Universidad de Valladolid

Escuela de Ingeniería Informática

Trabajo de Fin de Grado

Grado en Ingeniería Informática

Mención en Ingeniería de Software

Explotación de Frameworks Deep Learning Aplicados a la Astronomía

Autor:

D. Darío de la Torre Guinaldo

Tutores:

D. Benjamín Sahelices Fernández

D. Fernando Buitrago Alonso

Agradecimientos

A todos los que me han apoyado en la realización de este TFG, en especial a mis tutores Benjamín, por guiarme e introducirme en el mundo del deep learning y de las redes neuronales, y a Fernando, por enseñarme el apasionante mundo de la astronomía.

Resumen

Este trabajo tiene como objetivos principales aprender sobre machine learning y utilizar frameworks de deep learning de alto nivel para resolver problemas relacionados con el campo de la astronomía mediante la aplicación de técnicas de redes neuronales.

Primero se hará una introducción con los conceptos subyacentes de las redes neuronales. Después se hará un estudio comparativo mediante ejemplos de 4 de los principales frameworks de redes neuronales actuales: TensorFlow, Keras, PyTorch y Fastai. Por último, se utilizará Fastai para resolver problemas de clasificación, regresión y segmentación de imágenes astronómicas de galaxias del espacio profundo.

Índice general

1. Introducción **7**

2. Análisis del proyecto **8**

2.1. Metodología 8

2.2. Planificación 8

2.3. Análisis de riesgos 11

3. Redes neuronales **14**

3.1. Inteligencia artificial 14

3.1.1. ¿Qué es la inteligencia artificial? 14

3.1.2. Breve historia de la inteligencia artificial 14

3.1.3. Áreas de aplicación del deep learning 15

3.2. Fundamentos básicos de las redes neuronales 16

3.2.1. Perceptrón 16

3.2.2. Neurona sigmoide 17

3.2.3. La arquitectura de las redes neuronales 17

3.2.4. Descenso del gradiente 18

3.2.5. EL código para implementar una red que reconozca dígitos manuscritos . . 19

3.3. Algoritmo de backpropagation 20

3.3.1. Una aproximación rápida basada en matrices para computar la salida de una red neuronal 20

3.3.2. Dos suposiciones necesarias sobre la función de coste 21

3.3.3. El producto de Hadamard 21

3.3.4. Las cuatro ecuaciones fundamentales detrás de la backpropagation 21

3.3.5. El algoritmo de backpropagation 22

3.4. Mejorar el aprendizaje de las redes neuronales 22

3.4.1. La función de coste de entropía cruzada 23

3.4.2. ¿De dónde viene la entropía cruzada? 23

3.4.3. Softmax 23

3.4.4. Sobreajuste 24

3.4.5. Regularización 24

3.4.6. ¿Por qué la regularización ayuda a reducir el sobreajuste? 24

3.4.7. Otras técnicas de regularización 25

3.4.8. Inicialización de pesos 27

3.4.9. Segunda visita al código de reconocimiento de dígitos manuscritos 27

3.4.10. ¿Cómo elegir los hiper-parámetros para nuestra red neuronal? 27
3.4.11. Otras técnicas 28
3.5. Pruebas visuales de que las redes neuronales pueden computar cualquier función . 29
3.5.1. Dos consideraciones 29
3.5.2. Universalidad con una entrada y una salida 29
3.5.3. Varias variables de entrada 30
3.5.4. Conclusión 32
3.6. ¿Por qué son difíciles de entrenar las redes neuronales profundas? 33
3.6.1. El problema del gradiente evanescente 33
3.6.2. ¿Qué está causando el problema del gradiente evanescente? 34
3.6.3. Otros obstáculos para el aprendizaje profundo 34
3.7. Redes convolucionales 34
3.7.1. ¿Qué es una convolución? 35
3.7.2. Introduciendo redes convolucionales 35
3.7.3. Tercera visita al código de reconocimiento de dígitos manuscritos 37
3.8. Deep Learning 38
3.8.1. Ingredientes de una red neuronal 38
3.8.2. IA responsable 39
3.8.3. Progreso reciente en el reconocimiento de imágenes 40
3.8.4. Otras aproximaciones a las redes neuronales 41
3.8.5. Redes de creencia profunda, modelos generativos y máquinas de Boltzmann 41
3.8.6. El futuro de las redes neuronales 42
3.8.7. Ejemplos de aplicaciones web que usan redes neuronales 42

4. Contexto tecnológico **43**
4.1. Herramientas básicas 43
4.1.1. Anaconda 43
4.1.2. Jupyter Notebook 44
4.1.3. Hardware 45
4.1.4. Frameworks para Deep Learning en Python 45
4.1.5. Bibliotecas de Python 46
4.2. Keras & TensorFlow 46
4.2.1. Clasificación de imágenes con Keras 47
4.2.2. Transfer learning con Keras 51
4.2.3. Construyendo un motor de búsqueda de imágenes inverso 56
4.3. FastAI & PyTorch 61
4.3.1. Introducción a fastai 62
4.3.2. De modelo a producto 63
4.3.3. Clasificación de imágenes con fastai 66
4.3.4. Otros problemas de visión artificial 70
4.3.5. Técnicas avanzadas para entrenar un modelo 74
4.3.6. Mixup 75
4.3.7. Redes neuronales convolucionales 76
4.4. Conclusión: Keras y TensorFlow vs Fastai y PyTorch 77

5. Astroinformática 78
5.1. Introducción 78
5.2. Herramientas 78
5.3. Trabajando con herramientas de astroinformática 79
5.3.1. Cómo recortar una imagen de una estrella o una galaxia a partir de una imagen astronómica con SExtractor y DS9 79
5.3.2. Cómo obtener las coordenadas de una estrella o una galaxia con SExtractor, Topcat y Aladin 84
5.4. Estado del arte del deep learning en astronomía: Morpheus 86

6. Clasificación de galaxias 87
6.1. Introducción 87
6.2. Artículo de Marc Huertas-Company 2015 87
6.3. Conjunto de datos 88
6.3.1. Conjunto de datos inicial 88
6.3.2. Conjunto de datos final 88
6.4. Modelo: ResNet 91
6.5. Clasificación morfológica 91
6.5.1. Descripción 91
6.5.2. Dataset simple 92
6.5.3. Dataset doble 96
6.5.4. Dataset triple 98
6.5.5. Conclusiones 100
6.6. Regresión de valores morfológicos 101
6.6.1. Descripción 101
6.6.2. Dataset simple 102
6.6.3. Dataset doble 106
6.6.4. Dataset triple 108
6.6.5. Conclusión 110

7. Detección de bordes de galaxias con segmentación 111
7.1. Introducción 111
7.2. Conjunto de datos 111
7.2.1. Conjunto de datos inicial 111
7.2.2. Conjunto de datos final 111
7.3. Modelo: U-Net 112
7.4. Segmentación mediante U-Net 113
7.4.1. Descripción 113
7.4.2. Sin transformaciones 115
7.4.3. Con transformaciones 119
7.4.4. Conclusiones 123
7.5. Segmentación mediante aprendizaje de redes federadas 123
7.5.1. Definición 123

7.5.2. Exportar y combinar modelos . . . 124
7.5.3. Resultados . . . 125
7.5.4. Conclusiones . . . 129
7.6. Segmentación mediante un modelo personalizado . . . 130
7.6.1. Creación del modelo . . . 130
7.6.2. Resultados . . . 132
7.6.3. Conclusiones . . . 134

8. Conclusiones finales **135**
8.0.1. Consecución de objetivos . . . 135
8.0.2. Futuras mejoras . . . 135

A. Jupyter Notebooks usados **137**
A.1. Tutorial de segmentación . . . 137
A.2. Ejemplo con .fits. Sección 4.3 . . . 138
A.3. Creación de imágenes RGB. Sección 5.3 . . . 139
A.3.1. Create_images_RGB_lupton . . . 139
A.4. Clasificación morfológica de galaxias. Sección 5.4 . . . 144
A.4.1. Clasificación simple ResNet18 No Augmentation . . . 144
A.4.2. Clasificación simple ResNet18 Augmentation . . . 145
A.4.3. AumentarNumeroImagenes . . . 146
A.5. Regresión morfológica de galaxias. Sección 5.5 . . . 146
A.5.1. CrearCsv . . . 146
A.5.2. Regresión Simple ResNet18 sin data augmentation . . . 148
A.6. Detección de bordes de galaxias. Sección 6.2 . . . 149
A.6.1. Creating_rgb_images_lupton . . . 149
A.7. Segmentación mediante aprendizaje de redes federadas. Sección 6.5 . . . 152

Capítulo 1

Introducción

Hoy en día, la inteligencia artificial está en auge, debido principalmente a la creación de nuevos frameworks cada vez de más alto nivel y a la disponibilidad de hardware cada vez más potente para ejecutarlos, como las GPU. Las **redes neuronales** es una de las técnicas de inteligencia artificial más usadas actualmente, y que ha tenido un gran desarrollo en la última década. Las posibles aplicaciones de las redes neuronales son prácticamente infinitas, ya que permiten emular el comportamiento humano en multitud de tareas, y con frecuencia, incluso desempeñarlas mejor que nosotros.

Uno de los posibles campos de aplicación es la **astronomía**. Los telescopios espaciales como el Hubble o el Euclid fotografían miles de objetos del espacio profundo, lo que conlleva un trabajo bastante pesado por parte de los expertos para clasificar y etiquetar estos objetos astronómicos. Una red neuronal bien entrenada para este tipo de tareas puede ahorrar mucho tiempo a los astrónomos y permitir que se centran en otras labores.

Por lo tanto, los **objetivos** de la realización de este trabajo se pueden concretar en:

- Estudiar y entender los fundamentos básicos del machine learning y del deep learning. En concreto, aprender los conceptos subyacentes de las redes neuronales.
- Conocer la forma en que se trabaja con distintos frameworks de deep learning y conocer las diferencias básicas entre ellos.
- Aplicar conocimientos de deep learning para para resolver problemas reales mediante frameworks de deep learning de alto nivel de abstracción.
- Aprender conocimientos sobre astronomía y sobre herramientas informáticas utilizadas en astronomía. En concreto, sobre galaxias u otros objetos del espacio profundo.
- Aplicar todos los conocimientos adquiridos en problemas de deep learning relacionados con las astronomía.

Capítulo 2

Análisis del proyecto

2.1. Metodología

La realización de este TFG se ha hecho en dos partes bien diferenciadas que se solapan. Por una parte el estudio de las redes neuronales y del deep learning mediante tres libros que se detallan en el apartado de planificación y por otra parte mediante reuniones semanales en las que se repasa lo hecho durante la semana y se decide lo que se va hacer durante la siguiente. La memoria del TFG se ha ido redactando conforme se ha ido realizando el trabajo.

Por lo tanto, se ha seguido una **metodología ágil de tipo SCRUM con sprints de una semana** cada uno, salvo algunas excepciones.

2.2. Planificación

La planificación para la realización de este trabajo ha sido la siguiente:

1. **Inicio**: 02/11/2020
2. **Lectura y estudio del libro *Neural Networks and Deep Learning* de Michael Nielsen[1]**:02/11/2020 a 22/11/2020
3. **Lectura y estudio de los 4 primeros capítulos del libro *Practical Deep Learning for Cloud, Mobile & Edge* de Anirudh Koul, Siddha Ganju y Meher Kasam[2]**: 23/11/2020 a 13/12/2020
4. **Lectura y estudio del libro *Deep Learning for Coders with Fastai and Pytorch* de Jeremy Howard y Sylvain Gugger[3]**: 14/12/2020 a 31/01/2021
5. **Reuniones semanales realizadas con los tutores del TFG**: 01/12/2020 a 30/05/2021, exceptuando las semanas de vacaciones de Navidad y de Semana Santa.

Lo realizado en cada sprint se detalla a continuación. Cada sprint corresponde a una semana, siendo el primer sprint la primera semana de noviembre de 2020.

Sprint	Reunión	Tema de la reunión	Trabajo personal	Fecha
1	Reunión inicial con el tutor	Explicación del TFG	Lectura y estudio de capítulos 1 y 2 del libro de Nielsen	2/11 a 8/11
2	-	-	Lectura y estudio de capítulos 3 y 4 del libro de Nielsen	9/11 a 15/11
3	-	-	Lectura y estudio de capítulos 5 y 6 del libro de Nielsen	16/11 a 22/11
4	Reunión con el tutor para confirmar la realización del TFG	Confirmación de realización del TFG.	Lectura y estudio de capítulos 1 y 2 del libro de Keras y TensorFlow	23/11 a 29/11
5	Primera reunión semanal con los tutores	Introducción a la astronomía	Lectura y estudio de capítulos 3 y 4 del libro de Keras y TensorFlow	30/11 a 6/12
6	Segunda reunión semanal con los tutores	Utilización de herramientas informáticas en la astronomía: DS9	Repasar lo explicado en la reunión y practicar con la herramienta explicada	7/12 a 13/12
7	Tercera reunión semanal con los tutores	Utilización de herramientas informáticas en la astronomía: SExctractor y Aladin	Repasar lo explicado en la reunión y practicar con las herramientas explicadas	14/12 a 20/12
8	Cuarta reunión semanal con los tutores	Introducción a Jupyter Notebook	Repasar lo explicado en la reunión.	21/12 a 27/12
9	-	-	Lectura y estudio de capítulos 1, 2 y 3del libro de Fastai y PyTorch	28/12 a 3/1
10	-	-	Lectura y estudio de capítulos 4, 5 y 6 del libro de Fastai y Pytorch	4/1 a 10/1
11	Quinta reunión semanal con los tutores	Planificación de la preparación de los datasets de imágenes de galaxias	Lectura y estudio de capítulos 7 y 13 del libro de Fastai y Pytorch	11/1 a 17/1
12	Sexta reunión semanal con los tutores	Introducción a PyTorch	Repasar lo explicado en la reunión	18/1 a 24/1
13	-	-	Lectura y estudio de capítulos 14 y 15 del libro de Fastai y PyTorch	25/1 a 31/1
14	Séptima reunión con los tutores	Dataset para el problema de clasificación galaxias	Generar el dataset para el problema de clasificación de galaxias	1/2 a 7/2

15	Octava reunión con los tutores	Entrenamiento de redes para el problema de clasificación de galaxias. Introducción de csv de log del entrenamiento	Entrenar redes para el problema de clasificación de galaxias con las modificaciones habladas durante la reunión	8/2 a 14/2
16	Novena reunión con los tutores	Comparación y análisis de resultados. Introducción de transformaciones	Entrenar redes para el problema de clasificación de galaxias con las modificaciones habladas durante la reunión	15/2 a 21/2
17	Décima reunión con los tutores	Comparación y análisis de resultados. Introducción de matrices de confusión para la interpretación de resultados	Entrenar redes para el problema de clasificación de galaxias con las modificaciones habladas durante la reunión	22/2 a 28/2
18	Decimoprimera reunión con los tutores	Introducción al problema de regresión y al artículo de Marc Huertas-Company	Lectura del artículo de Marc Huertas-Company	1/3 a 7/3
19	Decimosegunda reunión con los tutores	Dataset para el problema de regresión de valores morfológicos de galaxias	Generar el dataset para el problema de regresión	8/3 a 14/3
20	Decimotercera reunión con los tutores	Entrenamiento de redes para el problema de regresión	Entrenar redes para el problema de regresión	15/3 a 21/3
21	-	-	Entrenar redes para el problema de regresión	22/3 a 28/3
22	-	-	Entrenar redes para el problema de regresión	29/3 a 4/4
23	Decimocuarta reunión con los tutores	Introducción al problema de detección de bordes de galaxias mediante segmentación	Lectura y estudio de documentación de Fastai para el problema de segmentación	5/4 a 11/4
24	Decimoquinta reunión con los tutores	Dataset para el problema de segmentación	Generar el dataset para el problema de segmentación	12/4 a 18/4
25	Decimosexta reunión con los tutores	Entrenamiento de redes para el problema de segmentación	Entrenar redes para el problema de segmentación	19/4 a 25/4

26	-	-	Entrenar redes para el problema de segmentación	26/4 a 2/5
27	Decimoséptima reunión con los tutores	Entrenamiento de redes para el problema de segmentación mediante aprendizaje federado	Estudio y documentación sobre el aprendizaje federado y cómo realizarlo en Fastai y Pytorch	3/5 a 9/5
28	-	-	Entrenaer redes para el problema de segmentación aplicando aprendizaje federado.	10/5 a 16/5
29	Decimoctava reunión con los tutores	Entrenamiento de redes para el problema de segmentación mediante una ResNet propia	Entrenar redes para el problema de segmentación mediante una ResNet propia	17/5 a 23/5
30	Decimonovena reunión con los tutores	Revisión de la memoria	Corrección de la memoria con las indicaciones recibidas en la reunión	24/5 a 30/5

Cuadro 2.1

2.3. Análisis de riesgos

A continuación se detalla la estructura utilizada para describir los riesgos de este proyecto, adaptado de la metodología descrita en el capítulo 7 del libro *Software Project Management* de Bob Hughes y Mike Cotterell[4]:

- **Descripción**: Explicación del riesgo en cuestión
- **Nivel de probabilidad**; Probabilidad de que la situación del riesgo llegue a darse. Puede dividirse cualitativamente en:
  - Alto: Más del 50 % de probabilidad de que ocurra
  - Significativo: 30-50 % probabilidad de que ocurra
  - Moderado: 10-29 % probabilidad de que ocurra
  - Bajo: Menos del 10 % de probabilidad de que ocurra
- **Nivel de impacto**: Impacto en el tiempo de realización previsto del proyecto. Puede dividirse cualitativamente en:
  - Alto: Aumento de más del 30 % del tiempo total previsto
  - Significativo: Aumento de entre 20 y 29 % del tiempo total previsto
  - Moderado: Aumento de entre 10 y 19 % del tiempo total previsto

- Bajo: Aumento de menos del 10 % del tiempo total previsto

- **Plan de mitigación**: Acciones que se deben tomar para reducir la probabilidad de que ocurra el riesgo.
- **Plan de contingencia**: Acciones que se deben tomar para reducir el impacto del riesgo cuando llega a ocurrir.

Riesgo 1	**Enfermedad**
Descripción	Posibilidad de enfermar por Covid-19 debido a la situación sanitaria actual
Nivel de probabilidad	Bajo
Nivel de impacto	Depende de la gravedad de la efectos del Covid-19. Desde Bajo a Alto
Plan de mitigación	Cumplir las medidas recomendadas por las autoridades sanitarias: Uso de mascarilla, distancia de seguridad, lavado de manos, etc.
Plan de contingencia	Replanificación del proyecto en la medida de los posible.

Cuadro 2.2: Riesgo 1: Enfermedad

Riesgo 2	**Pérdida de datos**
Descripción	Posibilidad de perder la totalidad o una parte de los datos o del contenido del proyecto
Nivel de probabilidad	Bajo
Nivel de impacto	Depende de la cantidad de datos perdidos. Desde Bajo a Alto
Plan de mitigación	Realizar copias de seguridad periódicamente
Plan de contingencia	Restauración del contenido perdido mediante las copias de seguridad realizadas. Si no es posible, reelaboración del contenido perdido

Cuadro 2.3: Riesgo 2: Pérdida de datos

Riesgo 3	**Equipo de trabajo averiado**
Descripción	Posibilidad de perder la totalidad o una parte de los datos o del contenido del proyecto
Nivel de probabilidad	Bajo
Nivel de impacto	Depende de la gravedad de la avería de la máquina. Desde Bajo a Alto
Plan de mitigación	Realizar un mantenimiento correcto y un uso adecuado del equipo de trabajo con el que se desarrolla el proyecto. Realizar copias de seguridad periódicamente en un disco externo.
Plan de contingencia	Reparar la máquina averiada. Si no es posible, recuperar los datos que sea posible y continuar en otra máquina el desarrollo del proyecto

Cuadro 2.4: Riesgo 3: Equipo de trabajo averiado

Riesgo 4	**Sprint incompleto**
Descripción	Posibilidad de no completar los objetivos propuestos para uno o varios sprints en una tarea determinada
Nivel de probabilidad	Significativo
Nivel de impacto	Bajo
Plan de mitigación	Planificar las tareas de cada Sprint de forma que se tenga mucha certeza de que van a ser completadas.
Plan de contingencia	Replanificar el proyecto en la medida de los posible. Trasladar las tareas sin completar al siguiente Sprint

Cuadro 2.5: Riesgo 4: Sprint incompleto

Capítulo 3

Redes neuronales

En este capítulo se introducirán los conceptos que hay que conocer y entender antes de trabajar con redes neuronales. La estructura de este capítulo está basado principalmente en el libro online gratuito de Michael Nielsen [1]. A lo largo del capítulo se irán explicando los fundamentos teóricos intercalando con referencias al código del repositorio de GitHub que Michael Nielsen proporciona, en el que se implementa una red neuronal sencilla desde cero: neural-networks-and-deep-learning

3.1. Inteligencia artificial

3.1.1. ¿Qué es la inteligencia artificial?

Lo primero que debemos hacer es saber diferenciar los siguientes términos, que aparecerán de forma recurrente a lo largo del trabajo:

1. **Inteligencia artificial**: Consiste en dar a las máquinas la capacidad de imitar el comportamiento humano.

2. **Aprendizaje automático o machine learning**: Es una rama de la inteligencia artificial en la que las máquinas usan técnicas estadísticas para aprender a partir de información o experiencia anterior.

3. **Aprendizaje profundo o deep learning**: Es una rama del machine learning en la que se usan redes neuronales con múltiples capas para resolver distintos problemas, entre los que destacan la visión artificial, el reconocimiento del habla o la comprensión del lenguaje natural.

3.1.2. Breve historia de la inteligencia artificial

Todo empezó en 1943, año en el que Warren McCulloch y Walter Pitts desarrollaron un modelo matemático para una neurona artificial, en su artículo: *A Logical Calculus of the Ideas Immanent in Nervous Activity*[5], que sentó las bases de la inteligencia artificial. En 1950, Alan Turing propuso lo que se llamó como el Test de Turing: Si una máquina puede comportarse como un humano sin que se note la diferencia, es humano. Entonces comenzó un profundo debate filosófico sobre este tema. El término *inteligencia artificial* fue acuñado por John McCarthy en 1956. Se

empezó a discutir sobre las posibilidades que podría tener un ordenador, pero todo de forma teórica, y la inteligencia artificial se volvió, por primera vez, un campo de estudio.

En 1957, Frank Rosenblatt, influenciado por el trabajo de McCulloch y Pitts créo los cimientos para las redes neuronales mediante su artículo "Perceptron: A Perceiving and Reconginizing Automaton". El perceptrón es una neurona artificial que pretende imitar el comportamiento del cerebro humano. En vez de usar un sistema basado en reglas, usaría datos estadísticos para hacer predicciones. En 1965, Alexey Ivakhnenko, considerado el padre del deep learning, publicó la primera red neuronal funcional en su artículo "Group Method of Data Handling - A Rival Method of Stochastic Approximation"[6]. Los gobiernos de todo el mundo vieron las grandes posibilidades que tenía esta tecnología y comenzaron a financiar multitud de proyectos en este campo, pero esta fiebre del oro solo duró hasta mediados de los '70.

Cuando los sistemas se pusieron en práctica, se desvanecieron las esperanzas puestas en la inteligencia artificial. Los perceptrones son redes de una sola capa, es decir, funciones lineales, y por ello no sirven para predecir problemas del mundo real, donde se necesitarían funciones mucho más complejas. Este periodo de pérdida de fe en la inteligencia artificial se conoce como Ïnvierno de la inteligencia artificial".

Durante este periodo se frenó la cantidad de dinero invertido en proyectos de IA pero se hicieron nuevos descubrimientos, como que se podían unir varios perceptrones, uno detrás de otro, para formar redes con múltiples capas, que representarían funciones no lineales, más útiles para predecir fenómenos del mundo real. Solo quedaba saber cómo entrenar a una red, pero en 1986, Geoffrey Hinton dio con la respuesta. Se trataba del algoritmo de *backpropagation.* Más tarde, varios investigadores del NIST (National Institute of Standards and Technology(NIST) construyeron una red neuronal que reconociera caracteres escritos. Esto útlimo sentó los cimientos de lo que se conoce como visión artificial o *computer vision.*

Pero aunque los fundamentos teóricos estaban muy avanzados, todavía no era posible demostrarlos de forma práctica debido a las limitaciones del hardware y a la escasez de datos para entrenar las redes. En 1995 se introdujeron las Máquinas de Vectores de Soporte (SVM), una técnica de aprendizaje automático más rápida y que necesitaba mucha menos información que las redes neuronales.

Con el nuevo milenio llegó el Internet de alta velocidad, los móviles inteligentes, los videojuegos y una gran cantidad de sitios web. Se acabó el problema de la escasez de información para entrenar a una red neuronal, y con el crecimiento de los videojuegos en PC y en consola, fabricantes como NVIDIA y AMD, construyeron GPUs cada vez más potentes, que servían perfectamente para ejecutar las operaciones que requiere el entrenamiento de una red neuronal. En la última década, el crecimiento del deep learning ha sido exponencial.

3.1.3. Áreas de aplicación del deep learning

Hoy en día, el deep learning es usado en multitud de tareas, por ejemplo:

1. **Procesamiento del lenguaje natural**: Responder preguntas, reconocimiento del habla, clasificar documentos, encontrar nombres, fechas, conceptos, etc. en documentos.

2. **Visión artificial**: Reconocimiento de rostros, subtitulado de imágenes, conducción autónoma, interpretación de imágenes de satélites.

3. **Medicina**: Encontrar anomalías en imágenes de distintos tipos de análisis, realizar diagnósticos.

4. **Biología**: Tareas relacionadas con la clasificación de proteínas, células o mutaciones genéticas de genomas.

5. **Generación de imágenes**: Colorear imágenes, incrementar la resolución de las imágenes, eliminar ruido de las imágenes, convertir una imagen a un estilo artístico determinado.

6. **Sistemas de recomendación**: Búsqueda en la web, recomendaciones de productos en comercio electrónico.

7. **Juegos**: Ajedrez, Go y varios videojuegos de estrategia en tiempo real.

8. **Robótica**: Manejar objetos que son difíciles de coger o de localizar.

9. **Astronomía**: Realizar tareas relacionadas con imágenes astronómicas, un tipo de imágenes complicadas de interpretar por el ojo humano, y que una red neuronal puede hacer de forma más eficiente. Este trabajo se centrará en este aspecto.

10. **Otras**: Predicción financiera, texto a voz, antivirus y muchas más.

3.2. Fundamentos básicos de las redes neuronales

3.2.1. Perceptrón

Un perceptrón es un tipo de neurona artifical que recibe un conjunto de entradas binarias y produce una sola salida binaria. Cada entrada x_j tiene un peso w_j asociado.

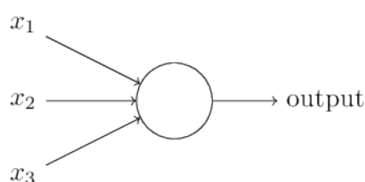


Figura 3.1: Esquema de un perceptrón, imagen tomada de [1]

Esta salida depende de si el valor de la suma ponderada $\sum_j w_j x_j$ es mayor o menor que un cierto valor límite(threshold). Se pueden establecer varias capas de perceptrones para que la salida de unos sirvan como entrada de otros y realizar cálculos más complejos. Podemos redefinir los perceptrones haciendo que $w \cdot x \equiv \sum_j w_j x_j$, donde w y x son vectores cuyas componentes son los pesos y las entradas, y definiendo el sesgo (bias) b como $b \equiv -threshold$. Por lo tanto, a mayor b, mayor probabilidad de que la salida sea 1. De esta forma, la fórmula para calcular la salida de un perceptrón es la siguiente:

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

Los perceptrones también se pueden usar para implementar puertas lógicas. Por ejemplo, se puede usar para simular un circuito formado por puertas NAND, que son puertas universales, ya que con puertas de este tipo se pueden implementar el resto de puertas lógicas.

Pero la mayor utilidad de las neuronas artificiales no es que sirvan como dispositivos de computación, sino que se les pueden aplicar algoritmos de aprendizaje para que aprendan por sí mismas a resolver problemas. Aunque, hoy en día el perceptrón es un tipo de neurona artificial menos usada que otros tipos de neuronas artificiales, como la neurona sigmoide.

3.2.2. Neurona sigmoide

Para hacer que una red neuronal aprenda habría que hacer pequeños cambios en los pesos y en el sesgo de la red que produjeran pequeños cambios en el output. Pero esto en los perceptrones es imposible, ya que su output es 0 o 1, y un pequeño cambio en uno de los pesos puede hacer que la salida resultante cambie completamente.

Por ello, tenemos que introducir un nuevo tipo de neurona artificial, la neurona sigmoide. Esta neurona ahora tiene entradas y salidas con valores comprendidos entre 0 y 1, y la salida viene dada por la fórmula:

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

donde σ es la función sigmoide:

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

Mediante esta función se consigue que pequeños cambios en los pesos o en el sesgo, produzcan pequeños cambios en la salida de la red. Es decir, es directamente proporcional. La función sigmoide es una de las muchas funciones de activación que se pueden usar en redes neuronales.

3.2.3. La arquitectura de las redes neuronales

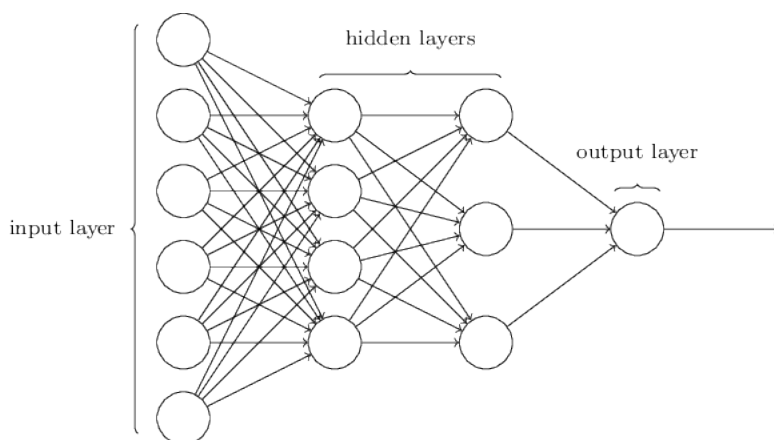


Figura 3.2: Arquitectura básica de una red neuronal, imagen tomada de [1]

Partes de una red neuronal:

1. **Input layer o capa de entrada**: Es la capa de más a la izquierda y contiene las neuronas de entrada.

2. **Output layer o capa de salida**: Es la capa de más a la derecha y contiene las neuronas de salida.

3. **Hidden layers o capas ocultas**: Son las capas que se encuentran entre la capa de entrada y la capa de salida

Este tipo de redes se llaman *feedforward neural networks*(redes de prealimentación), porque las salidas de una capa sirven como entradas de la siguiente y no hay bucles. Existen otro tipo de redes neuronales llamadas *recurrent neural networks* (redes de retroalimentación), que sí pueden tener bucles.

3.2.4. Descenso del gradiente

Lo primero que necesita una red neuronal para aprender es un conjunto de datos. En nuestro caso, este conjunto de datos estará formado por un conjunto de imágenes de entrenamiento, y otro conjunto de imágenes para probar la red una vez entrenada. Denotaremos el *training input* mediante x y el *output deseado* mediante $y = y(x)$. Lo que queremos es un algoritmo que nos permita encontrar los pesos y los sesgos de forma que el *output* aproxime $y(x)$ para toda entrada x. Para ello definimos la función de coste cuadrática:

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

donde w son los pesos, b son los sesgos, n es el número de entradas de entrenamiento y a son las salidas. El valor de C nunca es negativo, y se aproxima a 0 cuando $y(x)$ se aproxima a a, que es nuestro objetivo. Si C es muy grande, nuestro algoritmo de entrenamiento no es bueno. Es decir, la **función de coste** es lo que nos permitirá mejorar el rendimiento de nuestra red neuronal, así que debemos encontrar un conjunto de pesos y sesgos (que son los parámetros de la función) que minimice la función de coste, y para ello usaremos el descenso del gradiente.

El **descenso del gradiente** es un método matemático que permite hallar el mínimo de una función de forma iterativa. Supongamos que tenemos un conjunto de variables θ, y tenemos la función multidimensional f. Para unos valores iniciales de θ, calculamos el gradiente de la función, ∇f, que es un vector formado por las derivadas parciales de la función. Este vector indica hacia qué dirección es positiva la pendiente, por lo que usaremos el valor opuesto:

$$\theta := \theta - \nabla f$$

Falta añadir el parámetro η, que es el ratio de aprendizaje, que indicará cuánto nos desplazamos cada iteración, quedando la fórmula resultante así:

$$\theta := \theta - \eta \nabla f$$

En este vídeo [7] se puede ver gráficamente cómo funciona este método. Si η tiene un valor muy bajo, tendrá que calcular muchas veces el gradiente, ralentizando la operación, y puede que nunca lo encuentre, y si es muy alto, puede que nunca caiga en la zona donde está el mínimo. En este artículo interactivo [8] se puede observar y probar los diferentes comportamientos que puede tener. En la práctica, calcular el gradiente ∇C implica calcular el gradiente para cada x, que es

extremadamente lento. Para acelerar el proceso de aprendizaje, se usa la técnica llamada descenso del gradiente estocástico, que estima el gradiente ∇C calculando ∇C_x para un subconjunto aleatorio del total de entradas.

3.2.5. EL código para implementar una red que reconozca dígitos manuscritos

En el github de [1], se encuentra explicado de forma muy clara el código necesario para crear una red neuronal desde cero, sin ningún framework. Vamos a ir analizando este código para entender cómo se implementan los conceptos que vamos viendo y vamos a aplicar la red al problema del MNIST. Para probar este código necesitamos Python 2.7 y las librerías Numpy, Theano

MNIST [9] es un conjunto de 70000 imágenes de dígitoss escritos a mano de 28x28 píxeles. Fue recopilado por el NIST(National Institute of Standards and Technology) en 1999. Uno de los primeros desafíos de las redes neuronales fue crear una red que consiguiera reconocer estos dígitos con el mayor porcentaje de acierto. Aunque hoy está más que superado, es útil para demostrar cómo funcionan las redes neuronales.

Figura 3.3: Ejemplos de imágenes del MNIST

Empezamos por la clase `network.py`, que contiene la implementación más básica de una red neuronal. Vamos a crear una red que estará formada por 3 capas:

- Input layer: Formada por 784 neuronas, ya que vamos a analizar imágenes de 28x28 píxeles. El valor de cada neurona será 0.0 si representa el blanco, 1.0 si representa el negro. Los valores intermedios representan los distintos tonos de grises entre el blanco y el negro.

- Hidden layer: Formada por 15 neuronas.

- Output layer: Formada por 10 neuronas. Cada una representa el dígito analizado, del 0 al 9.

La clase `Network` tiene varios atributos:

- `num_layers`: número de capas que tiene la red.
- `sizes`: Vector que indica el número de neuronas en cada capa
- `biases`: Sesgos generados aleatoriamente en el constructor
- `weights`: Pesos generados aleatoriamente en el constructor

Tenemos además varias funciones:

- `sigmoid(z)`: Devuelve el valor de la función sigmoide del parámetro z
- `feedforward(self,a)`: Devuelve el output de la red a partir del input a
- `SGD(self, training data, epochs, mini_batch_size, eta, test_data=None)`: Este método implementa el descenso del gradiente estocástico. `epochs` es el número de "iteraciones" que hará el algoritmo. En cada epoch, divide el training_data en trozos más pequeños de tamaño igual a mini_batch_size.
- `update_mini_batch(self, mini_batch, eta)`: Este método aplica un solo paso de descenso del gradiente al subconjunto de los datos (`mini_batch`) y actualiza los pesos y los sesgos de la red. Este método llama a su vez a otro método llamado `backprop`, que implementa el algoritmo de backpropagation, que se explicará en la siguiente sección.

Al probarlo, obtenemos distintos resultados según el número de capas y el número de neuronas que le pasemos como argumento al constructor de la clase `Network`, y según el número de epochs que hagamos y del ratio de aprendizaje (`eta`) que le digamos. La dificultad está en encontrar los valores óptimos de estos parámetros, llamados hiperparámetros, ya que entrenar a una red neuronal lleva bastante tiempo, en función del hardware, del tamaño del conjunto de datos y del número de epochs, entre otras cosas. En este caso concreto, cada ejecución del programa puede llevar varios minutos.

3.3. Algoritmo de backpropagation

El algoritmo de backpropagation, ideado por Geoffrey Hinton, sirve para computar el gradiente de la función de coste de una forma eficiente. A continuación se hace resumen de los conceptos matemáticos sobre los que se basa el algoritmo sin entrar en detalle, ya que no es el objetivo de este trabajo aprender de forma completa la base matemática subyacente de las redes neuronales.

3.3.1. Una aproximación rápida basada en matrices para computar la salida de una red neuronal

Notación:

- w_{jk}^l es el peso de la conexión entre la k-ésima neurona de la (l-1)-ésima capa y la j-ésima neurona de la l-ésima capa.

- b_j^l es el sesgo de la j-ésima neurona de la l-ésima capa.
- a_j^l es la activación de la j-ésima neurona en la l-ésima capa.

Ahora definimos una matriz de pesos w^l para cada capa l. Las entradas de la matriz serán los pesos que conectan a la l-ésima capa, así que la entrada de la fila j, columna k es w_{jk}^l. También definimos un vector de sesgos b^l y de activaciones a^l. Por último, lo juntamos en una fórmula que dé una idea de cómo las activaciones de una capa dependen de las activaciones de la capa anterior.

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

Uno de los cálculos intermedios que se realizan en la ecuación anterior es $z^l = w^l a^{l-1} + b^l$. Esta cantidad recibe el nombre de *weighted input* o *entrada ponderada*.

3.3.2. Dos suposiciones necesarias sobre la función de coste

1. La función de coste se puede escribir como la media $C = \frac{1}{n}\sum_x C_x$ de las funciones de coste C_x para ejemplos de entrenamiento individuales. De esta forma asumimos que x es un valor fijo.
2. La función de coste se puede escribir como una función de las salidas de la red: $C = C(a^L)$. Como x era un parámetro fijado, y también lo será. Así que solo tenemos en cuenta las activaciones de salida, a^L.

3.3.3. El producto de Hadamard

Es una operación algebraica que se aplica entre matrices de la misma dimensión. Dadas las matrices s y t el producto de Hadamard, $s \odot t$ es la matriz cuyas entradas son el resultado de multiplicar los elementos de s por los elementos de t que tienen la misma posición en la matriz.

