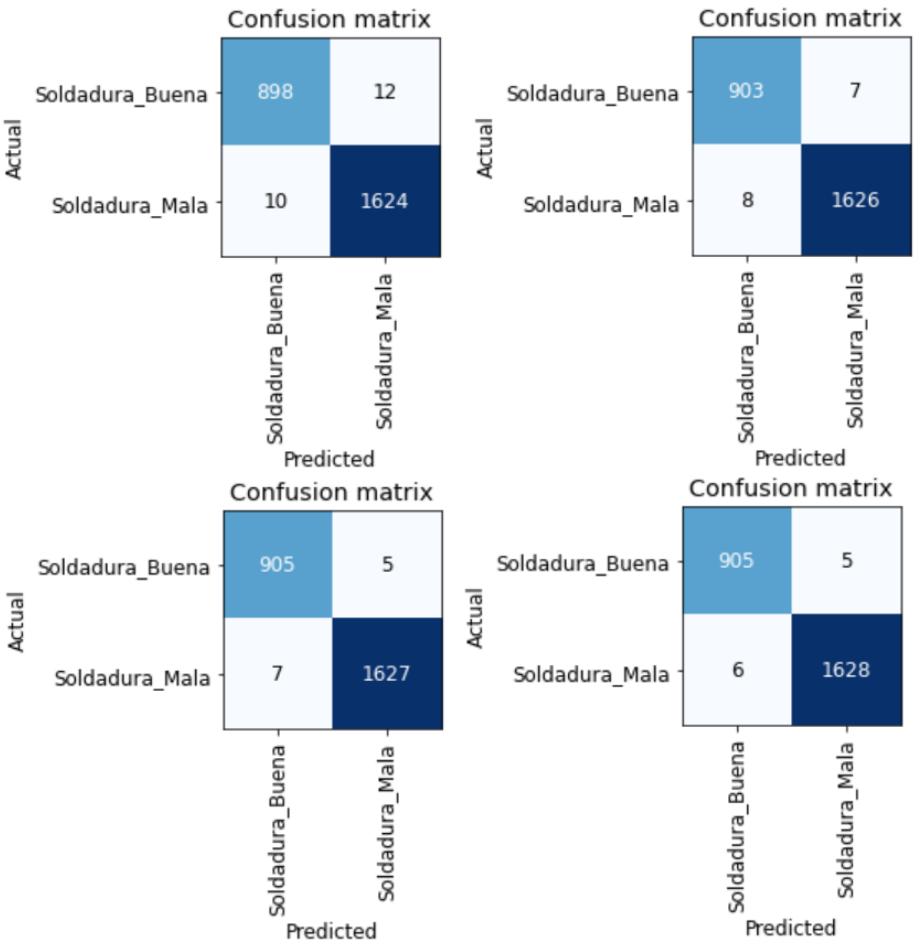

Figura 6.36: Matriz de confusión para tipo 4 para ResNet18, ResNet34, ResNet50 y ResNet101

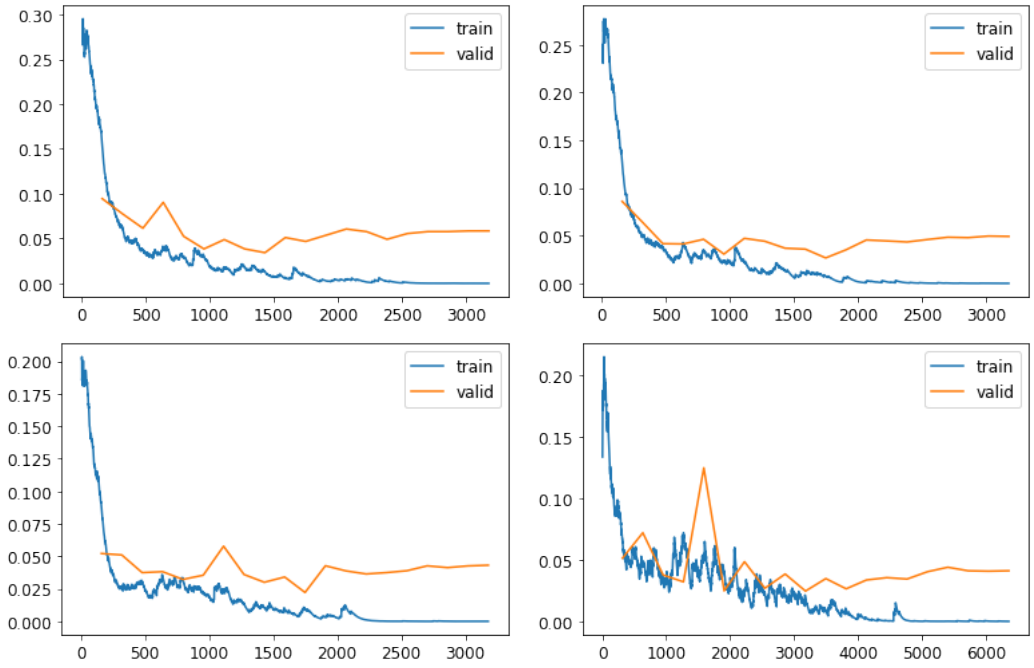

Figura 6.37: Evolución de pérdida en validación y entrenamiento para tipo 4 para ResNet18, ResNet34, ResNet50 y ResNet101

El “Tipo 4” sigue presentando problemas en cuanto a la evolución de su función de pérdida ya que sigue presentando overfitting, la función de pérdida tiende a aumentar, aunque si

bien es cierto los resultados en cuanto a porcentaje de error ha mejorado.

- **Tipo 5**

  - **Imágenes de soldaduras buenas**: 13672
  - **Imágenes de soldaduras malas**: 21483

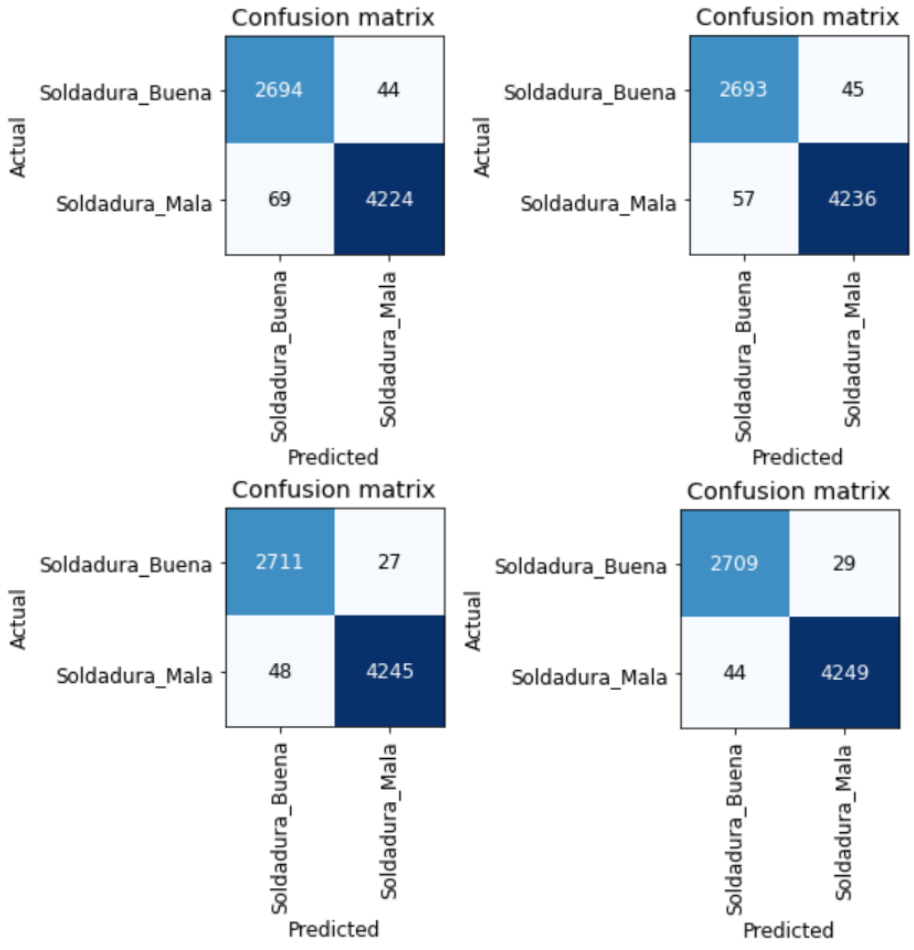

Figura 6.38: Matriz de confusión para tipo 5 para ResNet18, ResNet34, ResNet50 y ResNet101

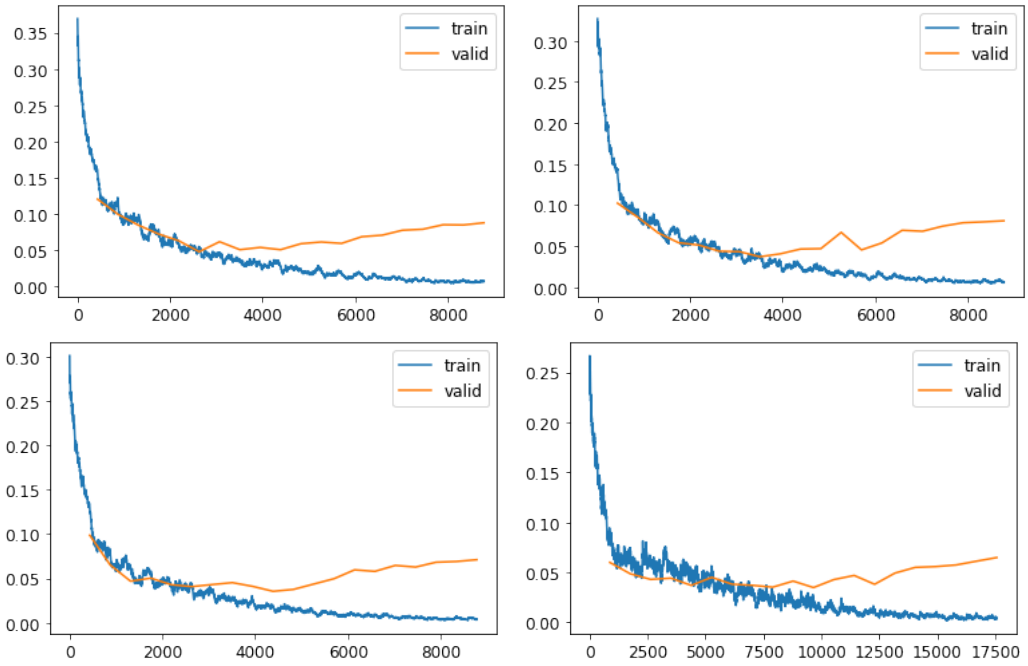

Figura 6.39: Evolución de pérdida en validación y entrenamiento para tipo 5 para ResNet18, ResNet34, ResNet50 y ResNet101

Respecto a este tipo hemos conseguido reducir el ritmo con el que la función de perdida en validación aumenta, haciendo que el overfitting sea menor, esto se puede ver reflejado en los resultados, aunque este tipo seguirá siendo objeto de mejora, ya que su error es elevado.

- **Tipo 7**

  - **Imágenes de soldaduras buenas**: 4183

  - **Imágenes de soldaduras malas**: 10664

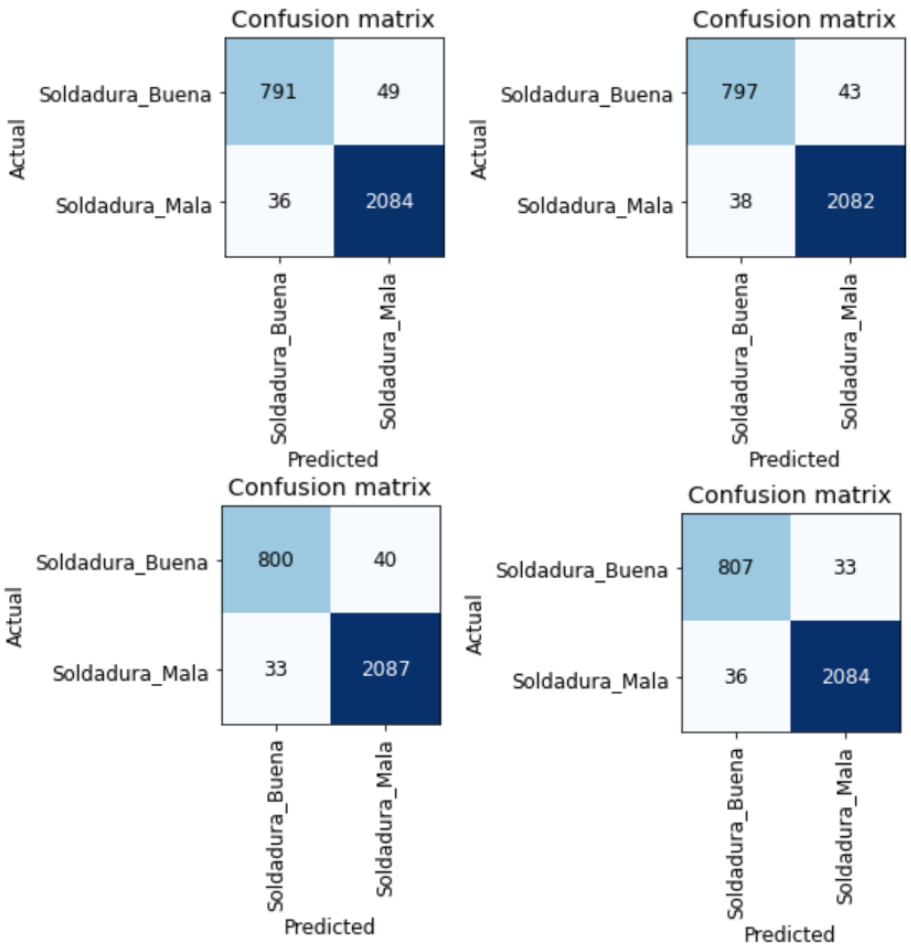

Figura 6.40: Matriz de confusión para tipo 7 para ResNet18, ResNet34, ResNet50 y ResNet101