3.3.4. Las cuatro ecuaciones fundamentales detrás de la backpropagation

Denotamos mediante δ_j^l el error en la j-ésima neurona de la l-ésima capa. Se puede calcular como $\delta_j^l \equiv \frac{\partial C}{\partial z_j^l}$. δ^l será el vector de errores de la capa l-ésima

Una ecuación para el error en la capa de salida δ^L :

$$\delta^L = \nabla_a C \odot \sigma'(z^L) \tag{3.1}$$

Una ecuación para el error δ^l en términos del error en la siguiente capa δ^{l+1} :

$$\delta^l = ((w^{l+1})^T \delta^{l+1}) \odot \sigma'(z^l) \tag{3.2}$$

Una ecuación para el ratio de cambio del coste con respecto a cualquier sesgo de la red:

$$\frac{\partial C}{\partial b_j^l} = \delta_j^l \tag{3.3}$$

Una ecuación para el ratio de cambio del coste con respecto a cualquier peso de la red:

$$\frac{\partial C}{\partial w_{jk}^l} = a_k^{l-1} \delta_j^l \tag{3.4}$$

Conclusiones

Un peso en la capa final aprenderá muy lentamente si la neurona de salida tiene una activación muy baja (≈ 0) o muy alta (≈ 1). En este caso se dice que la neurona está saturada y deja de aprender. Algo similar pasa con los sesgos.
Cualquier *weighted input* de una neurona saturada aprenderá lentamente. Es decir, un peso aprenderá lentamente si la neurona de entrada tiene una activación muy baja o la neurona de salida está saturada.
Además, estas cuatro ecuaciones sirven para cualquier función de activación, no solo la función sigmoide, así que se pueden usar para diseñar funciones de activación que tengan propiedades deseables.

3.3.5. El algoritmo de backpropagation

Elementos del algoritmo:

- **Input,** x: son los valores de activación para las neuronas de la capa de entrada
- **Feedforward**: Calcular el vector de weighted inputs, z^l, y de activaciones, a^l, para cada capa l.
- **Output error** δ^L: Calcular el vector de errores de la capa de salida mediante la primera ecuación fundamental (3.3.4).
- **Backpropagate the error**: Para cada capa l, excepto la de salida, calcular el vector de errores δ^l mediante la segunda ecuación fundamental (3.2).
- **Output**: Obtener el gradiente de la función de coste mediante la tercera (3.3) y la cuarta (3.4) ecuación fundamental.

Resumiendo, este algoritmo calcula el error de la capa de salida y después lo propaga hacia atrás, recalculando los w y los b.

La parte más interesante del backpropagation es que puede realizar todos los cálculos en una pasada hacia delante, seguida de una pasada hacia atrás, lo que lo hace mucho más rápido que otras formas de calcular el gradiente de la función de coste. Para entender un poco mejor cómo funciona, supongamos que hacemos un pequeño cambio en uno de los pesos de la red, w^l_{jk}. Esto producirá cambios en la activación de la correspondiente neurona, que a su vez provocará cambios en todas las activaciones de las neuronas de la siguiente capa, que a su vez afectará a la siguiente y así sucesivamente. Pero para realizar el algoritmo completo, hay que tener en cuenta todos los pesos de cada capa, no solo uno, de forma que los cambios de una capa dependen de los cambios de las capas anteriores.

3.4. Mejorar el aprendizaje de las redes neuronales

En esta sección se introducirán técnicas usadas comúnmente y conceptos importantes para mejorar la eficiencia de las redes neuronales

3.4.1. La función de coste de entropía cruzada

La función de coste cuadrático que hemos estado usando hasta ahora tiene una gran desventaja, y es que si la red comienza con unos valores de pesos y sesgos muy distantes a los óptimos, la red tardará mucho más en aprender. Por ello, vamos a usar una función de coste que arregle justo eso.

Definimos la función de coste de entropía cruzada para una neurona como:

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

donde n es el número de items en los datos de entrenamiento, el sumatorio se aplica a todas las entradas de entrenamiento x, e y es la salida deseada correspondiente. Esta función sirve como función de coste ya que $C > 0$ se cumple siempre y si la salida real se acerca a la salida deseada, el valor de la función será cercano a 0, como pasaba con la función de coste cuadrático. Calculando $\frac{\partial C}{\partial w_j}$ y simplificando obtenemos:

$$\frac{\partial C}{\partial w_j} = \frac{1}{n}\sum_x x_j(\sigma(z) - y)$$

Esto indica que el ratio al que la red aprende sobre los pesos viene dado por $\sigma(z) - y$, que es el error en la salida. Así que, cuanto mayor sea el error, más rápido aprenderá la neurona. Lo mismo pasa con los sesgos. La fórmula anterior era para una neurona, ahora hay que generalizarlo para una red entera, quedando de la siguiente forma:

$$C = -\frac{1}{n}\sum_x \sum_j [y_j lna_j^L + (1-y_j)ln(1-a_j^L)]$$

3.4.2. ¿De dónde viene la entropía cruzada?

Supongamos que hemos descubierto que el problema de que una neurona aprenda lentamente es el término $\sigma'(z)$ de la función de coste, entonces el objetivo es encontrar una función de coste donde este término desaparezca, para hacer que cuanto mayor sea el error inicial, mayor sea la velocidad a la que la neurona aprende. Sobre el significado intuitivo de la entropía cruzada, podríamos decir que mide cuánto estamos de sorprendidos cuando calculamos el valor de la salida.

3.4.3. Softmax

Softmax consiste en definir un nuevo tipo de capa de salida en la que, en vez de aplicar la función sigmoide para hallar la salida, aplicamos la función softmax a las entradas ponderadas z_j^L. De esta forma, la activación a_j^L de la j-ésima neurona de salida es

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

La característica principal de esta función es que hace que todas las activaciones de una capa sumen 1, transformando las salidas en una distribución de probabilidad.

Una vez tenemos la función de activación, debemos definir la función de coste, que será la función de coste $log - likelihood$, que es

$$C \equiv -lna_y^L$$

Esta función soluciona el problema de la ralentización del aprendizaje para la función de activación softmax. En conclusión, se puede decir que una capa de salida softmax con una función de coste $log - likelihood$ es similar a una capa de salida sigmoide con una función de coste de entropía cruzada.

3.4.4. Sobreajuste

Hay veces, a la hora de entrenar la red neuronal, que se no se observa ninguna mejoría en la precisión a pesar de aumentar el número de epochs de entrenamiento. En este caso se dice que nuestra red está sobreajustada o sobreentrenada. Esto puede pasar porque la red está memorizando las imágenes de entrenamiento, en vez de entender lo que hay en las imágenes.

Hay que detectar cuándo estamos sobreentrenando una red. La forma más común sería observando los resultados con diferentes parámetros y viendo cuando el aprendizaje se estanca. Y uno de los mejores métodos de reducir el sobreajuste es incrementar el tamaño de los datos de entrenamiento, aunque esto no siempre es posible, ya que puede ser difícil o caro conseguir estos datos.

3.4.5. Regularización

Hay otras técnicas que ayudan a reducir el sobreajuste llamadas técnicas de regularización. Una de ellas es la regularización L2 o decaimiento del peso ($weightdecay$). Consiste en añadir un término extra a la función de coste llamado término de regularización:

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

C_0 es la función de coste original, n es el tamaño de nuestro conjunto de datos de entrenamiento y λ es el parámetro de regularización y cumple $\lambda > 0$. Intuitivamente, significa que la red preferirá aprender pesos pequeños, y solo permitirá pesos grandes si mejoran considerablemente la primera parte de la función, que es la función original.

Si aplicamos este método a nuestro programa, observamos que los resultados mejoran incluso cuando el tamaño del conjunto de entrenamiento es pequeño (tamaño 1000).
Además, cuando no aplicamos regularización, puede que la función de coste se quede atascada en mínimos locales, debido a que el vector de pesos puede crecer sin parar, haciendo que el descenso del gradiente produzca cambios tan pequeños en la dirección, que no avance.

3.4.6. ¿Por qué la regularización ayuda a reducir el sobreajuste?

Supongamos que tenemos un conjunto de puntos(x,y) de una función desconocida y queremos hallar una función para ajustarlos. Si usamos un polinomio de grado igual al número de puntos obtendremos un ajuste perfecto para ese conjunto de datos, figura 3.4. Pero si añadimos después más puntos, este polinomio no ajustará nada bien, ya que solo sirve para esos datos concretos. Sin

embargo, podemos ajustarlos mediante una función lineal, que, aunque no ajuste perfectamente los puntos, a la larga aproximará mejor los puntos que añadamos después, figura 3.5.

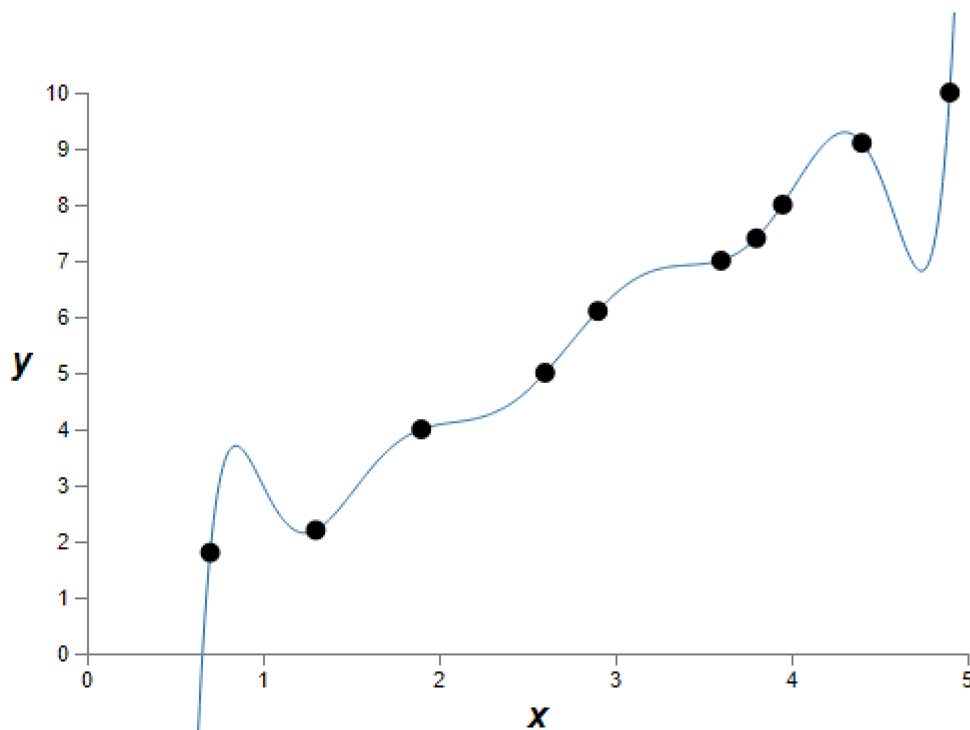


Figura 3.4: Polinomio muy ajustado

Algo similar ocurre con la regularización, que evita que la red se ajuste perfectamente al conjunto de datos de entrenamiento para que dé mejores resultados con otros datos distintos. Metafóricamente, podemos decir, que la regularización obliga a la red a entender y aprender, en vez de memorizar.

Por último, hay que aclarar por qué no se tienen en cuenta los sesgos en el término de regularización, y es porque permitir sesgos grandes da a nuestra red más flexibilidad en el comportamiento, ya que hace que las neuronas se saturen más fácilmente, que, en ocasiones, puede ser deseable.

3.4.7. Otras técnicas de regularización

Hay muchas otras técnicas de regularización aparte de la L2. A continuación, se describen 3 tipos diferentes:

Regularización L1

Este método añade un término de regularización que contiene el sumatorio del valor absoluto de los pesos:

$$C = C_0 + \frac{\lambda}{n} \sum_{w} |w|$$

Igual que la L2, penaliza los pesos grandes y hace que la red prefiera los pequeños. Pero se diferencian en que en la L1 los pesos se encogen por una cantidad constante hacia 0, mientras que en la L2 es por una cantidad proporcional a w.

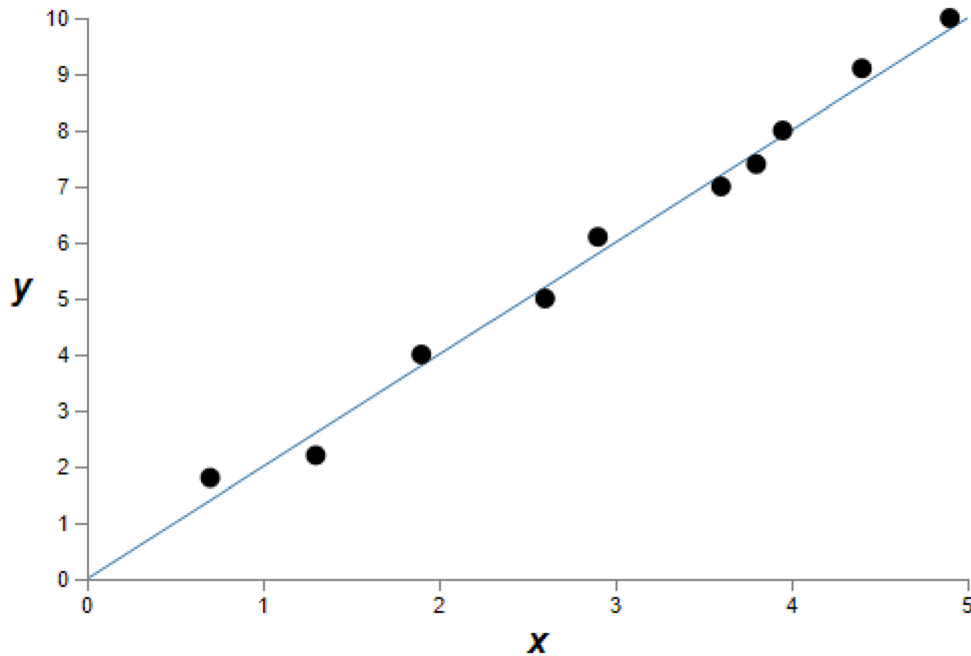


Figura 3.5: Polinomio poco ajustado

Dropout

Es una técnica que consiste en eliminar aleatoriamente y temporalmente la mitad de las neuronas de las capas escondidas. A continuación se propaga hacia delante el input x y se aplica el algoritmo de backpropagation de la manera usual (pero siempre con la mitad de neuronas intermedias eliminadas). Después se repite el proceso restaurando las neuronas que se habían eliminado y escogiendo otras de forma aleatoria para eliminarlas.

Este método funciona porque es como si estuviéramos entrenando varias redes diferentes y después promediando los resultados, haciendo que el modelo sea robusto ante la pérdida de alguna conexión individual.

Expandir artificialmente los datos de entrenamiento

Como hemos dicho anteriormente, una forma de mejorar el rendimiento de nuestra red es aumentar el tamaño del conjunto de datos de entrenamiento, pero esto a veces es complicado. Sin embargo, hay varios métodos para aumentarlo artificialmente.

En nuestro caso de reconocimiento de caracteres escritos, podríamos rotar las imágenes varios grados hacia la derecha o hacia la izquierda(no más de 15º aproximadamente ya que si no no correspondería con el carácter de verdad), de forma que el carácter sigue siendo legible pero para la red neuronal los píxeles han variado completamente. Otra técnica que se usa son las distorsiones elásticas, un tipo de distorsión de imagen que emula las oscilaciones de los músculos de la mano. Si nuestro objetivo fuera el reconocimiento de voz, podríamos alterar los datos añadiendo o quitando ruido de fondo.

3.4.8. Inicialización de pesos

Hasta ahora, para inicializar los pesos hemos usado una distribución normal de Gauss, con media 0 y desviación estándar 1, esto hace que las neuronas se saturen fácilmente y el aprendizaje se ralentice. Para solucionarlo, podemos inicializar los pesos con variable aleatorias Gaussianas con media 0 pero con desviación estándar $\frac{1}{\sqrt{n_{in}}}$, de forma que las neuronas no se saturen tan fácilmente. Hay muchas otras formas de inicializar los pesos.

3.4.9. Segunda visita al código de reconocimiento de dígitos manuscritos

En el fichero `network2.py` del Github de Nielsen están implementadas algunas de estas nuevas técnicas, como una inicialización de pesos mejor, y una regularización L2 en la función de coste, que ahora es de entropía cruzada. Al añadir todas estas técnicas hemos mejorado el rendimiento de nuestra red neuronal en comparación con la inicial de `network.py`.

3.4.10. ¿Cómo elegir los hiper-parámetros para nuestra red neuronal?

- **Estrategia general:** Debemos intentar que cada ejecución del programa sea corta para poder probar muchas combinaciones posibles de hiper-parámetros. Por ello, debemos reducir el tamaño de los datos de entrenamiento a una cantidad que nos permita probar muchas cosas y ver resultados de forma rápida, para ver si vamos por buen camino o no.

- **Ratio de aprendizaje:** Deberíamos probar diferentes valores para η y ver si el coste desciende en los primeros epochs. Puede pasar que el coste oscile mucho debido a que el ratio de aprendizaje es demasiado alto y el descenso del gradiente nunca encuentre el mínimo, en ese caso, debemos ir variando el orden de magnitud de η hasta encontrar el adecuado.

- **Parando temprano para determinar el número de epochs:** Si observamos que no hay ninguna mejoría durante varios epochs habría que pensar en parar la ejecución y cambiar los hiper-parámetros.

- **Programar el ratio de aprendizaje:** Podemos ir variando η durante los epochs en vez de mantenerlo constante, empezando con un ratio muy alto e ir reduciéndolo lentamente.

- **El parámetro de regularización:** Se recomienda empezar sin regularización ($\lambda = 0{,}0$) y después ir incrementándolo o decrementándolo por factores de 10.

- **Tamaño del mini-batch:** Un tamaño demasiado pequeño ralentizará el procedimiento porque no aprovecharemos los beneficios de las librerías de matrices bien optimizadas. Un tamaño demasiado grande hará que los pesos y los sesgos no se actualicen lo suficiente. Este parámetro es independiente del resto, así que primero habría que centrarse en los demás, y una vez que tengamos buenos valores en el resto de hiper-parámetros, centrarse en el tamaño del mini-batch.

Este campo está en constante investigación y no hay fórmulas mágicas para hallar los hiperparámetros óptimos. Lo que hay es multitud de artículos que exponen sus propios métodos y técnicas para optimizar las redes neuronales.

3.4.11. Otras técnicas

Variaciones del descenso del gradiente estocástico

- **Técnica Hessiana:** Esta técnica converge a un mínimo en menos pasos que el descenso del gradiente estándar, pero no se usa en la práctica porque requiere muchísima carga computacional en comparación a otras técnicas.

- **Descenso del gradiente basado en el momento:** Hasta ahora el descenso del gradiente se podía imaginar como una bola que bajaba por un valle buscando el punto más bajo, pero sin seguir estrictamente las leyes de la física. El descenso del gradiente basado en el momento añade un nuevo parámetro μ que representaría la fricción de la bola. Este parámetro debería tener un valor entre 0 y 1, para que la velocidad se mantenga en unos valores aceptables, a la vez que evitamos el $overshooting$, que es pasarnos del punto donde está el mínimo. A diferencia de la técnica Hessiana, esta sí es muy fácil de implementar y de computar.

Existen muchos otros métodos para minimizar la función de coste, pero no hay unanimidad sobre cuál es el mejor.

Otros modelos de neurona artificial

- **Neurona tanh:** Reemplaza la función sigmoide por la función tangente hiperbólica. Esta función tiene una forma parecida a la sigmoide, pero ahora toma valores entre -1 y 1, en vez de entre 0 y 1.

- **Neurona lineal rectificada:** La salida de la neurona viene dada por la siguiente función:

$$max(0, w \cdot x + b)$$

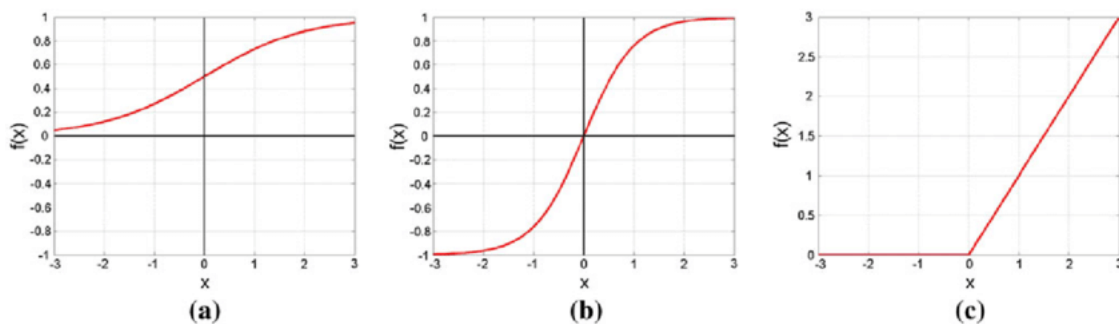


Figura 3.6: Función sigmoide (a), tanh (b), y lineal rectificada (c) [10]

Todas las neuronas pueden computar cualquier función, solo se diferencian en la eficiencia a la hora de hacer que la red aprenda, pero no hay consenso tampoco en este aspecto. Casper Hansen, en su artículo *Activation Functions Explained*, explica 6 tipos diferentes de funciones de

activación que se pueden usar a la hora de construir redes neuronales, para dar una idea de la variedad de opciones que existen.

En el campo de las redes neuronales, al contrario que en otras áreas de investigación científica, no se tienen pruebas sólidas sobre si una técnica es mejor que otra o viceversa a la hora de construir una red neuronal. Hoy en día, únicamente tenemos intuiciones, heurísticas, hipótesis y pruebas empíricas, pero no pruebas matemáticas.

3.5. Pruebas visuales de que las redes neuronales pueden computar cualquier función

Las redes neuronales se basan en el teorema de aproximación universal, que dice que una función, con las suficientes variables y los suficientes parámetros, puede aproximar y representar cualquier problema del mundo real. La universalidad de las redes neuronales es su característica más impresionante. En este capítulo se intenta explicar el por qué de esta propiedad de forma gráfica. Se tomará como referencia este artículo de Shiva Sankeerth Reddy [11], quien a su vez toma de referencia el capítulo 4 de [1].

3.5.1. Dos consideraciones

Primera: En realidad, una red neuronal, no puede computar cualquier función de manera exacta, sino que realiza una aproximación más o menos precisa. Supongamos que queremos computar una función $f(x)$ con una precisión $\epsilon > 0$. Está garantizado que usando una cantidad de neuronas escondidas suficiente, obtendremos una red neuronal que cumple $|g(x) - f(x)| < \epsilon$ para cualquier entrada x.

Segunda: Una red neuronal solo puede computar funciones continuas, ya que, por dentro, este es el tipo de funciones que usa para sus cálculos. Pero si queremos computar funciones discontinuas, podemos aproximarlas mediante funciones continuas.

3.5.2. Universalidad con una entrada y una salida

Supongamos que queremos aproximar una función cuya gráfica resulta a primera vista complicada, como la de la figura 3.7.

Comenzamos con una red neuronal con una neurona de entrada, una de salida y dos en una capa escondida. Recordemos que la función de activación de una neurona es la función $\sigma(wx+b)$, donde σ es la función sigmoide. Si visualizamos gráficamente la salida de una de las neuronas escondidas, podemos comprobar cómo, aumentando w a una cantidad muy grande, la función sigmoide parece una función escalonada (step function) (figura 3.8), donde la posición del escalón en el eje X es proporcional a b e inversamente proporcional a w. Definimos entonces que el escalón está en la posición $s = -b/w$.

Combinando las salidas de cada neurona escondida obtenemos la salida ponderada de la capa escondida, que tiene forma de escalera como en la figura 3.9. La posición y la altura de cada escalón depende de los valores de s de las neuronas escondidas y de los pesos w que llegan a la neurona de salida. Si probamos otras combinaciones de valores puede no tener forma de escalera, sino que la gráfica sea una sucesión de "baches"hacia arriba o hacia abajo, (figura 3.10). Combinando

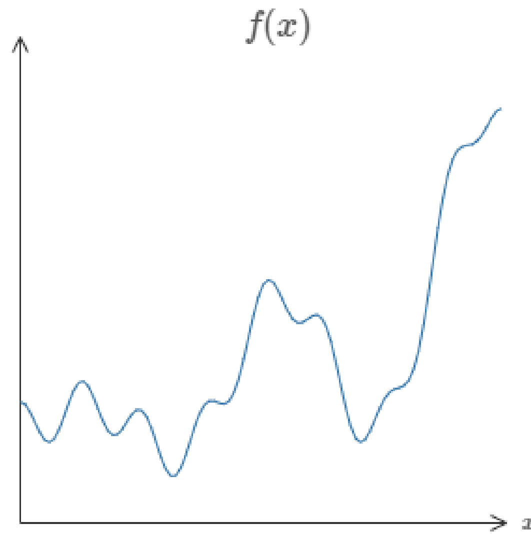


Figura 3.7: Función complicada de ejemplo

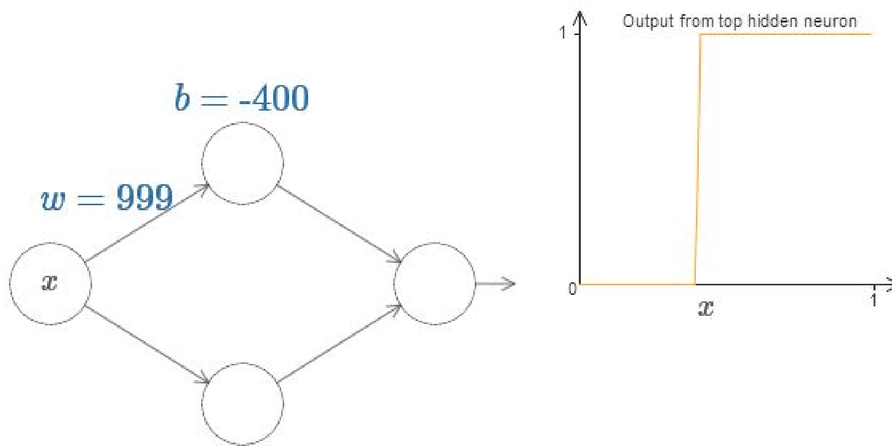


Figura 3.8: Función escalonada

muchas neuronas, podemos hacer un cantidad de baches mayor que aproximará la función que queríamos al principio, con mayor o menor precisión, como se puede observar en la figura 3.11. Cuantas más neuronas escondidas (como se ha dicho en la primera consideración del apartado anterior), más baches podremos hacer y mejor será la aproximación.

3.5.3. Varias variables de entrada

A continuación, vamos a hacerlo con dos variables de entrada. Supongamos que tenemos 2 neuronas de entrada, x e y, 1 intermedia y 1 de salida. Si hacemos que w_1 (el peso de la neurona x), tenga un valor muy grande y w_2 (el peso de la neurona y) tenga valor 0, crearemos una función escalonada tridimensional en la que la dirección del escalón es en el eje x, como en la figura 3.12. Si intercambiamos los valores de las neuronas de entrada, el escalón estará en la dirección del eje y.

Ahora, si usamos dos neuronas intermedias, podremos crear baches tridimensionales, y si usamos más podremos crear funciones de "torre", como en la figura 3.13. Combinando muchas de estas, podremos aproximar cualquier función de 2 variables. Cuantas más neuronas, más torres podremos hacer, y por lo tanto, podremos aproximar la función con más detalle. Cuando tenemos

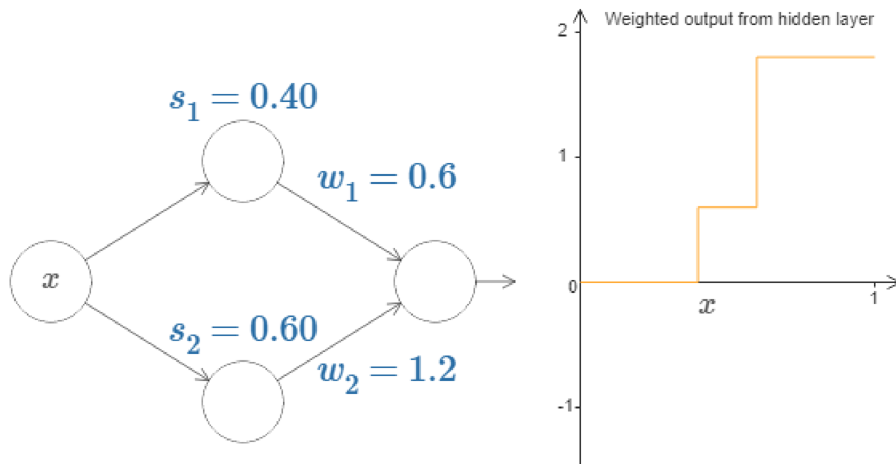


Figura 3.9: Función con forma de escalera

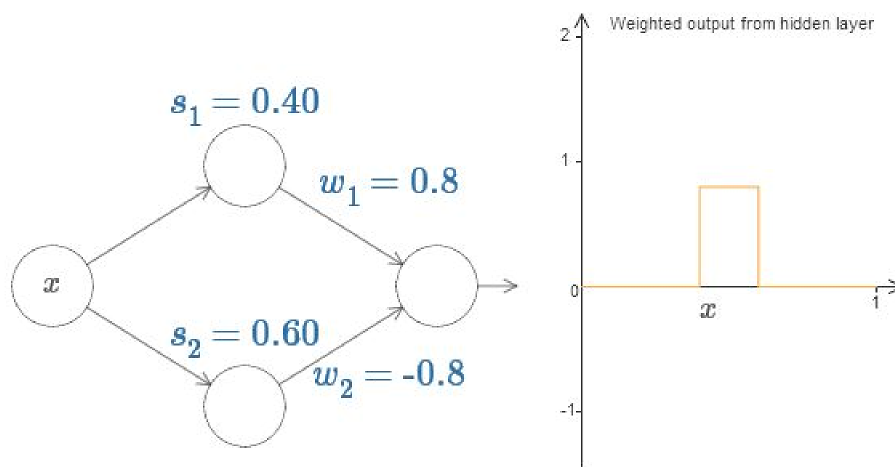


Figura 3.10: Función con forma de bache

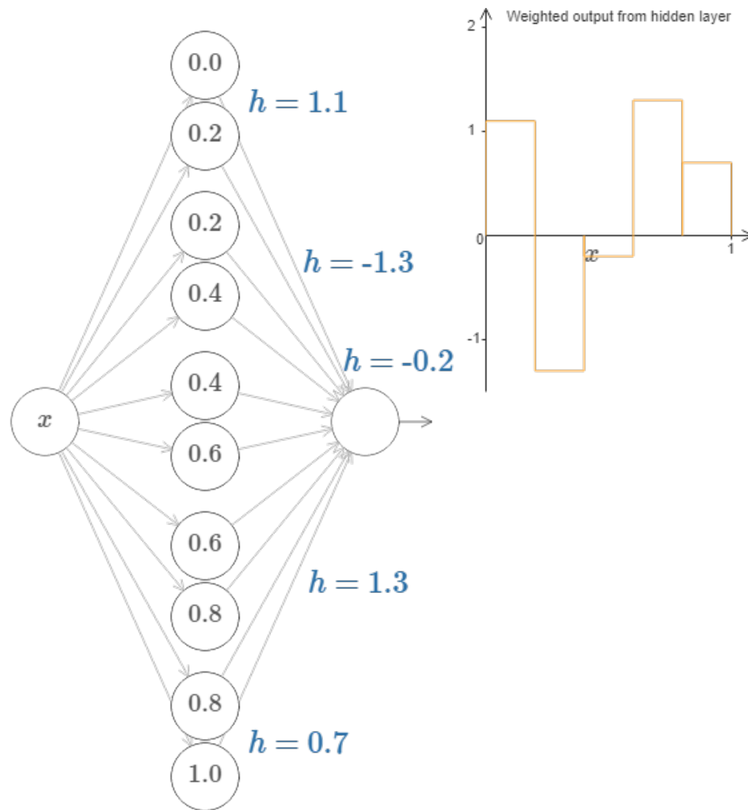


Figura 3.11: Función con varios "baches"

más de 2 variables de entrada ya no podemos imaginarlo gráficamente, pero la idea es la misma.

3.5.4. Conclusión

Hemos probado que las redes neuronales que usan la función sigmoide como función de activación pueden computar cualquier función. Podemos usar otras funciones $s(z)$ que también puedan hacer esto, aunque deben cumplir que $s(z)$ esté bien definida cuando $z \longrightarrow -\infty$ y $z \longrightarrow \infty$ y que además sean diferentes, para que podamos aproximar la función a una función escalonada con los métodos que hemos dicho anteriormente.

Si hemos demostrado que una red neuronal con una sola capa escondida puede computar cualquier función, ¿por qué usar redes neuronales profundas que requieren más tiempo de cómputo?

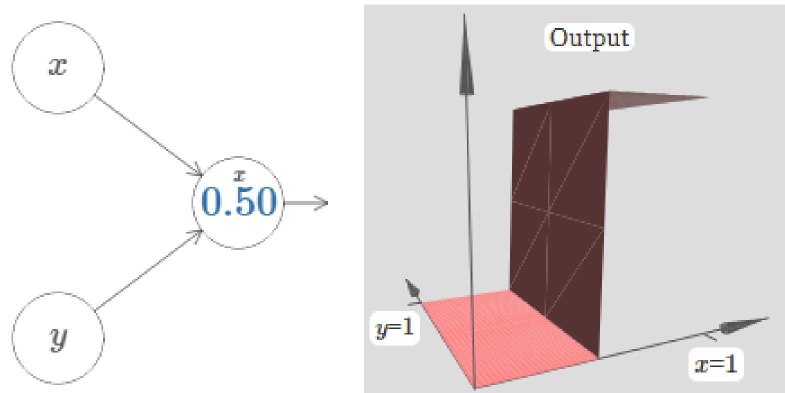


Figura 3.12: Función escalonada con 2 variables de entrada

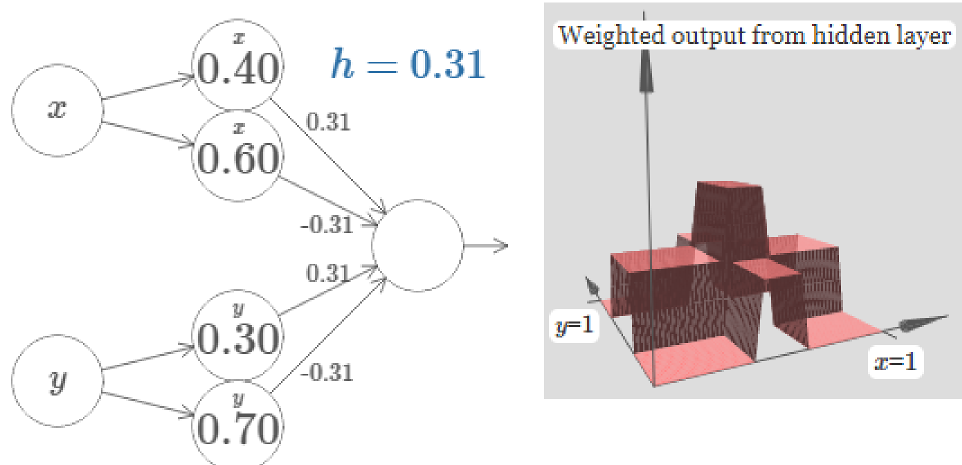


Figura 3.13: Función con forma de "torre"

Pues porque las redes profundas tienen una estructura jerárquica que las hace especialmente útiles para resolver muchos problemas del mundo real, como se verá posteriormente.

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

En la práctica, para resolver un problema, normalmente resolvemos subproblemas más pequeños y después vamos combinando e integrando las soluciones. Es decir, usamos capas de abstracción. En una red neuronal, usar muchas capas intermedias puede ayudar a resolver problemas complejos del mundo real que serían muy complicados de resolver con una sola capa escondida. Sin embargo, las redes profundas tienen el inconveniente de que son muy difíciles de entrenar.

3.6.1. El problema del gradiente evanescente

Vamos a volver a nuestro programa `network2` y vamos a ejecutarlo con más capas escondidas. Observamos que el rendimiento no solo no mejora, sino que empeora ligeramente. En teoría, con más capas y neuronas, la red debería poder aprender mejores funciones de clasificación, pero no es así. Para averiguar el problema, vamos a comparar las velocidades de aprendizaje de las capas. Comenzamos por definir el gradiente de la j-ésima neurona de la l-ésima capa como

$$\delta_j^l = \frac{\partial C}{\partial b_j^l}$$

Así, podemos pensar en δ^l como un vector cuyas entradas indican cómo de rápido aprende la capa. Para obtener una medida final de la velocidad de aprendizaje de una capa calculamos la norma de ese vector $\|\delta^l\|$.

Después de realizar estos cálculos con varias capas, se obtienen unos resultados en los que se cumple que la velocidad de aprendizaje va aumentando significativamente a medida que avanzamos en la red, lo que significa que las primeras capas escondidas aprenden muy despacio. Esto es el problema de gradiente evanescente. Hay veces que ocurre justo lo contrario, y las primeras capas

aprenden demasiado rápido. Es el problema del gradiente explosivo. En general, el gradiente en las redes neuronales profundas es inestable, y para poder entrenarlas, debemos intentar solucionar este problema.

3.6.2. ¿Qué está causando el problema del gradiente evanescente?

Supongamos que tenemos una red neuronal con 3 capas escondidas con una sola neurona en cada capa, la función de activación sigmoide y una función de coste C. La fórmula para calcular el gradiente del coste asociado a la primera capa escondida sería

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

Es decir, que depende de todos los términos, y si estos términos son menores que 1, al ir multiplicándose entre ellos, darán un resultado muy pequeño. Si comparamos con la fórmula para calcular un gradiente más adelantado en la red, por ejemplo, el de la tercera capa escondida, vemos que tiene menos factores

$$\frac{\partial C}{\partial b_3} = \sigma'(z_3) w_4 \sigma'(z_4) \frac{\partial C}{\partial a_4}$$

Esta es la idea intuitiva que está detrás del problema del gradiente evanescente. Por otro lado, si se da que los valores son mayores que 1, cuantos más factores haya, mayor será el gradiente. Este es el gradiente explosivo. Por lo tanto, para evitar el problema del gradiente inestable, los valores de los pesos y los sesgos deberían estar dentro de un rango de valores muy ajustado que permita mantener el equilibrio entre los gradientes de las distintas capas, algo que normalmente no ocurre.

En este ejemplo hemos usado una red de juguete, con una sola neurona en cada capa, pero el funcionamiento en una red compleja, con muchas neuronas por capa, es similar. En una red compleja, además, como hay muchos más elementos, el gradiente se desvanece de forma exponencial, por lo que el aprendizaje en las primeras capas se ralentiza muchísimo.

3.6.3. Otros obstáculos para el aprendizaje profundo

El problema del gradiente inestable es solo una de las muchas dificultades que surgen a la hora de entrenar una red neuronal profunda. También hay que tener en cuenta la función de activación usada, la forma de inicializar los pesos o la implementación del descenso del gradiente, además de la elección de la arquitectura de la red y de los hiper-parámetros.

3.7. Redes convolucionales

En este capítulo se hará uso de varios de los conceptos explicados en capítulos anteriores para aplicarlos al entrenamiento de redes neuronales profundas. También se explicará uno de los tipos de redes neuronales profundas más usadas, las redes convolucionales, y se aplicará al problema del reconocimiento de caracteres escritos del MNIST.

3.7.1. ¿Qué es una convolución?

Una convolución es una operación matemática que transforma dos funciones f y g en una tercera función que en cierto sentido representa la magnitud en la que se superponen f y una versión trasladada e invertida de g. Tiene aplicaciones en muchos campos como la estadística, la física o las telecomunicaciones. En el caso de las redes neuronales, se aplica al tratamiento de imágenes. Un filtro de imagen implementado en programas de edición como Photoshop o Gimp es una matriz convolucional, llamado *kernel de imagen*, que se aplica a cada uno de los pixeles de la imagen original. En este artículo[12] de Victor Powell hay una demostración visual y muy didáctica sobre el funcionamiento de estos filtros. La red neuronal convolucional aplicará este tipo de filtros para extraer las características principales con las que quiera quedarse.

3.7.2. Introduciendo redes convolucionales

Hasta ahora hemos usado redes completamente conectadas, donde cada neurona está conectada a todas las neuronas de la capa posterior. Esta arquitectura de redes neuronales no es buena para clasificar imágenes porque no tiene en cuenta la estructura espacial de las imágenes.

Cuando nos enfrentamos a problemas más complejos, como decir si una fotografía contiene una cara o no, hay que descomponer el problema en subproblemas, que a su vez se pueden descomponer en otros subproblemas más pequeños, creando una red neuronal con varias capas escondidas. Cada subred responderá preguntas simples que se irán componiendo hasta tener la respuesta final.

Las redes neuronales convolucionales usan una arquitectura concreta que está especialmente pensada para clasificar imágenes, haciéndoles más fáciles de entrenar. Usan tres ideas básicas: campos receptivos locales (*local receptive fields*), pesos compartidos (*shared weights*) y agrupación (*pooling*).

Campos receptivos locales

En las capas completamente conectadas, las neuronas de entrada se veían gráficamente como una línea vertical de neuronas; ahora hay que imaginarlas como un cuadrado de neuronas (figura 3.14), donde cada neurona de la primera capa escondida está conectada a solo una pequeña región de neuronas de entrada. Esta región es el campo receptivo local de la neurona escondida. Cada conexión aprende un peso y la neurona escondida aprende un sesgo. Después, esta región la vamos desplazando, según el paso (*stride*) especificado, para conectar cada neurona escondida con su grupo de neuronas de entrada. Esto hará que la capa escondidas tenga menos neuronas que la capa de entrada.

Pesos y sesgos compartidos

Vamos a usar los mismos pesos y sesgos para todas las neuronas de la capa escondida. Si tenemos una imagen de 28x28 píxeles y hemos creado regiones de 5x5, la capa escondida será de 24x24 neuronas, de forma que la salida de la neurona escondida de la fila j, columna k es :

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

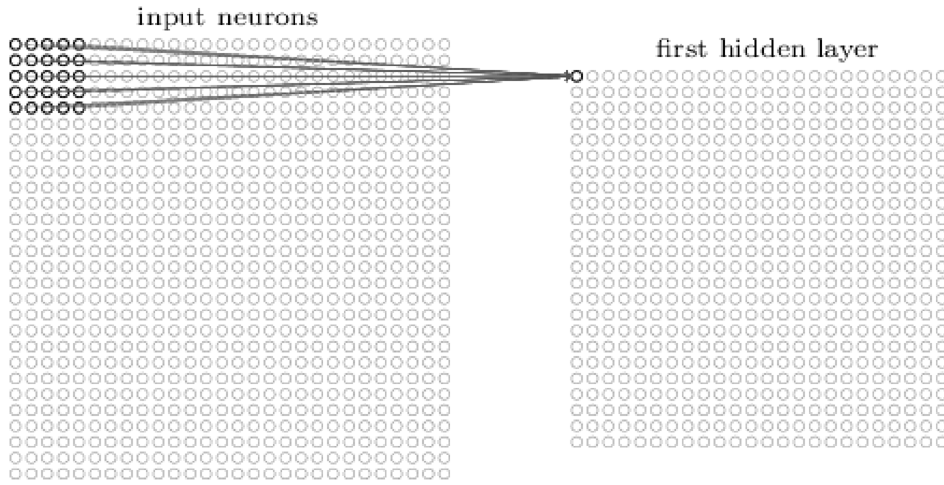


Figura 3.14: Campos receptivos locales, imagen tomada de [1]

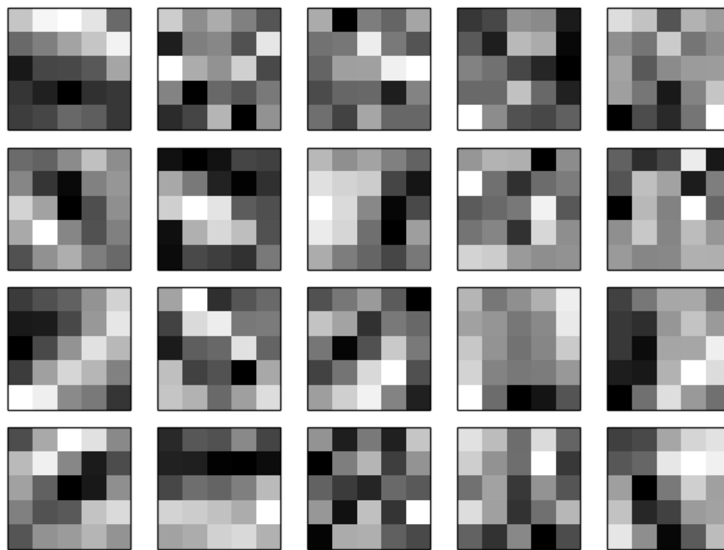


Figura 3.15: Mapas de características que una capa convolucional genera con el MNIST

Aquí, b es el valor del sesgo compartido, $w_{l,m}$ es un array de pesos compartidos de 5x5, y $a_{x,y}$ es la activación de entrada en la posición x, y.

Esto significa que todas las neuronas de la capa escondida detectan la misma característica (*feature*), pero en diferentes partes de la imagen. A veces, al mapeado entre la capa de entrada y la capa de salida, se le llama mapa de características o *feature map* (figura 3.15). Este mapa está definido por los pesos compartidos y el sesgo compartido; a la combinación de estos dos valores se le llama *kernel* o filtro (*filter*).

Pero para el reconocimiento de imágenes, no es suficiente un solo mapa, sino varios, dependiendo del problema. Cada mapa con sus propios pesos y sesgo compartidos, para que detecten características diferentes teniendo en cuenta la estructura espacial de la imagen. Otra ventaja de usar estos mapas es que reduce enormemente el número de parámetros de la red.

Capas de agrupación

Una capa de agrupación toma la salida de un *feature map* de la capa convolucional y crea un *feature map* condensado. Uno de los métodos comunes es el *max-pooling*, donde una unidad de

agrupación, por ejemplo, de 2x2, da como salida la máxima activación de la región de entrada. De forma que si teníamos capas convolucionales de 24x24 del paso anterior, ahora tendremos capas de agrupación de 12x12 neuronas. Por cada mapa de características, habrá una capa de agrupación. La idea intuitiva es que, cuando encontramos una característica, no nos importa la posición exacta, sino la posición aproximada relativa a otras características. Existen otros tipos de *pooling* como el *L2 pooling*.

3.7.3. Tercera visita al código de reconocimiento de dígitos manuscritos

En nuestro ejemplo del MNIST tendríamos una primera capa de entrada de 28x28 neuronas, que es el tamaño de las imágenes que analizamos, una capa convolucional que usa un campo receptivo local de 5x5, que da como resultado 3 mapas de 24x24, y una capa de agrupación, de regiones de 2x2, que tendrá 3 mapas de 12x12 neuronas. Por último conectamos todas las neuronas de la capa de agrupación a todas las neuronas de la capa de salida. La entrenaremos mediante descenso del gradiente estocástico y backpropagation, aunque habrá que modificar las implementaciones, ya que ahora la arquitectura de la red es distinta.

Para ello vamos a usar el último programa del Github de [1]: `network3.py`. A diferencia de los anteriores `network.py` y `network2.py`, en este no están implementados desde cero los algoritmos de backpropagation y del descenso del gradiente estocástico (usando la biblioteca de matrices Numpy), sino que usa una biblioteca de *machine learning* llamada Theano, que puede hacer uso de la GPU de nuestro ordenador para acelerar el aprendizaje.

Analicemos el código del programa `network3.py`. Tenemos la clase `FullyConnectedLayer`, cuyos pesos y sesgos se inicializan aleatoriamente. Después se cargan los pesos y sesgos en unas variables compartidas de Theano para que puedan ser procesadas por la GPU. El método `set_inpt` se utiliza para introducir las entradas a la capa; además, puede recibir un parámetro que indique el número de neuronas que desaparecen en cada iteración si queremos usar dropout.

Las clases `ConvPoolLayer` y `SoftmaxLayer` son similares a la anterior, salvo en la forma de computar el output. Theano proporciona operaciones para calcular convoluciones, max-pooling y la función softmax. La clase principal del programa es la clase `Network` que contiene un constructor y el método `SGD`, que sirve para entrenar a la red.

Primero vamos a probar una arquitectura poco profunda con una sola capa escondida, 60 epochs, ratio de aprendizaje $\eta = 0{,}1$ un tamaño de mini-batch de 10 y sin regularización. Estamos usando una función de activación sigmoide, una capa de salida softmax y una función de coste log-likelihood. Obtenemos una precisión del 97,80 %.

A continuación, vamos a transformarla en una red profunda. Insertamos una capa convolucional, con un campo receptivo local de 5x5, un paso de 1 y 20 mapas de características. También insertamos una capa de max-pooling de agrupaciones de 2x2, y añadimos una capa extra completamente conectada antes de la capa de salida. Obtenemos una precisión del 98,78 %

Después insertamos una capa convolucional igual que la anterior justo a continuación de esta. Obtenemos una precisión del 99,06 %. La segunda capa convolucional recibe como entrada una versión de la imagen original más abstracta y condensada, y tiene acceso a todas las características detectadas en la anterior capa convolucional.

Vamos a usar otras técnicas para intentar aumentar la precisión de la red neuronal:

1. **Usando unidades lineales rectificadas:** Si cambiamos la función de activación sigmoide por la función de activación lineal rectificada, esto es, $f(z) \equiv max(0, z)$, con 60 epochs, $\eta = 0{,}03$ y usando regularización L2 con el parámetro de regularización $\lambda = 0{,}1$, obtenemos una precisión del 99,23 %.

2. **Expandiendo los datos de entrenamiento:** Vamos a expandir nuestros datos mediante el programa `expand_mnist.py`. Esto aumentará el tamaño del conjunto de datos de entrenamiento de 50.000 a 250.000 imágenes. Obtenemos una precisión del 99,37 %.

3. **Insertando una capa extra completamente conectada:** Con 300 y 1000 neuronas obtenemos respectivamente unas precisiones de 99,46 % y 99,43 %. No parece una gran mejoría.

4. **Usando dropout:** Esta técnica consistía en hacer desaparecer varias neuronas escondidas en cada iteración. Por eso, hemos reducido el número de epochs a 40 y hemos aumentado el número de neuronas de las capas completamente conectas a 1000 neuronas. Obtenemos una precisión del 99,60 %

5. **Usando un conjunto de redes:** Esta técnica consiste en hacer que varias redes voten para determinar la mejor clasificación, ya que las redes normalmente cometen errores distintos debido a que cada una se inicializa de forma aleatoria. Supongamos que hemos entrenado 5 redes neuronales como hasta ahora, cada una con una precisión del 99,60 %. Si las juntamos para que voten cada imagen, obtenemos una precisión del 99,67 %. Es decir, solo han fallado 33 de 10.000 imágenes.

Hemos conseguido sobreponernos al problema del gradiente inestable gracias a varias cosas:

1. Usar redes convolucionales para reducir enormemente el número de parámetros de esas capas.

2. Usar técnicas de regularización como el dropout para reducir el sobreajuste.

3. Usar unidades lineales rectificadas en vez de neuronas sigmoides.

4. Usar GPUs para acelerar el proceso de cómputo.

3.8. Deep Learning

En esta sección se sintetizan los conceptos necesarios para crear una red neuronal y se hablan sobre temas de actualidad del Deep Learning.

3.8.1. Ingredientes de una red neuronal

Para construir una buena red neuronal hay que tener en cuenta muchos aspectos, algunos son propiamente técnicos, pero otros, de los que se habla después, son derivados de la ética y la responsabilidad social:

Conjuntos de datos o datasets

Los métodos tradicionales de aprendizaje automático que se usaban en los 80' y 90' funcionaban con unos pocos cientos o miles de datos de ejemplo, a diferencia de las redes neuronales, que necesitaban una cantidad de datos mucho mayor para realizar predicciones decentemente. Hoy en día, estamos teniendo una explosión de datos, pero para hacer un uso efectivo de estos datos, hay que etiquetarlos. Este etiquetado actúa como profesor de nuestra IA y es muy costoso tanto en tiempo como en dinero. Cada tipo de dato requiere de un proceso de etiquetado diferente. Hay compañías, como Appen o Scale Ai, que se dedican a proporcionar servicios de etiquetado de datos.

Pero no hace falta gastarse una fortuna para entrenar una red neuronal, gracias a dos cosas:

1. Grandes compañías y organizaciones han hechos públicos muchos conjuntos de datos etiquetados.

2. La técnica del *transfer learning* o aprendizaje de transferencia, que permite entrenar nuestros modelos con pocos datos si ya han sido entrenados por otra persona con una gran cantidad de datos en un problema similar.

Arquitectura del modelo

A alto nivel, el modelo es simplemente una función, que tiene una o más entradas, que pueden ser fotos, texto, audio, vídeo, etc., y devuelve una salida, que es una predicción. Un buen modelo será aquel cuyas predicciones correspondan con la realidad. Por dentro, el modelo de la red neuronal es un grafo que tiene nodos y aristas. La disposición de los nodos y aristas es la *arquitectura del modelo*. La arquitectura de la red neuronal dependerá del problema que estemos tratando. Por ejemplo, las redes neuronales convolucionales (CNNs) se usan para tratar imágenes o audio, mientras que las redes neuronales recurrentes (RNNs) se usan para procesamiento de textos.

Hardware

Las compañías de hardware, como Microsoft o NVIDIA, han experimentado un crecimiento brutal en la última década, y la variedad de hardware es muy amplio. Hay que saber bien qué es lo que se necesita para la labor que se quiere realizar. En nuestro caso, se trata de entrenar redes neuronales, que puede llevar mucho tiempo, desde horas a días. El valor que nos aporta un hardware más potente es la capacidad ejecutar de más iteraciones y, por lo tanto, experimentar más veces en menos tiempo. Típicamente, las GPUs el proceso de entrenamiento de la red neuronal entre 10 y 15 veces más en comparación con las CPUs.

3.8.2. IA responsable

Está claro que la inteligencia artificial tiene un gran potencial para mejorar la vida de las personas y la sociedad en general, pero mal usada puede causar grandes daños, intencionadamente o no. Y no es culpa de la inteligencia artificial, sino de sus diseñadores. Hay que tener en cuenta varios aspectos a la hora de crear una IA.

Parcialidad

Nadie es completamente objetivo e imparcial, todos tenemos nuestros defectos y prejuicios. Y eso se refleja en el trabajo que desempeñamos, muchas veces de forma inconsciente. Por eso, a la hora de crear una inteligencia artificial, hay que asegurarse de que no genera discriminación u ofensa hacia nadie y que es objetiva e imparcial cuando realiza sus funciones.

Rendición de cuentas y explicabilidad

Con los métodos tradicionales de machine learning, los científicos de datos escogían manualmente las características de los datos que el modelo aprendería. Este método era más engorroso pero daba a los creadores un mayor control sobre cómo saldría la predicción. Pero con el deep learning, estas características son escogidas automáticamente. Los científicos de datos proveen a las redes neuronales de un enorme conjunto de datos y estas acaban realizando predicciones más o menos fiables, pero los científicos no saben exactamente cómo funcionan. Esto es un grave problema ya que hoy en día, el deep learning se usa para tomar decisiones importantes que afectan a la sociedad, y no podemos dejar estas decisiones a una "caja negra" que no sabemos cómo funciona. Se está empezando a invertir en *Explainable AI*, en la que el modelo puede explicar qué factores han llevado a la predicción que ha hecho.

Reproducibilidad

La investigación realizada en un área de la ciencia gana aceptabilidad entre la comunidad científica si el experimento es reproducible, es decir, que cualquiera puede replicar las condiciones en que se hizo y obtener los mismos resultados. Gradualmente, cada vez más investigadores están publicando sus resultados y el código que han usado de forma abierta para que todo el mundo pueda aprender.

Robustez

Una IA robusta es aquella que es capaz de seguir haciendo predicciones fiables a pesar de que haya ruido, pequeñas desviaciones o manipulación intencionada en los datos.

Privacidad

En su empeño por por construir mejores modelos, las compañías tiene que recolectar una gran cantidad de datos. Y a veces, rozan los límites de la legalidad y entran en conflicto con el derecho universal de todo ser humano a su privacidad.

3.8.3. Progreso reciente en el reconocimiento de imágenes

El MNIST se introdujo en 1998, época en la que era un desafío. Pero, hoy en día, el problema del MNIST está más que superado y se buscan problemas de reconocimiento de imágenes más complicados.

El artículo de 2012 LRMD[13]: En este artículo, los autores explican cómo crearon una red neuronal que clasificara imágenes de ImageNet, una base de datos online de imágenes. Usaron 16

millones de imágenes en color de 20000 categorías diferentes. La red neuronal que construyeron obtuvo una precisión del 15,8 %.

El artículo de 2012 KSH[14]: Se intentó clasificar 1.2 millones de imágenes de ImageNet de 1000 categorías, usando una red profunda de 5 capas convolucionales (3 de ellas con max-pooling) y 2 capas completamente conectadas. También usaron una variante de la regularización L2, dropout y descenso del gradiente estocástico basado en el momento. Se obtuvo una precisión del 84,7 %.

Otras actividades: Actualmente hay muchísimos problemas sobre reconocimiento de imágenes que se están investigando. Un ejemplo es el problema de reconocer números en los portales de las casas en las imágenes de Google Street View. Un equipo de Google consiguió hacer una red neuronal que reconoció casi 100 millones de números con una precisión similar a la de un ser humano.

3.8.4. Otras aproximaciones a las redes neuronales

En este trabajo nos centraremos en las redes neuronales convolucionales y en el tratamiento de imágenes, pero hay otros métodos para utilizar las redes neuronales que funcionan muy bien según el problema concreto que se quiera tratar.

Redes neuronales recurrentes

Este tipo de redes se diferencian de las redes neuronales prealimentadas(feedforward) en que la salida de una capa escondida no depende solo de su entrada, sino de sus propias salidas anteriores. Cambian dinámicamente, no son estáticas y con una estructura predeterminada, y por eso son muy útiles para analizar datos o procesos que cambian con el tiempo, como el reconocimiento del habla.

Long short-term memory units (LSTMs)

Las redes neuronales recurrentes sufren del problema del gradiente inestable mucho más que las prealimentadas, debido a que ahora el gradiente se propaga en el tiempo. Para solucionar este problema se idearon las LSTMs.

3.8.5. Redes de creencia profunda, modelos generativos y máquinas de Boltzmann

Las redes de creencia profunda (DBN) son un tipo de modelo generativo. En una red prealimentada, solo especificamos las entradas y después se calculan las activaciones de las siguientes neuronas. En un modelo generativo, podemos especificar los valores de algunas activaciones y ejecutar el proceso al revés, generando las entradas para esas activaciones. En el problema del reconocimiento de dígitos escritos se podría usar para generar imágenes que parezcan dígitos escritos. Las máquinas de Boltzmann son una de las técnicas necesarias para implementar este tipo de redes.

3.8.6. El futuro de las redes neuronales

Interfaces de usuario dirigidas por la intención (intention-driven):

Esta idea consiste en hacer interfaces de usuario tolerantes a imprecisiones humanas, y que no respondan de forma literal a lo que pide el usuario, sino que intenten discernir qué es lo que de verdad quiere hacer. Es algo parecido al mensaje de Google cuando cometes una errata al buscar algo y te dice "Quizá quisiste decir ...".

Ciencia de datos:

Otro campo de aplicación es la ciencia de datos, donde el aprendizaje automático se usa para analizar los datos y encontrar los *known unknowns*, es decir, lo que sabemos que está pero no entendemos, entre los datos de que disponemos. Esto puede resultar muy rentable para las empresas, las cuales pueden invertir en machine learning, contratando más profesionales en este campo y contribuyendo al desarrollo de este.

El rol de las redes neuronales y del deep learning:

En la década de 1980 las redes neuronales fueron muy usadas debido a la invención del backpropagation, pero en los 90, fueron perdiendo importancia en favor de otras técnicas de aprendizaje automático. Hoy en día, han vuelto a tomar fuerza, pero no se sabe si han vuelto para quedarse o surgirán otras técnicas de machine learning que sean más eficaces que las redes neuronales.

3.8.7. Ejemplos de aplicaciones web que usan redes neuronales

A continuación se enumeran algunos sitios web cuyo objetivo es demostrar de forma sencilla de lo que son capaces las redes neuronales.

- **TalkToTransformer**: Herramienta que redacta texto a partir de un tema y párrafo inicial dados. TalkToTransformer es una pequeña demo que forma parte del software Inferkit.
- **Nado name generator**: Herramienta que genera nombres aleatorios que encajen dentro de una temática concreta. Nado name generator forma parte de Onitools.
- **ThisPersonDoesNotExist**: Esta herramienta es una web que genera, cada vez que recargas la página, una foto realista generada artificialmente de una persona que no existe. ThisPersonDoesNotExist utiliza la técnica de red generativa antagónica (*generative adversarial network*, GAN) para generar las imágenes.

Capítulo 4

Contexto tecnológico

En este capítulo se detallan las herramientas y los frameworks de deep learning de Python utilizados durante la realización de este trabajo.

4.1. Herramientas básicas

4.1.1. Anaconda

Anaconda[15] es una distribución de Python enfocada a la ciencia de datos y al aprendizaje automático. Permite instalar paquetes de Python muy fácilmente y ejecutarlos de distintas formas, entre las que destaca Jupyter Notebook. Anaconda instala por defecto más de 100 paquetes de Python, por ello, existe una versión más ligera llamada Miniconda, que instala por defecto solo los paquetes necesarios para la instalación.

Una vez instalado, lo recomendable es ejecutar el **Anaconda Navigator**, (Figura 4.1), una interfaz gráfica muy amigable para gestionar toda la distribución. Todos los programas y paquetes que aparecen se pueden instalar y abrir sin necesidad de abrir el Navigator.

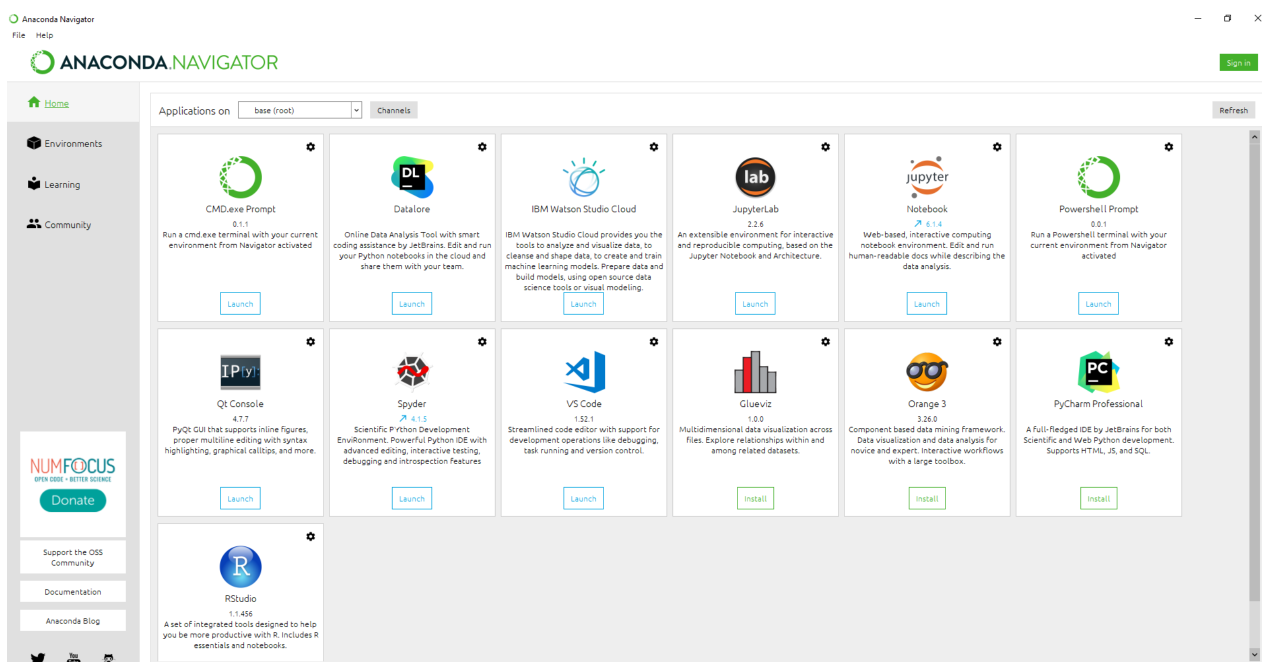


Figura 4.1: Anaconda Navigator, la interfaz gráfica de Anaconda

Anaconda se organiza en entornos(environments), que podemos crear y eliminar. En cada entorno podemos tener una versión de Python diferente y los paquetes que queramos. De esta forma, podemos ir cambiando de entorno según necesitemos una versión concreta de Python o de algún paquete de una forma muy cómoda. Para crear un entorno abrimos el CMD.exe Prompt del Navigator e introducimos:

```
conda create --name NOMBRE python=VERSION
```

Donde NOMBRE es el nombre que le queramos dar al entorno y VERSION es la versión de Python que queramos. Anaconda por defecto viene con un entorno cuyo nombre es *base*. Si queremos trabajar con otro entorno hay que introducir:

```
conda activate NOMBRE
```

Donde NOMBRE es el nombre de uno de los entornos que hayamos creado. Para instalar paquetes podemos usar cualquiera de los siguientes comandos:

```
pip install PAQUETE
pip install PAQUETE==VERSION
conda install PAQUETE
conda install PAQUETE==VERSION
```

Donde PAQUETE es el nombre del paquete que queremos instalar y VERSION es la versión concreta del paquete que queremos instalar. Si no especificamos versión se instala la versión más reciente compatible con la versión de Python del entorno. Es preferible instalar todos los paquetes con conda y los que no se pueda, con pip, aunque no hay grandes diferencias [16]. Una vez que tenemos instalados los paquetes que queramos necesitamos un entorno de ejecución. Como se puede observar en la figura 4.1, hay multitud de herramientas disponibles. En este trabajo se va a hacer uso de Jupyter Notebook

4.1.2. Jupyter Notebook

Jupyter Notebook [17] es una herramienta que permite incluir texto formateado, imágenes, vídeos y lo más importante, fragmentos de código ejecutable, en un documento interactivo, llamado cuaderno(notebook)(Figura 4.2). Es muy útil para explicar código ya que permite ejecutarlo poco a poco y añadir explicaciones entre los fragmentos. Un notebook está formado por celdas (cells). Estas pueden ser de dos tipos:

1. **Celdas de Markdown**: Contienen texto, imágenes, vídeos y más.

2. **Celdas de código**: Contiene código que se puede ejecutar.

Para ejecutar una celda de código simplemente hay que seleccionar una celda y pulsar shift+enter o clicando en el botón de Run. Debajo de la celda nos saldrá la salida del código que hayamos escrito, si es que tiene y los errores en caso de que los haya habido.

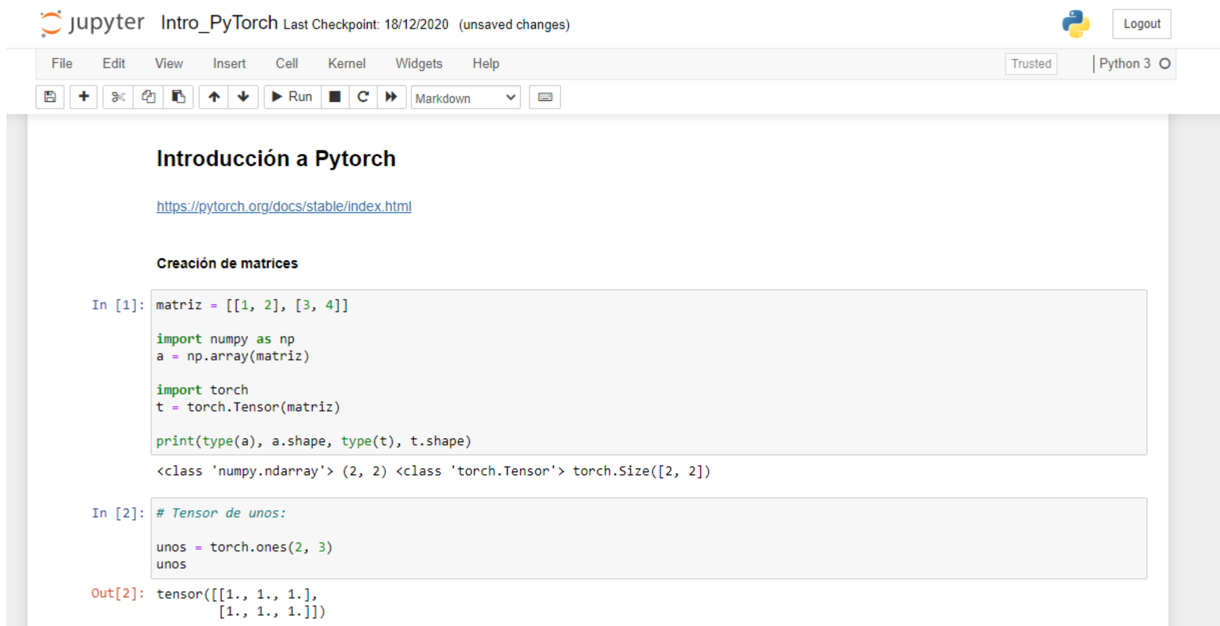


Figura 4.2: Ejemplo de un Jupyter Notebook

4.1.3. Hardware

Los notebooks pueden ser ejecutados de dos formas principalmente:

1. **Localmente**: Requiere configurar todo el sistema para que funcione perfectamente, lo que puede llevar bastante tiempo, pero se ejecutará en el ordenador local sin necesidad de depender de conexión a internet. Es la mejor opción si se dispone de un hardware potente.

2. En la nube mediante **Google Colab**: Permite ejecutar notebooks sin necesidad de ninguna configuración previa y con un hardware medio de forma totalmente gratuita. Es la mejor opción si no se dispone de hardware potente y en especial, si no se dispone de tarjeta gráfica, y no se quiere gastar tiempo en configurar el sistema.

Para trabajar con redes neuronales es altamente recomendable usar una tarjeta gráfica. Entrenar una red neuronal requiere alimentarla con una gran cantidad de datos del mismo tipo, en nuestro caso imágenes. Si ejecutamos el entrenamiento de la red en una CPU, solo podemos darle imágenes de 1 en 1, pero con una GPU, podemos aumentar el tamaño de batch a muchas más, dependiendo de la capacidad de la tarjeta gráfica. De esta forma, se disminuye el tiempo de entrenamiento de la red considerablemente.

Si se opta por trabajar con una tarjeta gráfica de forma local, es recomendable usar Nvidia, ya que tiene compatibilidades con los principales frameworks de deep learning gracias a Nvidia Cuda. Si usa una de AMD, la configuración puede ser más complicada. En este trabajo se ha ejecutado todo localmente usando una tarjeta gráfica Nvidia RTX 3070.

4.1.4. Frameworks para Deep Learning en Python

Hay distintos frameworks que simplifican la tarea de construir redes neuronales y abstraen los detalles más complicados de la implementación que hemos visto en el capítulo 1. En los capítulos siguientes se trabajará con todos ellos.

TensorFlow

Desarrollado por Google, salió a la luz en noviembre de 2015. Tuvo una gran recepción , pero a medida que aumentó el número de usuarios, aumentaron las quejas sobre la dificultad de uso del framework. El equipo de TensorFlow reconoció que si querían democratizar la inteligencia artificial, tenían que hacer sus herramientas más accesibles. Por ello, TensorFlow acabó incorporando Keras.

Keras

Como respuesta a la dificultad de uso que tenían otros frameworks de deep learning (TensorFlow aún no había salido, pero había muchos otros), François Chollet sacó en marzo de 2015 Keras. Keras provee una interfaz mucho más intuitiva y fácil de usar que el resto de frameworks, y como framework de backend computacional usa TensorFlow. Es decir, Keras es una API para TensorFlow.

PyTorch

Desarrollado por Facebook, fue lanzado a principios de 2016. Los desarrolladores de PyTorch tenían la ventaja de haber visto las limitaciones de TensorFlow, e interaron hacerlo más fácil de usar.

Fastai

Framework basado en PyTorch que crea un nuevo nivel de abstracción por encima. Es decir, Fastai es una API para usar PyTorch, así que es equivalente a lo que hace Keras con Tensorflow. Fue creado por Jeremy Howard y Rachel Thomas y lanzado en 2018. El objetivo principal de fastai es hacer el deep learning lo más accesible posible.

4.1.5. Bibliotecas de Python

En los notebooks explicados en este trabajo se hace uso de diferentes bibliotecas de Python:

- **Numpy**: Paquete que permite realizar computación numérica en Python de forma eficiente, como si se estuviera programando en C.
- **Matplotlib**: Paquete que sirve para visualizar datos en Python de muchas formas.
- **Astropy**: Paquete que proporciona herramientas para tratar ficheros de imágenes astronómicas y otras utilidades para astronomía.
- **Pandas**: Paquete que permite manipular y analizar muchos tipos de datos, por ejemplo, datos tabulares.

4.2. Keras & TensorFlow

En esta sección se realizarán algunos ejemplos prácticos de redes neuronales tomados del libro *Practical Deep Learning for Cloud, Mobile & Edge, Real-World AI & Computer-Vision Projects*

Using Python, Keras & Tensorflow de Anirudh Koul, Siddha Ganju y Meher Kasam, [2]. Este libro tiene un repositorio Github en el que están subidos los Jupyter Notebooks que se explican en cada capítulo. Todos los ejemplos de este capítulo están hechos con los frameworks **Keras** y **Tensorflow**. También se introducirán algunos conceptos nuevos usados en los ejemplos que no se hayan explicado anteriormente.

4.2.1. Clasificación de imágenes con Keras

En esta sección se hará un ejemplo introductorio con el framework Keras. Clasificaremos imágenes usando una red ya existente y sin necesidad de conjuntos de datos gracias al transfer learning.

Configuración del entorno

Una vez que hemos instalado Anaconda debemos crear un entorno con la versión de Python que necesitemos, en este caso, la 3.7, para ello abrimos el CMD.exe desde la interfaz gráfica Anaconda Navigator para que se nos abra el prompt del entorno base e introducimos el siguientes comandos:

```
conda create env --name py37 python==3.7
```

Ahora instalamos los siguientes paquetes:

```
conda install tensorflow==2.0.0
conda install keras
conda install numpy
conda install matplotlib
conda install pillow
conda install pathlib
```

Predicción de la categoría de una imagen

La clasificación de imágenes consiste en decir que una imagen contiene el objeto X con una cierta probabilidad, donde X es una de las categorías predefinidas para el modelo. Si la probabilidad sobrepasa cierto umbral, entonces hay es muy posible que la imagen contenga una o más instancias de X.

En este primer ejercicio vamos a seguir los pasos que se explican en el notebook del capítulo 2 de [2]. Primero importaremos las bibliotecas que necesitamos:

```python
1 import tensorflow as tf
2 from tensorflow.keras.applications.resnet50 import preprocess_input,
    decode_predictions
3 from tensorflow.keras.preprocessing import image
4 import numpy as np
5 import matplotlib.pyplot as plt
```

Fragmento de código 4.1: Importar bibliotecas de Keras y TensorFlow

Ahora cargamos la imagen que queremos analizar, en nuestro caso será una foto de un gato:

```
IMG_PATH = '../../sample-images/cat.jpg'
img = image.load_img(IMG_PATH, target_size=(224, 224))
plt.imshow(img)
plt.show()
```

Fragmento de código 4.2: Cargar imagen y mostrarla con Matplotlib

Antes de alimentar a Keras con una imagen hay que convertirla a un formato estándar, ya que, normalmente, los modelos preentrenados esperan que el input sea de un tamaño determinado. La mayoría de modelos esperan un lote o **batch** de imágenes como entrada. Si tenemos solo una imagen, creamos un batch de una sola imagen. Por ejemplo, un batch de 64 imágenes de 224x224 píxeles cada una, cada una con tres canales, que corresponden a los colores de la imagen (RGB), tendrá una forma de 64x224x224x3. En machine learning, los modelos funcionan mejor cuando se les da valores que están dentro de un rango, típicamente [0,1] o [-1,1]. Como los valores de los píxeles de una imagen toman valores entre 0 y 255 (debido a que cada uno de los tres valores de RGB se representa con 8 bits), debemos normalizarlos antes mediante la función de Keras `preprocess_input`. La normalización o *feature scaling* es clave en el preprocesamiento de imágenes que servirán como input para las redes neuronales.