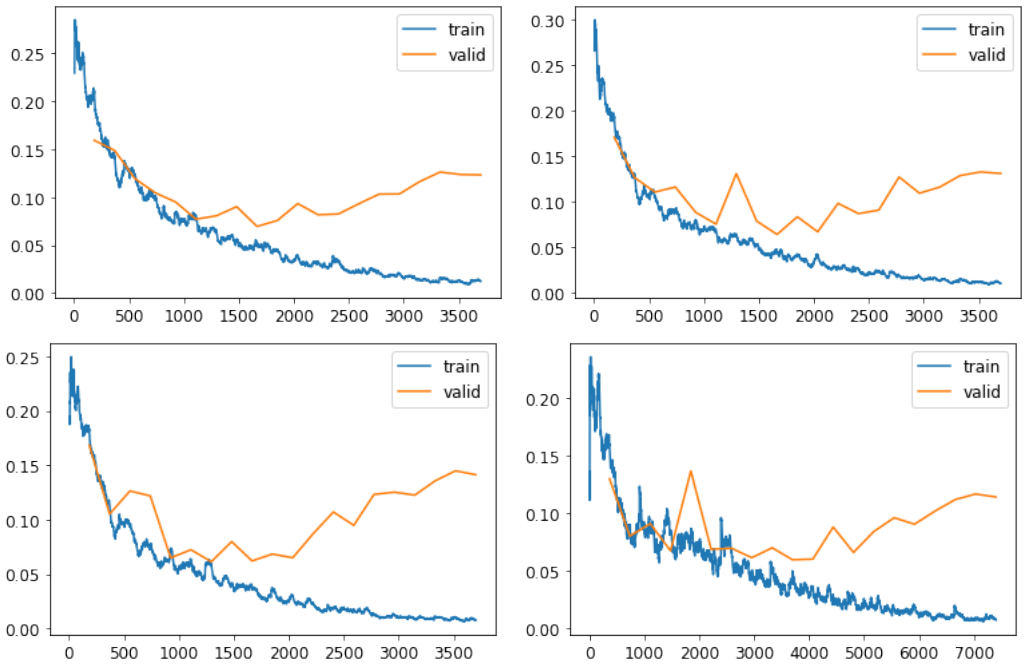

Figura 6.41: Evolución de pérdida en validación y entrenamiento para tipo 7 para ResNet18, ResNet34, ResNet50 y ResNet101

El experimento en este tipo no ha servido para lograr ningún tipo de mejora, se ha mantenido de forma similar e incluso ha aumentado la cantidad de imágenes que falla en validación,

aunque exista una mejora en la tasa de de error debido al aumento de datos. Esto nos hace pensar que el problema puede residir en los datos que dispone de forma que pueda haber datos mal clasificados, es decir, que se encuentren en el conjunto que no deben.

- **Tipo 7-Subtipo 1**

  - **Imágenes de soldaduras buenas**: 2066
  - **Imágenes de soldaduras malas**: 5040

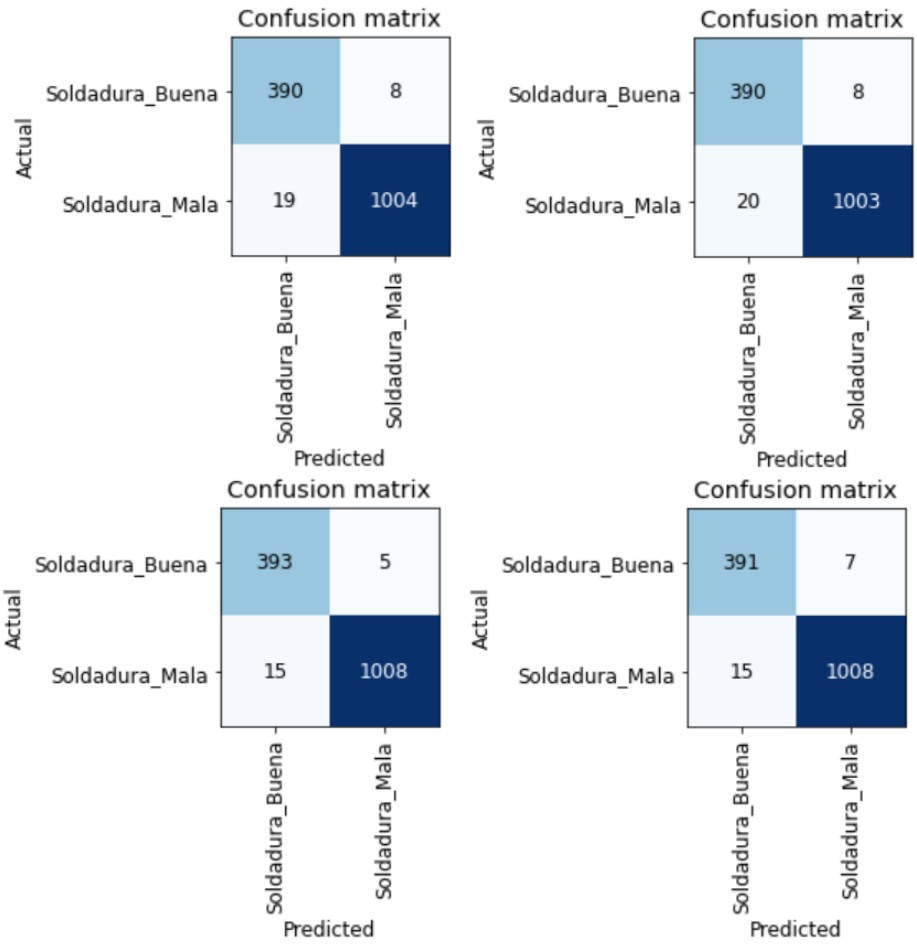

Figura 6.42: Matriz de confusión para tipo 7 subtipo 1 para ResNet18, ResNet34, ResNet50 y ResNet101

- **Tipo 7-Subtipo 2**

  - **Imágenes de soldaduras buenas**: 2072
  - **Imágenes de soldaduras malas**: 5632

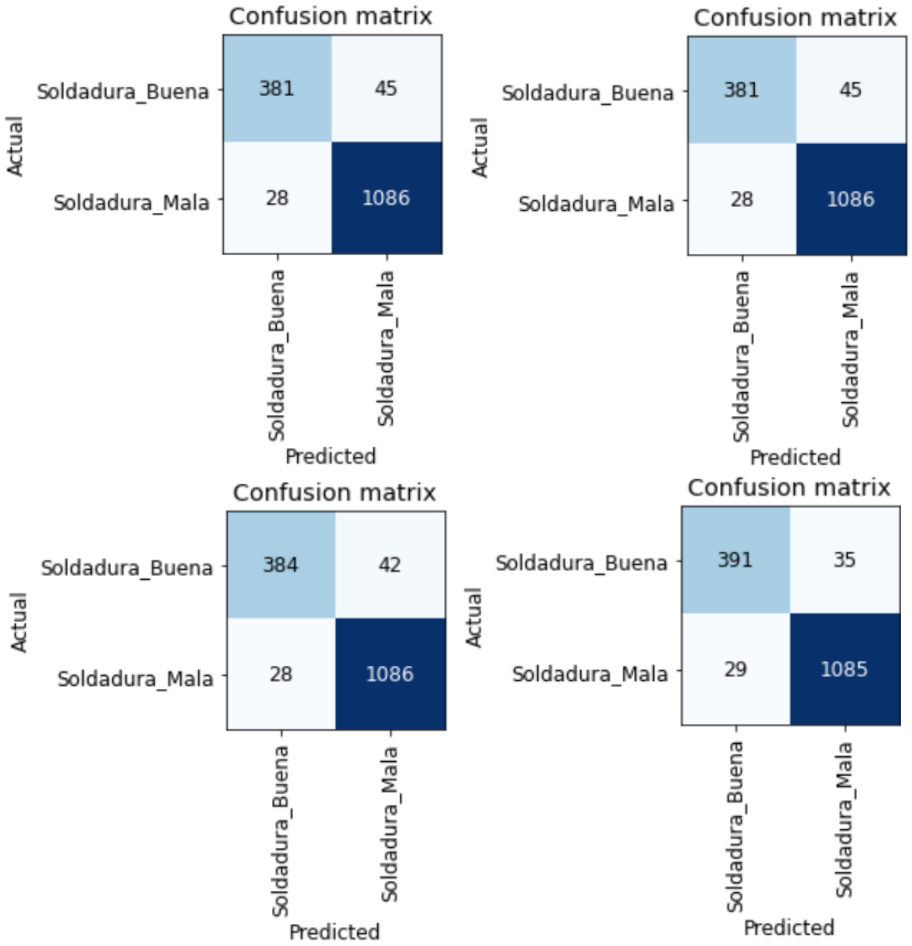

Figura 6.43: Matriz de confusión para tipo 7 subtipo 2 para ResNet18, ResNet34, ResNet50 y ResNet101

La división en dos subtipos no logra ninguna mejora con respecto a mantenerlo en el mismo conjunto, ya que obtiene resultados peores, fallando más imágenes de las que fallaba manteniendo los dos conjuntos unidos con lo que será una línea de investigación que acabaremos descartando.

- **Tipo 8**

  - **Imágenes de soldaduras buenas**: 2642
  - **Imágenes de soldaduras malas**: 4160

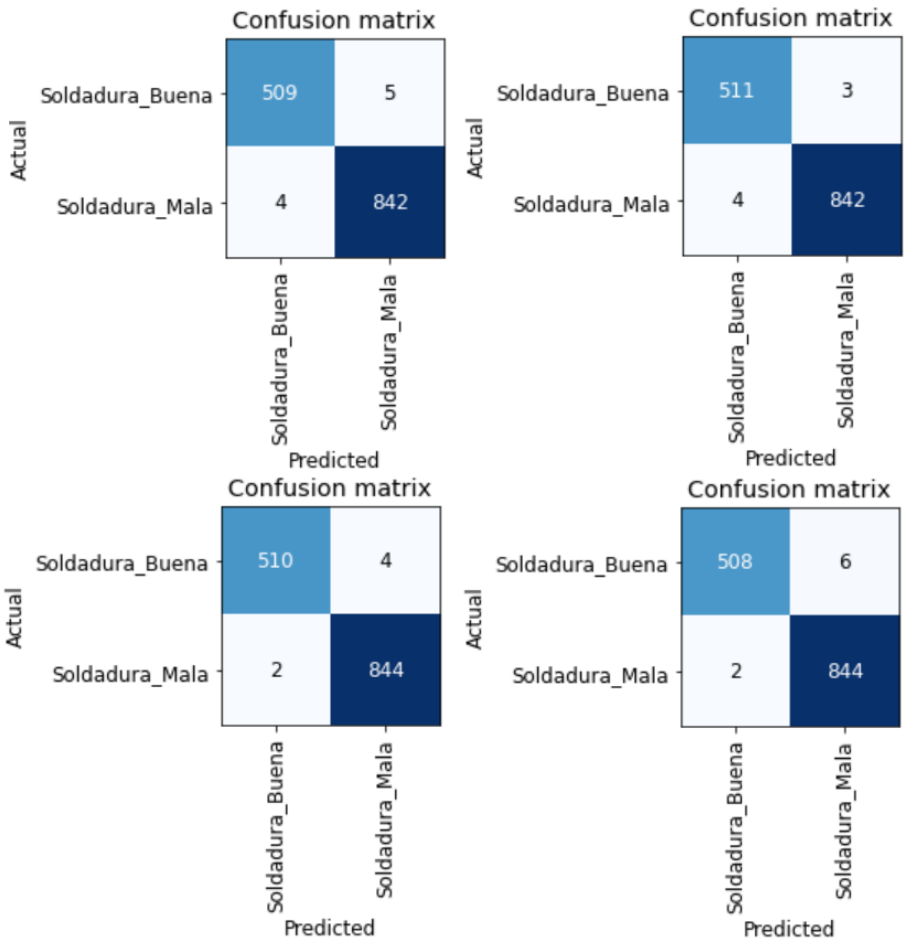

Figura 6.44: Matriz de confusión para tipo 8 para ResNet18, ResNet34, ResNet50 y ResNet101