Para el modelo usaremos una red neuronal convolucional llamada **ResNet-50**. Keras nos permite importarla mediante una sola función, que descargará la red neuronal desde un servidor remoto y la cacheará localmente:

```
model = tf.keras.applications.resnet50.ResNet50()
```

Fragmento de código 4.3: Definir arquitectura de red

Ahora definimos la función completa, donde, entre otras cosas, normalizaremos la imagen:

```
def predict(img_path):
    img = image.load_img(img_path, target_size=(224, 224))
    model = tf.keras.applications.resnet50.ResNet50()
    img_array = image.img_to_array(img)
    img_batch = np.expand_dims(img_array, axis=0)
    img_preprocessed = preprocess_input(img_batch)
    prediction = model.predict(img_preprocessed)
    print(decode_predictions(prediction, top=3)[0])
```

Fragmento de código 4.4: Definir función de predicción

Por último lo ejecutamos con la imagen 4.3a:

```
    predict(IMG_PATH)
```

Fragmento de código 4.5: Ejecutar función de predicción

Obtenemos los siguientes resultados:

```
[('n02123045', 'tabby', 0.5681726),
('n02124075', 'Egyptian_cat', 0.22223175),
('n02123159', 'tiger_cat', 0.061033934)]
```

Cada número que aparece en tercera posición es la probabilidad de que corresponda a la categoría que aparece en segunda posición. La red ResNet-50 fue entrenada para clasificar imágenes con una granularidad muy fina, por eso llega hasta el punto de clasificar por razas de gatos. Con la otra foto de ejemplo del libro (imagen 4.3b) obtenemos:

```
[('n02113186', 'Cardigan', 0.83495283),
('n02113023', 'Pembroke', 0.1554946),
('n02110806', 'basenji', 0.0027703578)]
```

Podemos probar con otras fotos que queramos, no tiene por qué ser de perros o gatos, ya que ResNet-50 clasifica cualquier cosa, por ejemplo una foto de nuestro ratón (imagen 4.3c):

```
[('n03793489', 'mouse', 0.9860136),
('n03777754', 'modem', 0.0027862787),
('n03602883', 'joystick', 0.0009623466)]
```

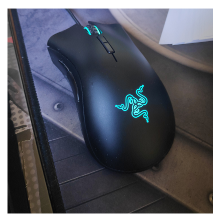
O a una taza (imagen 4.3d):

```
[('n03063599', 'coffee_mug', 0.9408508),
('n07930864', 'cup', 0.02910366),
('n04579145', 'whiskey_jug', 0.018378533)]
```

Como podemos observar, no hace falta que las fotos tengan una calidad profesional, sino que una simple foto sacada con la cámara de nuestro teléfono móvil ya funciona perfectamente, gracias a que la red ResNet-50 está muy bien entrenada.

(a) Gato

(b) Perro



(c) Ratón

(d) Taza

Figura 4.3: Imágenes usadas en la sección 2.2

Sobre ImageNet, Kaggle y los zoos de modelos

ImageNet es una base de datos online de imágenes usada para la investigación en visión artificial. Está formada por más de 14 millones de imágenes de 20000 categorías diferentes. Todos los años se celebra una competición llamada *ImageNet Large Scale Visual Recognition Challenge* (ILSVRC) para ver quién hace la mejor red neuronal para el conjunto de ImageNet, en la que se han mostrado a lo largo de los años algunos de los avances recientes más importantes en el campo de las redes neuronales.

Kaggle es una comunidad de profesionales de la inteligencia artificial en la que se celebran también competiciones de redes neuronales. Es muy interesante para ver ejemplos de datasets y modelos que la gente ha creado. A lo largo del trabajo tomaremos algunos de datasets colgados en Kaggle para realizar algunos ejemplos.

Los zoos de modelos son lugares donde organizaciones o usuarios individuales pueden publicar y subir sus modelos para compartirlos con la comunidad. Estos modelos pueden ser entrenados usando cualquier framework(Keras, TensorFlow, PyTorch, etc.), para cualquier tarea (clasificación, detección, etc.) o entrenados con cualquier conjunto de datos(ImageNet, Street View House Numbers, etc.). Un ejemplo es el zoo de modelos de Keras, que contiene varias redes neuronales

Figura 4.4: Mapa de calor de la foto de la taza

que han sido entrenadas con el conjunto de datos ImageNet, entre ellas, la que hemos estado usando: ResNet-50.

Mapas de activación de clase

La *image saliency*, es la técnica que estudia a qué parte de una imagen mira más la gente. Es una parte importante en la investigación en UX (User Experience), ya que permite mejorar las interfaces gráficas en el software o mejorar las campañas de márketing en diferentes aspectos. En los seres humanos esto se realiza mediante estudios de eye-tracking y mapas de calor. Sería interesante poder saber en qué partes de una imagen se fija una red neuronal. Para este experimento seguiremos el segundo notebook del capítulo 2 de [2]. Debemos instalar antes dos cosas más:

```
conda install tensorflow-gpu
pip install tf-explain==0.1.0
```

Para obtener el **mapa de calor** de una foto nos servimos del programa `visualization.py`, disponible en el GitHub del libro, y ejecutamos el siguiente comando:

```
!python visualization.py --process image --path ../../sample-images/cup.jpg
```

Fragmento de código 4.6: Obtener mapa de calor en Keras y TensorFlow para una imagen

El resultado se puede ver en la figura 4.4 Estas técnicas se pueden aplicar a un vídeo entero. Primero habría que dividir el vídeo en frames con algún programa, como FFmpeg, después se aplica el programa `visualization.py` al conjunto de imágenes, y por último, los frames con mapas de calor que se han generado se juntan con FFmpeg:

```
!ffmpeg -i nombreVideoInput.mp4 -vf fps=25 data/carpetaFrames/thumb\%04d.jpg -hide_banner
!python visualization.py --process video --path data/carpetaFrames/
!ffmpeg -framerate 25 -i data/nombreVideoOutput/result-\%04d.jpg nombreVideoOutput.mp4
```

Fragmento de código 4.7: Obtener mapa de calor en Keras y TensorFlow para un vídeo

El resultado es el vídeo original en una mitad de la pantalla y en la otra el vídeo con el mapa de calor en cada frame. Es una técnica útil para visualizar en qué se fija el modelo cuando analiza las imágenes.

4.2.2. Transfer learning con Keras

El aprendizaje de transferencia o ***transfer learning*** permite reutilizar modelos ya entrenados y adaptarlos a nuestro problema, ahorrándonos una gran cantidad de tiempo y dinero. En deep learning para visión artificial, estos modelos son normalmente redes neuronales convolucionales (CNNs). En estas redes, la complejidad de lo que una capa puede reconocer aumenta a medida que nos acercamos a la capa de salida, y a la vez, la reusabilidad de la capa disminuye.

Aprendizaje de transferencia o transfer learning

Si queremos transferir conocimiento de un modelo a otro, debemos reutilizar más capas *genéricas* cercanas al input y menos capas *específicas* cercanas al output. Es decir, eliminaremos algunas de las últimas capas y añadiremos nuevas que encajen con la nueva tarea que queremos realizar. Una vez que empecemos a entrenar la red, las capas genéricas se *congelan*, y solo se permite modificar las nuevas capas añadidas.

Si queremos aumentar la precisión de nuestro modelo, podemos darle permiso para que modifique algunas de las capas que estaban congeladas. Es lo que se conoce como *fine tuning*.

Construcción de un clasificador de imágenes con Keras y transfer learning

Esta tarea la vamos a realizar en varios pasos que se detallan a continuación siguiendo el notebook del capítulo 3 de [2].

Organizar los datos

Un conjunto de datos para una red neuronal se divide, típicamente, en 3 partes:

1. **Entrenamiento**: Es el conjunto de mayor tamaño, ya que servirá para que la red aprenda y reajuste sus parámetros en función de los resultados de la función de coste y de métrica obtenidos con estos datos tras cada epoch.

2. **Validación**: Es un conjunto de menor tamaño que el de entrenamiento y sirve para medir la eficacia de la red neuronal después de haber aprendido con el conjunto de entrenamiento. La proporción en la que se divide un dataset es típicamente 80 % para entrenamiento y 20 % para validación. Cuando la red está recibiendo datos de validación no debe modificarse de ninguna forma, solo debe medir los resultados de la función de coste y de la métrica en cada epoch.

3. **Test**: Es un conjunto extra que se usa para medir la eficacia de la red cuando ya se ha dejado de entrenar para siempre a la red neuronal.

Para nuestro ejemplo usaremos un conjunto de datos que debemos descargarnos de Kaggle.com. El conjunto de datos de entrenamiento contiene 25000 imágenes de perros y gatos, y el de testeo 12500. A continuación, creamos una carpeta `data`, en la cual crearemos dos carpetas

`train` y `val`, y dentro cada una de estas, dos carpetas `cat` y `dog`. Una vez hecho esto, escogemos aleatoriamente 250 fotos de cada clase introduciendo los siguientes comandos: (Estos comandos son comandos Unix; en un sistema Windows habría que, o bien usar los equivalentes en este sistema operativo, o bien usar el WSL, lo cual es más recomendable si se está trabajando en Windows.)

```
ls | grep cat | sort -R | head -250 | xargs -I {} mv {} train/cat/
ls | grep dog | sort -R | head -250 | xargs -I {} mv {} train/dog/
ls | grep cat | sort -R | head -250 | xargs -I {} mv {} val/cat/
ls | grep dog | sort -R | head -250 | xargs -I {} mv {} val/dog/
```

Esto es una forma “manual” de dividir el conjunto de datos. En capítulos posteriores usaremos formas de hacerlo mucho más “automáticas” proporcionadas por los frameworks para no perder tiempo en esta tarea.

Construir la tubería de datos

Mediante los siguientes comandos prepararemos los paquetes necesarios para el escenario:

```
1 import tensorflow as tf
2 from tensorflow.keras.preprocessing.image import ImageDataGenerator
3 from tensorflow.keras.models import Model
4 from tensorflow.keras.layers import Input, Flatten, Dense, Dropout,
    GlobalAveragePooling2D
5 from tensorflow.keras.applications.mobilenet import MobileNet,
    preprocess_input
6 import math
```

Fragmento de código 4.8: Importar paquetes de Keras y TensorFlow necesarios

A continuación indicamos que queremos que se ejecute en la GPU todo lo que sea posible.

```
1 physical_devices = tensorflow.config.list_physical_devices('GPU')
2 tensorflow.config.experimental.set_memory_growth(physical_devices[0], True)
```

Fragmento de código 4.9: Indicar que se ejecute en la GPU

Después hay que definir varios parámetros necesarios:

```
1 TRAIN_DATA_DIR = 'train/'
2 VALIDATION_DATA_DIR = 'val/'
3 TRAIN_SAMPLES = 500
4 VALIDATION_SAMPLES = 500
5 NUM_CLASSES = 2
6 IMG_WIDTH, IMG_HEIGHT = 224, 224
7 BATCH_SIZE = 64
```

Fragmento de código 4.10: Definir parámetros necesarios

`NUM_CLASSES` es el número de clases entre las que distinguirá nuestro modelo. Con 2 clases puede haber 2 enfoques:

1. **Clasificación binaria**: Para cada imagen dada, el modelo dará una sola probabilidad, que corresponderá en nuestro caso a la clase *cat*. Por lo tanto, la probabilidad de *dog* será $1 - P(cat)$.

2. **Clasificación multiclase**: El modelo da probabilidades separadas para cada clase, y la que tenga probabilidad más alta gana. Este enfoque es mucho más útil para problemas del mundo real y además hace que el modelo se pueda reutilizar en otros problemas, así que es el tipo de clasificación que usaremos.

`BATCH_SIZE` es el número de imágenes que el modelo puede analizar al mismo tiempo. Si usamos una CPU solo podremos usar batch de tamaño 1, porque no puede procesar más de una imagen en paralelo, pero si estamos usando una GPU para entrenar, aprovecharemos la capacidad de paralelización de esta metiendo muchas imágenes a la vez. En este caso, se ha usado un batch de 64 imágenes. Según la GPU que se esté usando, puede que haya que reducir este número hasta que el batch quepa en la memoria de la GPU.

Aumentar el tamaño del conjunto de datos

También hemos hablado anteriormente del sobreajuste y sus efectos, y de que es la razón por la que hay que aplicar distintas técnicas para mitigarlo. En este ejercicio vamos a usar la técnica del aumento del tamaño del dataset. Keras proporciona la función `ImageDataGenerator` para realizar esto de forma sencilla:

```
train_datagen = ImageDataGenerator(preprocessing_function=preprocess_input,
                                   rotation_range=20,
                                   width_shift_range=0.2,
                                   height_shift_range=0.2,
                                   zoom_range=0.2)
val_datagen = ImageDataGenerator(preprocessing_function=preprocess_input)
```

Fragmento de código 4.11: Definir transformaciones de los datasets en Keras

Solo se debe ampliar el conjunto de entrenamiento, nunca el conjunto de validación. Después se agrupan las imágenes en batch como se explica en el notebook para aumentar la eficiencia de nuestro entrenamiento, y que la red no esté analizando solo una imagen a la vez.

```
train_generator = train_datagen.flow_from_directory(TRAIN_DATA_DIR,
                                                    target_size=(IMG_WIDTH,
                                                                 IMG_HEIGHT),
                                                    batch_size=BATCH_SIZE,
                                                    shuffle=True,
                                                    seed=12345,
                                                    class_mode='categorical')
validation_generator = val_datagen.flow_from_directory(
                                    VALIDATION_DATA_DIR,
                                    target_size=(IMG_WIDTH, IMG_HEIGHT),
                                    batch_size=BATCH_SIZE,
                                    shuffle=False,
                                    class_mode='categorical')
```

Fragmento de código 4.12: Definir datasets en Keras

Definir el modelo

Vamos a reusar una CNN que fue entrenada con ImageNet, quitar las últimas capas y reemplazarlas con nuestras propias capas especialmente diseñadas para esta tarea.

```python
def model_maker():
    base_model = MobileNet(include_top=False,
                input_shape=(IMG_WIDTH, IMG_HEIGHT, 3))
    for layer in base_model.layers[:]:
        layer.trainable = False
    input = Input(shape=(IMG_WIDTH, IMG_HEIGHT, 3))
    custom_model = base_model(input)
    custom_model = GlobalAveragePooling2D()(custom_model)
    custom_model = Dense(64, activation='relu')(custom_model)
    custom_model = Dropout(0.5)(custom_model)
    predictions = Dense(NUM_CLASSES, activation='softmax')(custom_model)
    return Model(inputs=input, outputs=predictions)
```

Fragmento de código 4.13: Definir modelo en Keras

Entrenar el modelo

En transfer learning, entrenar el modelo es equivalente a “adaptar el modelo a los datos”. Primero debemos establecer varios parámetros:

1. La función de coste
2. El optimizador : Algoritmo que sirve para ayudar a minimizar la función de coste. (Backpropagation)
3. El ratio de aprendizaje
4. La métrica: Que juzgará el rendimiento de nuestro modelo. La precisión es una buena métrica para este ejemplo.

Ahora ejecutamos el siguiente trozo de código para entrenar el modelo. Realizará 10 epochs que tardarán segundos o minutos cada uno, dependiendo de las capacidades de la máquina:

```python
model = model_maker()
model.compile(loss='categorical_crossentropy',
          optimizer=tf.keras.optimizers.Adam(0.001),
          metrics=['acc'])
model.fit_generator(
    train_generator,
    steps_per_epoch=math.ceil(float(TRAIN_SAMPLES) / BATCH_SIZE),
    epochs=10,
    validation_data=validation_generator,
    validation_steps=math.ceil(float(VALIDATION_SAMPLES) / BATCH_SIZE))
```

Fragmento de código 4.14: Entrenar el modelo en Keras

Cada epoch tarda una media de 12 segundos para ejecutarse, y llegamos a obtener una precisión de 98 %. Ahora hay que guardar nuestro modelo entrenado con el nombre que queramos:

```python
model.save('model.h5')
```

Fragmento de código 4.15: Guardar el modelo en Keras

Una vez hecho esto, podemos cargar el modelo para probarlo con las imágenes del capítulo anterior:

```
model = load_model('model.h5')
img_path = '../../sample-images/dog.jpg'
img = image.load_img(img_path, target_size=(224, 224))
img_array = image.img_to_array(img)
expanded_img_array = np.expand_dims(img_array, axis=0)
preprocessed_img = expanded_img_array / 255.
prediction = model.predict(preprocessed_img)
print(prediction)
print(validation_generator.class_indices)
```

Fragmento de código 4.16: Cargar y probar el modelo en Keras

Y ,efectivamente, nos da el resultado correcto:

```
[[0.00828362 0.99171644]]
{'cat': 0, 'dog': 1}
```

Esto es todo el código necesario para entrenar un clasificador. Si nuestro problema tuviera más de dos clases sería recomendable usar `categorical_crossentropy` como función de coste y `softmax` como función de activación de la última capa.

Analizar los resultados

El código sobre el que se basa esta sección está en el segundo notebook del capítulo 3 de [2], realizando un análisis en detalle de los resultados del modelo. Primero creamos un diccionario que almacene el índice y el *ground truth*, es decir, la categoría a la que corresponde de verdad, para cada imagen. A continuación haremos que nos muestre 10 imágenes de cada una de las siguientes categorías:

- Imágenes con la probabilidad más alta de contener perros: Salen imágenes que contienen varios perros vistos de forma no ambigua.

- Imágenes con la probabilidad más baja de contener perros: Estas son imágenes que la red ha clasificado como perros con una probabilidad muy cercana al 0.5. Son imágenes bastante ambiguas y hay algunas que las ha clasificado mal. Una forma de mejorar la red sería entrenar con un conjunto de datos mayor. Y si queremos evitar que clasifique las imágenes dudosas, podríamos poner un umbral de 0.75 en vez de 0.5, con las imágenes que estuvieran por debajo de ese umbral, ya decidiremos lo que haremos.

- Imágenes de gatos con la probabilidad más alta de contener perros: Esto se refiere a las imágenes en cuyo nombre pone cat, pero que la red los clasifica como perros. Resulta que algunas de estas imágenes contienen perros y gatos a la vez, así que no es culpa del modelo sino de que los datos no son del todo correctos. Esto suele pasar en conjuntos de datos muy grandes.

- Imágenes con la probabilidad más alta de contener gatos: Al igual que con los perros, son imágenes muy claras que contienen varios gatos.

- Imágenes con la probabilidad más baja de contener gatos: Al igual que con los perros, son imágenes ambiguas y en algunas se equivoca la red.

- Imágenes de perros con la probabilidad más alta de contener gatos: Al igual que con los perros, son imágenes que contienen tanto perros como gatos y, por lo tanto, no es culpa de la red.

Podríamos concluir que los fallos en las predicciones han sido causados por varios motivos: poca iluminación, dificultad para distinguir el fondo, falta de textura, objeto que queremos analizar muy pequeño en comparación con el tamaño de la imagen. En resumen. imágenes ambiguas, pero que pueden suceder en situaciones del mundo real, por eso, es una buena práctica entrenar a la red con imágenes que sean similares a las que se va a encontrar en el problema real. Si solo le entrenamos con imágenes que estén perfectamente claras, a la hora de la verdad, su rendimiento decaerá.

4.2.3. Construyendo un motor de búsqueda de imágenes inverso

La búsqueda inversa de imágenes (*reverse image searching* o *instance retrieval*) es una técnica que permite a los desarrolladores construir aplicaciones que no se restrinjan a la búsqueda o comparación de palabras clave. Los buscadores de Google o Bing implementan esta técnica y permiten, a partir de una imagen, buscar las páginas web en las que aparezca.

Esta técnica se puede implementar mediante varios enfoques. El más simple sería comparar imágenes o partes de imágenes pixel por pixel y ver si los valores coinciden. Esto puede ser útil para encontrar imágenes exactamente o casi iguales, pero una pequeña rotación en la imagen haría que todos los pixeles cambiaran. Otra forma es crear un histograma de los valores RGB de la imagen y compararla con otras. Esta técnica se usa para encontrar imágenes duplicadas almacenadas cuando, por ejemplo, se hace una ráfaga de fotografías y más tarde se quiere elegir una de ellas. La desventaja es que si se aplican filtros de color, los histogramas de RGB cambian completamente.

Un enfoque más robusto basado en visión por computadora consiste en encontrar características visuales cerca de los bordes mediante algoritmos como SIFT[18] y SURF[19] y comparar el número de características similares entre las fotos. Se pasa de una comprensión a nivel de imagen a una comprensión a nivel de objeto. Funciona bien con objetos rígidos que siempre tienen la misma posición pero empieza a fallar con persona o animales, que son deformables. Otro enfoque es usar deep learning para hallar la categoría de un objeto para después encontrar imágenes con la misma categoría.

Pero nuestro objetivo es buscar entre millones de imágenes, por lo tanto, necesitamos una técnica que resuma la información de una imagen formada por miles o millones de píxeles. Eso es justo lo que las redes neuronales convolucionales son capaces de hacer; toman una imagen como entrada y la convierten en un vector de características (este a su vez sirve como entrada de las siguientes capas de la red para clasificar la imagen). Este vector de características es una colección de miles de números de punto flotante. Se puede entrenar a la red para que moldee estos valores de forma las imágenes que pertenecen a la misma categoría tengan una distancia euclídea pequeña entre ellos y una distancia grande si son completamente diferentes.

Extracción de características

Seguiremos el primer notebook del capítulo 4 de [2]. Primero vamos a descargar el conjunto de datos, que será el Caltech101, que contiene 9000 imágenes de 101 categorías y ocupa unos 131MB. A continuación, debemos importar las bibliotecas que necesitemos. Se ha creado una función para elegir entre varios modelos preentrenados, a los que se les quita las capas superiores. Vamos a elegir `ResNet50`, y vamos a hacer una función que cargue una foto, la redimensione para que la red lo acepte, extraiga sus características y las normalice:

```
def extract_features(img_path, model):
    input_shape = (224, 224, 3)
    img = image.load_img(img_path,
                         target_size=(input_shape[0], input_shape[1]))
    img_array = image.img_to_array(img)
    expanded_img_array = np.expand_dims(img_array, axis=0)
    preprocessed_img = preprocess_input(expanded_img_array)
    features = model.predict(preprocessed_img)
    flattened_features = features.flatten()
    normalized_features = flattened_features / norm(flattened_features)
    return normalized_features
```

Fragmento de código 4.17: Definir función de extracción de características

Podemos probar a ejecutar esta función con cualquier imagen y mostrar el número de características que obtiene y el tiempo que tarda:

```
%timeit features = extract_features('../../sample-images/cat.jpg', model)
print(len(features))
```

Fragmento de código 4.18: Ejecutar función de extracción de características

Ahora debemos aplicar esta función al conjunto de datos entero. Para ello, nos va a ser útil definir una función `get_file_list(root_dir)` que obtenga la lista de imágenes de un directorio para después, mediante un bucle, extraer las características de cada imagen de la lista:

```
root_dir = '../../datasets/caltech101'
filenames = sorted(get_file_list(root_dir))
feature_list = []
for i in tqdm_notebook(range(len(filenames))):
    feature_list.append(extract_features(filenames[i], model))
```

Fragmento de código 4.19: Extraer características de las imágenes de un directorio

Esta tarea puede ser muy costosa en función del hardware disponible, ya que en una CPU va a tardar 1 hora pero una GPU solo unos pocos minutos. Por último, debemos guardar estos datos en un fichero *pickle* para no tener que recalcularlos:

```
pickle.dump(feature_list, open('../../datasets/features-caltech101-resnet.
    pickle','wb'))
pickle.dump(filenames, open('../../datasets/filenames-caltech101.pickle','wb')
    )
```

Fragmento de código 4.20: Guardar lista de características extraídas

Ahora vamos a realizar la misma tarea pero utilizando las funciones de Image Generator de Keras, que se ejecutan en unos pocos segundos con una GPU y guardaremos los siguientes datos:

```
pickle.dump(generator.classes, open('./data/class_ids-caltech101.pickle','wb')
    )
pickle.dump(filenames, open('./data/filenames-caltech101.pickle', 'wb'))
pickle.dump(
    feature_list,
    open('./data/features-caltech101-' + model_architecture + '.pickle', 'wb')
    )
```

Fragmento de código 4.21: Guardar lista de características extraídas mediante Image Generator

Por último, vamos a entrenar un modelo nosotros mismos que realice la misma función que los dos anteriores. Aquí se tardará bastante más en ejecutar, ya que hay que realizar como mínimo 10 epochs para que el modelo sea fiable:

```
model_finetuned = model_maker()
model_finetuned.compile(loss='categorical_crossentropy',
              optimizer=tensorflow.keras.optimizers.Adam(0.001),
              metrics=['acc'])
model_finetuned.fit_generator(
    train_generator,
    steps_per_epoch=math.ceil(float(TRAIN_SAMPLES) / batch_size),
    epochs=10)
```

Fragmento de código 4.22: Entrenar modelo de extracción de características

Cuando ya esté listo, lo guardamos también.

Búsqueda de semejanzas

En este apartado seguiremos el segundo notebook del capítulo 3. Lo primero que debemos hacer es cargar los ficheros pickle que guardamos anteriormente:

```
filenames = pickle.load(open('./data/filenames-caltech101.pickle', 'rb'))
feature_list = pickle.load(open('data/features-caltech101-resnet.pickle', 'rb'
    ))
class_ids = pickle.load(open('./data/class_ids-caltech101.pickle', 'rb'))
```

Fragmento de código 4.23: Cargar características extraídas anteriormente

A partir de una imagen cualquiera, podemos encontrar los *nearest neighbors* según sus características. Para ello, usaremos la biblioteca `scikit-learn`, en concreto, un modelo nearest-neighbor que utiliza un algoritmo de fuerza bruta para hallar el vecino más cercano según la distancia euclídea:

```
neighbors = NearestNeighbors(n_neighbors=5,
                    algorithm='brute',
                    metric='euclidean').fit(feature_list)
```

Fragmento de código 4.24: Hallar nearest neighbors

Después elegimos una imagen al azar y obtenemos la lista ordenada de vecinos más cercanos. La primera de la lista será ella misma, ya que tiene una distancia euclídea de 0. Como es un poco incómodo realizar esto para cada imagen, vamos a crear una función `plot_images` que obtenga la lista de los 4 vecinos más cercanos de una imagen, sin contarse a ella misma. Después, mediante un bucle, se llama a esta función el número de veces que queramos con índices de imágenes aleatorios.

Ahora vamos a realizar el experimento con el conjunto de datos entero. Usaremos el algoritmo t-SNE (t-distributed stochastic neighbor embedding), que reduce los vectores de características multidimensionales a 2D. Primero los mostraremos en un mapa de calor (figura 4.5) y después en un mapa con todas las fotos (figura 4.6). En este último es complicado distinguir algo ya que hay zonas muy densas en las que las imágenes son muy parecidas, así que vamos a crear la función `tsne_to_grid_plotter` que muestra una imagen representativa de cada zona de forma que queden espaciadas (figura 4.7).

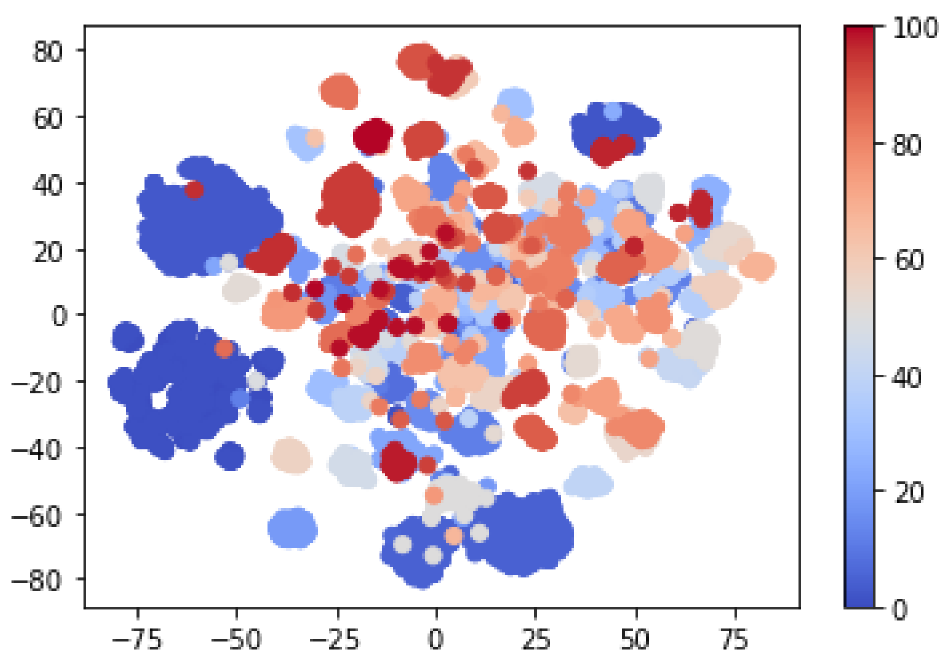


Figura 4.5: Mapa de calor generado por el t-SNE

Algoritmos para mejorar la velocidad de la búsqueda de semejanzas

Por último, vamos a seguir el tercer notebook del capítulo 4 del libro. Para mejorar la velocidad de la búsqueda de semejanzas entre imágenes podemos seguir dos estrategias: reducir el número de características o usar un algoritmo de búsqueda mejor.

Para reducir el número de características utilizaremos una técnica conocida como PCA (Principal Component Analysis). PCA es un procedimiento estadístico que cuestiona si todas las características son igual de importantes. No elimina características, sino que crea un nuevo conjunto de características que son combinación lineal de las originales. Estas nuevas características son ortogonales entre sí y, por lo tanto, no son redundantes. Normalmente elegiremos que reduzca a 100 características, porque a partir de este valor ya no hay mucha diferencia en cuanto a importancia pero cuantas más características añadamos, más lento va a ir. PCA puede decirnos cómo de importante es cada característica, y gracias a ello, podemos encontrar el equilibrio que deseemos entre número de características y rendimiento.

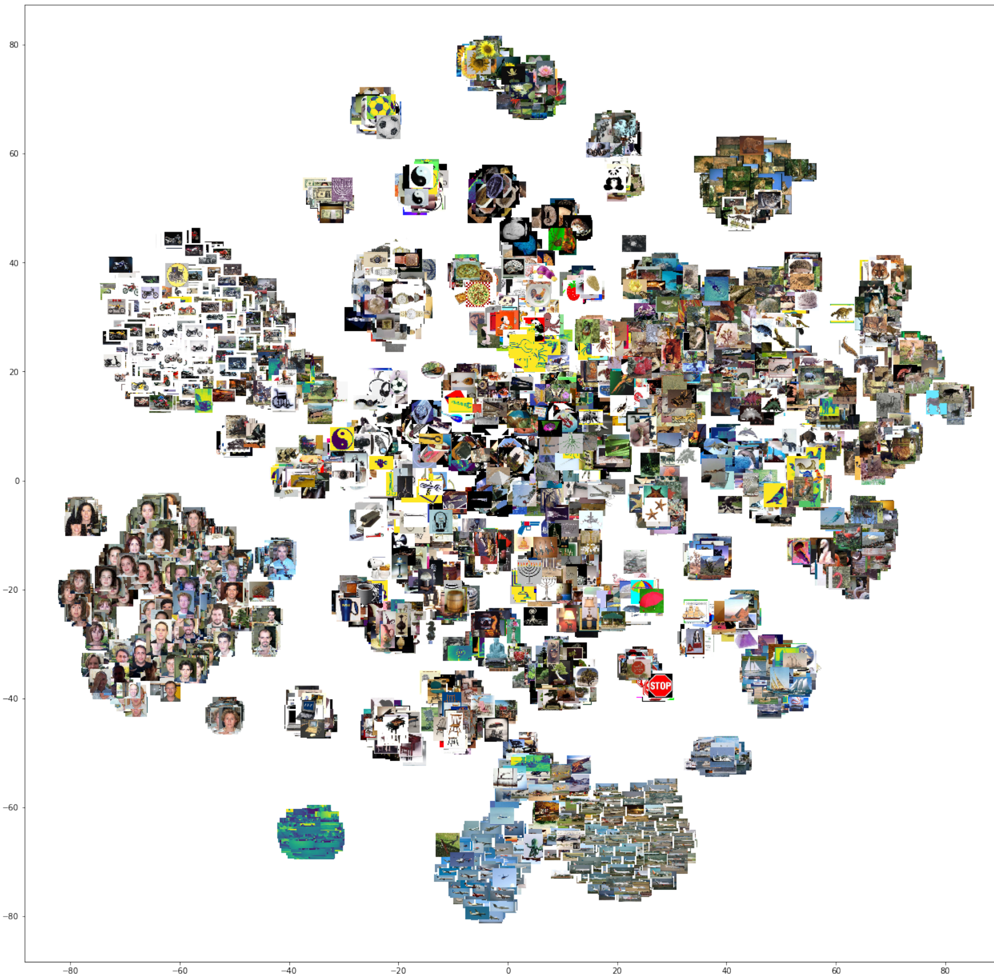


Figura 4.6: Mapa de fotos generado por el t-SNE

Una vez que hemos aplicado PCA, es importante elegir el algoritmo *Nearest Neightbor* que mejor rendimiento nos proporcione en nuestro problema. Para hallarlo es necesario realizar *benchmarks* para cada uno y compararlos. Algunos de los más usados son:

- **Fuerza bruta**: Es el algoritmo básico y el más lento. Compara cada elemento con todos los demás para hallar la distancia mínima.

- **k-d Tree**: Este algoritmo ordena los elementos en un árbol k-d [20] y después busca los elementos del árbol más cercano al elemento dado como entrada. Es más eficiente porque elimina muy rápidamente grandes trozos del espacio de búsqueda.

- **Ball Tree**: Este algoritmo ordena los elementos en un ball tree [21] y después aplica una búsqueda aprovechando las características de esta estructura de datos, similar al anterior.

El problema del Nearest Neighbor es muy recurrente y surge en multitud de contextos. Por ello, aparte de los algoritmos básicos, existen varias bibliotecas creadas por grandes organizaciones y empresas que tienen versiones muy bien optimizadas de estos algoritmos. La mayoría se programaron originalmente en C++ con un *wrapper* para Python. Algunos de los más destacados son:

- **Annoy**[22]: Approximate Nearest Neighbors Oh Yeah es una biblioteca desarrollada por Spotify y es la que usa para realizar las recomendaciones a los usuarios de la aplicación.

Figura 4.7: Mapa de fotos generado por la función tsne_to_grid_plotter

- **NGT**[23]: Neighborhood Graph and Tree fue desarrollada por Yahoo Japón y es especialmente adecuada para grandes conjunto de datos(millones) con muchas dimensiones(miles).

- **Faiss**[24]: Este algoritmo, creado por Facebook, es el más eficiente cuando se tienen conjuntos de datos de miles de millones de elementos.

- **Falconn**[25]: FAst Lookups of Cosine and Other Nearest Neighbors fue desarrollado por el MIT y está basado en el LSH [26] (Locality-Sensitive Hashing).

4.3. FastAI & PyTorch

En este capítulo se realizarán algunos ejemplos prácticos de redes neuronales tomados del libro *Deep Learning for Coders with Fastai & PyTorch, AI Applications Without a PhD* de Jeremy Howard & Sylvain Gugger, [2]. Este libro está íntegramente publicado en forma de Jupyter Notebooks en su repositorio de Github. Todos los ejemplos de este capítulo están hechos con el framework **Fastai**, que a su vez necesita el framework **PyTorch**. También se introducirán algunos conceptos nuevos usados en los ejemplos que no se hayan explicado anteriormente.

4.3.1. Introducción a fastai

Comenzaremos realizando el ejemplo básico del notebook del primer capítulo del libro [3] para conocer los fundamentos básicos de fastai. Utlizando una red preentrenada vamos a clasificar imágenes de perros y gatos, al igual que hicimos con Keras al principio del capítulo anterior.

Lo primero que se debe hacer es descargar y descomprimir el conjunto de datos que se va a usar. En este caso será el The Oxford-IIIT Pet Dataset. Debemos definir una función booleana que devuelva true si el nombre de la imagen indica que es una gato y false en caso contrario, es decir, una función para etiquetar los datos. Esto es lo que servirá a la red para saber si se ha equivocado al clasificar:

```
def is_cat(x): return x[0].isupper()
```

Fragmento de código 4.25: Definir función de etiquetado en Fastai

Después se indica qué clase de conjunto de datos es y cómo está estructurado. Uno de los parámetros importantes es la función de etiquetado que se acaba de definir. A los datos se les puede aplicar transformaciones, fastai tiene muchas predefinidas y se pueden crear nuevas. En este caso son imágenes, por lo que se les puede aplicar una transformación para redimensionarlas mediante `item_tfms=Resize(NUMERO_PIXELES)`. Así todas son iguales y totalmente compatibles con el modelo preentrenado concreto que se quiera usar. Con `valid_pct` indicamos el porcentaje del conjunto de datos que queremos dedicar a la validación; el resto irá al entrenamiento. Así que creamos lo que se llama en Fastai como `DataLoaders` a partir de los parámetros anteriores:

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

Fragmento de código 4.26: Definir ImageDataLoaders

Si estamos trabajando en **Windows** debemos añadir siempre al crear un DataLoaders el siguiente parámetro: `num_workers=0`. O de lo contrario, saltará una excepción a la hora de entrenar la red más tarde. (Es un detalle que proviene de cómo está implementado PyTorch por debajo.)

Después, se crea la red neuronal propiamente dicha. Se le pasa el DataLoaders (en el libro suelen llamar a la variable de este tipo siempre `dls`), el modelo que va a usar, en este caso resnet34, y la métrica. La métrica es la función que mide la calidad de las predicciones del modelo en el conjunto de validación, y se imprimirá después de cada epoch. Por último, hay que indicar la forma en que tiene que entrenarse. Poner `fine_tune(1)` significa que el modelo usará las técnicas que él considere mejor para entrenar y entrenará durante 1 epoch.

```
learn = cnn_learner(dls, resnet34, metrics=error_rate)
learn.fine_tune(1)
```

Fragmento de código 4.27: Definir Learner y entrenar el modelo en Fastai

Ahora podemos probar nuestro modelo. Mediante el siguiente código se crea un botón en notebook que permite subir una imagen manualmente desde nuestro equipo:

```
uploader = widgets.FileUpload()
```

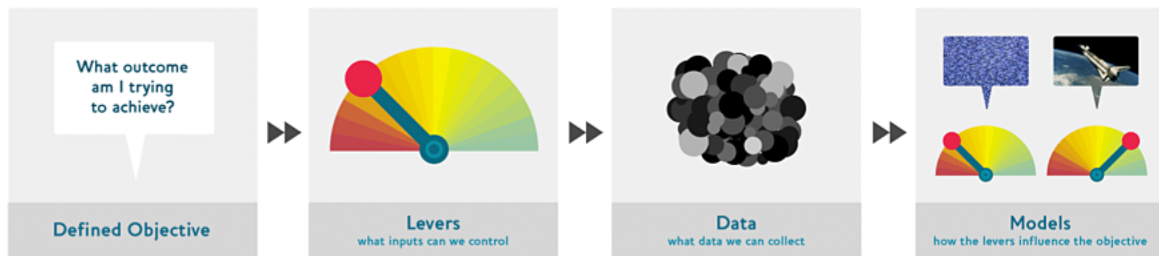


Figura 4.8: Enfoque Drivetrain de Jeremy Howard

```
2 uploader
```

Fragmento de código 4.28: Cargar imagen en Jupyter Notebook mediante botón

También se puede especificar una ruta:

```
1 uploader = SimpleNamespace(data = ['RUTA'])
```

Fragmento de código 4.29: Cargar imagen en Jupyter Notebook mediante ruta

Una vez subida la imagen, se le pasa al modelo para que realice su predicción:

```
1 img = PILImage.create(uploader.data[0])
2 is_cat,_,probs = learn.predict(img)
3 print(f"Is this a cat?: {is_cat}.")
4 print(f"Probability it's a cat: {probs[1].item():.6f}")
```

Fragmento de código 4.30: Realizar predicción de una imagen en Fastai

Esto sería lo necesario para crear un clasificador de imágenes básico mediante Fastai y utilizando una red preentrenada.

4.3.2. De modelo a producto

En esta sección vamos a implementar un clasificador de imágenes utilizando más técnicas que fastai ofrece siguiendo el notebook del segundo capítulo de [3].

Drivetrain approach

El enfoque Drivetrain es una metodología para resolver problemas de machine learning introducida por Jeremy Howard, uno de los creadores de fastai, en 2012. Consisten en identificar varias cosas antes de ponerse a codificar el modelo como tal.

1. Identificar un objetivo claro.

2. Identificar las acciones que podemos tomar para alcanzar el objetivo, es decir, los inputs que podemos controlar.

3. Identificar los datos que podemos recolectar para poder llevar a cabo las acciones.

Solo después de estos tres pasos, hay que empezar a pensar en construir el modelo.

Recolectar los datos

Una de las formas más rápidas para recolectar imágenes de un tema concreto, es usar APIs de búsqueda de imágenes como Bing Image Search API, de Microsoft. Normalmente, se requiere una suscripción para poder usar estos servicios, aunque se puede probar gratuitamente durante unos pocos días. En el libro usan la API de Bing para recolectar 150 imágenes de 3 tipos de osos diferentes que la red aprenderá a distinguir. Para evitar usar APIs de búsqueda, en esta sección vamos a usar un dataset de Kaggle llamado Rock-Paper-Scissors preparado por Julien de la Bruère-Terreault. Contiene 2000 imágenes de manos en posición de piedra, papel o tijeras. Por lo tanto, son imágenes de 3 categorías diferentes, similar al ejemplo original del libro.

Cargar los datos con Dataloaders

Fastai ofrece la clase DataLoaders, que puede almacenar varios DataLoader, normalmente el conjunto de de entrenamiento y el conjunto de validación. Para transformar los datos en un DataLoader, debemos especificar varias cosas:

1. El tipo de datos con el que estamos trabajando.

2. Cómo obtener la lista de items.

3. Cómo etiquetar los items.

4. Cómo crear el conjunto de validación.

Fastai ofrece varios métodos factoría predefinidos para crear los DataLoader. Pero si no hay ninguno que encaje con nuestro problema, podemos usar la clase DataBlock para definir el nuestro, que es el procedimiento que se usará principalmente a lo largo de este trabajo. Esta clase recibe varios parámetros:

```
1 hands = DataBlock(
2     blocks=(ImageBlock, CategoryBlock),
3     get_items=get_image_files,
4     splitter=RandomSplitter(valid_pct=0.2, seed=42),
5     get_y=parent_label,
6     item_tfms=Resize(128))
```

Fragmento de código 4.31: Crear DataBlock

En la segunda línea, le especificamos la variable independiente, es decir, lo que estamos usando para realizar las predicciones, en nuestro caso, `ImageBlock`, y la variable dependiente, que es nuestro objetivo, en nuestro caso, `CategoryBlock`. En la tercera línea especificamos cómo obtiene los ficheros, en este caso a partir de una ruta que recibirá más tarde. En la cuarta línea, dividimos el conjunto de datos en dos partes, el de entrenamiento y el de validación, asignando el 20 % al de validación, y de forma aleatoria, pero con una semilla fija. EN la quinta línea, indicamos cómo crear las etiquetas. En nuestros caso, hemos dividido en carpetas las imágenes, y el nombre de la carpeta es el nombre de la categoría. En la sexta línea le aplicamos transformaciones para redimensionar las imágenes.

Después usamos este DataBlock para crear el DataLoaders, especificando la ruta a los datos:

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

Fragmento de código 4.32: Crear DataLoaders

Se puede algún modificar parámetro del DataBlock, por ejemplo, los `Transforms`. Con el siguiente código estamos recortando la imagen de forma diferente en cada epoch y además aplicando técnicas de *data augmentation*:

```
hands = hands.new(
    item_tfms=RandomResizedCrop(224, min_scale=0.5),
    batch_tfms=aug_transforms())
dls = hands.dataloaders(path)
```

Fragmento de código 4.33: Aplicar transformaciones en Fastai

El entrenamiento de la red

Por último creamos el `Learner` y hacemos *fine-tuning* durante 4 epochs:

```
learn = cnn_learner(dls, resnet18, metrics=error_rate)
learn.fine_tune(4)
```

Fragmento de código 4.34: Definir Learner y entrenar

Los resultados son los de la tabla 4.1:

epoch	train_loss	valid_loss	error_rate	time
0	1.156238	0.190012	0.067039	00:08

epoch	train_loss	valid_loss	error_rate	time
0	0.178623	0.078071	0.016760	00:04
1	0.134914	0.068634	0.011173	00:03
2	0.109041	0.081353	0.011173	00:03
3	0.088000	0.077930	0.011173	00:03

Cuadro 4.1: Resultados del entrenamiento del dataset de piedra, papel o tijeras

Vamos a explicar los resultados que da la función `fine_tune` en la tabla 4.1. Esta función realiza siempre un epoch con solo la última capa de la red descongelada, es lo que se muestra en la primera fila de la tabla. Después, descongela el resto de capas y entrena durante el número de epochs que se le pasa como parámetro, como se muestra en las 4 últimas filas.

La columna `epoch` indica obviamente el epoch en que se está entrenando. La columna `train_loss` indica el valor de la función de coste en el conjunto de entrenamiento en ese epoch. De forma similar, la columna `valid_loss` indica el valor de la función de coste en el conjunto de validación en ese epoch. Cuanto más alto sea el valor, más se ha equivocado, lo ideal sería que fueran bajando los dos valores a medida que avanzan los epochs, pero siempre llegará un momento en el que se estancará y no pueda aprender más. La columna `erro_rate`, es la métrica que hemos elegido en este caso, e indica el porcentaje de error de la red. Como podemos observar, la red ha conseguido un porcentaje de error muy pequeño, aproximadamente un 1 %. Por último, la columna `time` indica el tiempo que ha transcurrido en cada epoch. Una vez que tenemos los resultados, podemos visualizar el número de errores de forma gráfica mediante:

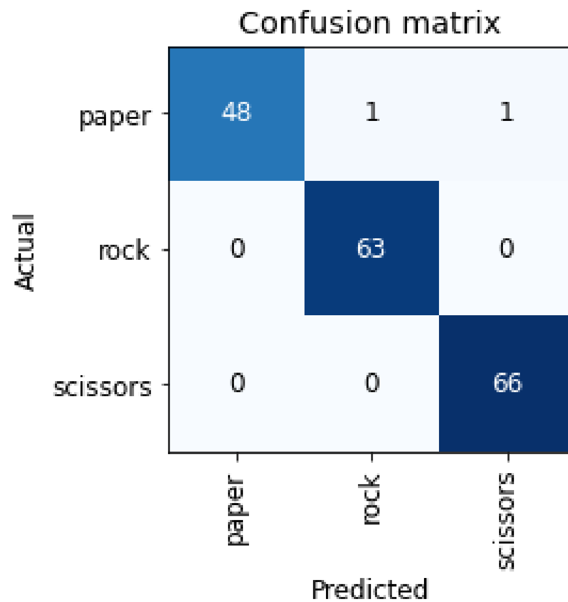


Figura 4.9: Matriz de confusión del dataset de piedra, papel o tijeras

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

Fragmento de código 4.35: Obtener matriz de confusión

Esto creará una matriz(figura 4.9) en la que en la diagonal está el número de aciertos y en el resto de casillas, los fallos por categoría.

Exportar el modelo

El modelo está formado por la arquitectura y los parámetros. Para guardar estas dos cosas después de entrenarlas se debe hacer los siguiente:

```
learn.export()
```

Fragmento de código 4.36: Exportar el modelo

Esto creará un fichero llamado `export.pkl`. Este fichero se puede cargar en otro Notebook diferente mediante:

```
learn = load_learner(path/'export.pkl')
```

Fragmento de código 4.37: Cargar el modelo

Ahora se le puede pasar una imagen concreta para que infiera su categoría:

```
learn.predict(path)
```

Fragmento de código 4.38: Predecir categoría de imagen

Mediante el paquete Voilà se podría transformar este modelo en una aplicación web de forma muy sencilla.

4.3.3. Clasificación de imágenes con fastai

En esta sección se van a ver algunas técnicas que ofrece fastai para mejorar los modelos siguiendo el notebook del capítulo 5 de [3]. Para mostrar estas técnicas se va usar un conjunto de

datos de 7000 imágenes de razas de perros de 37 categorías llamado The Oxford-IIIT Pet Dataset.

Creando el conjunto de datos

Los datos normalmente vienen dados de diferentes maneras:

- En ficheros individuales que representan los distintos items de datos, como imágenes o documentos de texto, que pueden estar organizados en directorios o con nombres que aporten información sobre ellos.
- En tablas de datos (como ficheros .csv), donde cada fila representa un item e incluye nombres de ficheros que conectan los datos de la tabla con datos en otros formatos.
- En formatos de bases de datos binarias o flujos de red. Esta categoría es muy poco común y se usa en dominios muy específicos como el análisis de genomas.

Una vez descargado el dataset, se crea el DataBlock y el DataLoaders:

```
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)
```

Fragmento de código 4.39: Crear DataBlock de dataset de mascotas y DataLoaders

En la línea 4, para definir las etiquetas, se ha usado una expresión regular, ya que las imágenes vienen todas juntas en la misma carpeta y no es posible usar la función `parent_label` como en la sección anterior. Las líneas 5 y 6 juntas realizan una estrategia de data augmentation única de fastai llamada *presizing*, que consiste en:

1. Redimensionar las imágenes a un tamaño relativamente grande, es decir, más grande que las dimensiones finales que tendrán las imágenes del conjunto de entrenamiento.
2. Componer todas las operaciones comunes de data augmentation en una sola y realizarla solo una vez al final del proceso en la GPU.

De esta forma, se consigue un aumento de la velocidad, al realizar todas las operaciones en la GPU a la vez; y un aumento en la precisión del modelo, ya que, así no hay partes en las imágenes que no aportan nada al modelo y no se pierde tanta calidad en las imágenes de entrenamiento.

Antes de entrenar el modelo, es recomendable comprobar que el conjunto de datos está correctamente etiquetado. En fastai se puede hacer mediante el método `show_batch`, especificando el número de imágenes organizados en filas y columnas:

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

Fragmento de código 4.40: Mostrar fragmento de batch

Otro método es `summary`, que intentará crear un batch a partir del dataset y mostrará toda la información relacionada posible, incluyendo errores si los hubiera. Por ejemplo, nos avisará si no hemos redimensionado las imágenes, de forma que tienen tamaños diferentes y no se puede meter el batch en la GPU.Cuando está todo correcto se puede comenzar el entrenamiento:

```
learn = cnn_learner(dls, resnet34, metrics=error_rate)
learn.fine_tune(2)
```

Fragmento de código 4.41: Entrenar el modelo con dataset de mascotas

Los resultados del entrenamiento se encuentran en la tabla 4.2:

epoch	train_loss	valid_loss	error_rate	time
0	1.540678	0.325591	0.104195	01:26

epoch	train_loss	valid_loss	error_rate	time
0	0.514307	0.304233	0.100135	01:08
1	0.345417	0.205364	0.062923	01:10

Cuadro 4.2: Resultados del entrenamiento del dataset de razas de perros

Interpretando el modelo

Para interpretar los datos se puede crear una matriz de confusión como en la sección anterior (Listing 4.35), donde se puede ver cuántas veces ha acertado y cuántas veces se ha equivocado y en qué categorías:

En este caso, como hay 37 categorías, crea una matriz de 37x37 (figura 4.10), que puede ser demasiado compleja para lo que se necesita saber. El método `most_confused` muestra las celdas con mayor número de equivocaciones a partir de un mínimo:

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

Fragmento de código 4.42: Obtener categorías con más equivocaciones

La salida de esta función es una lista con las categorías en las que se ha equivocado al menos 5 veces junto con el número de veces que se ha equivocado:

```
[('american_pit_bull_terrier', 'staffordshire_bull_terrier', 6),
('Birman', 'Ragdoll', 5),
('Russian_Blue', 'British_Shorthair', 5)]
```

Aplicando Transfer Learning

En clasificación de imágenes, la capa final de un modelo tiene una matriz con tantas columnas como el número de clases del modelo. Cuando hacemos transfer learning, esta capa final hay que quitarla y reemplazarla por una que encaje en nuestro modelo. Esta nueva capa final tendrá los parámetros inicializados aleatoriamente. La tarea a realizar es entrenar esta última capa sin afectar a las demás, que ya están perfectamente entrenadas para la clasificación de imágenes. Lo que se hace es congelar las capas preentrenadas.

Al crear un modelo a partir de una red preentrenada, por defecto, fastai congela todas las capas preentrenadas. Como se ha explicado anteriormente, el método `fine_tune` de fastai realiza 2 cosas:

1. Entrena las nuevas capas durante un epoch, manteniendo el resto congeladas.

Confusion matrix

Actual \ Predicted	Abyssinian	Bengal	Birman	Bombay	British_Shorthair	Egyptian_Mau	Maine_Coon	Persian	Ragdoll	Russian_Blue	Siamese	Sphynx	american_bulldog	american_pit_bull_terrier	basset_hound	beagle	boxer	chihuahua	english_cocker_spaniel	english_setter	german_shorthaired	great_pyrenees	havanese	japanese_chin	keeshond	leonberger	miniature_pinscher	newfoundland	pomeranian	pug	saint_bernard	samoyed	scottish_terrier	shiba_inu	staffordshire_bull_terrier	wheaten_terrier	yorkshire_terrier
Abyssinian	31	1	0	0	0	0	0	0	0	0	1	0	0	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
Bengal	1	33	0	0	0	6	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
Birman	0	0	35	0	0	0	0	0	2	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	1	0	0	0	0	0	0	0	0	0	0	0	0
Bombay	0	0	0	38	0	0	0	0	0	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
British_Shorthair	0	0	0	2	41	0	0	0	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
Egyptian_Mau	0	0	0	0	0	34	0	0	0	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
Maine_Coon	2	1	0	1	0	0	33	1	2	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
Persian	0	0	0	0	0	0	1	48	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
Ragdoll	0	0	5	0	0	0	0	1	28	0	0	0	0	0	0	0	0	0	0	0	0	0	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0
Russian_Blue	0	0	0	1	5	0	0	0	0	43	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
Siamese	0	0	2	0	0	0	0	0	0	0	51	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
Sphynx	1	0	0	0	0	0	0	0	0	0	0	30	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
american_bulldog	0	0	0	0	0	0	0	0	0	0	0	0	41	1	0	0	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	1	0	0
american_pit_bull_terrier	0	0	0	0	0	0	0	0	0	0	0	0	2	24	0	1	0	0	0	0	0	0	0	0	0	0	1	0	0	0	0	0	0	0	5	0	0
basset_hound	0	0	0	0	0	0	0	0	0	0	0	0	0	0	45	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
beagle	0	0	0	0	0	0	0	0	0	0	0	0	0	0	1	37	0	0	0	0	1	0	0	0	0	0	0	0	0	0	1	0	0	0	0	0	0
boxer	0	0	0	0	0	0	0	0	0	0	0	0	2	0	0	0	35	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
chihuahua	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	42	0	0	0	0	0	0	0	0	1	0	0	0	0	0	0	0	0	0	0
english_cocker_spaniel	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	42	0	0	0	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0
english_setter	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	47	0	0	0	0	0	0	0	0	0	0	1	0	0	0	0	0	0
german_shorthaired	0	0	0	0	0	1	0	0	0	0	0	0	0	1	0	0	0	0	0	0	38	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
great_pyrenees	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	31	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
havanese	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	1	0	0	0	30	0	0	0	0	0	0	0	0	0	0	0	0	1	0
japanese_chin	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	35	0	0	0	0	0	0	0	0	0	0	0	0	0
keeshond	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	38	1	0	0	0	0	0	0	0	0	0	0	0
leonberger	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	40	0	0	0	0	0	0	0	0	0	0	0
miniature_pinscher	0	0	0	0	0	0	0	0	0	0	0	1	0	2	0	0	0	1	0	0	0	0	0	0	0	0	39	0	0	0	0	0	0	0	1	0	0
newfoundland	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	39	0	0	0	0	0	0	0	0	0
pomeranian	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	1	0	0	0	0	0	0	0	1	0	0	40	0	0	0	0	0	0	0	0
pug	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	47	0	0	0	0	0	0	0
saint_bernard	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	43	0	0	0	0	0	0
samoyed	0	0	0	0	0	0	0	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	44	0	0	0	0	0
scottish_terrier	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	1	0	0	0	1	0	0	0	0	0	0	0	0	0	34	0	0	0	0
shiba_inu	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	1	0	32	0	0	0
staffordshire_bull_terrier	0	0	0	0	0	0	0	0	0	0	0	0	1	2	0	0	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	35	0	0
wheaten_terrier	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	1	0	0	0	0	0	0	0	0	0	0	0	0	36	0
yorkshire_terrier	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	3	0	0	0	0	0	0	0	0	0	1	0	0	1	26

Figura 4.10: Matriz de confusión del dataset de las razas de perros

2. Descongela todas las capas y las entrena durante el número de epochs especificado.

Este es un enfoque adoptado por los creadores de fastai que funciona razonablemente bien en la mayoría de casos, pero si se quiere decidir qué capas se descongelan y cuántos epochs se entrenan hay que usar otras funciones. En el siguiente fragmento de código se realiza un pequeño ejemplo:

```
learn = cnn_learner(dls, resnet34, metrics=error_rate)
learn.fit_one_cycle(3, 3e-3)
learn.unfreeze()
learn.fit_one_cycle(12, lr_max=slice(1e-6,1e-4))
```

Fragmento de código 4.43: Entrenar mediante fit_one_cycle

En la línea 1 se crea el Learner como de costumbre. En la línea 2 se usa el método `fit_one_cycle` para entrenar las nuevas capas durante 3 epochs con un ratio de aprendizaje concreto. En la línea 3, se descongelan todas las capas, de forma que en la línea 4, se indica que hay que entrenar la red entera durante 12 epochs. El segundo parámetro es un Slice de Python que indica el ratio de aprendizaje mínimo y máximo. Esto último es necesario porque las capas preentrenadas no necesitan un ratio de aprendizaje tan alto como las nuevas capas. Es lo que se conoce como ratio de aprendizaje discriminatorio (*discriminative learning rate*. El ratio mínimo se asignará a la capa inferior, el ratio máximo a la capa superior, y las capas intermedias irán adoptando ratios equidistantes entre sí en ese rango de valores.

4.3.4. Otros problemas de visión artificial

En esta sección vamos a ver 3 tipos de problemas típicos de visión artificial que será muy útil conocer a la hora de empezar a trabajar con las imágenes astronómicas en el capítulo 6 de este trabajo. Los 2 primeros problemas está detallados en el notebook del capítulo 6 de [3]. El tercero está en el tutorial online de Fastai: Tutorial de segmentación:

- **Clasificación multietiqueta**: Consiste en predecir más de una categoría para una imagen.
- **Regresión**: Consiste en predecir un número o un conjunto de número a partir de una imagen.
- **Segmentación**: Consiste en predecir una cateogría para cada píxel de una imagen.

Clasificación multietiqueta

Para este ejemplo se va a usar el conjunto de datos PASCAL_2007, que no está estructurado por nombre de los ficheros sino por un fichero .csv que indica que etiquetas corresponden a cada imagen. Se puede visualizar el contenido del fichero .csv con un `DataFrame` de la biblioteca Pandas:

```
df = pd.read_csv(path/'train.csv')
df.head()
```

Fragmento de código 4.44: Cargar CSV en DataFrame

Pandas es una biblioteca de Python muy utilizada para manipular datos tabulares y series temporales. Un `DataFrame` es una tabla formada por filas y columnas. Se puede crear un `DataFrame` a partir de un fichero .csv, de una tabla de una base de datos, de un diccionario de Python, etc.

Primero hay que crear el `DataBlock` y la forma de obtener las etiquetas:

```
def get_x(r): return path/'train'/r['fname']
def get_y(r): return r['labels'].split(' ')
dblock = DataBlock(blocks=(ImageBlock, MultiCategoryBlock),
                   get_x = get_x, get_y = get_y)
dsets = dblock.datasets(df)
dsets.train[0]
```

Fragmento de código 4.45: Definir funciones de etiquetado y DataBlock

Al crear el `DataBlock`, para el output, hay que usar un `MultiCategoryBlock`, que recibe una lista de strings, y codifica de la siguiente forma: Una cadena de 0s y 1s, donde un 1 indica que sí pertenece a esa categoría y un 0 en caso contrario. Esto se conoce como *one-hot encoding*. A continuación se podría crear una forma de dividir el conjunto de datos:

```
def splitter(df):
train = df.index[~df['is_valid']].tolist()
valid = df.index[df['is_valid']].tolist()
return train,valid
```

Fragmento de código 4.46: Definir splitter y datasets para clasificación multietiqueta

Se le asigna al atributo `splitter` del `DataBlock` y se le aplican las transformaciones que queramos:

```
dblock = DataBlock(blocks=(ImageBlock, MultiCategoryBlock),
                   splitter=splitter,
                   get_x=get_x,
                   get_y=get_y,
                   item_tfms = RandomResizedCrop(128, min_scale=0.35))
dls = dblock.dataloaders(df, num_workers=0)
```

Fragmento de código 4.47: Definir DataBlock y DataLoaders para clasificación multietiqueta

Como estamos haciendo clasificación multietiqueta, fastai elegirá por defecto como función de coste la entropía cruzada binaria (Binary Cross Entropy). Por último, como métrica, no podemos elegir la precisión a secas. Con una sola categoría, el modelo elegía la que tuviera mayor valor entre 0 y 1 (ya que hemos aplicado la función sigmoide). Con varias categorías, debemos definir un umbral(threshold), de forma que, las categorías cuyo valor esté por encima del umbral las trate como 1s, y las que estén por debajo como 0s. Esto se hace con una sencilla función que se le pasa al modelo después:

```
def accuracy_multi(inp, targ, thresh=0.5, sigmoid=True):
    if sigmoid: inp = inp.sigmoid()
    return ((inp>thresh)==targ.bool()).float().mean()

```

```
5 learn = cnn_learner(dls, resnet50, metrics=partial(accuracy_multi, thresh=0.2)
    )
6 learn.fine_tune(3, base_lr=3e-3, freeze_epochs=4)
```

Fragmento de código 4.48: Definir métrica de precisión para clasificación multietiqueta

Lo correcto sería probar con diferentes valores del umbral(El parámetro `thresh` de la función anterior) hasta encontrar el óptimo.

Regresión

La regresión consiste en que la red genere un número determinado de valores a partir de la entrada. Para ello, hay que alimentar a la red con unos inputs, que pueden ser imágenes, y con sus valores relacionados, para que cuando le demos una imagen, nos haga una predicción de los valores que corresponden a esa imagen.

Para este ejemplo se va a usar el conjunto de datos Biwi Kinect Head Pose, formado por 24 directorios. En cada directorio hay ficheros de imágenes .jpg y ficheros .txt con información sobre cada imagen. Estos ficheros de texto contienen las coordenadas del centro de la cabeza de la perosna que aparece en la imagen. Vamos a crear un *Key Point Model*. El *key point* se refiere a un lugar concreto de una imagen, en este caso el centro de la cabeza de la imagen. El objetivo es que el modelo genere las coordenadas que el cree que son el centro de la cabeza de la persona de la imagen.

Primero hay que crear la función que cambia los nombres de las imágenes para crear las etiquetas correctamente:

```
1 def get_ctr(f):
2     ctr = np.genfromtxt(img2pose(f), skip_header=3)
3     c1 = ctr[0] * cal[0][0]/ctr[2] + cal[0][2]
4     c2 = ctr[1] * cal[1][1]/ctr[2] + cal[1][2]
5     return tensor([c1,c2])
```

Fragmento de código 4.49: Definir función de etiquetado para regresión

Después se crea el `DataBlock`, con un tipo de bloque de salida llamado `PointBlock`, para que fastai sepa que las etiquetas representan coordenadas.

```
1 biwi = DataBlock(
2     blocks=(ImageBlock, PointBlock),
3     get_items=get_image_files,
4     get_y=get_ctr,
5     splitter=FuncSplitter(lambda o: o.parent.name=='13'),
6     batch_tfms=[*aug_transforms(size=(240,320)),
7                 Normalize.from_stats(*imagenet_stats)])
```

Fragmento de código 4.50: Definir DataBlock para regresión

Por último, creamos el `Learner` y el ratio de aprendizaje, lo entrenamos y mostramos algunos de los resultados comparándolos con el *ground truth*:

```
1 dls = biwi.dataloaders(path, num_workers=0)
2 learn = cnn_learner(dls, resnet18, y_range=(-1,1))
3 lr = 1e-2
4 learn.fine_tune(3, lr)
```

```
learn.show_results(ds_idx=1, nrows=3, figsize=(6,8))
```

Fragmento de código 4.51: Entrenar modelo para regresión

En este caso, lo más complicado ha sido encontrar la forma de etiquetar las imágenes y alimentar a la red con estos datos; el resto del procedimiento es el mismo de siempre.

Funciones de coste

Normalmente cada tipo de problema tiene una función de coste óptima para ese problema concreto:

- Clasificación unietiqueta: `nn.CrossEntropyLoss`
- Clasificación multietiqueta: `nn.BCEWithLogitsLoss`
- Regresión: `nn.MSELoss`

Segmentación

El código completo que se describe a continuación se encuentra en el apéndice A.1. Para este ejemplo vamos a usar el Camvid dataset, un conjunto de datos formado por imágenes de coches tomadas por cámaras de vigilancia. Primero descargamos el dataset:

```
path = untar_data(URLs.CAMVID_TINY)
```

Fragmento de código 4.52: Obtener dataset para segmentación

En el dataset que nos hemos descargado hay 3 elementos:

- El directorio `images`, que contiene las imágenes.
- El directorio `labels`, que contiene las máscaras de segmentación para cada imagen.
- El fichero `codes.txt`, que contiene la correspondencia entre entero y clase, ya que cada pixel de las máscaras tiene un valor determinado que corresponde a una de estas clases.

A continuación, cargamos el fichero `codes.txt` para definir la correspondencia entre ints y classes y definimos la función que etiquetará una imagen a partir de su máscara:

```
codes = np.loadtxt(path/'codes.txt', dtype=str)
def label_func(fn): return path/"labels"/f"{fn.stem}_P{fn.suffix}"
```

Fragmento de código 4.53: Obtener codes y etiquetas de segmentación

Después, creamos el DataBlock y el DataLoaders. En el DataBlock el bloque de output debe ser un MaskBlock que toma como parámetros las clases de pixeles que hay:

```
camvid = DataBlock(blocks=(ImageBlock, MaskBlock(codes)),
                   get_items = get_image_files,
                   get_y = label_func,
                   splitter=RandomSplitter(),
                   batch_tfms=aug_transforms(size=(120,160)))
dls = camvid.dataloaders(path/"images", path=path, bs=8, num_workers=0)
```

Fragmento de código 4.54: Crear DataBlock y DataLoaders para segmentación

Por último, creamos el learner y entrenamos. En segmentación se debe usar un `unet_learner`:

```
learn = unet_learner(dls, resnet34)
learn.fine_tune(6)
```

Fragmento de código 4.55: Crear y entrenar unet_learner

Podemos mostrar algunos resultados del entrenamiento. La imagen de la izquierda es la máscara real y la imagen de la derecha es la máscara predicha por la red.

```
learn.show_results(max_n=6, figsize=(7,8))
```

Fragmento de código 4.56: Mostrar resultados de segmentación

4.3.5. Técnicas avanzadas para entrenar un modelo

En esta sección se discutirán algunas de las técnicas más recientes para el entrenamiento de modelos del estado del arte. Estas técnicas están explicadas con ejemplos en el capítulo 7 de [3]. Para ilustrar estas técnicas se va a usar un subconjunto de ImageNet formado por 10 categorías llamado Imagenette.

Normalización

Al entrenar un modelo, suele ayudar que los datos de entrada estén normalizados, es decir, con media 0 y desviación estándar 1. Pero la mayoría de imágenes contienen valores entre 0 y 255 o entre 0 y 1, por lo que no se cumple esta condición. Con fastai, normalizar el conjunto de datos es muy fácil; simplemente hay que aplicar la transformación `Normalize` y pasarle la media y la desviación estándar que queramos. En la siguiente función se define un `DataLoaders` aplicando esta técnica:

```
def get_dls(bs, size):
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)])
return dblock.dataloaders(path, bs=bs, num_workers=0)
```

Fragmento de código 4.57: Crear Datablock aplicando normalización

Cuando usamos un modelo preentrenado mediante `cnn_learner`, fastai automáticamente añade la transformación de normalización adecuada.

Redimensionamiento progresivo

Consiste en ir aumentando gradualmente el tamaño de las imágenes a medida que se entrena el modelo. Esto funciona porque las características que aprenden las redes convolucionales no tienen que ver con el tamaño de la imagen. Además, es otra forma de realizar data augmentation. Para implementar esta técnica, lo más fácil es crear una función `get_dls` como en el apartado anterior y pasarle el tamaño de imagen y de batch.

Test Time Augmentation

Consiste en aplicar técnicas de data augmentation al conjunto de validación, y después tomar la media o el máximo de las predicciones para cada versión aumentada de la imagen. En fastai, se pasa el método `tta` a un `DataLoader`:

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

Fragmento de código 4.58: Aplicar Test Time Augmentation

4.3.6. Mixup

Mixup es una técnica de aumento de datos muy potente, introducida en 2017 por el paper *mixup: Beyond Empirical Risk Minimization*. Para cada imagen, hace lo siguiente:

1. Selecciona otra imagen aleatoriamente del conjunto de datos.

2. Elige un peso aleatorio.

3. Realiza una media ponderada (usando el peso del paso 2) de la imagen seleccionada en el paso 1 con la imagen seleccionada antes de empezar. Esta será la variable independiente.

4. Realiza una media ponderada (usando el peso del paso 2) de las etiquetas de la imagen seleccionada en el paso 1 con las etiquetas de la imagen seleccionada antes de empezar

Para que esto funcione, la salida debe estar *one-hot encoded*. En resumen, creamos una imagen que es combinación lineal de otras 2, y en cada epoch, no se entrena con la misma imagen, sino que es una combinación lineal diferente cada vez. El inconveniente es que requiere más epochs para alcanzar una alta precisión en comparación a otras técnicas. En fastai se hace añadiendo un *callback* al `Learner`. Los `Callbacks` se usan para definir un comportamiento personalizado en cada epoch.

```
learn = Learner(dls, model, loss_func=CrossEntropyLossFlat(),
                metrics=accuracy, cbs=MixUp())
```

Fragmento de código 4.59: Aplicar Mixup

Suavizado de etiquetas

El suavizado de etiquetas o *label smoothing* es una alternativa al one-hot. Consiste en reemplazar los 1s con un número ligeramente inferior a 1 y los 0s con un número ligeramente superior a 0. En fastai se aplica cambiando la función de loss:

```
learn = Learner(dls, model, loss_func=LabelSmoothingCrossEntropy(),
                    metrics=accuracy)
```

Fragmento de código 4.60: Aplicar label smoothing

4.3.7. Redes neuronales convolucionales

A continuación se detalla una forma de crear una red neuronal convolucional desde 0 usando fastai y Pytorch. Este ejemplo se ha tomado del notebook del capítulo 13 de [3]. Vamos a usar el MNIST para este ejemplo.

Primero creamos el DataLoader:

```
def get_dls(bs=64):
    return DataBlock(
        blocks=(ImageBlock(cls=PILImageBW), CategoryBlock),
        get_items=get_image_files,
        splitter=GrandparentSplitter('training','testing'),
        get_y=parent_label,
        batch_tfms=Normalize()
    ).dataloaders(path, bs=bs, num_workers=0)
dls = get_dls()
```

Fragmento de código 4.61: Preparar DataLoaders para red convolucional

A continuación definimos una capa convolucional:

```
def conv(ni, nf, ks=3, act=True):
    res = nn.Conv2d(ni, nf, stride=2, kernel_size=ks, padding=ks//2)
    if act: res = nn.Sequential(res, nn.ReLU())
    return res
```

Fragmento de código 4.62: Definir capa convolucional

Esta tiene un *stride*=2 y un *kernel size* de 3x3. Normalmente se debe duplicar el número de filtros cada vez que tengamos una capa con un stride=2. Para ello, se puede aplicar un kernel más grande a la primera capa. Una red simple quedaría de la siguiente forma:

```
def simple_cnn():
    return sequential(
        conv(1 ,8, ks=5),         #14x14
        conv(8 ,16),              #7x7
        conv(16,32),              #4x4
        conv(32,64),              #2x2
        conv(64,10, act=False),   #1x1
        Flatten(),
    )
```

Fragmento de código 4.63: Definir red convolucional

Los comentarios indican el tamaño del mapa de activación después de cada capa. Para medir el rendimiento interno de nuestra red, se puede usar el callback `ActivationStats`, que graba la media, la desviación estándar y el histograma del mapa de activaciones de cada capa que se entrene. Normalmente, un modelo deberá tener una media y desviación típica relativamente estables durante el entrenamiento, si no, el entrenamiento no está siendo adecuado. Hay varias formas de estabilizar el entrenamiento:

- **Aumentar el tamaño del batch**: Los lotes más grandes tienen gradientes más precisos, ya que se calculan a partir de más datos.

- **1cycle training**: Leslie Smith introdujo esta técnica en su artículo: *Super-Convergence: Very Fast Training of Neural Networks Using Large Learning Rates*[27]. Consiste en separar el tratamiento del ratio de aprendizaje en dos fases: Una en la que el ratio crece desde el valor mínimo hasta el valor máximo (*warmup*), y otra en la que decrece hasta encontrar el valor mínimo (*annealing*). Es la técnica usada por defecto en la función `fine_tune` de fastai.

- **Batch normalization**: Consiste en tomar las medias y desviaciones estándar medias de las activaciones de una capa y usarlas para normalizar las activaciones del resto de capas.

4.4. Conclusión: Keras y TensorFlow vs Fastai y PyTorch

El objetivo de este trabajo no es comprender completamente los fundamentos subyacentes de las redes neuronales y aprender a usar frameworks de bajo de nivel como TensorFlow y PyTorch, sino aprender a utilizar redes neuronales para resolver problemas de la vida real utlizando frameworks de alto nivel como Keras y fastai.

Tras hacer varios ejemplos similares con cada pareja de frameworks, se ha decidido realizar el resto del trabajo con **fastai** y **PyTorch**. La principal razón es que fastai es el framework más sencillo de usar y que provee las funciones de más alto nivel, lo que permite centrarse en los problemas que de verdad se quieren realizar. No hay que preocuparse de los detalles de implementación y además el libro de Jeremy Howard sobre fastai [3] explica los conceptos y ejemplos suficientes para llevar a cabo los objetivos de este TFG. El mayor inconveniente es que al ser el framework más nuevo de los cuatro, no hay muchos más ejemplos en Internet que los del libro y los de la documentación oficial.

Aunque Keras también tiene una interfaz bastante intuitiva, en mi opinión, fastai es más fácil de entender para alguien que está utilizando redes neuronales por primera vez. Como Keras es una API para TensorFlow, este último tampoco se va a usar. PyTorch solo se usará en momentos puntuales, cuando la API de fastai no permite realizar una tarea concreta, como crear redes desde cero.

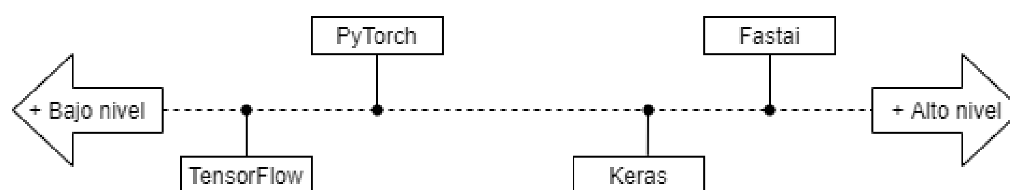


Figura 4.11: Frameworks de Deep learning

Capítulo 5

Astroinformática

En este capítulo se dejará temporalmente de lado el tema de las redes neuronales para hablar sobre astronomía y herramientas informáticas usadas en este campo.

5.1. Introducción

La **astroinformática** es un campo interdisciplinar en el que se combinan astronomía, informática, ciencia de datos y tecnologías de la información y la comunicación. Tiene como principal objetivo desarrollar métodos, aplicaciones y herramientas de ciencia de datos y estadística para aplicarlos al estudio de la astronomía [28].

Es un área de investigación muy amplia que tiene herramientas y conceptos propios que son necesarios conocer y manejar para entender el contexto del trabajo. Después se realizarán ejemplos de uso de las herramientas en tareas concretas.

5.2. Herramientas

Para la realización de este trabajo se ha usado un sistema operativo Windows 10 de 64 bits. Todas las herramientas utilizadas son compatibles con Windows 10 salvo el SExtractor descrito en los capítulos siguientes. Para poder usar esta última herramienta y poder ejecutar comandos linux cuando se necesite se ha usado el **WSL**(Windows Subsystem for Linux)[29]. Este es un software oficial de Microsoft que permite utilizar varias distribuciones Unix en Windows en modo consola. Las distribuciones para el WSL descargan desde la Microsoft Store. A continuación se detallan varias herramientas de la astroinformática usadas en este trabajo.

- SAOImageDS9: Herramienta cuya principal función es visualizar imágenes astronómicas. Permite reescalar y aplicar diferentes filtros para resaltar las imágenes de forma que podamos fijarnos en lo que más nos interese. Está disponible para Windows, MacOS y Linux.

- Topcat: Es la abreviatura de “Tool for OPerations on Catalogues And Tables”. Es una herramienta muy útil para visualizar datos tabulares de una forma rápida y sencilla. Está disponible para todos los sistemas operativos en forma de archivo .jar.

- Aladin Sky Atlas: Permite visualizar imágenes astronómicas de una forma más interactiva que con DS9, y además permite utilizar imágenes directamente del observatorio virtual. Está disponible para Windows, MacOS y Linux.

- SExtractor:Este programa permite detectar (y segmentar algorítmicamente) un catálogo de los objetos que haya en una imagen astronómica. Solo está disponible para distribuciones Unix, pero se puede instalar fácilmente en Windows con el WSL.

Las herramientas mencionadas trabajan con imágenes tomadas por telescopios, cuyas características son muy particulares, y por ello, las imágenes astronómicas usan el formato .fits, *Flexible Image Transport System* [30], que se creó específicamente para el almacenamiento, la transmisión y el procesamiento de este tipo de imágenes. Estas imágenes requieren telescopios capaces de realizarlas, en nuestro caso el Telescopio Espacial Hubble, y se realizan mediante procedimientos llamados **cartografiado de galaxias** o *galaxy surveys*. Ha habido una gran cantidad de estos estudios a lo largo de los años y cada uno tiene unos objetivos concretos[31]. El cartografiado de galaxias en el que nos centraremos y del que sacaremos las imágenes para las redes neuronales es el **CANDELS**[32], que en principio está destinado para observar galaxias ultra lejanas (no en vano es el cartografiado de galaxias al que más tiempo le ha dedicado –984 órbitas– el citado telescopio espacial).