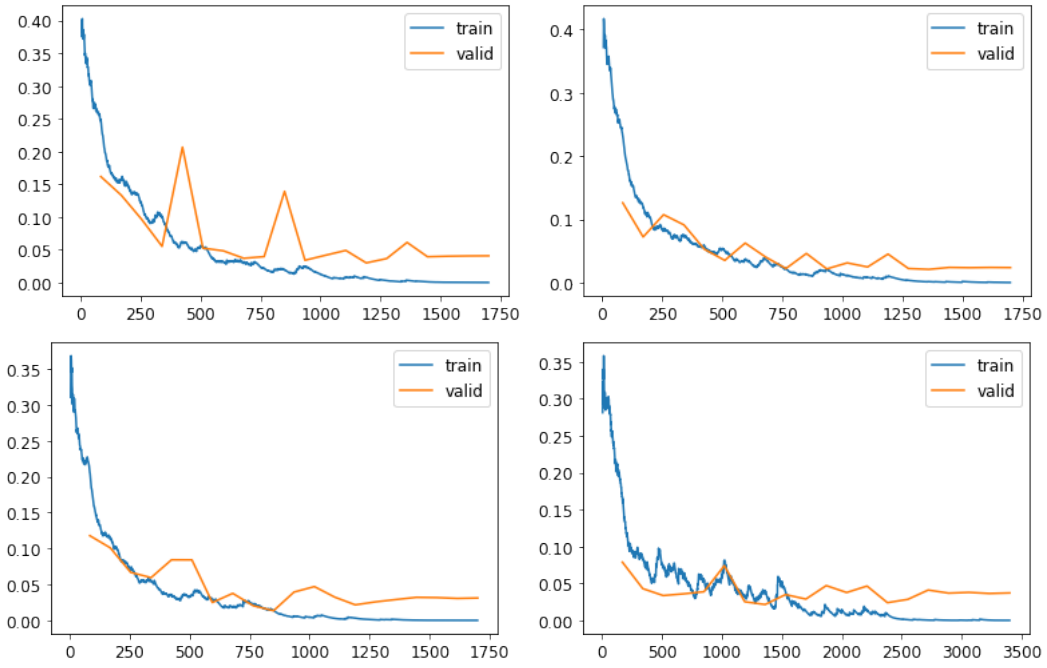

Figura 6.45: Evolución de pérdida en validación y entrenamiento para tipo 8 para ResNet18, ResNet34, ResNet50 y ResNet101

En esta tipología observamos, a diferencia de lo visto en la figura 6.4.3, como la función de perdida en validación se estabiliza, haciendo que desaparezca el overfitting, de forma que

también se ve reflejado en los resultados.

	DataSet2			
	ResNet18	ResNet34	ResNet50	ResNet101
Tipo 2	0.08%	0%	0%	0.08%
Tipo 3	0.28%	0.11%	0.18%	0.19%
Tipo 4	0.86%	0.59%	0.47%	0.43%
Tipo 5	1.61%	1.45%	1.07%	1.04%
Tipo 7	2.87%	2.74%	2.25%	2.33%
Tipo 7-1	1.90%	1.97%	1.40%	1.55%
Tipo7-2	4.74%	4.74%	4.54%	4.16%
Tipo 8	0.66%	0.51%	0.44%	0.59%
Tipo 9	0%	0%	0%	0%
Tipo 10	0%	0%	0%	0%
Tipo 11	0.09%	0.04%	0.04%	0.13%

Figura 6.46: Tabla de resultados de la métrica tasa de error de clasificación para los diferentes tipos utilizando diferentes arquitecturas

En la tabla mostrada se observa una mejora general con respecto a lo mostrado en la tabla 6.4.3 aunque el tipo 5,7 y 8 siguen siendo problemáticos.

6.5.4. Conclusiones

Lo primero que observamos a la vista de los resultados es que si existe una mejora en los tipos más problemáticos respecto al conjunto de datos del primer experimento, sobretodo en las tipologías 4 y 8, la tasa de fallos de las tipologías 5 y 7 aún es mejorable. Respecto al tipo 7 podemos ver que al separar las imágenes en dos subtipos no solo no vemos mejora si no que su tasa de fallo empeora, por lo que descartamos esta opción como posible mejora para la clasificación, por lo que posiblemente la tasa de error tan alta se deba a una mala clasificación dentro del conjunto de datos, habiendo datos que estén clasificados como buenos siendo malos y viceversa.

Otra conclusión que observamos en vista de las dos iteraciones realizadas es que las arquitecturas más complejas, si bien suelen obtener mejores resultados, el tiempo requerido es mayor y el resultado no es mucho mejor, con lo que para este trabajo de investigación y en vista a futuras iteraciones, a partir de este punto realizaremos solo la ejecución utilizando ResNet18.

6.6. Tercera iteración sobre el conjunto de datos: Limpieza del conjunto de datos

6.6.1. Descripción del problema

En vista de los resultados de la segunda iteración, seguimos observando que la red pese a obtener resultados correctos sigue presentando tasas de error altas en algunas tipologías de soldaduras. Esto, nos lleva a creer que una posibilidad de los resultados obtenidos sea que algunos datos estén clasificados en el conjunto que no deben, por lo que realizando una limpieza de los datos, moviendo o eliminando las imágenes que se detecten que se encuentren en el conjunto incorrecto.

6.6.2. Diseño del experimento

Para la realización de esta experimentación optamos por seguir dos líneas:

- Por un lado separaremos los defectos de las soldaduras malas con el fin de ver cuales son los defectos que más problemas dan a la red en términos de diferenciarlos de una soldadura buena. Para ello se crea una subcarpeta con cada uno de los defectos y se ejecuta el mismo código que en las secciones anteriores. Con ello obtendremos matrices de confusiones similares a la mostrada a continuación:

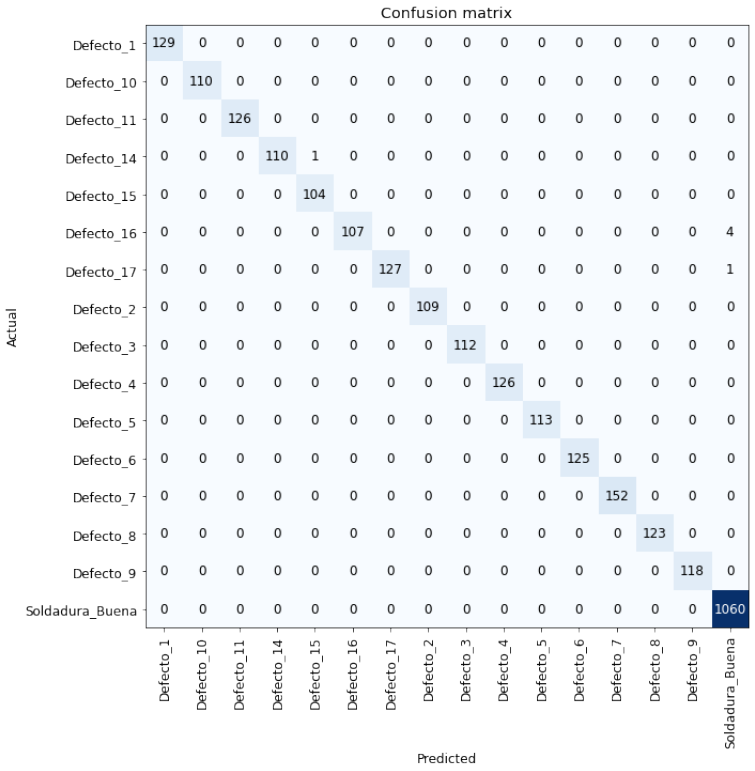

Figura 6.47: Matriz de confusión del Tipo 3 ejecutada separando los defectos en subcarpetas

  En ella nos fijaremos si existe un defecto que confunda con frecuencia con la categoría "Soldadura_Buena", en caso de que esto ocurra observaremos el porque, y si fuera necesario eliminaríamos la tipología de error o corregiríamos como se genera el tipo sintéticamente.

- Por otro lado usaremos una sección de codigo para imprimir aquellas imágenes que estén saliendo equivocadas en la matriz de confusión, o aunque su clasificación sea correcta, su función de pérdida es mayor, es decir lo que denominamos como "top_losses".

```
x = 1
for i in interp.top_losses(n).indices:
    print(f"[{x}] {dls.valid_ds.items[i]}")
    x += 1
```

  En ella la variable x nos indica la posición del índice, mientras que n la cantidad de imágenes que queremos que nos muestre en el "ranking".

Como al establecer una "semilla", la división en conjuntos de pérdida y validación no es aleatoria, lo que haremos es realizar diferentes ejecuciones de las mismas para ver si estos top_losses son diferentes modificando ese valor.

Lo que haremos es una vez sacadas las conclusiones de las dos líneas seguidas, tomar decisiones sobre aquellas imágenes que salgan en los top_losses: mantenerlas, cambiarlas de conjunto o eliminarlas del conjunto de datos. Volveremos a ejecutar el código de la red neuronal y analizaremos sus resultados. Si los resultados no fueran aceptables iteraremos mediante subiteraciones sobre ellos, siguiendo las líneas que acabamos de mencionar.

En este caso solo se ha realizado la ejecución de la red con transfer learning de la arquitectura ResNet18 ya que como hemos mencionado en la iteración 2 el tiempo de entrenamiento de las arquitecturas más complejas es mucho mayor que la de las arquitecturas más sencillas, y el problema de fondo de porque los clasificadores no funcionan tan bien como deberían, no es por la arquitectura elegida por lo visto en las dos iteraciones anteriores, ya que si bien los resultados mejoran con arquitecturas más complejas, las tasas de error siguen siendo altas con estas en los tipos más problemáticos. Seguiremos utilizando como en los experimentos anteriores 20 épocas.

6.6.3. Resultados

Al igual que en las dos anteriores iteraciones se muestran los resultados de la realización del experimento. En ella se muestran las matrices de confusión de la última subiteración realizada dentro de la propia tercera iteración del experimento. También se muestra los gráficos de pérdida de aquellas tipologías problemáticas en la anterior iteración.

- **Tipo 5**

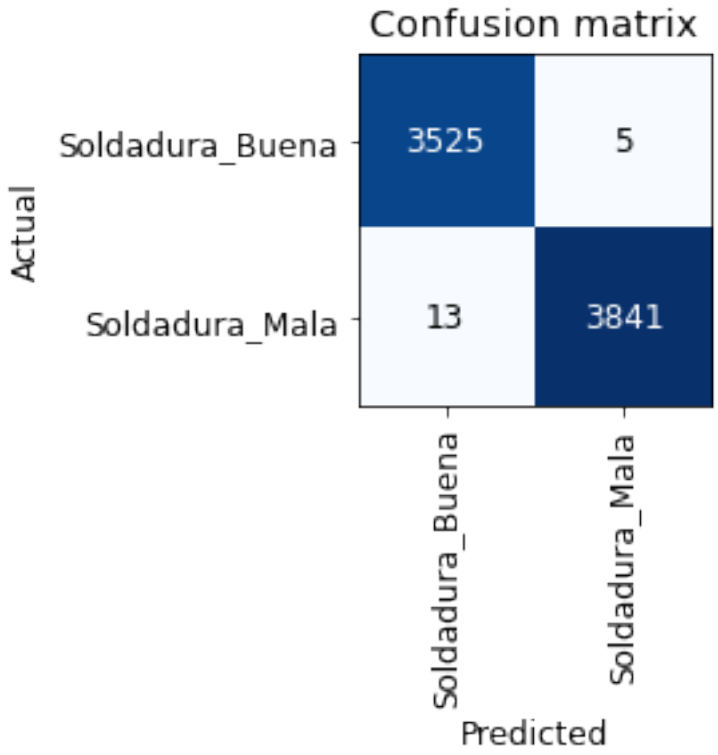

Figura 6.48: Matriz de confusión para tipo 5 para ResNet18

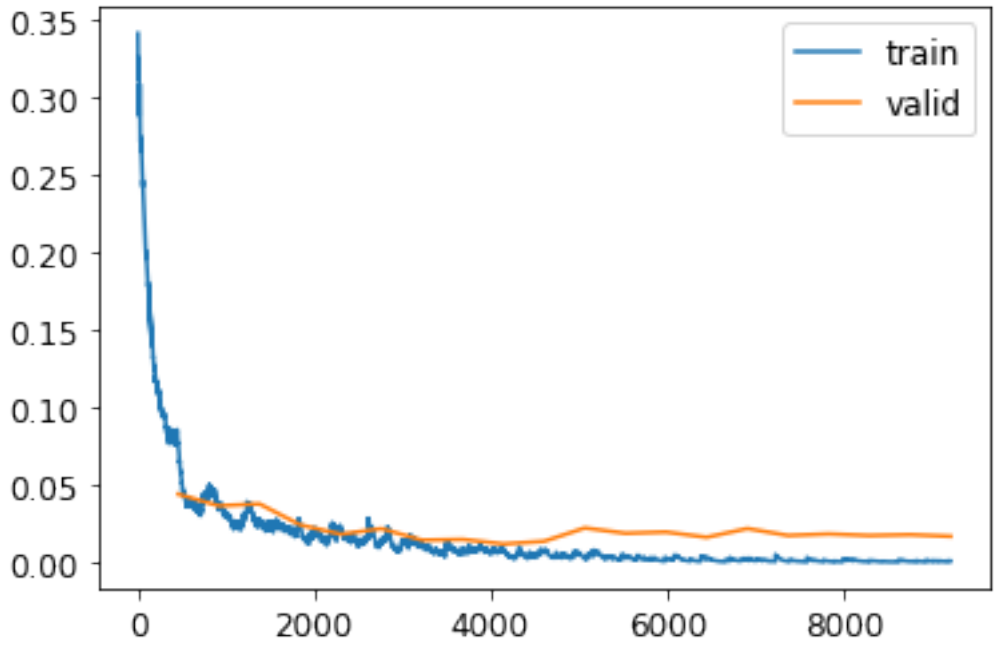

Figura 6.49: Evolución de pérdida en validación y entrenamiento para tipo 5 para ResNet18

El "Tipo 5" recibe una gran mejora en sus resultados, una de las cosas que hacen que esto ocurra es la regeneración de los defectos de cortes buenos, ya que al tener un tamaño de corte incorrecto hacia que estos fueran confundidos, esto también se refleja en la gráfica de pérdida.