5.3. Trabajando con herramientas de astroinformática

En esta sección se detallará el proceso para realizar algunas tareas concretas usando las herramientas mencionadas anteriormente con el objetivo de aprender a usar estas herramientas a un nivel básico.

5.3.1. Cómo recortar una imagen de una estrella o una galaxia a partir de una imagen astronómica con SExtractor y DS9

La imagen fuente que vamos a utilizar es `hlsp_candels_hst_wfc3_cos-tot_f160w_v1.0_drz.fits`, perteneciente a CANDELS, y se puede descargar en el repositorio de datos del telescopio espacial Hubble. Primero vamos a visualizar la imagen astronómica completa con SAOImageDS9:

1. Menú `File->Open` y elegimos el fichero .fits que deseemos. Al abrirlo, normalmente se verá negro y deberemos aplicarle filtros:

2. Menú `Scale->Log` y menú `Color->hsv`. Con esto logramos ver las imágenes como lo haría el ojo humano.

3. Menú `Scale->Scale parameters`. Se abrirá una nueva ventana (Figura 5.1). En los cuadros de texto de Low introducimos `-0.0001`, y en High, introducimos `10`. Con esto reducimos el rango dinámico de los píxeles a mostrar para fijarnos en rasgos de bajo brillo superficial.

En la figura 5.1 el eje Y representa el número total de píxeles a cada flujo, que está indicado en el eje X. Reduciendo nuestro rango entre -0.0001 y 10, hacemos que nuestra visión no esté sesgada a píxeles calientes (de valor muy alto) ni fríos (de valor muy bajo).

Ahora estamos distinguiendo con claridad las estrellas y galaxias que hay en esa imagen astronómica. En la figura 5.2 se puede observar una captura de cómo se vería un fragmento de una imagen astronómica. Arriba a la derecha se puede ver un minimapa con las dimensiones

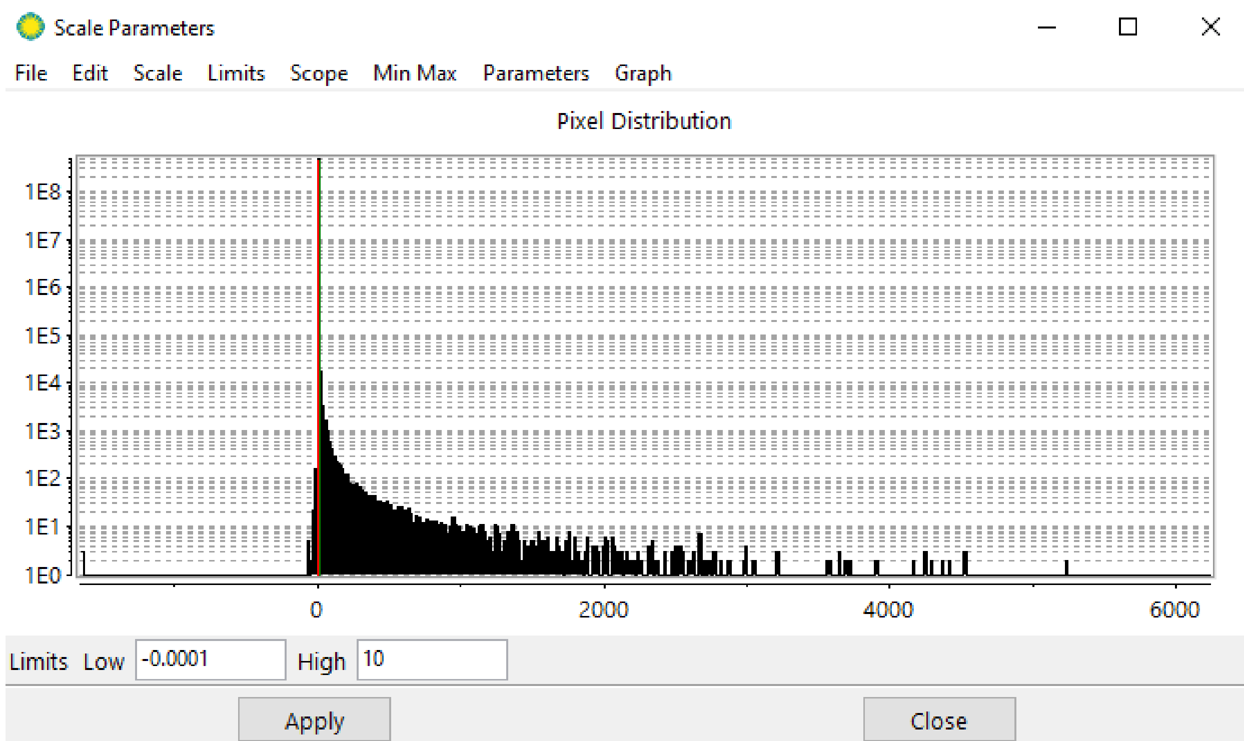


Figura 5.1: Ventana Scale Parameters

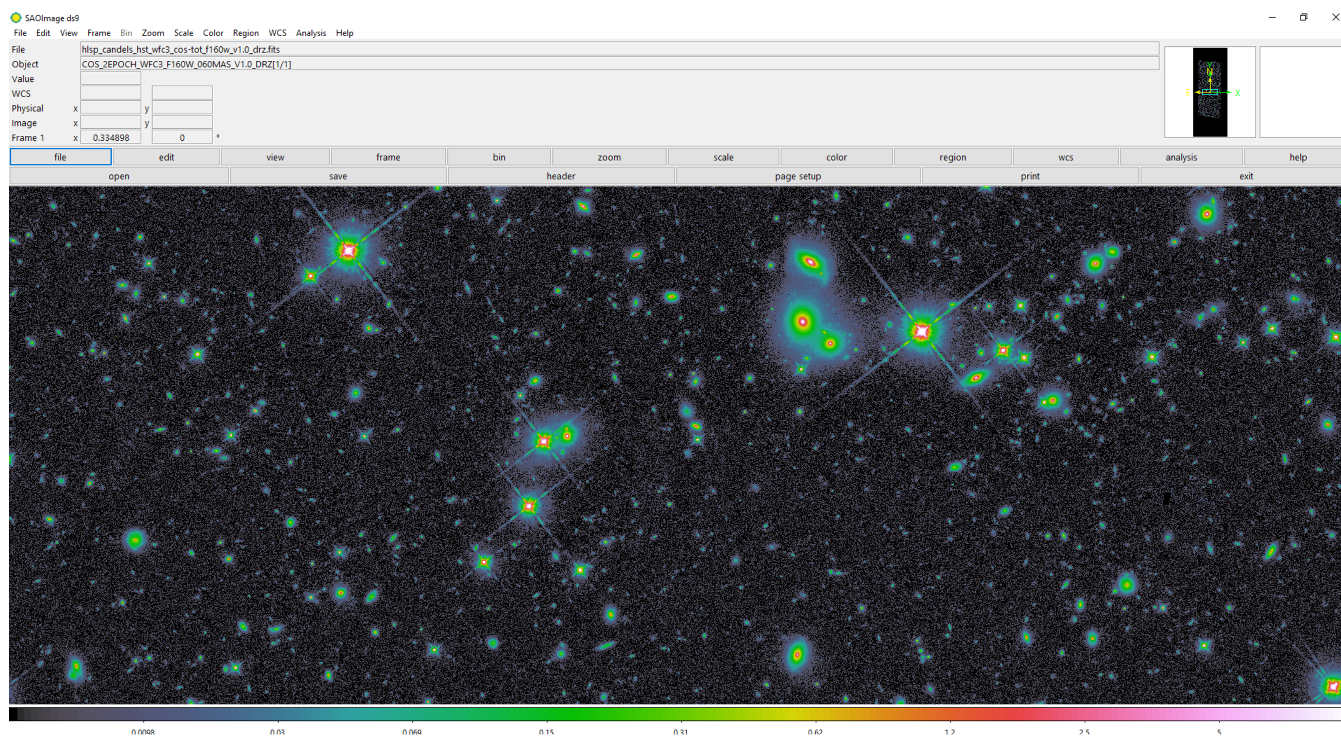


Figura 5.2: Ventana principal de DS9

reales de la imagen y en qué parte nos encontramos. Los objetos de cuatro puntas son estrellas, así es como se ve una fuente puntual debido al patrón de difracción del telescopio. Esta imagen en concreto es bastante grande y ocupa 1,9Gb.

Ahora queremos obtener imágenes de galaxias o estrellas individuales, pero no vamos a hacer eso simplemente haciendo zoom en la imagen, capturando la pantalla y buscando y recortando lo que queramos. Vamos a usar SExtractor, que nos dará las coordenadas exactas en las que está centrado cada objeto que detecte.

1. Abrimos una terminal Unix en el directorio en el que queramos realizar la tarea. Por comodidad, en este mismo directorio será recomendable tener los siguientes ficheros:

    - Imagen que queramos analizar.
    - Fichero de parámetros: En nuestro caso `default.param`. Es el fichero que contiene los parámetros que queremos que tenga en cuenta el programa
    - Fichero de configuración: En nuestro caso `wfc3_H.sex`
    - Otros ficheros: `default.conv`, `default.nnw`, `gauss_3.0_5x5.conv`

2. Ejecutamos el siguiente comando:

```
sextractor hlsp_candels_hst_wfc3_cos-tot_f160w_v1.0_drz.fits -c
↪ wfc3_H.sex -PARAMETERS_NAME default.param -WEIGHT_TYPE MAP_RMS
↪ -WEIGHT_IMAGE hlsp_candels_hst_wfc3_cos-tot_f160w_v1.0_rms.fits
↪ -CHECKIMAGE_TYPE SEGMENTATION -CHECKIMAGE_NAME
↪ segmentation.fits
```

    -PARAMETERS_NAME sirve para indicar el fichero de parámetros que queremos usar. -c sirve para indicar el fichero de configuración que queremos usar. Este comando analizará la imagen .fits y creará dos ficheros:

    - Un fichero `first.cat` que contendrá una lista con todos los objetos que hay en la imagen junto con sus coordenadas dentro de la imagen.
    - Un fichero `segmentation.fits` que es una imagen de las mismas dimensiones que la original en la que está resaltadas las siluetas de cada objeto.

Por último, vamos a obtener los recortes de los objetos que queramos mediante un sencillo programa en Python en el que usaremos las bibliotecas Numpy, Astropy y Matplotlib.

1. Importamos las bibliotecas necesarias:

```python
import numpy as np
from astropy.io import fits
from astropy.table import Table
import matplotlib.pyplot as plt
```

Fragmento de código 5.1: Importar bibliotecas Numpy, Astropy y Matplotlib

2. Leemos los campos del catálogo de objetos que ha creado SExtractor. Además, podemos usar poner la variable recién creada sin más en el intérprete para que muestre las dimensiones y los datos de la tabla:

```
tt = Table.read("first.cat",names=["id","ra","dec","mag_aper",
    "magerr_aper","x_image","y_image",
    "flux_radius","ellipticity","theta","kron","a"],
    format="ascii.commented_header")

tt

```

Fragmento de código 5.2: Leer catálogo de objetos con Astropy

3. Abrimos la imagen astronómica de la que queremos obtener fragmentos(Se debe sustituir PATH por nuestra ruta a la imagen):

```
hdu = fits.open("PATH")
img = hdu[0].data
hdr = hdu[0].header

```

Fragmento de código 5.3: Leer imagen .fits

4. Opcionalmente podemos intentar mostrar la imagen mediante:

```
plt.imshow(img)

```

Fragmento de código 5.4: Mostrar imagen con Matplotlib

Pero al igual que nos ha pasado antes con el DS9, la imagen sale oscura y no se distingue nada, porque el ojo humano es un detector logarítmico. Así que debemos aplicarle filtros:

```
plt.imshow(np.log10(img), origin = "lower", cmap = "cubehelix")

```

Fragmento de código 5.5: Mostrar imagen con Matplotlib aplicando filtros

Con esto nos saldría la imagen entera blanca salvo algunos puntos oscuros.

5. Lo último que debemos hacer es decirle a nuestro programa qué coordenadas de la imagen principal hay que recortar y mostrarla o guardarla en el formato que queramos:

```
radio=100
x=int(linea["x_image"])
y=int(linea["y_image"])
objeto=np.log10(img[y-radio:y+radio,x-radio:x+radio])
plt.imshow(np.log10(objeto), origin = "lower", cmap = "cubehelix")

```

Fragmento de código 5.6: Recortar imagen astronómica

Las variables `x_image` e `y_image` son las coordenadas x e y del centro del objeto (columnas 6 y 7 del first.cat) detectado en la imagen. También debemos especificar un radio para calcular las dimensiones totales del objeto recortado y, por último, lo recortamos aplicándole la función logarítmica para poder distinguir los colores.

En vez de ir de una en una, para este proceso se podría hacer un bucle que obtuviera una lista de objetos (filas del first.cat), según algún criterio que especifiquemos y después otro bucle para recortarlas e imprimirlas:

```
ListaObjetos=[]
for linea in tt:
    valor=float(linea["mag_aper"])
    if valor<15:
    ListaObjetos.append(int(linea["id"]))

for objeto in ListaObjetos:
    linea=tt[objeto-1]
    x=int(linea["x_image"])
    y=int(linea["y_image"])
    plt.axis('off')
    nuevoA=np.log10(img[y-radio:y+radio,x-radio:x+radio])
    plt.imshow(nuevoA, origin = "lower", cmap = "cubehelix")
    plt.show()
```

Fragmento de código 5.7: Recortar trozos de imagen en bucle siguiendo un criterio determinado

La línea 11 del código anterior sirve para quitar los ejes de coordenadas que Matplotlib añade por defecto. Para guardar la imagen en nuestro ordenador, habría que añadir la siguiente línea justo antes de `plt.show()`, donde el segundo parámetro hace que la imagen quede sin márgenes blancos:

```
plt.savefig(r"PATH"+".png",bbox_inches='tight')
```

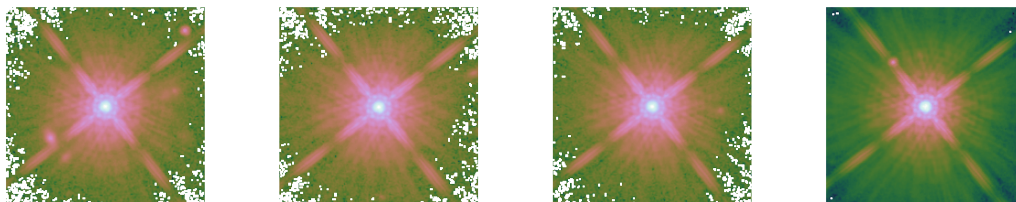
Fragmento de código 5.8: Guardar imagen con Matplotlib

Por último, también podríamos guardar nuestras imágenes como .fits para que el DS9 las pueda abrir. Se hace de la siguiente forma:

```
hdu1 = fits.PrimaryHDU(IMAGEN)
hdu1.writeto(r"PATH"+".fits",overwrite=True)
```

Fragmento de código 5.9: Guardar imagen en formato .fits con Astropy

Algunas de las estrellas que encuentra son las siguientes:



Subimágenes de estrellas

En el ejemplo anterior se han obtenido los objetos siguiendo el criterio `mag_aper<15` para obtener estrellas, pero se podrían usar otros criterios para obtener distintos objetos. Normalmente las estrellas son los objetos más luminosos en una imagen dada. Cogemos `mag_aper<15` porque esta magnitud es el parámetro astronómico que mide el brillo de un objeto, y es menor cuanto más brillante sea este.

5.3.2. Cómo obtener las coordenadas de una estrella o una galaxia con SExtractor, Topcat y Aladin

Ahora vamos a realizar una tarea parecida pero con otro enfoque. Partimos del fichero first.cat creado por SExtractor de la misma forma que se ha explicado en el apartado anterior, y lo abriremos mediante el programa Topcat. Para ello:

1. Ejecutamos Topcat. Este programa es un fichero .jar.
2. Seleccionamos `File->Load Table`. Se abrirá una nueva ventana.
3. En esta nueva ventana seleccionamos `Filestore Browser` y elegimos el fichero que deseamos abrir; en nuestro caso es `first.cat`. Es importante además seleccionar `ASCII` como `Table Format`, si no, puede que nuestro archivo genere un error.
4. Una vez está la tabla cargada en Topcat, podemos seleccionar `Views->Table Data` para que aparezca una nueva ventana con los datos de la tabla.

Ahora tenemos muchas opciones para visualizar gráficos sobre los datos de la tabla y otras funcionalidades. Una de ellas consiste en seleccionar `Views->Column Info`, entonces se abre una nueva ventana con información sobre las columnas. Podemos hacer varias cosas sobre ellas, como ocultarlas o cambiarlas el nombre. Ahora vamos con los siguientes pasos:

1. Ejecutamos Aladin.
2. En Aladin, cargamos la imagen astronómica que hemos usado anteriormente en DS9 y SExtractor. Esta imagen sale inicialmente en blanco y negro, por lo que es recomendable aplicarle algunos filtros de color. Para ello seleccionamos la opción `pixel`. En esta nueva ventana habrá dos menús desplegables que pondremos en `cubehlx` y `Log` respectivamente.
3. En Topcat nos saldrá una nueva opción `File->Send table to->Aladin`. Si realizamos esto, Aladin recibirá el archivo, pero no podrá hacer nada con él todavía, así que antes debemos realizar varias acciones en Topcat.
4. En Topcat, abrimos la ventana de opciones de columnas (`Views->Column Info`). Hacemos invisible todas las columnas excepto la 2 y la 3. A estas últimas las cambiaremos el nombre. A la 2 la llamaremos radž a la 3, "dec", ya que estas columnas representan respectivamente la ascensión recta (right ascension) y la declinación (declination).
5. Ahora sí, enviamos la tabla a Aladin.
6. En Aladin nos aparecerán unos puntitos rojos minúsculos en el centro de cada objeto astronómico. Podemos cambiar el diseño de estos puntitos en `prop->Default shape`

Gracias a esta funcionalidad podemos saber las coordenadas del centro del objeto que queramos y escribirlas en un programa en Python para que los recorte. Vamos a usar el Jupyter Notebook del apéndice A.2:

1. Primero vamos a crear un vector de dos dimensiones con la rad y la dec de diez galaxias que elijamos. Estos valores los obtenemos a mano del Aladin:

```
{galax =[[150.1143535,2.356457],
        [150.1138174, 2.3607032],
        [150.1124122, 2.3549241],
        [150.0909541, 2.3912195],
        [150.1056863, 2.3991092],
        [150.1189451, 2.4571717],
        [150.1203373, 2.4715464],
        [150.0907101, 2.2056509],
        [150.0951051, 2.1937415],
        [150.1018265, 2.2009676]]}

```

Fragmento de código 5.10: Crear vector de rad y dec de galaxias

2. Después obtenemos los objetos de la tabla cuyo rad y dec coinciden con los de la tabla:

```
ListaGalaxias = []
contadorGalax = 0
for gal in galax:
    valorRa = galax[contadorGalax][0]
    valorDec = galax[contadorGalax][1]
    for linea in tt:
        if valorRa == float(linea["ra"]):
            if  valorDec == float(linea["dec"]):
                contadorGalax = contadorGalax+1
                ListaGalaxias.append(int(linea["id"]))
print(contadorGalax)

```

Fragmento de código 5.11: Obtener objetos del catálogo a partir del vector de rad y dec

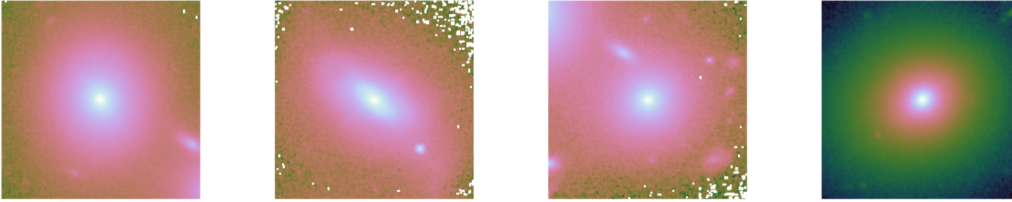
3. Por último, imprimimos los recortes de la imagen junto con otros datos, por ejemplo, el ángulo de posición de la galaxia:

```
!mkdir galaxias
contadorGalax = 0
for gal in ListaGalaxias:
    linea = tt[gal-1]
    contadorGalax=contadorGalax+1
    #Ahora obtenemos las coordenadas de los pixeles en la imagen
    x=int(linea["x_image"])
    y=int(linea["y_image"])
    plt.axis('off')
    nuevoA=np.log10(img[y-radio:y+radio,x-radio:x+radio])
    plt.imshow(nuevoA, origin = "lower", cmap = "cubehelix")
    plt.savefig(r"galaxias\foto"+str(contadorGalax)+".png",bbox_inches='
   tight')
    plt.show()
    print("Angulo de posicion: "+str(float(linea["theta"])))

```

Fragmento de código 5.12: Recortar y guardar imágenes de galaxias

Algunas de las galaxias que encuentra son las siguientes:



Subimágenes de galaxias

5.4. Estado del arte del deep learning en astronomía: Morpheus

Morpheus [33] es un framework de deep learning desarrollado en la Universidad de California cuya principal función es clasificar la morfología de objetos astronómicos a nivel de pixel y servir como alternativa a herramientas más tradicionales como Source Extractor.

Su red neuronal ha sido entrenada mediante imágenes FITS de galaxias CANDELS ya clasificadas y sus mapas de segmentación derivados de SExtractor. Morpheus identifica qué pixeles de una imagen tienen más probabilidad de contener flujo y lo separa de los pixeles del fondo. Estos pixeles los clasifica en en categorías morfológicas: spheroid, disk, irregular, point source/-compact y background. Morpheus admite imágenes FITS de cualquier tamaño, subdiviéndolas entre regiones que pueda procesar en paralelo.

Morpheus está implementado en Python3, con la biblioteca Tensorflow y utiliza una arquitectura de red neuronal inspirada por la de la U-Net [34]. Como función de activación utiliza la Unidad Lineal Rectificada, y como función de coste combina una función de entropía cruzada con una Dice loss para su estadística de optimización. El conjunto de entrenamiento está formado por 7629 galaxias del conjunto de datos K15. Las clasificaciones del K15 consisten en votos realizados por astrónomos expertos, que analizaban las galaxias y decidían entre una lista de categorías morfológicas.

También se han aplicado diferentes técnicas de aumento del tamaño del conjunto de datos, incrementando el número efectivo de imágenes por un factor de 574000, para reducir el sobreajuste. Para optimizar los parámetros del modelo, se usa el método del descenso del gradiente estocástico Adam. El algoritmo Adam usa el primer y segundo momentos de los gradientes de primer orden que se calculan mediante backpropagation para hallar el mínimo de la función de coste. Como salida, Morpheus produce distintas imágenes resaltando los pixeles que pertenecen cada categoría morfológica.

Morpheus es un ejemplo de lo que se puede llegar a hacer con redes neuronales en el campo de la astronomía. Obviamente, Morpheus está hecho por un grupo de profesionales expertos con años de experiencia, en este trabajo se tratarán y resolverán problemas más sencillos, como se verá en el capítulo siguiente.

Capítulo 6

Clasificación de galaxias

6.1. Introducción

En este capítulo haremos experimentos de clasificación y de regresión con un conjunto de datos de imágenes de galaxias proporcionado por el profesor Fernando Buitrago. Tomaremos como experimento de referencia un paper en el que se entrena a una red neuronal para que realice regresión de valores morfológicos a partir de una imagen de una galaxia. Realizaremos variaciones de los entrenamientos con diferentes parámetros para analizar cómo afectan al rendimiento de la red neuronal.

6.2. Artículo de Marc Huertas-Company 2015

En *A Catalog of visual-like morphologies in the 5 CANDELS fields using deep learning* [35], el profesor Marc Huertas-Company y colaboradores describen cómo han creado una red neuronal para predecir, a partir de una imagen de una galaxia, la morfología de esta. En concreto, cada galaxia tiene 5 valores en punto flotante entre 0 y 1 que corresponden a la clasificación morfológica a la que pertenecen: Disk, Spheroid, Irregular, Point Source y None. Pueden tener un poquito de cada categoría y sumar más de 1 entre todos los valores. La excepción es la categoría None, que significa que la galaxia no es de ningún tipo, y si pertenece a esta categoría, entonces no puede pertenecer a ninguna de las otras 4.

La clasificación morfológica utilizada en este artículo se basa en la **secuencia de Hubble**, que es la clasificación morfológica inventada por el astrónomo Edwin Hubble en 1926 [36]. Esta clasificación divide las galaxias en tres clases principales: **elípticas**, **lenticulares** y **espirales**, aunque también hay un cuarta clase para las irregulares. También se suele denominar diagrama de tenedor o de diapasón de Hubble por la forma de representar esta clasificación, figura 6.1.

Las elípticas se representan a la izquierda, tienen forma de elipse y una distribución de la luz bastante uniforme. Las lenticulares se representan en el centro, antes de que el gráfico se divida en dos ramas (S0); estas galaxias tienen un bulbo central brillante, rodeada de una estructura en forma de disco, pero al contrario que las espirales, no tienen brazos, porque son tan *viejas* que han consumido todo el gas que tenían para formar estrellas. Las galaxias espirales son las representadas a la derecha, que tienen un bulbo central brillante y un número par de brazos espirales que salen de él, pueden ser 2, 4, 6 o tantos que cuesta distinguir el número, estas son las llamadas galaxias espirales floculentas. La rama superior corresponde a las galaxias espirales

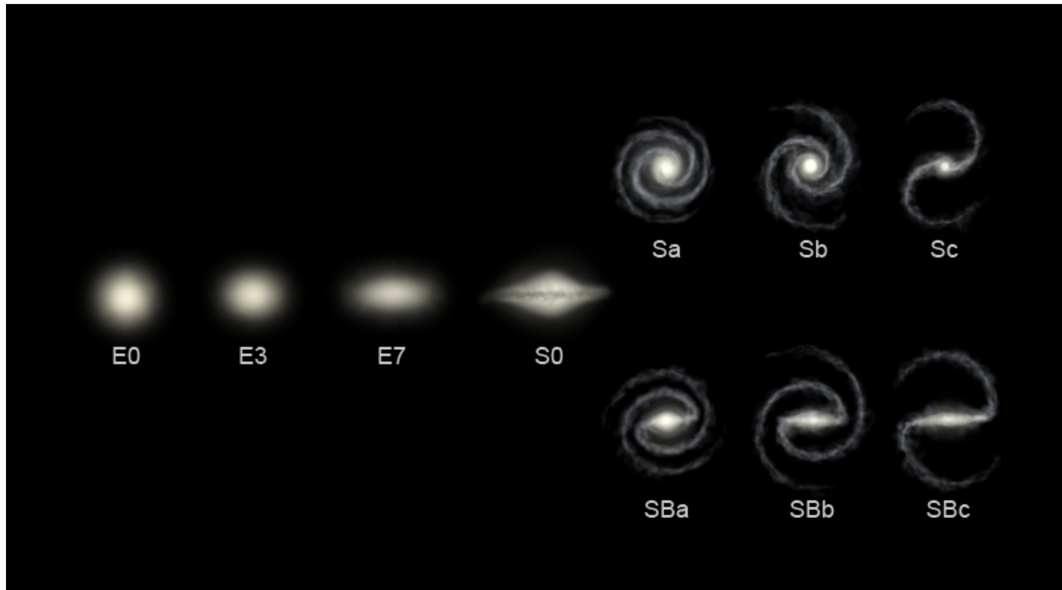


Figura 6.1: Secuencia de Hubble [36]

regulares y la rama inferior a las galaxias espirales barradas. Por último, las galaxias irregulares no tienen representación en el diagrama de tenedor aunque sí están identificadas por Hubble.

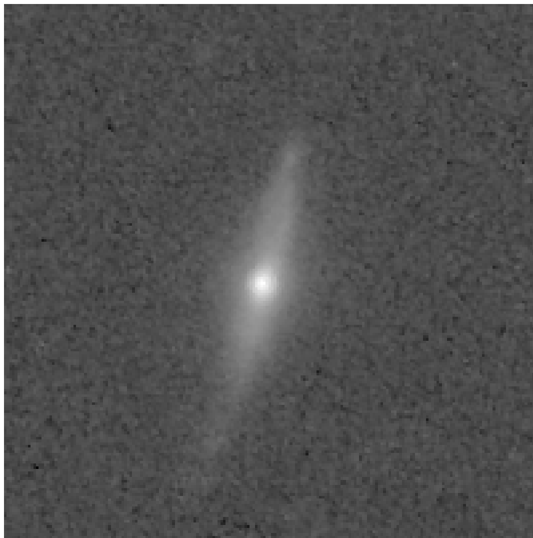
6.3. Conjunto de datos

6.3.1. Conjunto de datos inicial

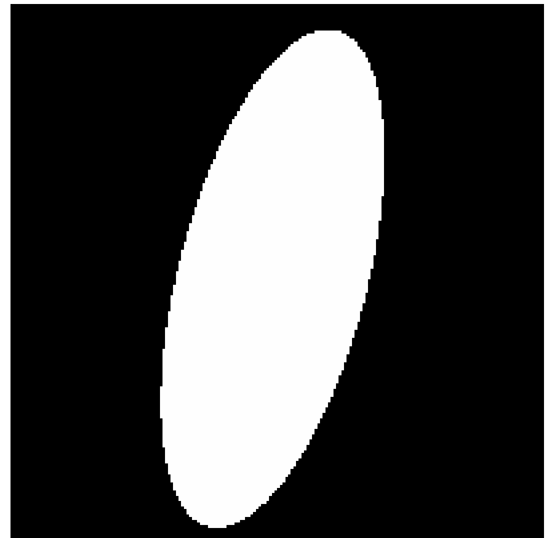
El profesor Fernando Buitrago nos ha proporcionado el conjunto de datos inicial, que está formado por unas 3400 imágenes de galaxias y sus respectivas máscaras en 4 filtros diferentes: I, J, V, H. Estos filtros son los del Telescopio Espacial Hubble, y cada uno representa una banda de frecuencias de la luz visible en el espectro electromagnético. En total, más de 26000 imágenes, una mitad imágenes de galaxias y la otra mitad de máscaras para estas galaxias. Una máscara es una imagen que sirve para quitar el ruido de otra imagen para centrarse en algo concreto. En este caso, nos queremos centrar en la galaxia que está en el centro y quitar posibles galaxias que hayan salido en los bordes de la imagen para no confundir a la red neuronal. En la figura 6.2 se muestra un ejemplo del conjunto datos. La primera imagen es la de una galaxia en el filtro H (aplicando un filtro logarítmico en DS9 para que se aprecie mejor la forma de la galaxia), la segunda imagen es su máscara. Junto con todas estas imágenes tenemos además un fichero .cat, que contiene, entre otras cosas, los valores morfológicos en punto flotante de cada galaxia del dataset. Este fichero no será necesario en el experimento de clasificación morfológica pero sí en el de regresión..

6.3.2. Conjunto de datos final

El objetivo es fusionar los distintos filtros de cada imagen y las máscaras para conseguir una imagen en color RGB de cada galaxia, para tener en total unas 3400 imágenes, y con las que podamos alimentar a la red neuronal. Para generar el dataset usamos un conjunto de funciones proporcionadas en un fichero Python(Apéndice A.3.1) por Fernando Buitrago y Benjamín Sahelices. La estructura principal de este código es la siguiente:



(a) Una galaxia tomada con el filtro H



(b) La máscara correspondiente a 6.2a

Figura 6.2: Una galaxia y su máscara

Primero se crean 6 directorios diferentes, uno para cada tipo de galaxia: DISK, DISKIRR, DISKSPH, IRR, NONE, SPH. Esto servirá para etiquetar los datos más tarde en la red neuronal. Cada imagen es un fichero .fits, que contiene una cabecera y los datos propiamente dichos, en este caso, una imagen. Para obtener la imagen de un .fits usamos el paquete astropy:

```
from astropy.io import fits
hdu = fits.open(filename_img_g)
img_g = hdu[0].data
hdr_g    = hdu[0].header
hud.close()
```

Fragmento de código 6.1: Obtener datos de imagen .fits

Debemos obtener las imágenes con los filtros I (azul), J (verde), H (rojo) de cada galaxia. Después multiplicamos cada filtro por la máscara de filtro H para juntarlas. Las imágenes con filtro V, por tanto, no se van a usar.

```
img_b=mask_r*img_b
img_g=mask_r*img_g
img_r=mask_r*img_r
```

Fragmento de código 6.2: Obtener imágenes enmascaradas

Una vez tenemos estas tres imágenes, las juntamos para obtener la imagen RGB:

```
rgb_original = createRGB(img_b, img_g, img_r, scale=scale, stretch=stretch, Q=
    QQ)
```

Fragmento de código 6.3: Crear imagen RGB

A continuación se elige el tamaño de la imagen con la siguiente línea:

```
fig = plt.figure(figsize=(2,2))
```

Fragmento de código 6.4: Definir tamaño de imagen

Los parámetros indican las pulgadas que tendrá la imagen, en este caso 2 pulgadas de ancho y de alto, para luego aplicarle 100 dpi (*dot per inch*, puntos por pulgada), cuando lo pasemos a

las funciones de matplotlib y así guardarlo como 200x200 píxeles. También hay que añadir la siguiente línea para quitar cualquier tipo de borde blanco que la función fuera a añadir a los márgenes de la imagen:

```
fig.subplots_adjust(0,0,1,1)
```

Fragmento de código 6.5: Quitar márgenes a imagen

Se le aplica algún procesamiento más y se le pasa a matplotlib para guardarla:

```
plt.savefig(path_output+morfo+"/"+morfo+"_"+gal+"_"+field[ii]+"_rgb_image.png"
    , format='png', dpi=100)
```

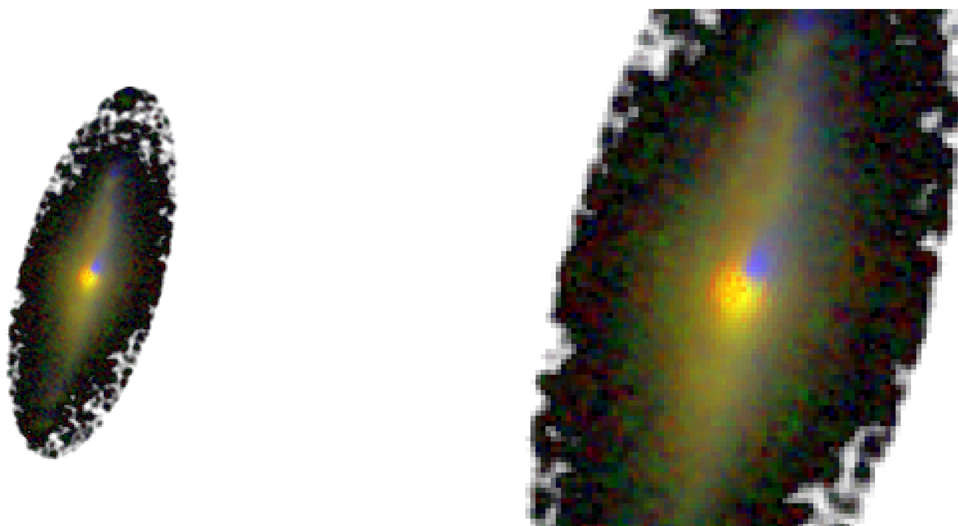
Fragmento de código 6.6: Guardar imagen para el dataset de galaxias

Se puede hacer una versión alternativa para recortar las imágenes y hacerlas de 100x100 píxeles. De esta forma, ocuparían menos y la red tardaría menos en entrenar, pero habría que ver cómo varían los resultados. La razón por la que se querría hacer esto es porque las imágenes de 200x200 tienen una gran cantidad de superficie sin información alrededor del centro de la imagen. Simplemente habría que cambiar las siguientes líneas:

```
img_b_recortada=img_b[49:150, 49:150]
img_g_recortada=img_g[49:150, 49:150]
img_r_recortada=img_r[49:150, 49:150]
rgb_original = createRGB(img_b_recortada, img_g_recortada, img_r_recortada,
    scale=scale, stretch=stretch, Q=QQ)
fig = plt.figure(figsize=(1,1))
```

Fragmento de código 6.7: Recortar imágenes de 200x200 a 100x100

Todo este proceso se realiza en un bucle para cada galaxia. En los experimentos siguientes se van a usar siempre imágenes de **200x200** píxeles en formato **.png** con 4 canales.



(a) Galaxia en rgb de 200x200　　(b) Galaxia en rgb de 100x100

Figura 6.3: Galaxia generada en rgb a partir de tres filtros

6.4. Modelo: ResNet

En varios ejemplos comentados anteriormente hemos usado como modelo una ResNet, y en este capítulo también la usaremos. La **ResNet** (Residual Network) fue introducido en 2015 por Kaiming et al. en su artículo *Deep Residual Learning for Image Recognition*[37] y actualmente es uno de los modelos más usados en deep learning. Las redes convolucionales anteriores a esta no podían tener muchas capas, ya que sufrían del problema del gradiente evanescente, explicado en el capítulo 2, pero las diferentes variantes ResNet daban grandes resultados teniendo muchísimas más capas que cualquiera de los modelos de la época. El motivo está en que la ResNet ideó las ***Skip Connections*** que se ilustran en la figura 6.4. El resultado no una capa está formado por el resultado de la capa normal + el valor inicial al entrar en la capa, $y = F(x) + x$. Esto significa que la red no predice y, sino la diferencia entre y y x. De ahí el nombre, la red está prediciendo residuos, que es predicción - objetivo. Existen muchas variantes de ResNet pero todas utilizan la

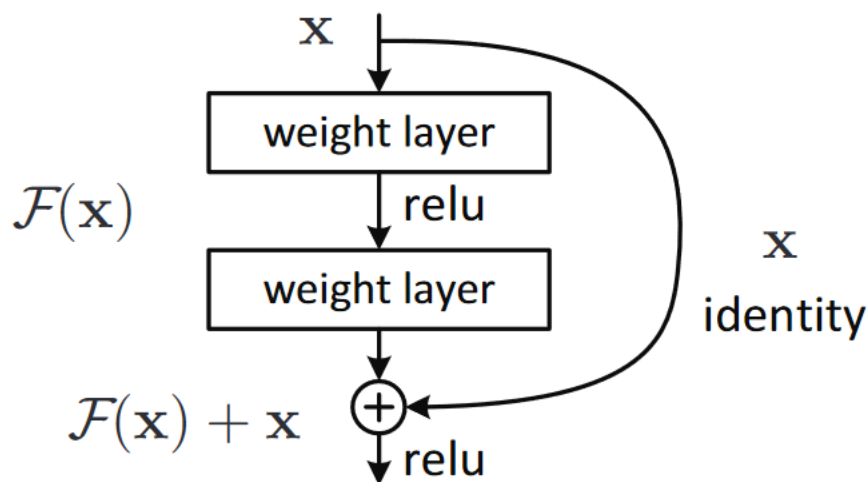


Figura 6.4: Bloque de ResNet con Skip Connections [3]

técnica de las *skip connections*. Se diferencian en el número de capas, indicado en el nombre, y el número de parámetros, por ejemplo [38]:

- ResNet18: 18 capas, 11,174 millones de parámetros.
- ResNet34: 34 capas, 21,282 millones de parámetros.
- ResNet50: 50 capas, 23,521 millones de parámetros.
- ResNet101: 101 capas, 42,513 millones de parámetros.
- ResNet152: 152 capas, 58,157 millones de parámetros.