- **Tipo 7**

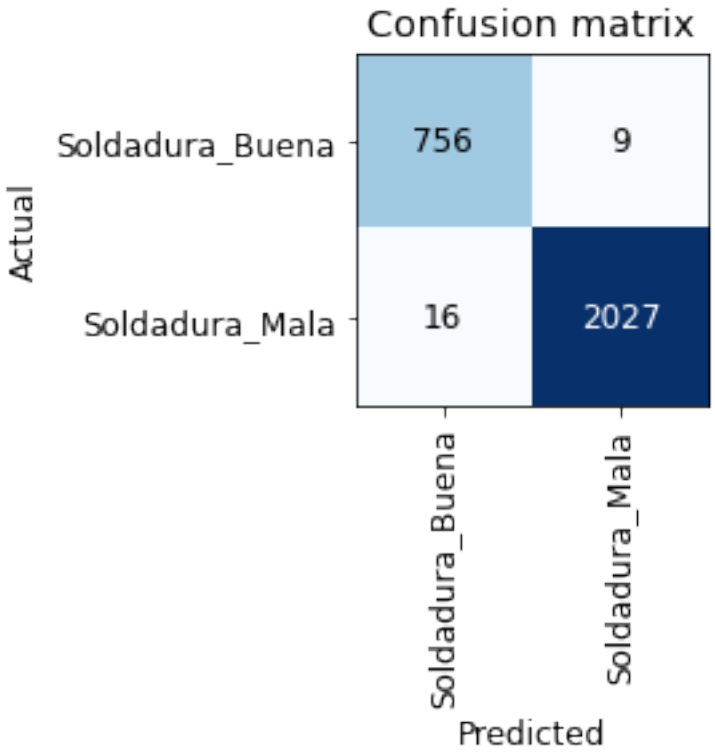

Figura 6.50: Matriz de confusión para tipo 7 para ResNet18

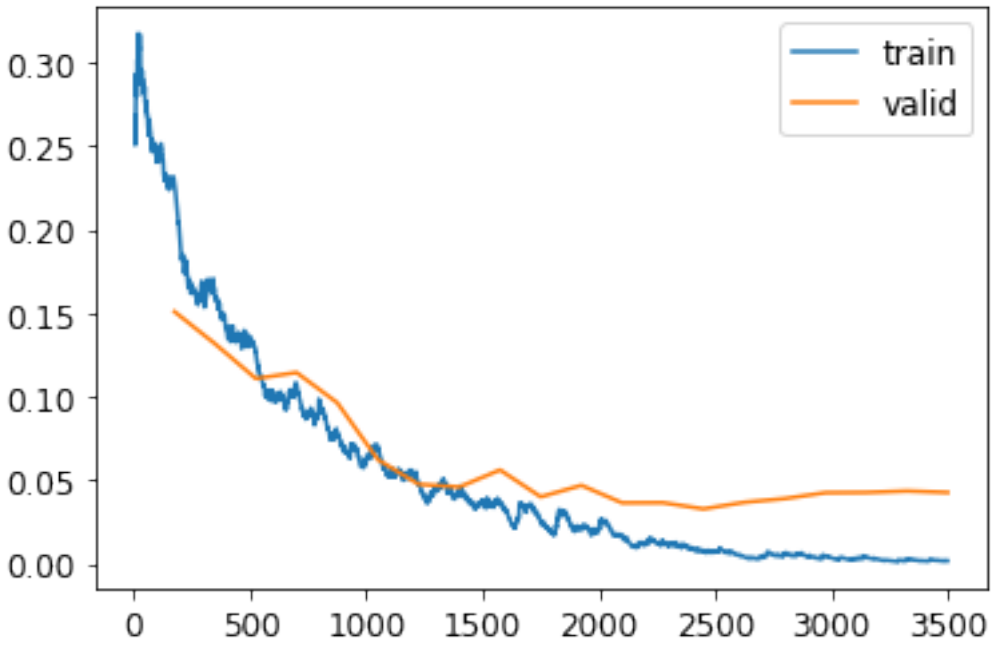

Figura 6.51: Evolución de pérdida en validación y entrenamiento para tipo 7 para ResNet18

En este conjunto de datos también se observa una notable mejora, ya que al corregir muchas imágenes mal clasificadas la red es capaz de detectar los defectos que se la piden. La gráfica de pérdida mejora notablemente respecto a la figura 6.5.3.

	DataSet3
	ResNet18
Tipo 2	0%
Tipo 3	0.10%
Tipo 4	0.30%
Tipo 5	0.20%
Tipo 7	0.89%
Tipo 8	0.40%
Tipo 9	0%
Tipo 10	0%
Tipo 11	0%

Figura 6.52: Tabla de resultados de la métrica tasa de error de clasificación para los diferentes tipos utilizando diferentes arquitecturas

Como vemos los resultados mejoran notablemente al limpiar el conjunto de datos, ya que de la otra forma lo que hacia era ver que ciertos elementos que se repetían en las imágenes de un conjunto aparecieran en el otro haciendo así que la red se confundiera o buscara otros elementos para diferenciar los conjuntos.

6.6.4. Conclusiones

Realizando estos ajustes en esta iteración hemos conseguido mejorar notablemente las tasas de error, esto se debe a que si ponemos imágenes erróneas en un subconjunto que no es el que pertenecen hace que la red no distinga el porque esa imagen pertenece a ese subconjunto pudiendo hacer que busque en otras imágenes características distintas a las que debe encontrar.

Al separar entre los diferentes defectos en las ejecuciones intermedias también ha ayudado a poder observar que subtipos de soldaduras malas confundía con soldaduras malas, teniendo que regenerar en algunos casos como en el tipo 5, el defecto de cortes buenos ya que tenían el mismo tamaño que los cortes malos.

Los resultados obtenidos parecen lo suficientemente correctos para tener una primera versión sobre las redes, de forma que se puedan usar para empezar a analizar soldaduras en un proceso de calidad. Durante el rodaje de la máquina se deberán introducir las imágenes que hagan fallar a nuestra red, de modo que ella misma sea capaz de aprender de sus fallos.

6.7. Conclusiones

Después de realizarse las tres iteraciones se han conseguido resultados bastante correctos para los diferentes tipos de soldadura, consiguiendo una primera clasificación que permite su integración en un proceso de producción. No obstante el error en algunas tipologías sigue siendo elevado, como es el caso del "Tipo 7", esto nos lleva a pensar en la posibilidad de realizar nuevos experimentos basados en técnicas aún no exploradas para intentar mejorar estos resultados.

Capítulo 7

Desarrollo de técnicas para la mejora de la exactitud

7.1. Introducción

Una vez obtenido un primer análisis de las tipologías previamente estudiadas mediante clasificadores utilizando la técnica de transferencia de conocimiento, podemos ver como no siempre se obtienen los resultados deseados. Por este motivo, en este capítulo, analizaremos las soldaduras de "Tipo 7" y exploraremos diferentes técnicas con el objetivo de mejorar los resultados obtenidos.

7.2. Análisis del Tipo 7

En vista de los resultados obtenidos en la tercera iteración, el "Tipo 7" se trata de la tipología que obtiene peores resultados, fallando 9 imágenes de Soldaduras_Buena, dándolas como Soldaduras_Mala y 16 imágenes de Soldaduras_Mala dándolas como Soldadura_Buena. Lo primero que haremos es realizar un estudio de cuales son estas imágenes que la red es incapaz de clasificar de forma correcta en ellas observamos imágenes como las siguientes:

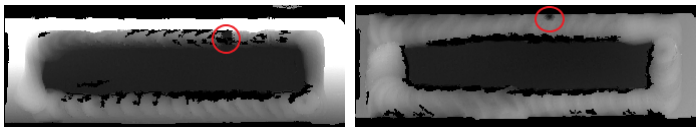

Figura 7.1: Imágenes clasificadas incorrectamente por la red de la tercera iteración

Ambas imágenes se encuentran en el conjunto correcto, Soldadura_Mala, ya que presentan defectos en el caso de la imagen de la izquierda un corte y en el de la derecha un poro. En vista de que los defectos si existen y aun así la clasificación es errónea nos lleva a plantearnos la posibilidad de usar diferentes técnicas que permitan la mejora de resultados.

7.3. Red desde cero

7.3.1. Descripción del problema

Una vez explotadas las redes preentrenadas utilizando transfer_learning es el momento de utilizar nuevas líneas de investigación que intenten obtener mejores resultados a los ya vistos, en este caso empezaremos con utilizar una red neuronal desde cero simple que permita servir de punto de partida a redes más complejas.

7.3.2. Descripción del experimento

Una primera aproximación para el problema es descartar el transfer_learning para optar por realizar una red neuronal desde cero, haciendo que su primer entrenamiento sea con las imágenes de las soldaduras de "Tipo 7". Lo primero que haremos es crear nuestra capa convolucional:

```
def conv(ni, nf, ks=3, act=True):
    layers = [nn.Conv2d(ni, nf, stride=2, kernel_size=ks, padding=ks//2)]
    layers.append(nn.BatchNorm2d(nf))
    if act: layers.append(nn.ReLU())
    return nn.Sequential(*layers)
```

En la que tendremos una capa convolucional, una capa de normalización y dependiendo de la entrada añadiremos una capa con la función ReLU o una secuencial. Posteriormente crearemos la red neuronal, para ello utilizaremos la siguiente estructura:

```
simple_cnn= sequential(
    conv(3 ,64),
    conv(64 ,128),
    conv(128 ,128),
    conv(128 ,128),
    conv(128 ,128),
    Flatten(),
    nn.Linear(5632,5632),
    nn.Linear(5632,5632),
    nn.Linear(5632,2)
)
```

En la que tenemos 4 capas convolucionales en las que obtenemos una salida de tamaño [64, 128, 4, 11] por lo que después de "Flatten()" el tamaño es 128 * 4* 11, es decir [64,5632] que es el tamaño que aplicaremos a las capas lineales salvo a la última cuya salida debe ser 2 ya que tendrá dos salidas posibles, Soldadura_Buena y Soldadura_Mala. Entrenaremos la red durante 20 épocas.

7.3.3. Resultados

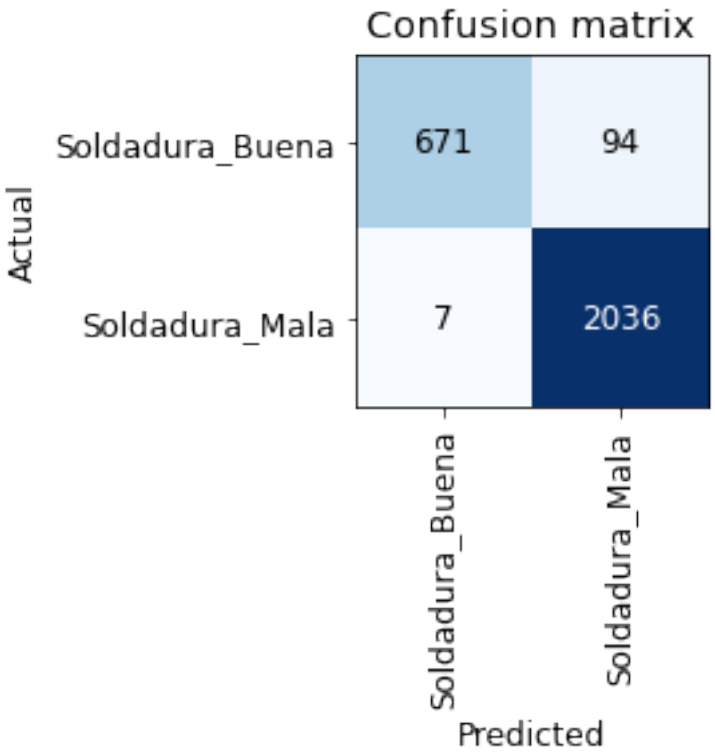

Figura 7.2: Matriz de confusión obtenida para una red creada desde cero utilizando 20 épocas de entrenamiento utilizando el conjunto de imágenes del "Tipo 7"

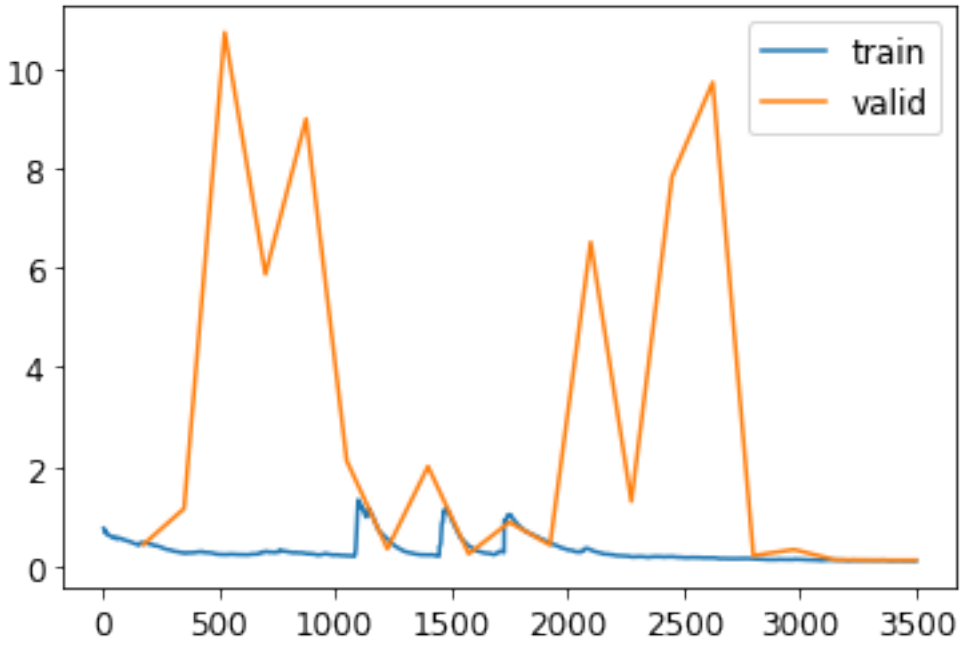

Figura 7.3: Gráfico de la evolución de la función de pérdida en validación y en entrenamiento para una red creada desde cero utilizando 20 épocas de entrenamiento utilizando el conjunto de imágenes del "Tipo 7"

Los resultados obtenidos en la matriz de confusión son peores a los obtenidos mediante transfer_learning en los que observamos que existe un alto caso de imágenes de la categoría Soldadu-

ra_Buena que son confundidos y clasificados en la otra categoría mientras que en el caso de la otra categoría esto ocurre con menor frecuencia, tan solo 7 casos. Otra cosa que se debe tener en cuenta es el proceso de evolución de la función de pérdida donde la pérdida en validación sigue una trayectoria en la que crece y decrece en dos ocasiones, aunque al final decrece haciendo parecer que no existe overfitting. La tasa de error obtenida en este experimento es de **3.6 %**.

7.3.4. Conclusiones

Como se ha detallado, el modelo no mejora los resultados obtenidos en la sección anterior habiendo una tasa de error mayor de aproximadamente un 2 %, pero se trata de un modelo creado que podría ser mejorado añadiendo capas o modificándolas. Por tanto observando los resultados concluimos que pese a poder existir un modelo de red neuronal desde cero que permita la mejora de resultados, encontrar la combinación que eso ocurra se trataría de un proceso largo, en el que normalmente obtendríamos resultados peores a realizar el entrenamiento utilizando transfer_learning.

7.4. Bag of tricks

7.4.1. Descripción del problema

Una vez visto que una red neuronal simple no obtiene resultados mejores, intentaremos volver al modelo de ResNet que fue usado cuando realizamos la primera experimentación. Para ello utilizaremos una red basada en ResNets en las que se aplican mejoras, lo que denominamos "Bag of Tricks" esperando que los resultados sean mejores que los obtenidos hasta ahora.

7.4.2. Descripción del experimento

En "Bag of Tricks for Image Classification with Convolutional Neural Networks"[7], Tong He estudió distintas variaciones de ResNet que no suponen casi ningún coste adicional en términos de número de parámetros o cálculo. Utilizando una arquitectura ResNet50 modificada y Mixup, consiguiendo una precisión del 94,6 % en el top-5 de ImageNet, en comparación con el 92,2 % de una ResNet50 normal sin Mixup. Este resultado es mejor que el obtenido por los modelos ResNet normales, que son el doble de profundos (y el doble de lentos, y mucho más propensos a tener overfitting).

Una implementación de una ResNet moderna, que utilice una "bag of tricks" podría ser la siguiente que utiliza cuatro grupos ResNet con 64, 128, 256 y luego 512 filtros. Cada grupo comienza con un bloque stride-2, excepto el primero, ya que está justo después de una capa MaxPooling:

```
class ResNet(nn.Sequential):
    def __init__(self, n_out, layers, expansion=1):
        stem = _resnet_stem(3,32,32,64)
        self.block_szs = [64, 64, 128, 256, 512]
        for i in range(1,5): self.block_szs[i] *= expansion
        blocks = [self._make_layer(*o) for o in enumerate(layers)]
```

```
        super().__init__(*stem, *blocks,
                         nn.AdaptiveAvgPool2d(1), Flatten(),
                         nn.Linear(self.block_szs[-1], n_out))

    def _make_layer(self, idx, n_layers):
        stride = 1 if idx==0 else 2
        ch_in,ch_out = self.block_szs[idx:idx+2]
        return nn.Sequential(*[
            ResBlock(ch_in if i==0 else ch_out, ch_out, stride if i==0 else 1)
            for i in range(n_layers)
        ])
```

Para hacer que nuestro modelo sea más profundo sin consumir demasiados cálculos o memoria, podemos utilizar otro tipo de capa introducida por el artículo de ResNet para ResNets con una profundidad de 50 o más: la capa cuello de botella.

En lugar de apilar dos convoluciones con un tamaño de núcleo de 3, las capas de cuello de botella utilizan tres convoluciones diferentes: dos 1×1 (al principio y al final) y una 3×3, como se muestra en la siguiente imagen

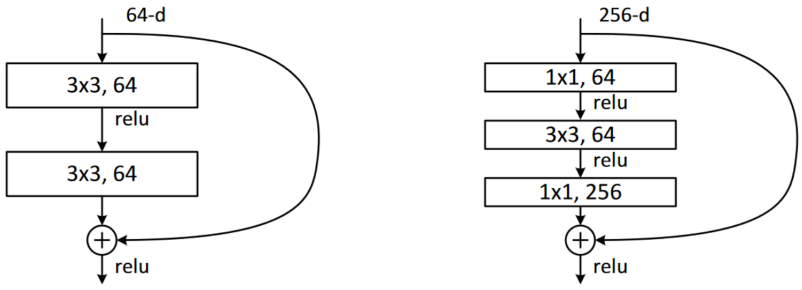

Figura 7.4: Diseño de cuello de botella para redes residuales tipo ResNet [1]

Esto es útil debido a que las convoluciones 1×1 son mucho más rápidas, con lo que este bloque se ejecuta más rápido. Esto nos permite utilizar más filtros, lo que disminuye el número de canales haciendo que podamos utilizar más filtros en el mismo tiempo.

Para ello utilizaremos en la clase ResBlock este diseño de cuello de botella:

```
def _conv_block(ni,nf,stride):
    return nn.Sequential(
        ConvLayer(ni, nf//4, 1),
        ConvLayer(nf//4, nf//4, stride=stride),
        ConvLayer(nf//4, nf, 1, act_cls=None, norm_type=NormType.BatchZero))

class ResBlock(Module):
    def __init__(self, ni, nf, stride=1):
        self.convs = _conv_block(ni,nf,stride)
        self.idconv = noop if ni==nf else ConvLayer(ni, nf, 1, act_cls=None)
        self.pool = noop if stride==1 else nn.AvgPool2d(2, ceil_mode=True)
```

```
    def forward(self, x):
        return F.relu(self.convs(x) + self.idconv(self.pool(x)))
```

El diseño de cuello botella suele ser utilizado en los modelos ResNet50, 101 y 152, mientras que los modelos ResNet18 y 34 suelen utilizar el diseño sin cuello de botella, aunque estas pueden tener mejores resultados utilizándolo.

Usaremos una función que nos devuelva el modelo de ResNet utilizando esta técnica en nuestro caso como queremos utilizar una ResNet18 le daremos los parámetros de entrada que se muestran en la siguiente porción de código, en caso de querer una arquitectura más compleja, tan solo deberíamos cambiar estos valores.

```
def get_model(pretrained=False):
    return ResNet(2,[2,2,2,2])
```

Por último crearemos un Learner utilizando un Dataloader y el modelo que acabamos de crear y lo entrenaremos durante 20 épocas mediante fine_tune

Utilizaremos este código para crear nuestra red neuronal en base al DataSet de la tercera iteración del "Tipo 7" para ver si conseguimos mejores resultados que al aplicar transfer_learning.

7.4.3. Resultados

Una vez ejecutado nuestra red neuronal residual utilizando "Bag of Tricks" vemos los siguientes resultados:

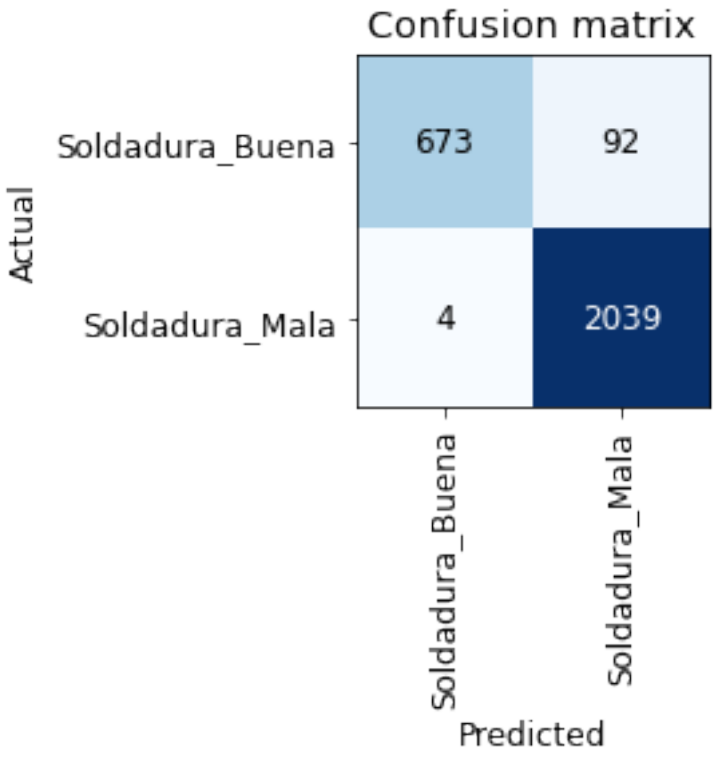

Figura 7.5: Matriz de confusión de "Tipo 7" utilizando "Bag of Trick" equivalente a ResNet18 entrenada 20 épocas

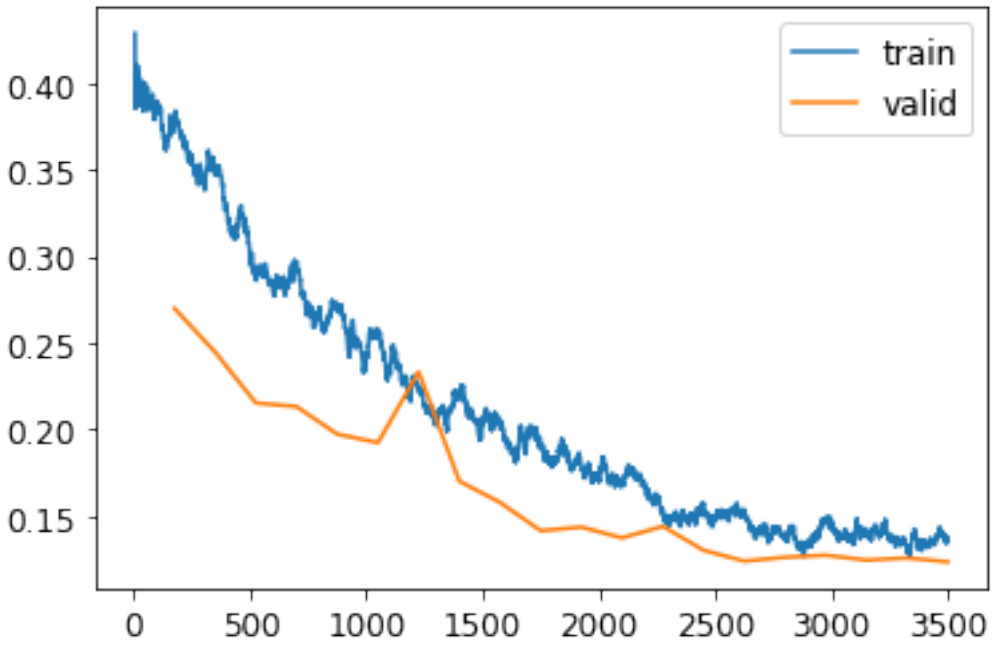

Figura 7.6: Gráfico de la evolución de perdida en entrenamiento y validación de "Tipo 7" utilizando "Bag of Trick" equivalente a ResNet18 entrenada 20 épocas

Al entrenar la red durante las 20 épocas se observa que la función de perdida en validación puede mejorar por lo que se ha decidido entrenar otra red durante 40 épocas en vez de 20.