6.5. Clasificación morfológica

6.5.1. Descripción

En este experimento se entrenará una red neuronal para que clasifique una imagen dada en una de las 6 categorías morfológicas que puede tener: DISK, DISK_IRR, DISK_SPH, SPH,

IRR y NONE. La estructura básica del experimento está en el Notebook del apéndice A.4.1. Se van a realizar pruebas con distintos modelos de red neuronal, con y sin data augmentation. Los que usen data augmentation siguen la estructura del Notebook del apéndice A.4.2. En este notebook definimos 2 funciones de data augmentation. La primera hará que las imágenes roten aleatoriamente solo en múltiplos de 90º:

```
def RotacionAleatoria(x):
batchsize=x.size(0)
result=torch.zeros(([batchsize]), dtype = torch.float, device='cuda')
for i in range(batchsize):
    valor=torch.rand(1)
    if(valor<0.25):
       result[i]=0.0
    elif(valor<0.5):
       result[i]=90.0
    elif(valor<0.75):
       result[i]=180.0
    else:
       result[i]=270.0
return result
```

Fragmento de código 6.8: Definir rotación aleatoria en múltiplos de 90º

La segunda introducirá un poco de ruido Gaussiano en las imágenes de forma aleatoria:

```
class RuidoGaussiano(Transform):
    order=100 #After normalize
def encodes(self, x:TensorImage):
    return x+torch.normal(mean=0.0, std=0.005, size=x.shape, device='cuda')
```

Fragmento de código 6.9: Definir ruido Gaussiano

Se van a realizar pruebas con 3 tipos de modelos de red neuronal: ResNet18, ResNet34 y ResNet50. Para todas las pruebas se entrenará durante 20 epochs y se mostrará:

1. Un gráfico lineal con la evolución de los valores de la función de coste en entrenamiento y en validación frente a los epochs.

2. Un gráfico lineal con la evolución de los valores de la métrica frente a los epochs.

3. La matriz de confusión resultante del entrenamiento

Los gráficos se generarán con Microsoft Excel a partir de los csv de resultados del entrenamiento que se generarán con el callback:

```
CSVLogger(fname="NOMBRE_FICHERO_CSV", append=true)
```

Fragmento de código 6.10: Callback que registra los resultados del entrenamiento en un fichero csv

6.5.2. Dataset simple

En este experimento el número de imágenes es el original. No se han multiplicado el número de imágenes. Resultados:

- **Resnet18 sin data augmentation**: Figura 6.5

- **Resnet18 con data augmentation**: Figura 6.6

- **Resnet34 sin data augmentation**: Figura 6.7

- **Resnet34 con data augmentation**: Figura 6.8

- **Resnet50 sin data augmentation**: Figura 6.9

- **Resnet50 con data augmentation**: Figura 6.10

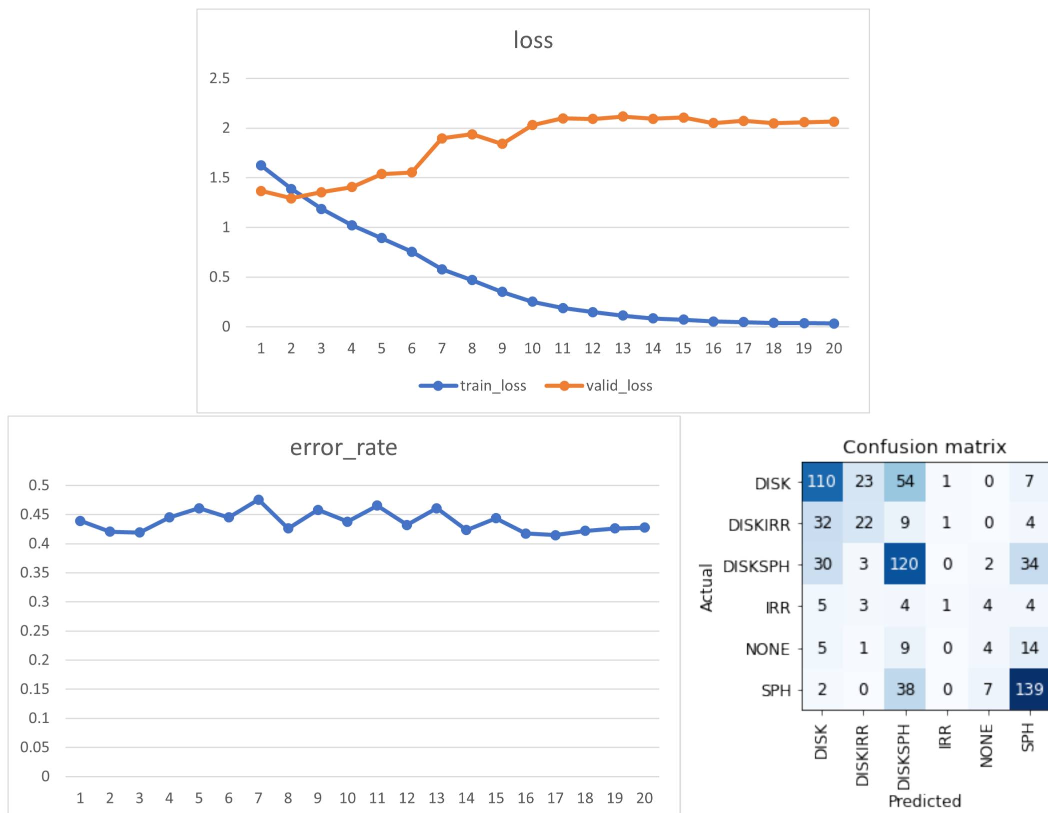


Figura 6.5: Clasificación sin data augmentation y ResNet18

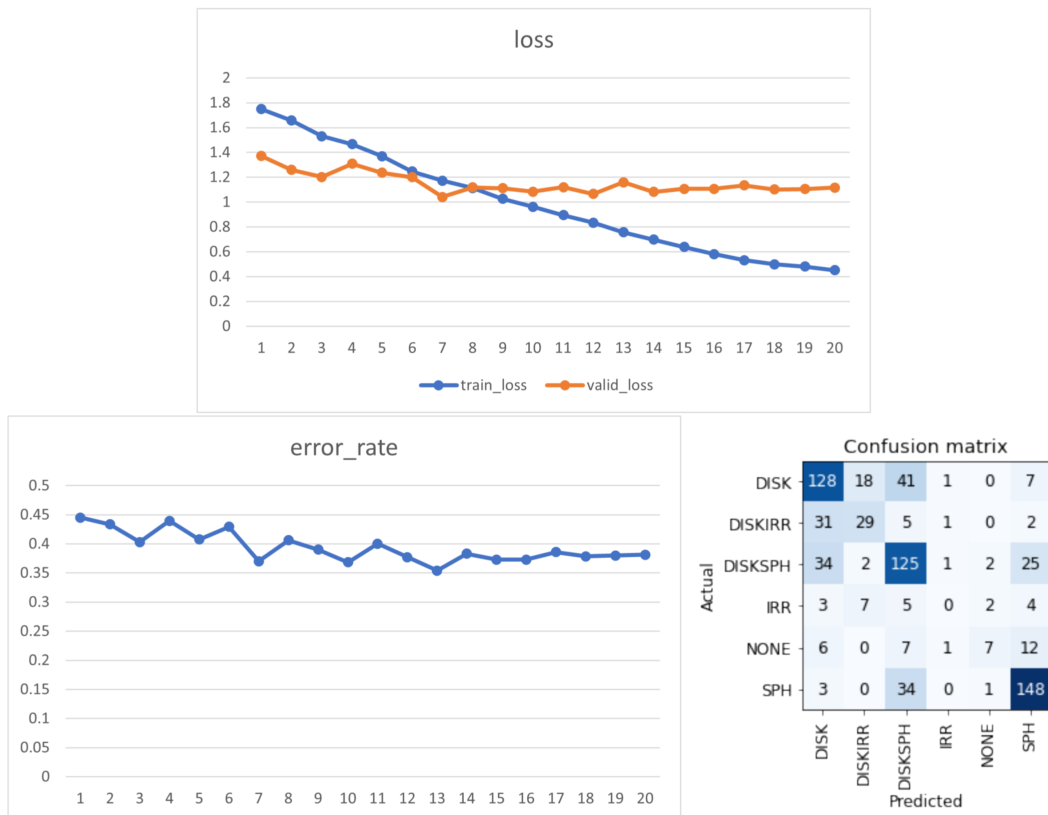


Figura 6.6: Clasificación con data augmentation y ResNet18

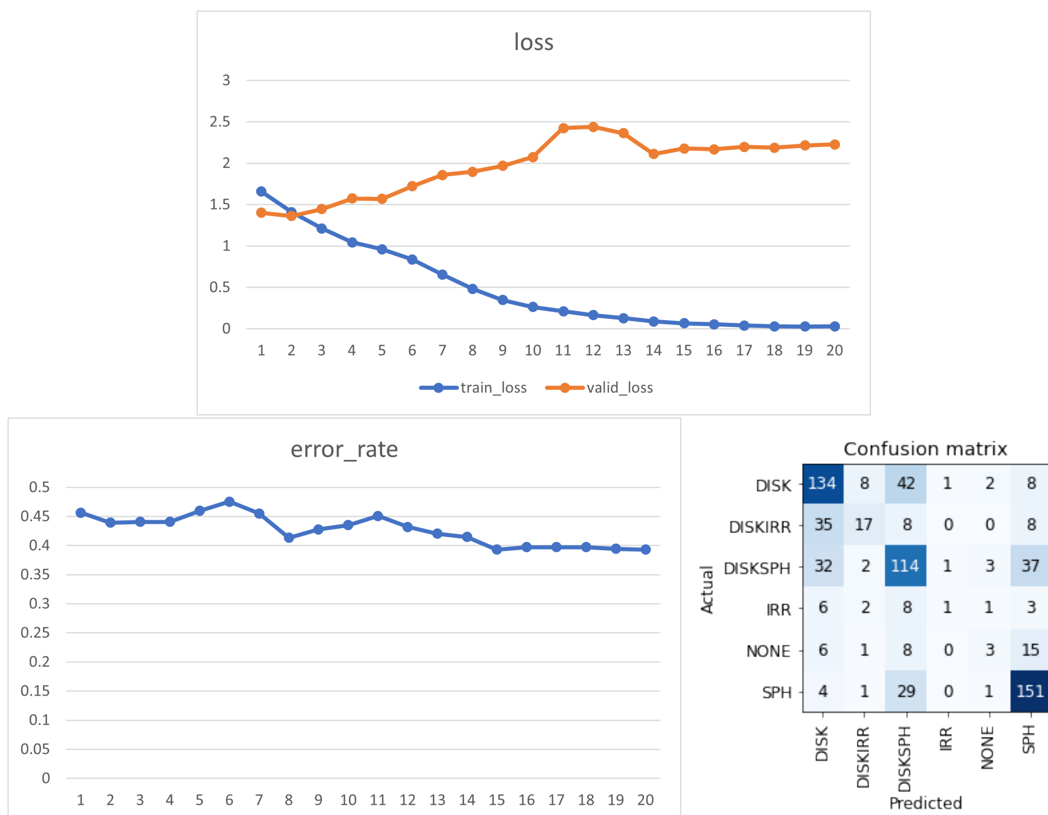


Figura 6.7: Clasificación sin data augmentation y ResNet34

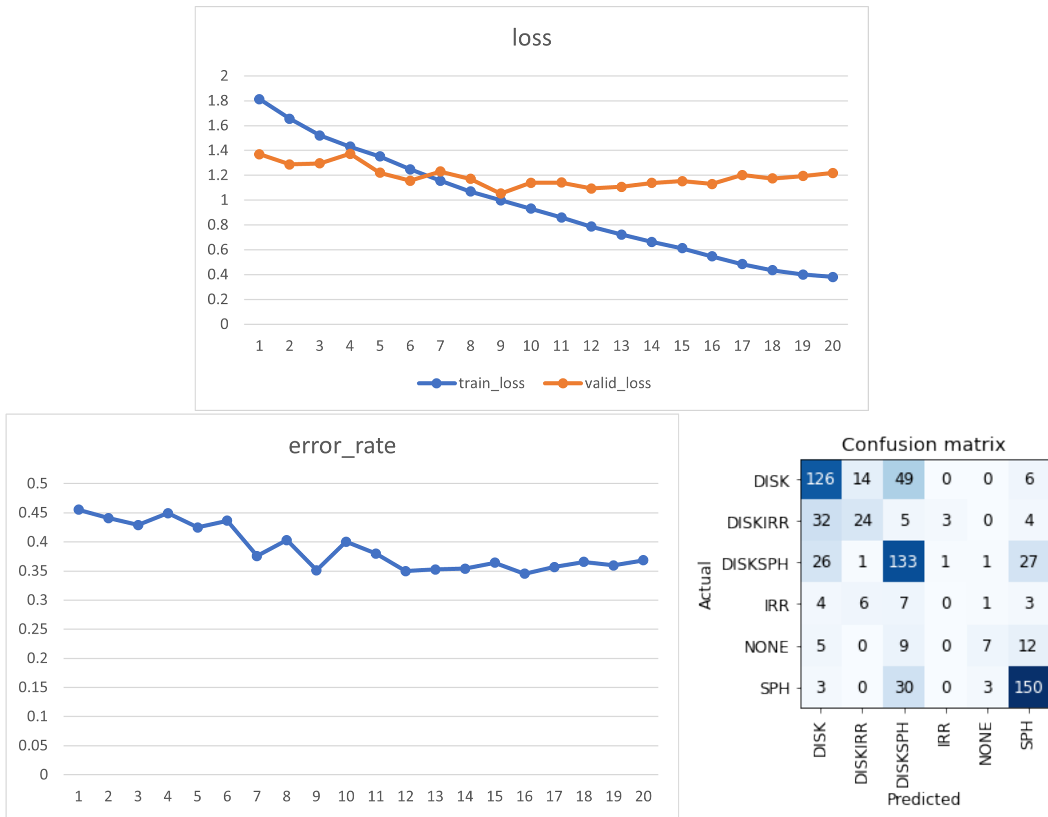


Figura 6.8: Clasificación con data augmentation y ResNet34

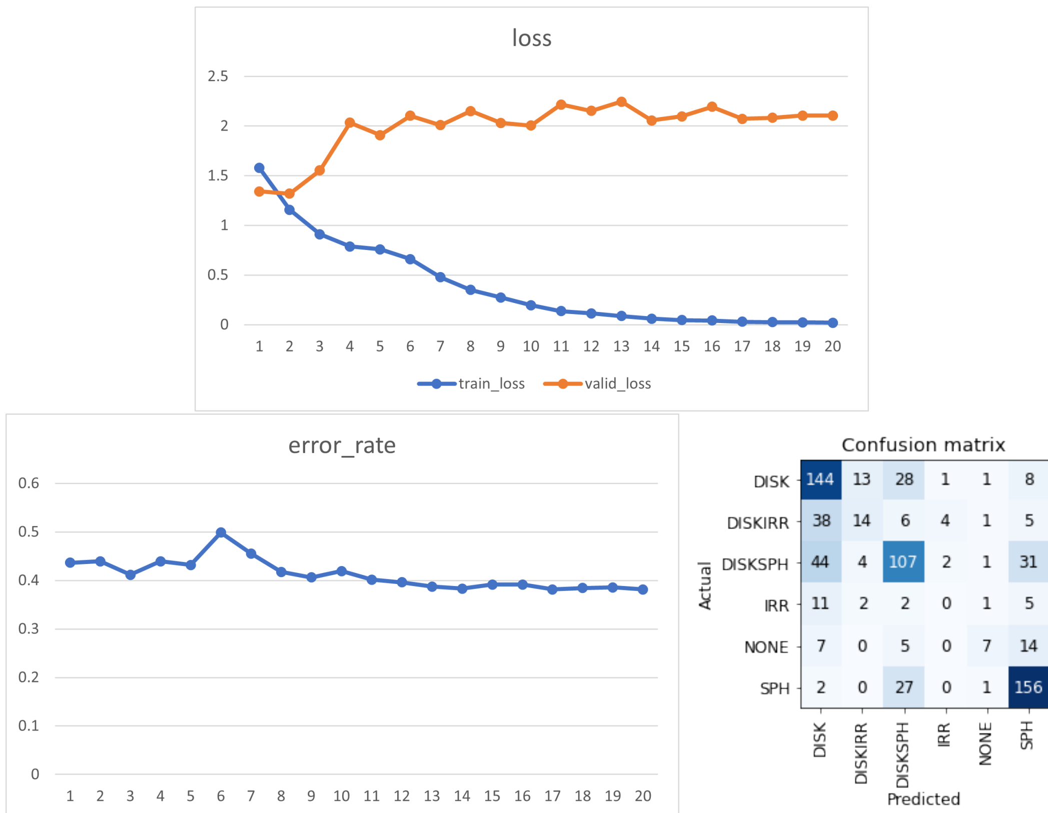


Figura 6.9: Clasificación sin data augmentation y ResNet50

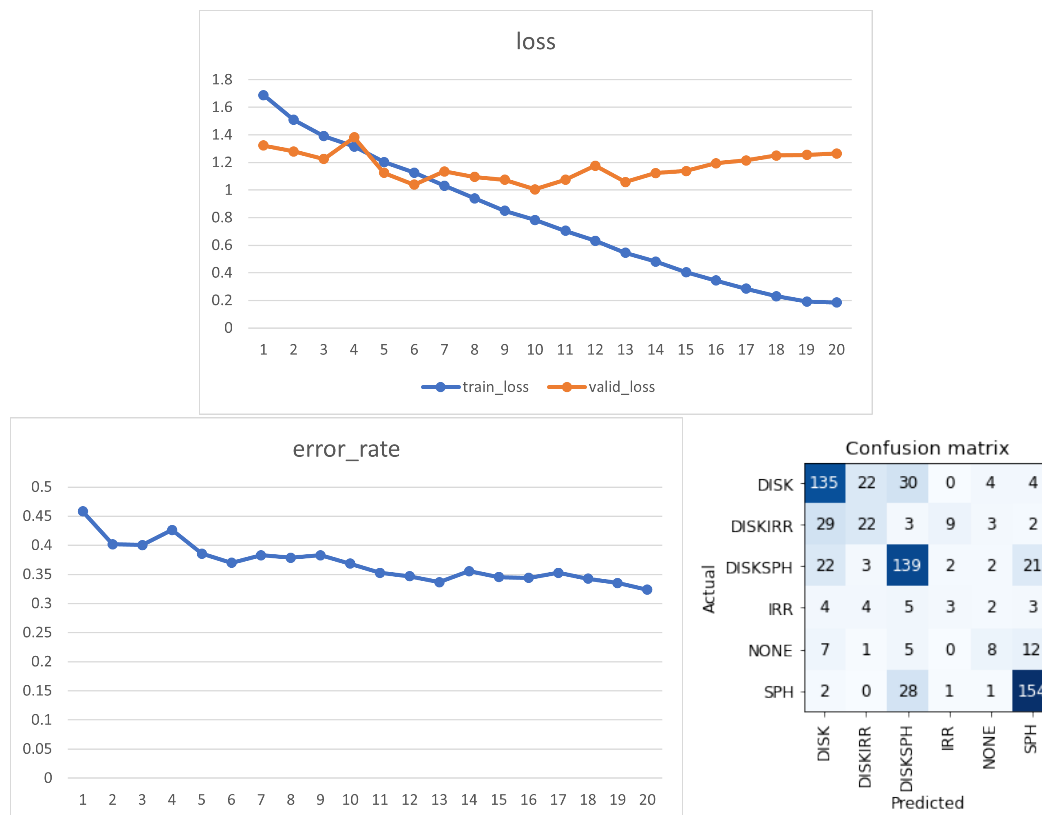


Figura 6.10: Clasificación con data augmentation y ResNet50

Sin data augmentation, el *train_loss* y el *valid_loss* se cortan en el epoch 2, lo que provoca que haya overfitting casi desde el principio, y con data augmentation, se cortan en los epochs 7 u 8, para cualquiera de los tres modelos de red. El *error_rate* es siempre unas décimas mejor con data augmentation que sin data augmentation con el mismo modelo. Además la resnet50 obtiene una pequeña mejoría respecto a la resnet34, y esta respecto a la resnet18. Con el dataset simple, por tanto, el mejor resultado es el de la resnet50 con data augmentation.

En cuanto a la matriz de confusión, se puede observar que las categoría en las que más se confunde es en DISK y SPH, que las predice como DISKSPH, y viceversa, DISKSPH las predice como DISK o como SPH, también pasa algo parecido aunque en menor medida con DISK y IRR.

6.5.3. Dataset doble

Ahora vamos a duplicar el número de imágenes del dataset mediante el sencillo programa del apéndice A.4.3, y siempre vamos a aplicar las técnicas de data augmentation que hemos usado en algunas pruebas del apartado anterior, ya que ahora tenemos duplicadas. Sería incorrecto alimentar a la red dos veces con las mismas imágenes porque solo conseguiría resultados buenos con las imágenes de este dataset, y sí cambiáramos de imágenes, el porcentaje de error aumentaría. Resultados:

- **Resnet18**: Figura 6.11
- **Resnet34**: Figura 6.12
- **Resnet50**: Figura 6.13

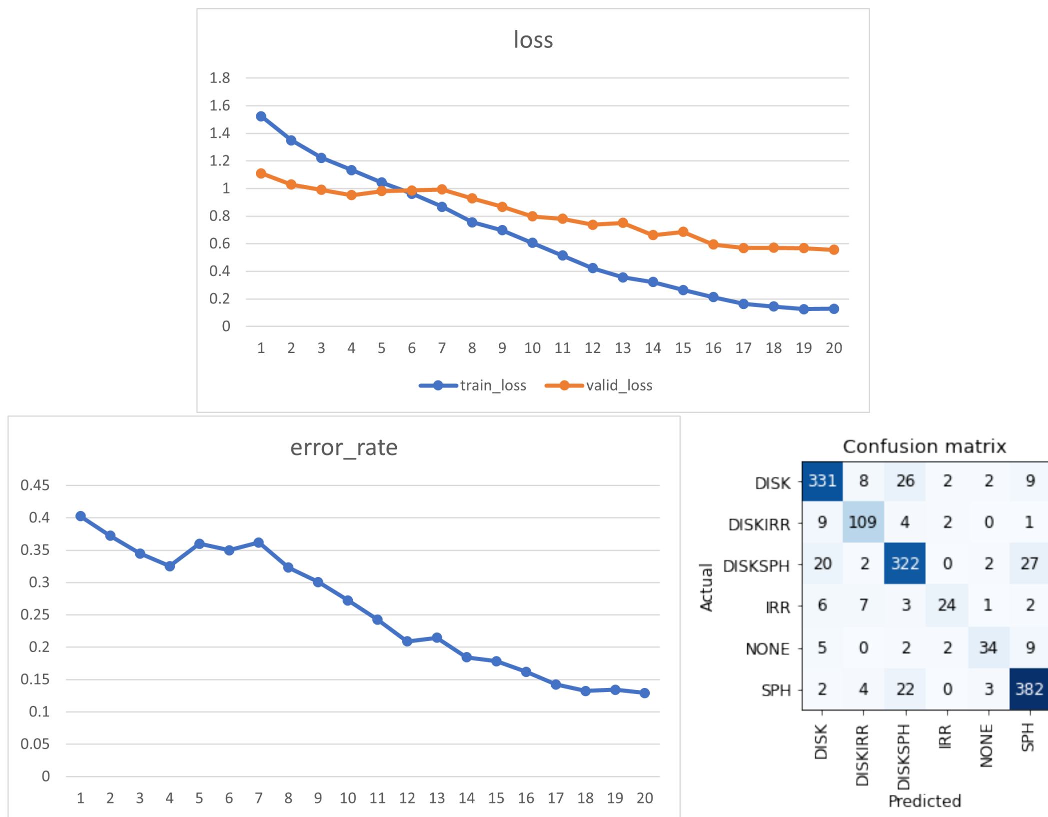


Figura 6.11: Clasificación con dataset doble y ResNet18

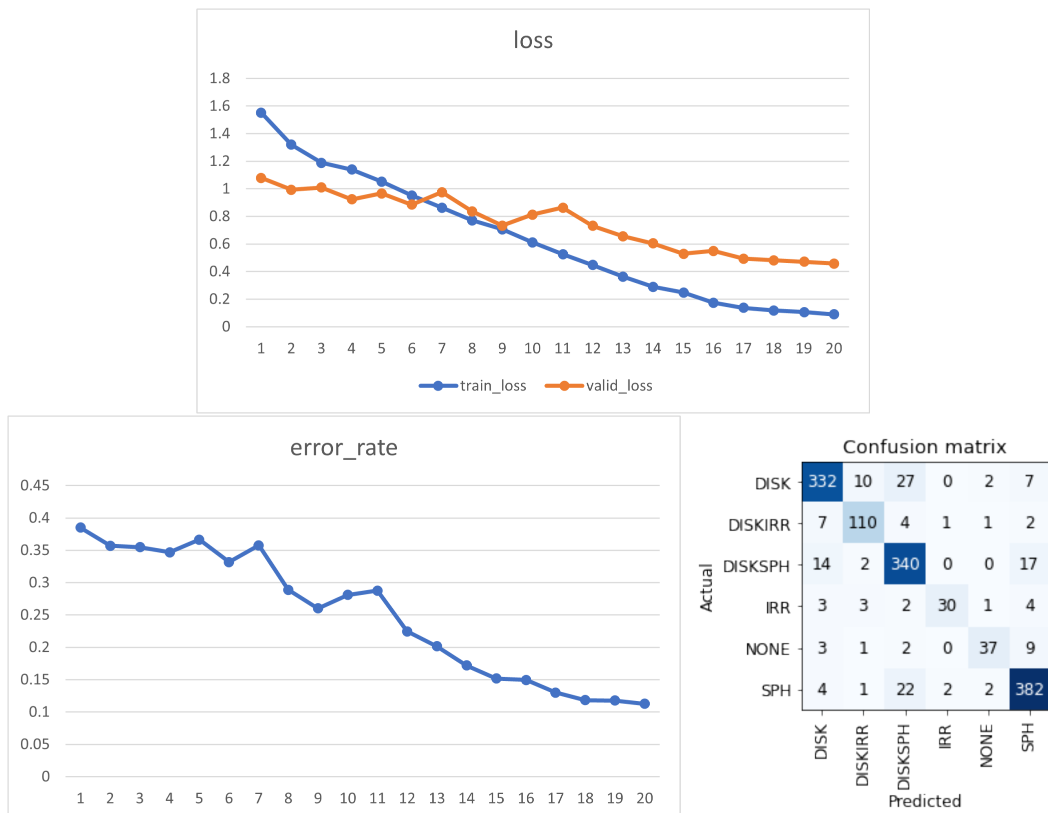


Figura 6.12: Clasificación con dataset doble y ResNet34

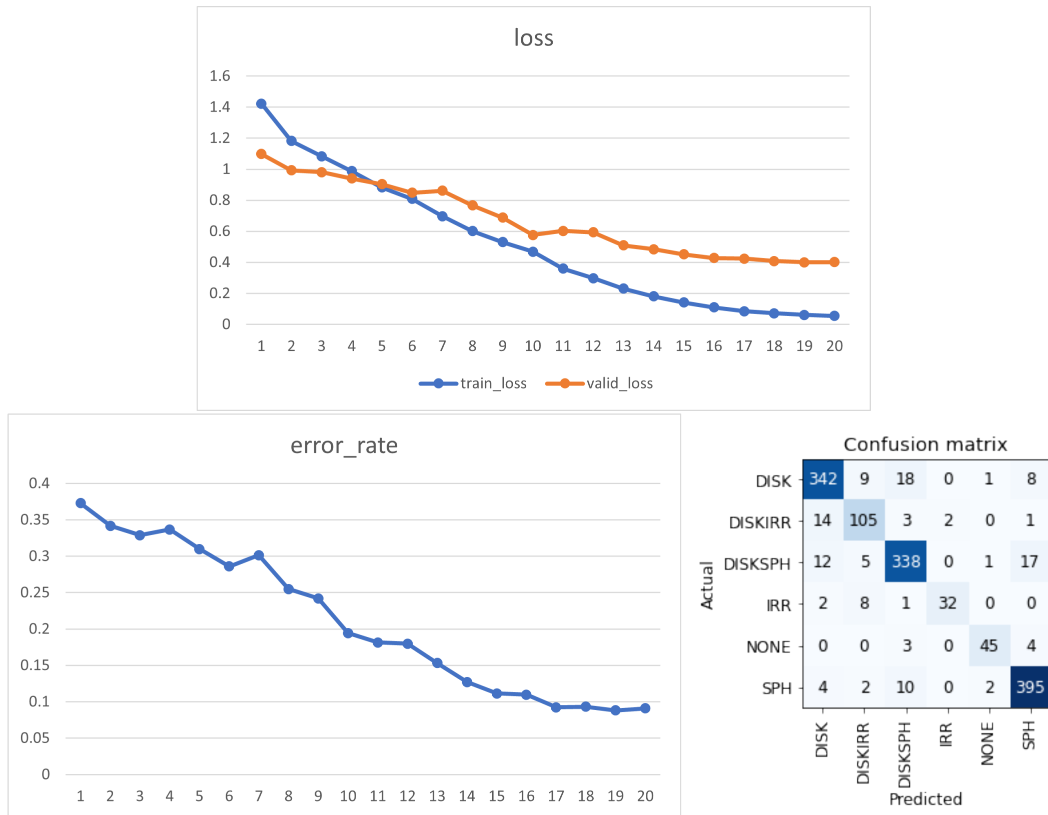


Figura 6.13: Clasificación con dataset doble y ResNet50

Aquí se puede ver una clara mejoría en todos los aspectos respecto al dataset simple. El *error_rate* con la resnet50 consigue llegar por debajo de 0.1. En la matriz de confusión se puede ver que, incluso en cifras absolutas, hay menos errores que con el dataset simple.

6.5.4. Dataset triple

Utilizando el mismo programa que en el apartado anterior, vamos a triplicar el dataset y aplicar las mismas técnicas de data augmentation. Resultados:

- **Resnet18**: Figura 6.14

- **Resnet34**: Figura 6.15

- **Resnet50**: Figura 6.16

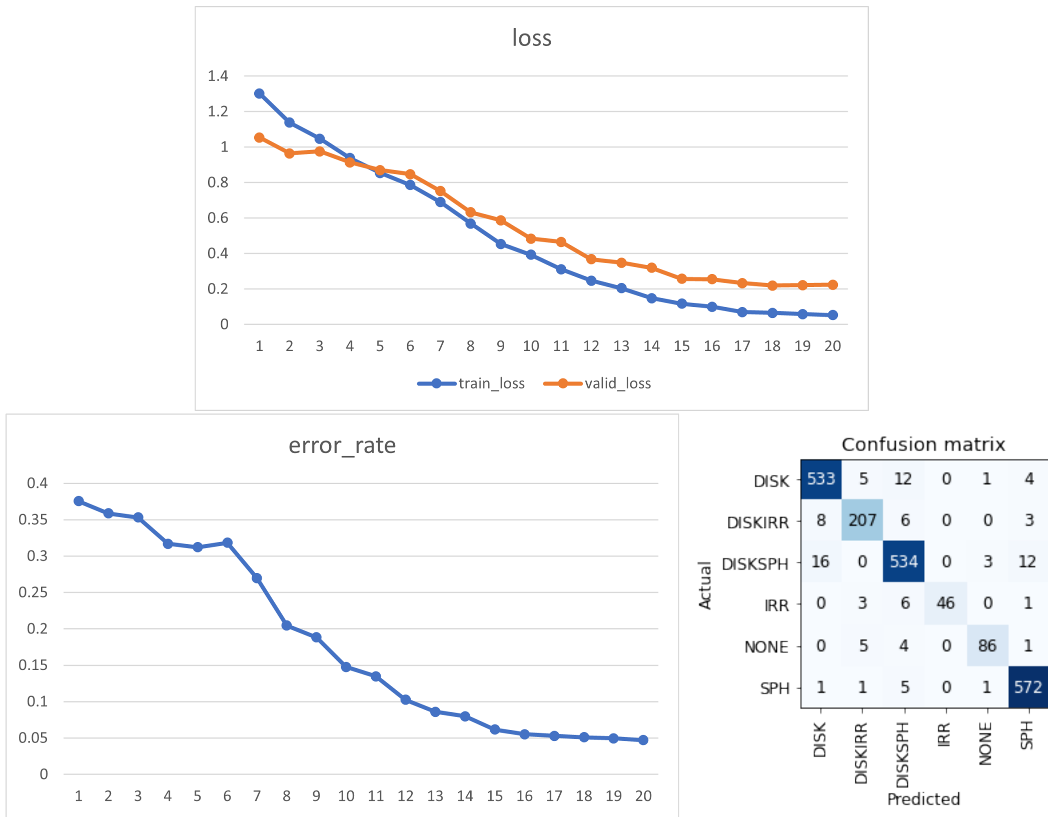


Figura 6.14: Clasificación con dataset triple y ResNet18

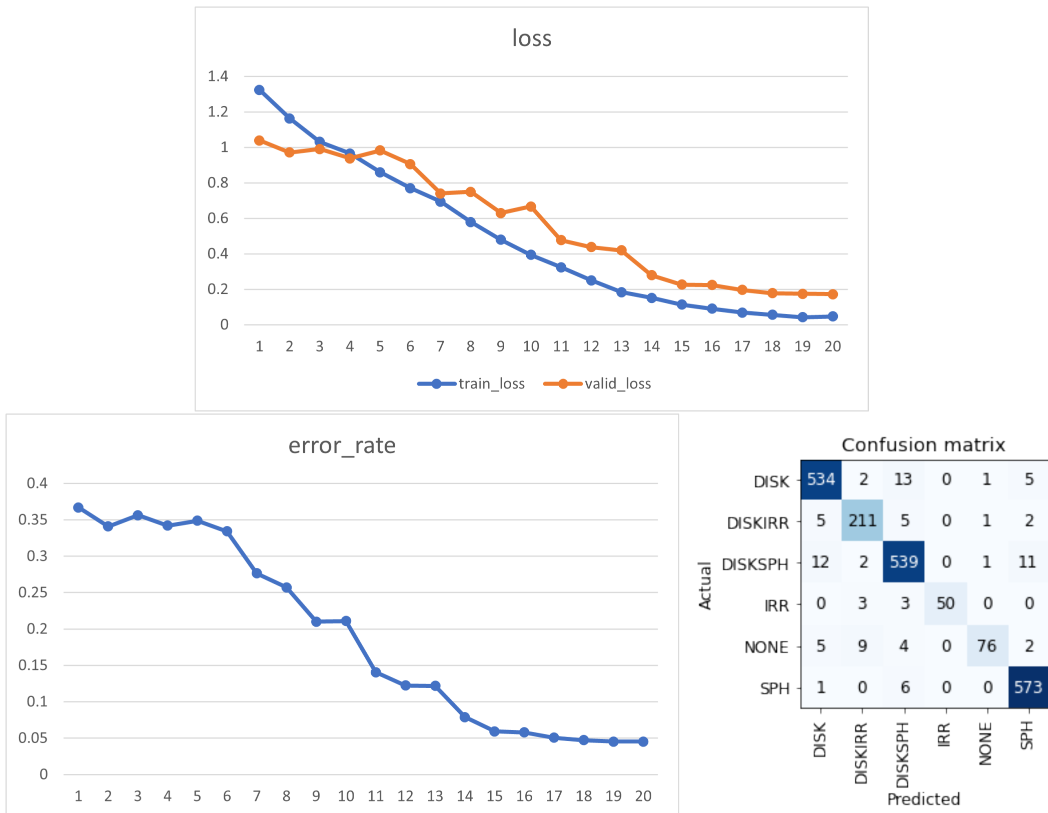


Figura 6.15: Clasificación con dataset triple y ResNet34

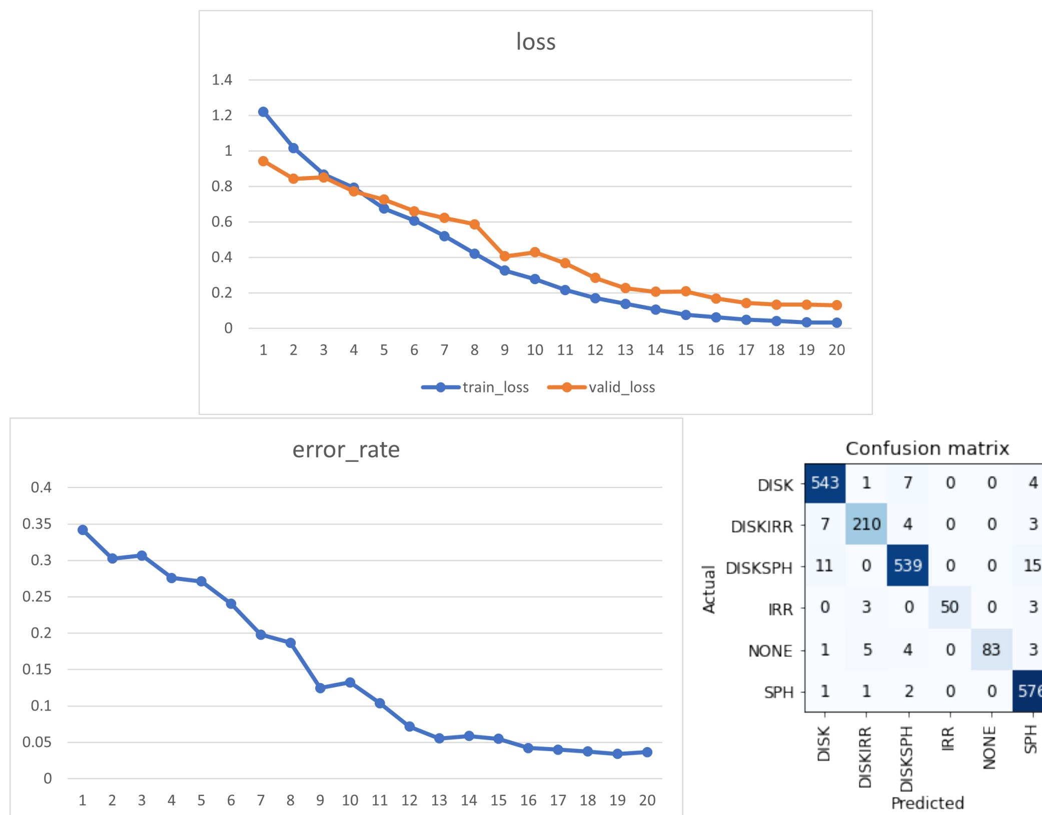


Figura 6.16: Clasificación con dataset triple y ResNet50

Con el triple de imágenes se obtiene otra mejoría en los resultados con respecto al dataset doble. El *error_rate* baja por debajo del 0.05 con cualquiera de los tres modelos de red, y el *train_loss* y *valid_loss* van evolucionando casi a la par, lo que indica que casi no hay overfitting.