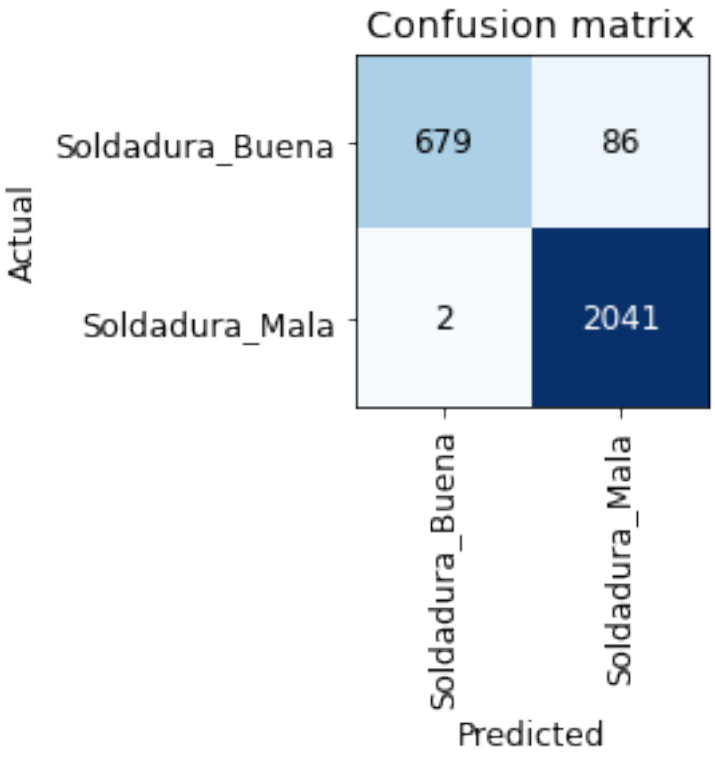

Figura 7.7: Matriz de confusión de "Tipo 7" utilizando "Bag of Trick" equivalente a ResNet18 entrenada 40 épocas

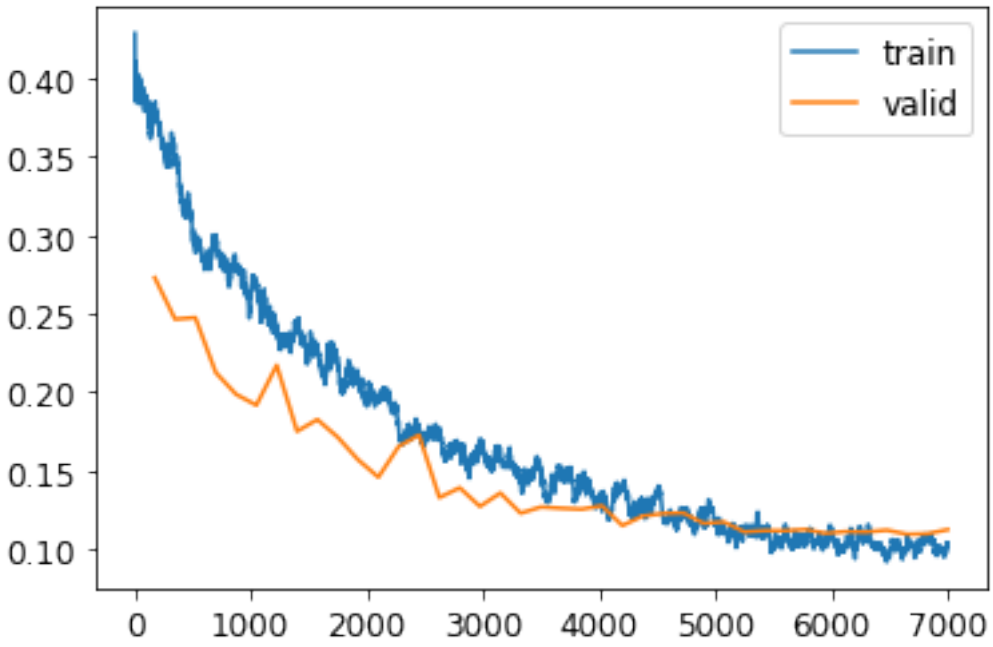

Figura 7.8: Gráfico de la evolución de perdida en entrenamiento y validación de "Tipo 7" utilizando "Bag of Trick" equivalente a ResNet18 entrenada 40 épocas

Como resultado del entrenamiento durante 40 épocas vemos como la red falla más imágenes que lo visto al hacer transfer_learning, donde tenía una tasa de error del **0.8 %** como se observa en la figura 6.6.3, mientras que el error utilizando esta estrategia es de un **3.1 %**

7.4.4. Conclusiones

Los resultados utilizando esta técnica de aprendizaje obtenidos son peores que los obtenidos para una ResNet18 utilizando transfer_learning. Destaca el aumento en el caso de errores de la categoría de Soldadura_Buena y por el contrario la reducción de casos de error en el conjunto de Soldadura_Mala, haciendo que clasifique mejor las soldaduras de la categoría Soldadura_Mala. La importancia de estos datos es que al tratarse de un proceso de calidad debe minimizarse el caso en el que la categoría Soldadura_Mala sea confundido con el conjunto Soldadura_Buena ya que si existen defectos, si se tratara de una pieza de seguridad podría tener graves consecuencias.

7.5. Redes Siamesas

7.5.1. Descripción del problema

En vista de que los dos experimentos pasados no han llevado a mejoras con respecto al transfer_learning nos hemos planteado realizar redes más complejas como lo son las redes siamesas que son capaces de identificar si dos imágenes pertenecen al mismo conjunto o no, permitiendo ver teniendo unas imágenes de prueba para cada conjunto si pertenecen al mismo conjunto. [8]

7.5.2. Descripción del experimento

Cuando hablamos de redes siamesas nos referimos a un tipo de redes que toman dos imágenes y deben determinar si pertenecen a la misma clase o no. Para realizar este proceso debemos preparar los datos para que sean procesados por este tipo de modelos y posteriormente entrenar el modelo con estos datos.

En este caso lo que haremos es que la imagen se muestre de la siguiente forma :

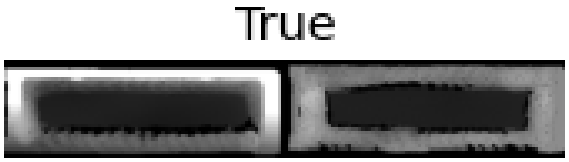

Figura 7.9: Ejemplo de salida de red siamesa

Para ello lo que haremos será crear una clase que contenga dos imágenes y un valor booleano que será True si las imágenes pertenecen a la misma clase o False si no ocurriese eso. Para mostrar la imagen hemos implementado el método show.

```
class SiameseImage(fastuple):
def show(self, ctx=None, **kwargs):
    img1,img2,same_breed = self
    if not isinstance(img1, Tensor):
        if img2.size != img1.size: img2 = img2.resize(img1.size)
        t1,t2 = tensor(img1),tensor(img2)
        t1,t2 = t1.permute(2,0,1),t2.permute(2,0,1)
    else: t1,t2 = img1,img2
    line = t1.new_zeros(t1.shape[0], t1.shape[1], 10)
    return show_image(torch.cat([t1,line,t2], dim=2),
                      title=same_breed, ctx=ctx)
```

Lo siguiente que haremos será construir la clase que permita que nuestros datos esten preparados para el modelo, en ella para cada imagen lo que hará es escoger una imagen de la misma clase (50 % de posibilidades) devolviendo una imagen de la clase SiameseImage con una etiqueta True o escoger una de otra distinta (50 % de posibilidades) devolviendo una imagen de la clase SiameseImage con una etiqueta False. Esto lo haremos dentro de la función _draw. Es importante tener en cuenta que en este tipo de modelos hay una diferencia entre los conjuntos de validación y de entrenamiento a los anteriormente vistos, en el conjunto de entrenamiento lo que haremos será escoger una imagen aleatoria cada vez que leamos la imagen, mientras que en el de validación escogeremos una imagen que no cambiará desde la inicialización. Así conseguimos más ejemplos durante el entrenamiento, pero siempre el mismo en validación. El código de la clase SiameseTransform se muestra a continuación:

```
class SiameseTransform(Transform):
    def __init__(self, files, label_func, splits):
```

```
        self.labels = files.map(parent_label).unique()
        self.lbl2files = {l: L(f for f in files if label_func(f) == l)
                          for l in self.labels}
        self.label_func = label_func
        self.valid = {f: self._draw(f) for f in files[splits[1]]}

    def encodes(self, f):
        f2,t = self.valid.get(f, self._draw(f))
        img1,img2 = PILImage.create(f),PILImage.create(f2)
        return SiameseImage(img1, img2, t)

    def _draw(self, f):
        same = random.random() < 0.5
        cls = self.label_func(f)
        if not same:
            cls = random.choice(L(l for l in self.labels if l != cls))
        return random.choice(self.lbl2files[cls]),same
```

Ahora nos encargaremos de crear nuestro conjunto de datos, lo primero que haremos será como siempre indicar la ruta donde se encuentran nuestras imágenes y en este caso obtener las imágenes de la ruta. Posteriormente utilizaremos nuestra transformación principal, SiameseTransform, después de dividir el conjunto de datos:

```
path = 'Tipo 7'
files=get_image_files(path)
splits = RandomSplitter()(files)
tfm = SiameseTransform(files, parent_label, splits)
```

Lo siguiente que haremos es crear una tupla para transformar nuestro conjunto, esto lo haremos ayudándonos de la clase TfmdLists de la siguiente forma:

```
tls = TfmdLists(files, tfm, splits=splits)
```

Por último utilizaremos la clase DataLoaders llamando al método dataloaders. Como este método no utiliza los item_tfms y batch_tfms como lo hacía DataBlock deberemos usar dos Hooks que se realizan después de los eventos como son after_item y after_batch en los que haremos las conversiones a tensores:

```
dls = tls.dataloaders(after_item=ToTensor,
    after_batch=IntToFloatTensor,num_workers=0)
```

Ahora es el momento de crear nuestro modelo para ello utilizaremos una arquitectur apreentrenada a la que le pasaremos nuestras dos imágenes, después concatenaremos los resultados y los mandaremos a una cabecera personalizada que nos devuelva la predicción. Por un lado tendremos nuestro modelo SiameseModel:

```
class SiameseModel(Module):
def __init__(self, encoder, head):
    self.encoder,self.head = encoder,head

def forward(self, x1, x2):
    ftrs = torch.cat([self.encoder(x1), self.encoder(x2)], dim=1)
    return self.head(ftrs)
```

Al que le pasaremos nuestro encoder, que en nuestro caso será un modelo de red ResNet18 preentrenado y nuestra cabecera. Posteriormente procederemos a construir el modelo:

```
encoder = create_body(resnet18, cut=-2)
head = create_head(512*4, 2, ps=0.5)
model = SiameseModel(encoder, head)
```

Antes de crear nuestro Learner, debemos definir dos cosas más. La primera será nuestra función de pérdida, utilizaremos la función de entropía cruzada pero como nuestro objetivo son booleanos debemos convertirlos a enteros o PyTorch lanzará un error:

```
def loss_func(out, targ):
    return nn.CrossEntropyLoss()(out, targ.long())
```

Lo segundo que debemos hacer para conseguir toda la ventaja que queremos es definir un separador personalizado. Un separador es una función que dice a la librería de fastai como debe dividir el modelo en grupos de parámetros, esto se usa para entrenar la cabeza del modelo cuando hacemos transfer_learning. En este caso utilizaremos el siguiente separador que tendrá dos grupos de parámetros uno para la cabeza y otro para el cuerpo:

```
def siamese_splitter(model):
    return [params(model.encoder), params(model.head)]
```

Después crearemos el Learner pasándole todo lo que hemos definido anteriormente y le entrenaremos durante 20 épocas:

```
learn = Learner(dls, model, loss_func=loss_func,
            splitter=siamese_splitter, metrics=error_rate)
learn.fine_tune(20)
```

Obteniendo los resultados mostrados en la siguiente sección.

7.5.3. Resultados

La ejecución de la red siamesa nos deja los siguientes resultados:

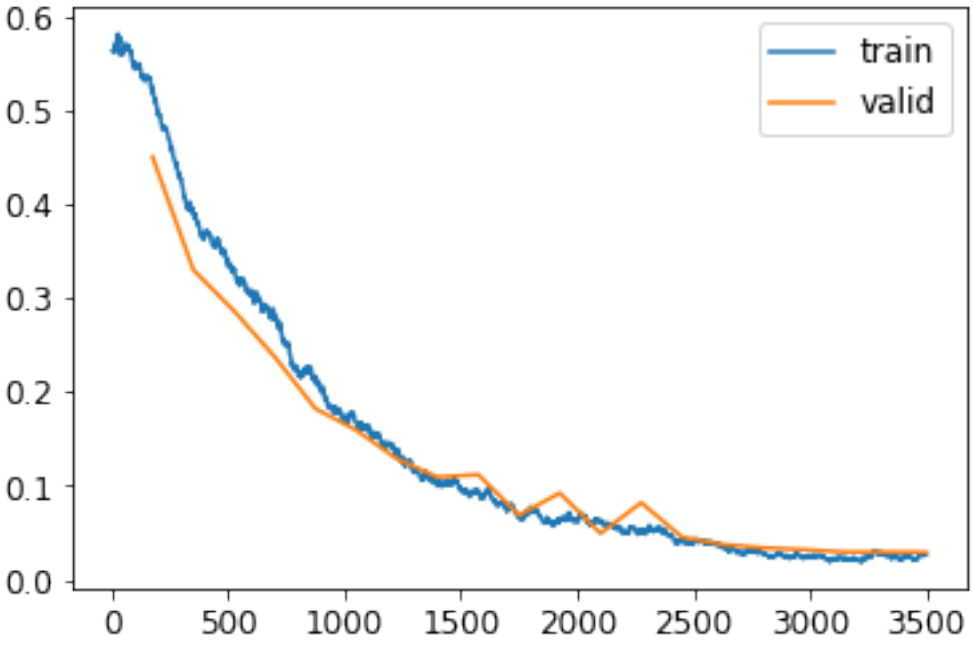

Figura 7.10: Gráfico de la evolución de perdida en entrenamiento y validación de "Tipo 7" utilizando una red siamesa entrenada 20 épocas

En esta ejecución se consigue una tasa de error similar a la que pudimos ver mediante transfer_learning en el capítulo anterior, de un **0.82 %** no obstante como vemos en la evolución de la función de perdida la red parece que sigue aprendiendo por lo que se ha decidido dejar la ejecución un total de 30 épocas.

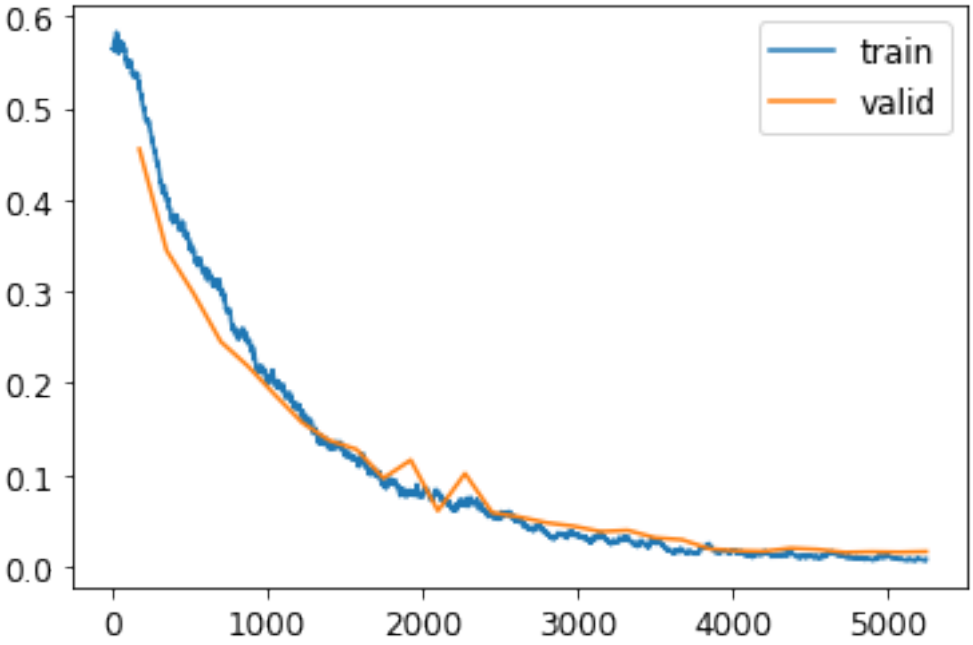

Figura 7.11: Gráfico de la evolución de perdida en entrenamiento y validación de "Tipo 7" utilizando una red siamesa entrenada 30 épocas

Al realizar esta ejecución durante 10 épocas más observamos que la red sigue aprendiendo y obtiene los mejores resultados vistos hasta el momento obteniendo una tasa de error del **0.4 %**,

aunque si bien es cierto la función de pérdida en validación se estabiliza indicándonos que más entrenamiento no hará que la red mejore.

7.5.4. Conclusiones

La realización de redes siamesas mejora el resultado obtenido en cualquier otra ejecución haciéndola la red que mejor funciona hasta el momento, indicando que es una buena línea de investigación en la que poder seguir para la mejora de resultados, pudiéndose así probar a utilizar encoders de más complejidad como es el caso de cuerpos de arquitecturas más profundas como ResNet34 o ResNet50 para intentar mejorar los resultados.

7.6. Detección de anomalías

7.6.1. Descripción del problema

Otro campo de exploración posible para la mejora de resultados es la detección de anomalías mediante redes neuronales. En esta sección se pretende dar una descripción de ellas y como se intentó abordar el problema mediante esta aproximación. Por ello es importante primero entender en que consiste este concepto. [**ImplementingAutoencoders**] [**Anomaly**] [9]

Denominamos anomalías, a los valores atípicos, eventos raros o desviados, puntos de datos o patrones en los datos que no se ajustan a lo que entendemos como un comportamiento normal. Por tanto diremos que la detección de anomalías es la tarea de encontrar estos valores atípicos que se salen de lo que veríamos normal en un conjunto de datos.

Para realizar esta tarea se pueden seguir muchas aproximaciones como el aprendizaje supervisado, el aprendizaje no supervisado o incluso el aprendizaje semi-supervisado. En nuestro caso debido al contenido del documento nos centraremos en una aproximación basada en Deep Learning utilizando Autoencoders.

Un **"autoencoder"** es una red neuronal diseñada para aprender a un nivel bajo de representación, partiendo de un dato de entrada. Consiste en dos componentes:

- **Encoder**: Elemento que mapea la entrada a un nivel de representación bajo.
- **Decoder**: Elemento que aprende a mapear la representación y la transforma en el elemento original.

Una vez obtenida la representación final la idea es entrenar a la red minimizando el error en la reconstrucción de la imagen, mediante la diferencia entre la imagen introducida inicialmente y la imagen reconstruida producida por el decoder, esto se realiza utilizando el error medio cuadrático.

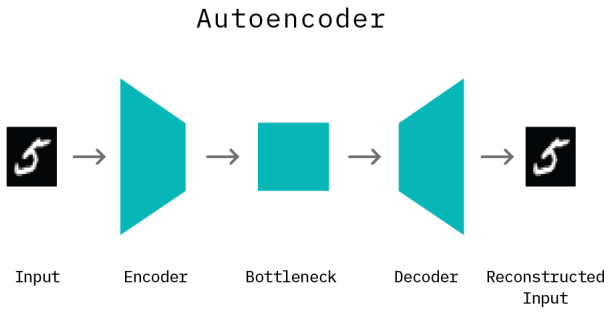

Figura 7.12: Componentes de un autoencoder [10]

Es importante tener en cuenta que un autoencoder funcionara "bien" en la reconstrucción de datos con los que haya sido entrenado, es decir, normalmente no será exitoso en la reconstrución de datos diferentes a los que vio en el proceso de entrenamiento.

7.6.2. Descripción del experimento

Una vez detallado el funcionamiento de un autoencoder llega el momento de implementar uno utilizando fastai, para ello lo primero que haremos será crear un conjunto de test, en este caso este conjunto se compondrá de un 10 % de las imágenes de la categoría Soldadura_Buena y un 10 % de las imágenes de la categoría Soldadura_Mala, después le indicaremos la ruta donde queremos que coja las imágenes, en este caso la ruta es la de las imágenes de la categoría Soldadura_Buena ya que lo que queremos es que no reconstruya bien las imágenes que no conoce, en nuestro caso las imágenes pertenecientes a la categoría Soldadura_Mala.

Posteriormente crearemos el DataBlock que será similar a los ya vistos pero con pequeñas modificaciones, lo que queremos es que nuestra red de como predicción otra imagen por ello el parámetro blocks será el siguiente blocks=(ImageBlock, ImageBlock), a diferencia del que estamos acostumbrados blocks=(ImageBlock, CategoryBlock), también cambiaremos la función de etiquetado y utilizaremos: "lambda o: o", que hará que la imagen que se use para entrenar sea etiquetada en el conjunto de entrenamiento como ella misma en vez de que pertenezca a una categoría en concreto:

```
path = "Tipo 7/Soldadura_Buena"

soldaduras = DataBlock(
    blocks=(ImageBlock, ImageBlock),
    get_items=get_image_files,
    splitter=RandomSplitter(valid_pct=0.2, seed=42),
    get_y= lambda o: o
)
```

Debido a la arquitectura escogida al realizar el dataloader tendremos que establecer que las

imágenes que vaya a procesar nuestra red tengan un tamaño de 128x128 esto lo haremos utilizando el campo item_tfms del DataBlock donde utilizaremos la clase Resize:

```
soldaduras=soldaduras.new(item_tfms=Resize(128, ResizeMethod.Pad, pad_mode='zeros'))
dls = soldaduras.dataloaders(path,num_workers=0)
```

Lo siguiente que haremos sera crear el cuerpo del modelo que vamos a utilizar como encoder en nuestro caso utilizaremos la arquitectura ResNet18, esto lo haremos utilizando la función create_body:

```
arch = create_body(resnet18, n_in=3).cuda()
```

A continuación crearemos el decodificador de nuestra red, para ello realizamos una clase que será la base de nuestro decodificador "UpsampleBlock" que contendrá las capas que utilizaremos dentro del codificador y por último para terminar el modelo crearemos el autoencoder que tendrá como entrada el encoder que se desa utilizar y el rango que queremos que tenga de salida la función sigmoide del decoder.

```
class UpsampleBlock(Module):
def __init__(self, up_in_c:int, final_div:bool=True, blur:bool=False, leaky:float=None
    self.shuf = PixelShuffle_ICNR(up_in_c, up_in_c//2, blur=blur, **kwargs)
    ni = up_in_c//2
    nf = ni if final_div else ni//2
    self.conv1 = ConvLayer(ni, nf, **kwargs)
    self.conv2 = ConvLayer(nf, nf, **kwargs)
    self.relu = nn.ReLU()

def forward(self, up_in:Tensor) -> Tensor:
    up_out = self.shuf(up_in)
    cat_x = self.relu(up_out)
    return self.conv2(self.conv1(cat_x))

def decoder_resnet(y_range, n_out=3):
return nn.Sequential(UpsampleBlock(512),
                     UpsampleBlock(256),
                     UpsampleBlock(128),
                     UpsampleBlock(64),
                     UpsampleBlock(32),
                     nn.Conv2d(16, n_out, 1),
                     SigmoidRange(*y_range)
                    )

def autoencoder(encoder, y_range): return nn.Sequential(encoder, decoder_resnet(y_rang

y_range = (-0.5,0.5)
ac_resnet = autoencoder(arch, y_range).cuda()
```

Ahora crearemos un Learner utilizando el dataloader que hemos creado anteriormente, el modelo que acabamos de definir y utilizaremos como función de pérdida el error cuadrático medio, entrenaremos la red durante 20 épocas:

```
learn = Learner(dls, ac_resnet, loss_func=MSELossFlat())
```

Una vez realizado el entrenamiento es el momento de probar la red para ello cargaremos todas las imágenes que están en el conjunto de test, haremos que el modelo haga una predicción sobre ellas, calcularemos el MSE entre las imágenes y estableceremos un umbral en el que si el error es mayor la imagen sea clasificada como mala y si es menor o igual sea clasificada como buena.

7.6.3. Resultados

Una vez entrenada la red los resultados obtenidos han sido los siguientes:

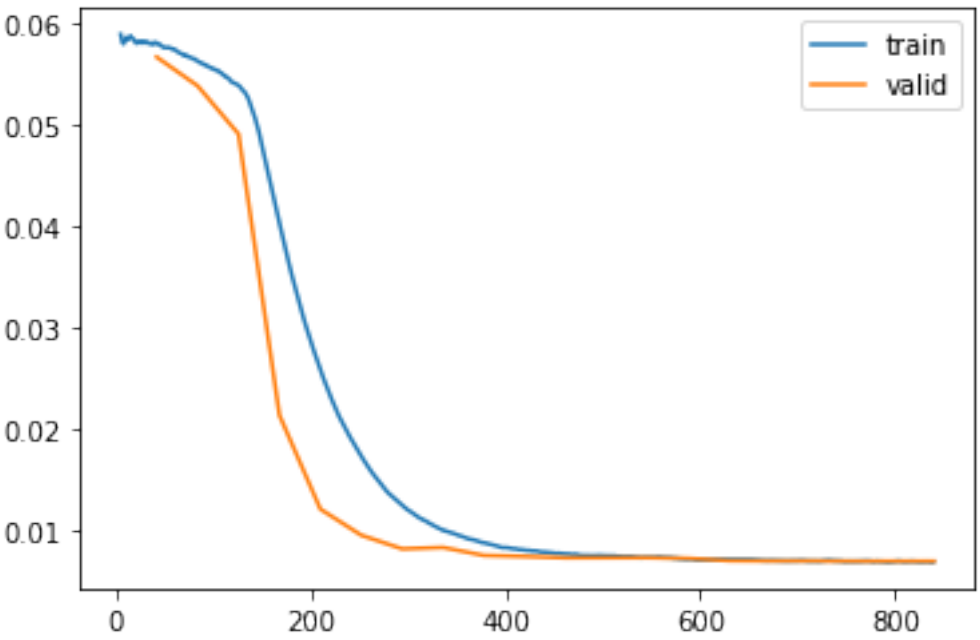

Figura 7.13: Gráfico de la evolución de perdida en entrenamiento y validación de "Tipo 7" utilizando una red para la detección de anomalías entrenada 20 épocas

Como vemos el aprendizaje se realiza de forma correcta ya que no existe overfitting, también podemos ver los resultados que produce la red en forma de imagen, donde vemos cual es la entrada, a qué imagen pretende llegar la red y por último cual es la imagen que ha generado:

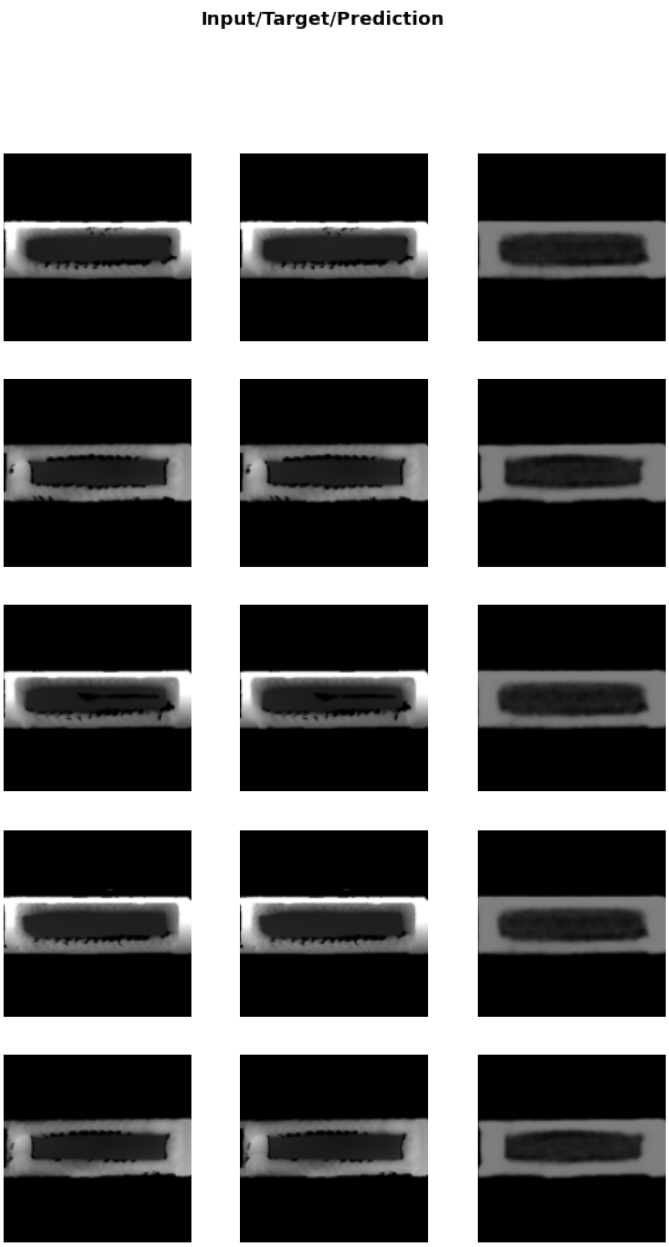

Figura 7.14: Salida de la red de detección de anomalías

Ahora usaremos el conjunto de test para obtener resultados. El valor usado como umbral ha sido la media de los resultados de calcular el MSE en las imágenes del conjunto de test de Soldadura_Buena, sabiendo esto, los resultados obtenidos para cada conjunto han sido los siguientes:

- Porcentaje de error del conjunto de Test de Soldadura_Buena:47.51 %
- Porcentaje de error del conjunto de Test de Soldadura_Mala: 49.02 %
- Porcentaje de error del conjunto de Test: 48.61 %

Estos resultados son mucho peores a los obtenidos en otras secciones del documento lo que nos hace ver que se trata de una línea de investigación compleja en la que quedaría por realizar un gran trabajo.

7.6.4. Conclusiones

El campo de las redes neuronales para la detección de anomalías es un campo interesante de exploración pero de gran complejidad saliéndose de los objetivos de este trabajo pudiendo ser de interés para futuras líneas de investigación. Por lo que pese haber investigado su funcionamiento y haber puesto en marcha una red neuronal mediante esta tecnología, para obtener unos resultados correctos quedaría mucho camino que recorrer. Una de las problemas más importante es establecer una métrica con la que poner un umbral para clasificar imágenes ya que esta cobra gran protagonismo. Como se ha descrito los resultados obtenidos no son buenos con lo que concluimos en que no hemos logrado buenos resultados utilizando este tipo de técnicas.

7.7. Conjunto de redes neuronales: comité de redes

7.7.1. Descripción del problema

En ocasiones obtener un modelo individual que obtenga buenos resultados para un conjunto de datos puede ser frustrante ya que no obtenemos los resultados que queremos. Un enfoque que permite la mejora de los resultados de un modelo individual es la combinación de varios modelos individuales, a esto se le denomina conjunto de redes neuronales (Ensemble Neural Network) [11] [12].

En esta sección exploraremos este método de aprendizaje ensamblando las redes mediante el procedimiento denominado "comité de redes" donde entrenaremos diferentes arquitecturas de red sobre el mismo conjunto de datos, posteriormente haremos que ellas nos indiquen a que categoría pertenece la imagen que queremos someter a su juicio.

7.7.2. Descripción del experimento

Para la realización de este experimento se ha creado un conjunto de test, en este caso este conjunto se compondrá de un 10 % de las imágenes de la categoría Soldadura_Buena y un 10 % de las imágenes de la categoría Soldadura_Mala. Posteriormente se ha procedido al entrenamiento de cuatro redes neuronales utilizando transfer_learning de las arquitecturas ResNet18, ResNet34, ResNet50 y ResNet101 , todas ellas entrenadas un total de 20 épocas.

Una vez entrenadas las cuatro redes se han realizado todas las combinaciones posibles entre ellas de modo que se haga la media de la salida del tensor que nos india las probabilidades de que una imagen pertenezca una categoría y se vuelve a determinar la categoría a la que la imagen pertenece.

También se ha explorado la posibilidad de realizar un sistema de votos en la que en vez de realizar la media entre los resultados del tensor de probabilidades obtenidos por las redes, se proceda de forma que si la mayoría de las redes indican que la imagen pertenece a una categoría se concluya que la imagen pertenece a esa categoría, en el caso de empate se calcula la media.

7.7.3. Resultados

Antes de mostrar los resultados obtenidos por el "comité de redes" es importante mostrar como funcionan las arquitecturas que pertenecen al conjunto frente a nuestro conjunto de test:

	Tasa de error en Test
ResNet18	0.9979%
ResNet34	0.4989%
ResNet50	0.4989%
ResNet101	0.6415%

Figura 7.15: Resultados sobre el conjunto de test de las nuevas redes entrenadas usando transfer_learning, entrenadas durante 20 épocas

Como vemos todas ellas presentan resultados bastante aceptables frente al conjunto de imágenes, pero en todas hay valores mejorables y es lo que intentaremos hacer utilizando esta nueva técnica. A continuación se muestran los resultados obtenidos para cada una de las combinaciones:

	ResNet18	ResNet34	ResNet50	ResNet101
ResNet18		0.4989%	0.7127%	0.8553%
ResNet34	0.4989%		0.4989%	0.4276%
ResNet50	0.4989%	0.4989%		0.5702%
ResNet101	0.8553%	0.4276%	0.5702%	
ResNet18+ResNet34			0.6414%	0.5702%
ResNet18+ResNet50		0.6414%		0.6414%
ResNet18+ResNet101		0.5702%	0.6414%	
ResNet34+ResNet50	0.6414%			0.2851%
ResNet34+ResNet101	0.5702%		0.2851%	
ResNet50+ResNet101	0.6414%	0.2851%		
ResNet18+ResNet34+ResNet50				0.5702%

Figura 7.16: Resultados sobre el conjunto de test de realizar todas las combinaciones del conjunto de redes

Como vemos en la tabla que acabamos de mostrar todos los resultados de combinar las redes mejorar los resultados obtenidos por la ResNet18, no obstante no todas las combinaciones obtienen mejores resultados a los obtenidos por las redes de forma individual ya que al hacer la media entre diferentes predicciones, si una red esta confundida y clasifica la imagen mal con cierta certeza podrá hacer que la o las redes con las que esta en conjunto no tengan el suficiente valor como para corregirla. No obstante observamos que la combinación de ResNet34+ResNet50+ResNet101 obtiene el mejor resultado de todas las combinaciones reduciendo a casi la mitad el mejor valor obtenido por cualquiera de las redes de forma individual.

Por último se muestra la tabla de resultados de realizar el segundo experimento descrito en la sección anterior:

	ResNet18	ResNet34	ResNet50	ResNet101
ResNet18+ResNet34			0.6414%	0.5702%
ResNet18+ResNet50		0.6414%		0.7127%
ResNet18+ResNet101		0.5702%	0.7127%	
ResNet34+ResNet50	0.6414%			0.3563%
ResNet34+ResNet101	0.5702%		0.3563%	
ResNet50+ResNet101	0.7127%	0.3563%		
ResNet18+ResNet34+ResNet50				0.5702

Figura 7.17: Resultados sobre el conjunto de test de realizar todas las combinaciones del conjunto de redes utilizando el sistema de votos

En el observamos que se obtienen peores resultados a lo visto en la tabla anterior, por lo que creemos que será peor método que lo mostrado anteriormente.

7.7.4. Conclusiones

El "comité de redes" se trata de una propuesta muy interesante que como hemos podido observar puede llegar a obtener una tasa de errores inferior a un modelo de red individual, tratándose de una buena opción si queremos mejorar la exactitud en un conjunto de datos. Además se trata de una propuesta que permite su mejora ya que en vez de usar métricas como la media para que todas las redes tengan la misma importancia podemos utilizar medias ponderadas haciendo que, redes con mejores resultados de forma individual adquieran más importancia que otras, pudiendo así obtener mejores resultados.

Capítulo 8

Conclusiones

8.1. Consecución de objetivos

- Adquisición de conocimientos sobre Deep Learning en concreto sobre redes neuronales.
- Aprendizaje de distintos frameworks para el desarrollo de redes neuronales como lo son Keras y Fastai.
- Realización de clasificadores para la resolución de diferentes tipos de problemas.
- Resolución de problemas utilizando Fastai acerca de visión artificial, en concreto en el ámbito industrial.
- Aprendizaje de diferentes técnicas para mejorar los resultados mediante redes neuronales.

8.2. Futuras mejoras

En este Trabajo de Fin de Grado se han utilizado clasificadores para la clasificación de un conjunto de imágenes de soldaduras, cuyos resultados obtenidos son mejorables. Una de las primeras mejoras en vista de los experimentos detallados es modificar el conjunto de datos de forma que se encuentre en un estado óptimo de clasificación, para ello se podría aumentar el número de imágenes así como la limpieza de datos que se encuentren en el conjunto incorrecto. También explorar diferentes técnicas a las citadas en el documento, como ampliar la investigación en el campo de la detección de anomalías puede ayudar a mejorar estos resultados. Las redes siamesas han conseguido obtener los mejores resultados en validación teniendo una precisión de alrededor de un 99.5 % por lo que serían otro punto de partida interesante para obtener mejores precisiones. También seguir profundizando en los modelos de redes creadas desde cero, creando nuevas capas y modificando las ya existentes se pueden lograr grandes resultados.

Bibliografía

[1] J. Howard y S. Gugger. *Deep Learning for Coders with Fastai and Pytorch: AI Applications Without a PhD*. O'Reilly Media, Incorporated, 2020. ISBN: 9781492045526. URL: https://books.google.no/books?id=xd6LxgEACAAJ.

[2] Michael A. Nielsen. *Neural Networks and Deep Learning*. misc. 2018. URL: http://neuralnetworksandcom/.

[3] A. Koul, S. Ganju y M. Kasam. *Practical Deep Learning for Cloud, Mobile and Edge: Real-World AI and Computer Vision Projects Using Python, Keras and TensorFlow*. O'Reilly Media, Incorporated, 2019. ISBN: 9781492034865. URL: https://www.oreilly.com/library/view/practical-deep-learning/9781492034858/.

[4] *EVGA GeForce RTX 3070 XC3 ULTRA GAMING 8GB GDDR6*. https://www.pccomponentes.com/evga-geforce-rtx-3070-xc3-ultra-gaming-8gb-gddr6.

[5] *RobotWorx - How Much Do Industrial Robots Cost?* URL: https://www.robots.com/faq/how-much-do-industrial-robots-cost.

[6] Agencia Estatal Boletín Oficial del Estado y España. *Boletín Oficial del Estado, Número 57, Martes 6 de marzo de 2018*. BOE, 2018. URL: https://www.boe.es/boe/dias/2018/03/06/pdfs/BOE-A-2018-3156.pdf.

[7] Tong He y col. *Bag of Tricks for Image Classification with Convolutional Neural Networks*. 2018. arXiv: 1812.01187.

[8] URL: https://docs.fast.ai/tutorial.siamese.html.

[9] *Autoencoder with fastai*. URL: https://colab.research.google.com/drive/1t9dn6qIdKc6rdF-A02KMdJ8UVGYPFh4v.

[10] *Deep Learning for Anomaly Detection*. Feb. de 2020. URL: https://ff12.fastforwardlabs.com/.

[11] Jason Brownlee. *Ensemble Learning Methods for Deep Learning Neural Networks*. https://machinelearningmastery.com/ensemble-methods-for-deep-learning-neural-networks/.

[12] Ludmila I. Kuncheva. *Combining Pattern Classifiers: Methods and Algorithms*. USA: Wiley-Interscience, 2004. ISBN: 0471210781.