6.5.5. Conclusiones

No hay diferencia real entre usar una ResNet18, una ResNet34 o una ResNet50 en este problema concreto. A igualdad del resto de parámetros, la mejoría que produce una ResNet50 respecto a una ResNet34, o esta respecto a una ResNet18 es mínima, y habría que valorar si merece la pena invertir el tiempo extra que necesitan las redes más profundas para entrenar.

El aspecto más relevante que se ha observado es que lo que marca la diferencia es aumentar el número de imágenes del dataset. Con el dataset original, el overfitting comienza muy pronto y es muy elevado, con el duplicado, el overfitting comienza más tarde y se reduce mucho, y con el triplicado, se reduce un poco más que con el duplicado. Si se siguiera aumentando el dataset, el *error_rate* y el *loss* seguiría bajando, pero el entrenamiento cada vez tardaría más. Una tasa de error de 0.05 en una clasificación de 6 categorías es un resultado bastante bueno.

Sin embargo, estos resultados son engañosos cuando duplicamos y triplicamos el dataset, ya que estamos introduciendo parte del conjunto de validación en el conjunto de entrenamiento. Aunque también estemos aplicando transformaciones a las imágenes cuando duplicamos y triplicamos el dataset, las imágenes siguen siendo muy parecidas y por eso la red obtiene tan buenos resultados. Es decir, está viendo en validación casi las mismas imágenes qu ene entrenamiento. Por lo tanto, la práctica de multiplicar el dataset de esta forma no es una forma válida de medir la efectividad del modelo, porque cuando le alimentemos con datos diferentes a los que nos han servido como entrenamiento, los resultados pueden no ser tan buenos como los que hemos

obtenido durante el entrenamiento.

6.6. Regresión de valores morfológicos

6.6.1. Descripción

En este experimento se entrenará a una red neuronal para que obtenga 5 valores morfológicos a partir de la imagen de una galaxia. Es decir, el input es una imagen de 200x200 píxeles, y el output son 5 valores en punto flotante. Los valores que corresponden a cada imagen vienen en el fichero `galaxies_main_props_with_struct_params_and_no_bad_objects_and_hudf_flags_and_morphs_and_z_flags6.cat`. Las columnas de este fichero que nos interesan son f_spheroid, f_disk, f_irr, f_ps y f_unc, que corresponden a cada una de las categorías morfológicas.

Este fichero .cat tiene muchas columnas que no son necesarias en este experimento, por lo tanto, vamos a crear un programa en Python (Notebook del apéndice A.5.1) que cree un nuevo fichero csv con las 5 columnas de valores morfológicos y con una columna extra que indica la ruta a la imagen que corresponde a los valores de esa fila.

Una vez que hemos creado el csv, pasamos al notebook que contiene el código para este experimento (Notebook A.5.2). Para manipular el fichero csv se utiliza la biblioteca Pandas, que contiene la clase DataFrame. Con las siguientes líneas leemos el fichero y mostramos las 5 primeras filas para comprobar que se ha leído bien:

```
df=pd.read_csv(path_csv, delim_whitespace=True)
df.head()
```

Fragmento de código 6.11: Cargar CSV en DataFrame

Las funciones de input y output de la red neuronal serían estas:

```
def get_x(r): return r['galPath']
def get_y(r): return (tensor([r['f_spheroid'], r['f_disk'], r['f_irr'], r['
    f_ps'], r['f_unc']]))
```

Fragmento de código 6.12: Definir funciones de etiquetado para regresión morfológica de galaxias

El input es la columna que contiene la ruta a la imagen y el output son las 5 columnas con los valores morfológicos. Es una forma muy rápida y simple de obtener los datos.

En este experimento, cuando el dataset sea el original, vamos a hacer pruebas con y sin trasnformaciones. Cuando dupliquemos y tripliquemos el dataset, vamos a usar siempre transformaciones, que serán las mismas que en el experimento de clasificación anterior:

```
tfms1=[Normalize.from_stats(mean=0.0, std=1.0), Rotate(p=1.0, draw=
    RotacionAleatoria), RuidoGaussiano]
```

Fragmento de código 6.13: Definir transformaciones para regresión morfológica de galaxias

El DataBlock es el siguiente:

```
galaxias= DataBlock(blocks=(ImageBlock, RegressionBlock),
        splitter = RandomSplitter(valid_pct=0.2, seed=1),
        batch_tfms=tfms1,
        get_x=get_x,
        get_y=get_y)
```

Fragmento de código 6.14: Definir DataBlock para regresión morfológica de galaxias

Al DataLoaders se le pasa el DataFrame del fichero csv, en vez de la ruta al directorio de las imágenes, como en el experimento de clasificación anterior:

```
dls = galaxias.dataloaders(df, num_workers=0)
```

Fragmento de código 6.15: Definir DataLoaders para regresión morfológica de galaxias

A continuación se muestran, respectivamente, las funciones de coste y de métrica que hemos definido nosotros mismo para este experimento:

```
def DistanciaLoss(preds, target):
return (preds-target).abs().mean()

def DistanciaLossLabelSmoothing(preds, target):
    target = target+(torch.rand(target.size(1), device='cuda')-0.5)*0.1
    return (preds-target).abs().mean()
```

Fragmento de código 6.16: Definir funciones de coste y métrica para regresión morfológica de galaxias

En estas funciones, `preds` y `targets` son 2 tensores de tamaño 5 que representan las salidas y las variables dependientes respectivamente. Por lo tanto estas funciones calculan la diferencia media absoluta entre los valores predichos por la red y los valores reales. La única diferencia entre las dos es que a la que usaremos como función de loss le aplicamos un poco de *label smoothing*, una técnica que se explicó anteriormente.

Se van a realizar pruebas con 3 tipos de modelos de red neuronal: ResNet18, ResNet34 y ResNet50.

Para todas las pruebas se entrenará durante 20 epochs y se mostrará:

1. Un gráfico lineal con la evolución de los valores de la función de coste en entrenamiento y en validación frente a los epochs.

2. Un gráfico lineal con la evolución de los valores de la métrica frente a los epochs.

Los gráficos se generarán con Microsoft Excel a partir de los csv de resultados del entrenamiento que se generarán con el callback:

```
CSVLogger(fname="NOMBRE_FICHERO_CSV", append=true)
```

6.6.2. Dataset simple

En este experimento el número de imágenes es el original. No se han multiplicado el número de imágenes. Resultados:

- **Resnet18 sin data augmentation**: Figura 6.17
- **Resnet34 sin data augmentation**: Figura 6.18
- **Resnet50 sin data augmentation**: Figura 6.19
- **Resnet18 con data augmentation**: Figura 6.20
- **Resnet34 con data augmentation**: Figura 6.21

- **Resnet50 con data augmentation**: Figura 6.22

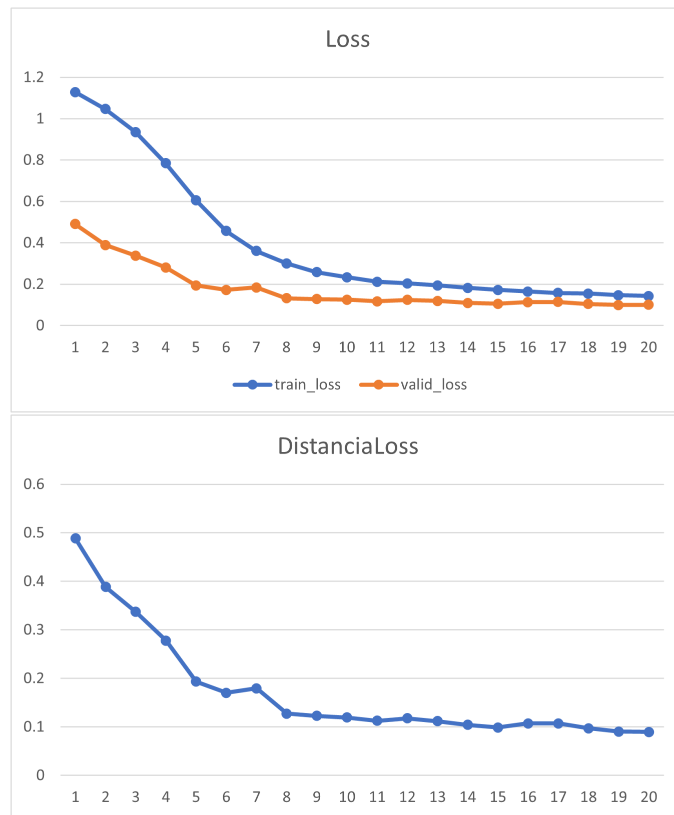


Figura 6.17: Regresión con dataset simple y ResNet18 sin data augmentation

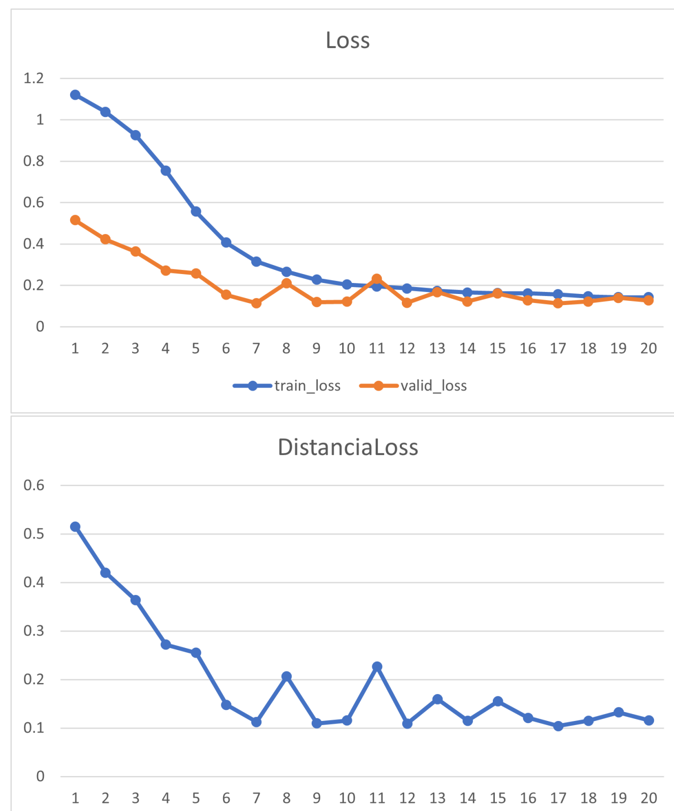


Figura 6.18: Regresión con dataset simple y ResNet34 sin data augmentation

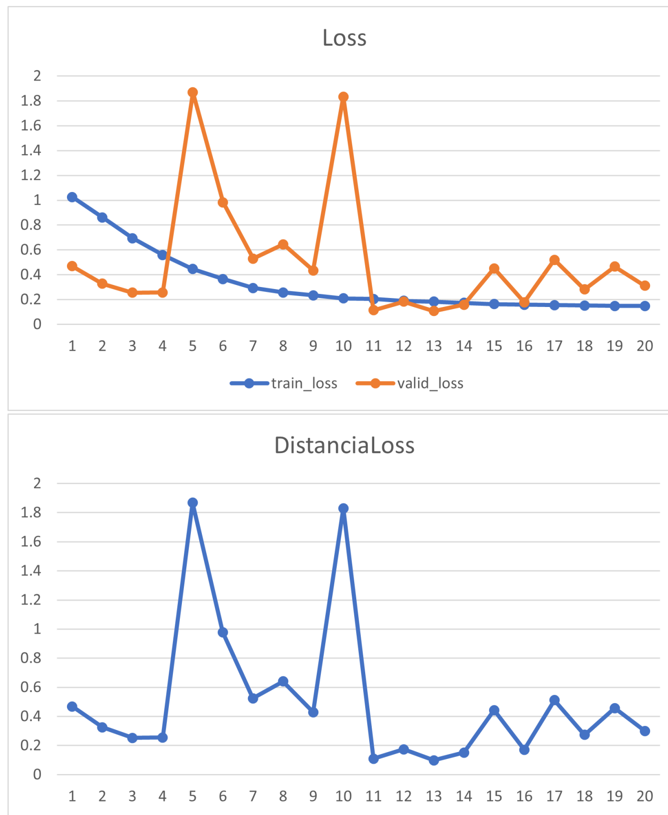


Figura 6.19: Regresión con dataset simple y ResNet50 sin data augmentation

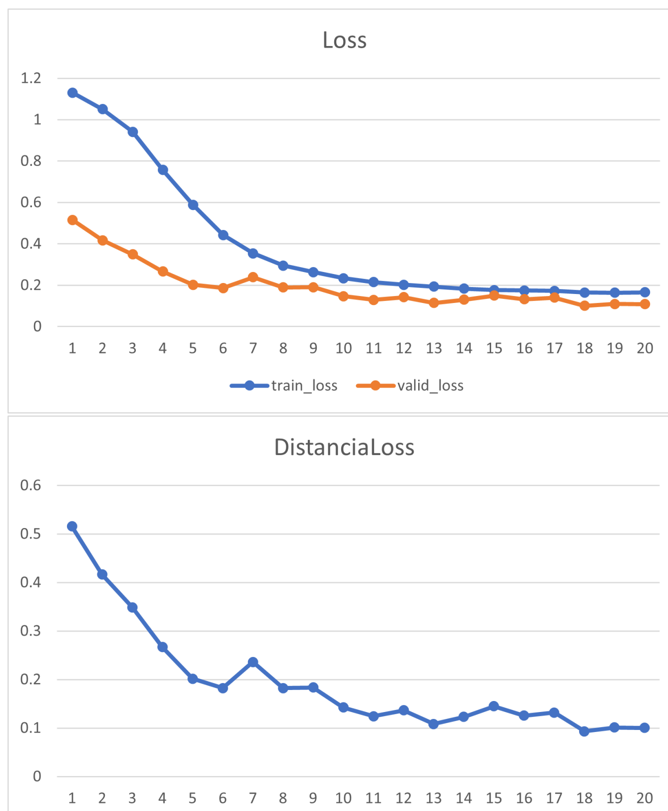


Figura 6.20: Regresión con dataset simple y ResNet18 con data augmentation

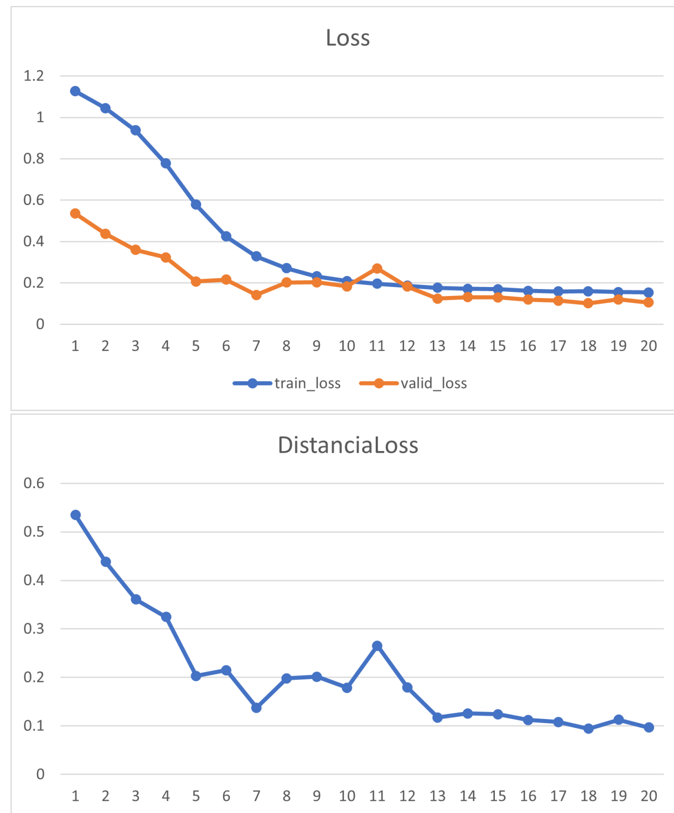


Figura 6.21: Regresión con dataset simple y ResNet34 con data augmentation

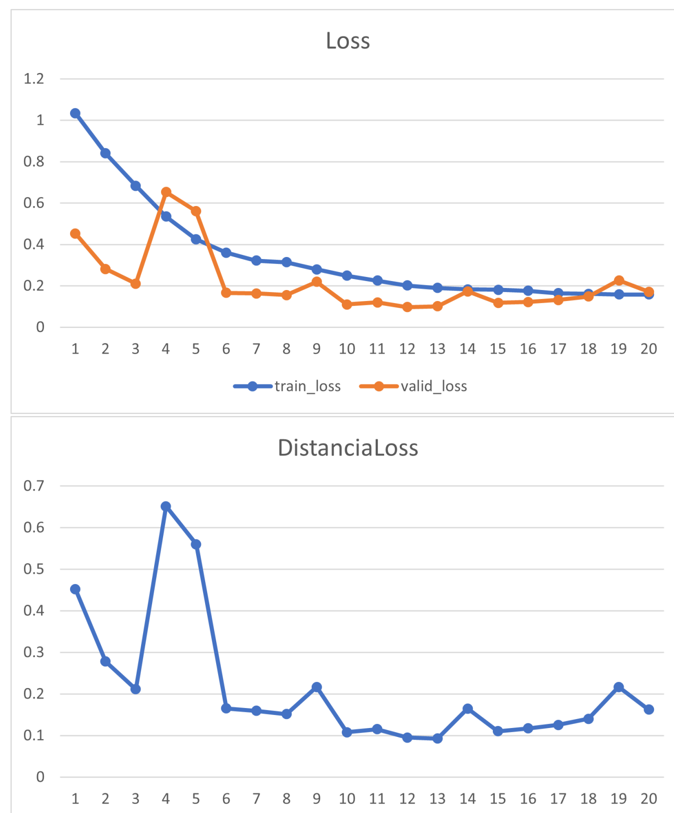


Figura 6.22: Regresión con dataset simple y ResNet50 con data augmentation

Con ResNet18 cpn y sin augmentation no hay nada de overfitting, ya que el train_loss y el valid_-loss se mantienen casi a la par durante todo el entrenamiento, y los resultados de la métrica son muy buenos, por debajo de 0.1. Con ResNet34 los resultados son muy similares, pero hay alguna

irregularidad en el valid_loss y en la métrica. Estas irregularidades se acentúan todavía más con ResNet50.

6.6.3. Dataset doble

Ahora vamos a generar un nuevo csv en el que cada línea aparezca duplicada utilizando el mismo código que para generar el csv de forma normal(Apéndice A.5.1), pero esta vez especificando el parámetro `numCopias=1`. De esta forma, podemos hacer que la red tome dos veces el mismo fichero de imagen durante el entrenamiento sin necesidad de duplicar las imágenes como en el experimento anterior.

- **Resnet18**: Figura 6.23
- **Resnet34**: Figura 6.24
- **Resnet50**: Figura 6.25

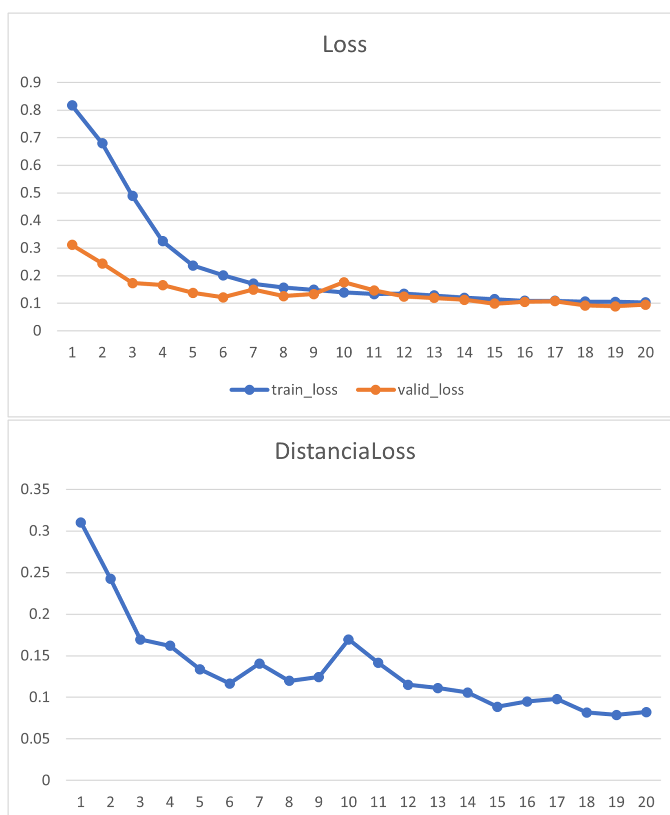


Figura 6.23: Regresión con dataset doble y ResNet18

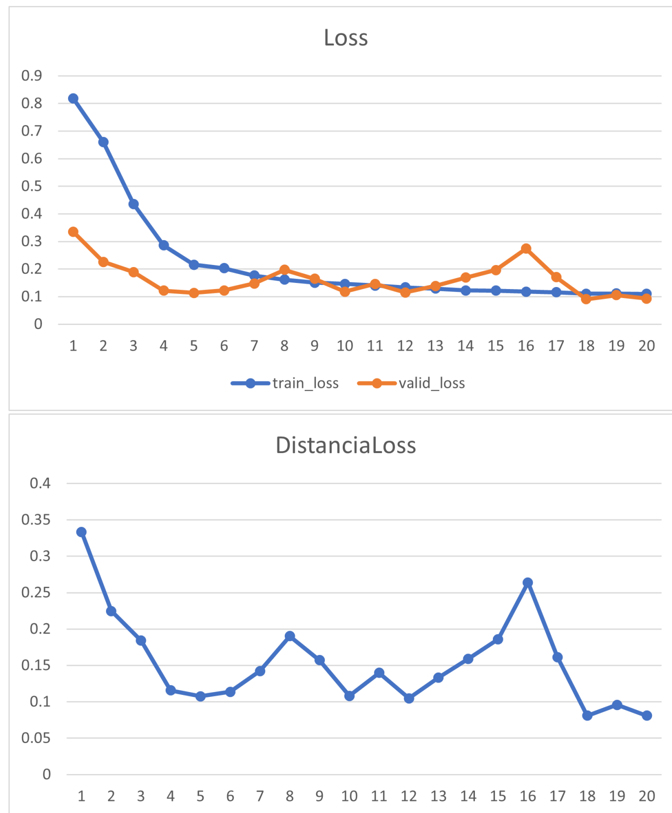


Figura 6.24: Regresión con dataset doble y ResNet34

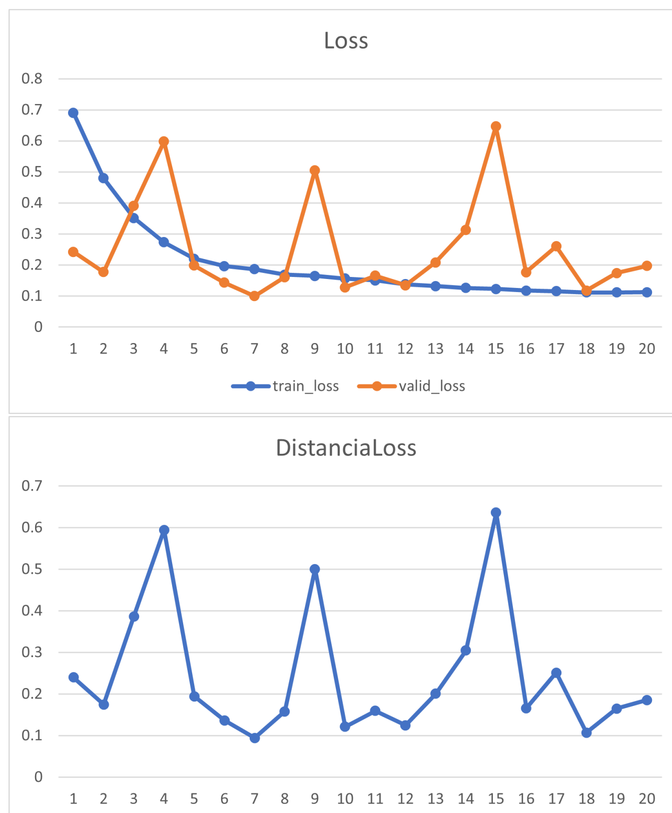


Figura 6.25: Regresión con dataset doble y ResNet50

Con el dataset duplicado no hay mejoras notables en loss y la métrica, al contrario que lo qe pasaba en el experimento de clasificación. Además las irregularidades en los gráficos del valid_loss y de la métrica con ResNet34 y ResNet50 se acentúan todavía más.

6.6.4. Dataset triple

Ahora vamos a generar un nuevo csv en el que cada línea aparezca triplicada utilizando el mismo procedimiento que en el apartado anterior.

- **Resnet18**: Figura 6.26

- **Resnet34**: Figura 6.27

- **Resnet50**: Figura 6.28

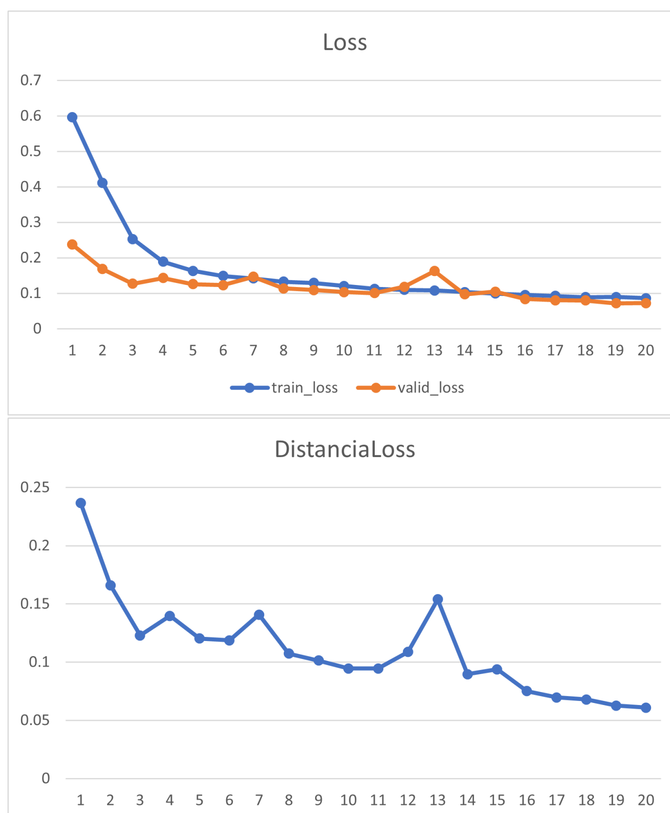


Figura 6.26: Regresión con dataset triple y ResNet18

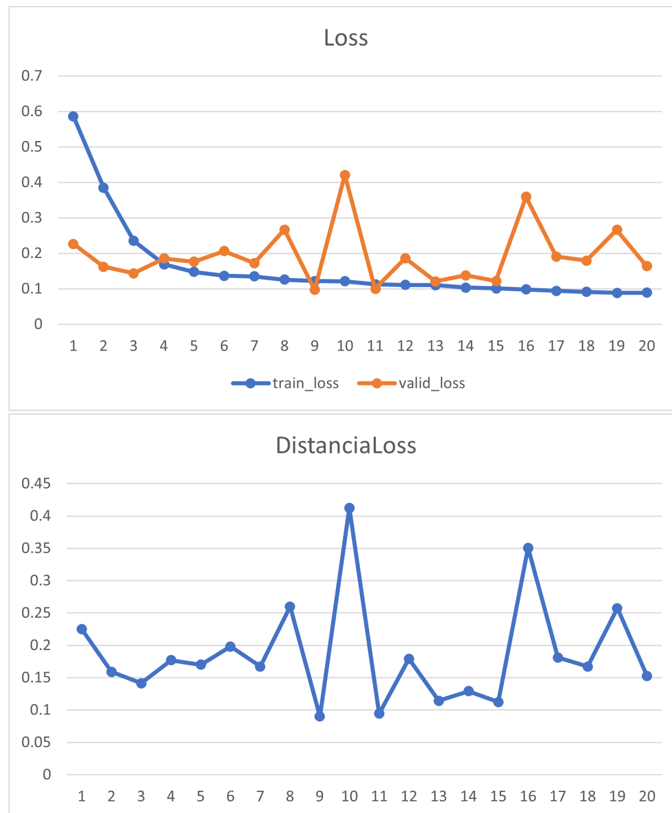


Figura 6.27: Regresión con dataset triple y ResNet34

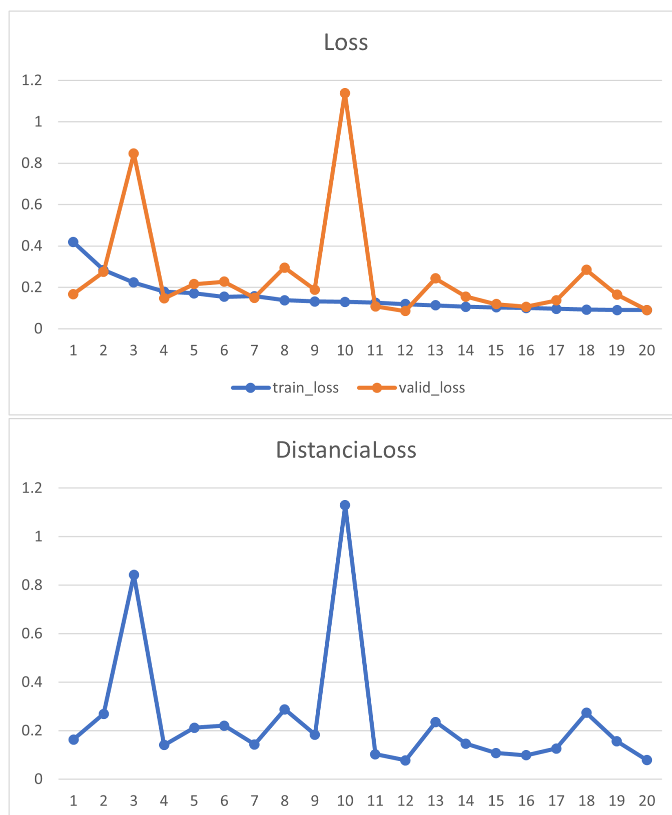


Figura 6.28: Regresión con dataset triple y ResNet50

Con el dataset triplicado, ocurre los mismo que con el duplicado. No hay ninguna mejora y las irregularidades con ResNet34 y ResNet50 son cada vez más grandes.

6.6.5. Conclusión

En este experimento el dataset simple ya obtiene muy buenos resultados y no hay casi diferencia entre mantener el dataset original y duplicarlo o triplicarlo. Aún así, al igual que se ha explicado en la conclusión del experimento de clasificación por categorías, no es recomendable aumentar el dataset de esta forma, ya que estamos introduciendo algunos datos iguales en el conjunto de entrenamiento y en el de validación, y por lo tanto, los resultados de efectividad de la red no son válidos, porque no estamos midiendo la precisión de la red de forma correcta. Aunque en este experimento de regresión, puede tener menos impacto porque estamos aplicando *label smoothing*.

La mejor arquitectura de red de las que se han probado es la ResNet18, porque es la no provoca saltos bruscos en la evolución del valid_loss y de la métrica. La ResNet34 y la ResNet50, en este problema concreto, sufren de overfitting solo en algunos epochs, eso, sumado a que cada epoch de estas redes requiere más tiempo que en la ResNet18, hace que no sean buenas elecciones para este experimento.

Capítulo 7

Detección de bordes de galaxias con segmentación

7.1. Introducción

En este capítulo se realizarán experimentos con redes neuronales para que aprendan a detectar, en imágenes de galaxias, los bordes de estas mediante la técnica de segmentación. Partiremos de un conjunto de datos inicial proporcionado por el profesor Fernando Buitrago, y los transformaremos en un dataset entendible por la red neuronal.

7.2. Conjunto de datos

7.2.1. Conjunto de datos inicial

El conjunto de datos inicial está formado por imágenes .fits de cada galaxia en 4 filtros diferentes: H, I, J y V. y por máscaras .fits para cada una de estas galaxias. El objetivo es juntar los distintos filtros de cada galaxia en una sola imagen RGB en formato .png, y transformar cada máscara en un fichero .png. Para crear las imágenes RGB solo necesitaremos 3 filtros(I, J y H), así que los ficheros de filtro V quedarán sin usar.

Los valores de cada píxel de las máscaras indican el tipo de objeto que hay en ese píxel en la imagen de la galaxia correspondiente:

- **1**: Indica que ese píxel no corresponde a ningún objeto relevante.
- **2**: Indica que ese píxel corresponde a la galaxia principal de la imagen.
- **3**: Indica que ese píxel corresponde a una galaxia vecina.

En las imágenes pueden aparecer varias galaxias, y la red debe aprender a distinguir entre la principal y las que puedan aparecer alrededor. Estos valores se indican se le indican a la red neuronal mediante un fichero .txt.

7.2.2. Conjunto de datos final

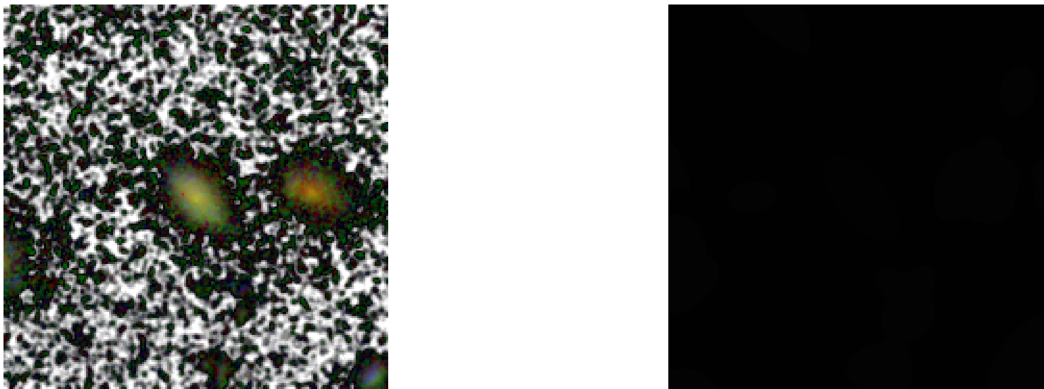
Para transformar los .fits en ficheros que la red neuronal entienda se ha usado el código descrito en el listado de código A.9 del apéndice. Este código es similar al usado en el capítulo anterior

para generar las imágenes para clasificación pero con algunas diferencias:

- La imagen rgb de la galaxia ya no se guarda con el parámetro `origin=lower`, que hacía que el índice(0,0) del array que representa la imagen estuviera en la esquina inferior izquierda de la imagen. Ahora es mejor quitarlo y que el origen del array esté por defecto en la esquina superior izquierda, para que coincidan con las máscaras.

- La máscara se guarda usando funciones de la biblioteca PIL, ya que con Matplotlib, los valores de los colores no se guardaban bien.

El resultado final es un directorio llamado `images_rgb` con 1053 imágenes de 200x200 píxeles en RGB y un directorio llamado `labels` que contiene 1053 máscaras, una para cada imagen. En la figura 7.1 se muestra un ejemplo de una galaxia y su máscara. En la máscara, los colores no son fácilmente distinguibles por el ojo humano, ya que solo hay una diferencia de una unidad entre los valores de los distintos colores. En la figura 7.2 se han cambiado los colores para apreciar mejor las diferencias.



(a) Una imagen en RGB de una galaxia

(b) La máscara correspondiente a 7.1a

Figura 7.1: Una galaxia y su máscara

Figura 7.2: La máscara correspondiente a 7.1a con colores legibles para el ojo humano

7.3. Modelo: U-Net

Para realizar segmentación se utiliza un modelo de red particular llamado *red en U* o *U-Net*, que fue introducida por primera vez en 2015 por Olaf Ronneberger et al. en su artículo

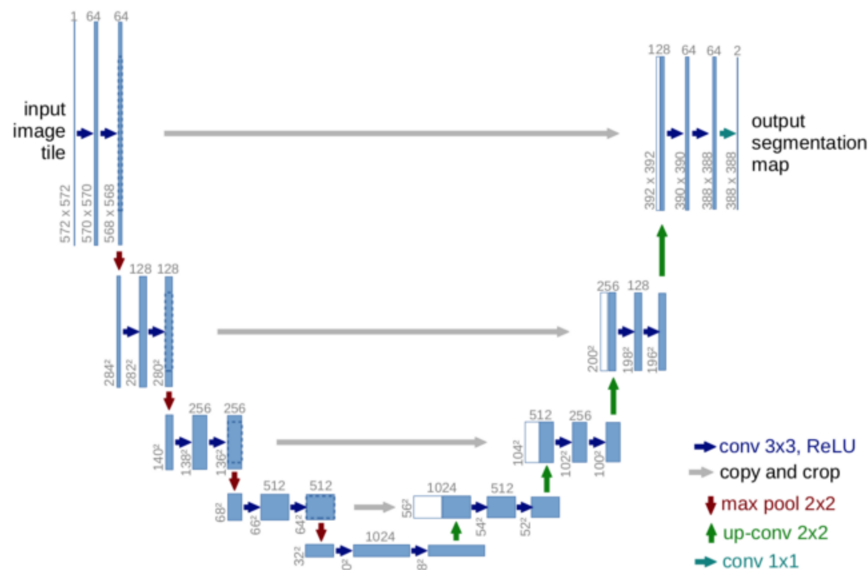


Figura 7.3: Arquitectura de una U-Net [39]

U-Net: Convolutional Networks for Biomedical Image Segmentation[34] y aplicada para procesar imágenes biomédicas, donde el objetivo no solo es necesario saber si hay o no una afección, sino saber además en qué zona se encuentra.

La U-Net es capaz de detectar bordes en la imagen realizando una clasificación de cada píxel, ya que el tamaño de la entrada y de la salida es el mismo. En la figura 7.3 se ilustra la arquitectura de una U-Net. Como su nombre indica, tiene forma de U, y consiste en dos partes principales y simétricas. La parte izquierda es el camino de contracción, que va reduciendo la imagen de entrada mediante convoluciones, y la parte derecha es el camino de expansión, que va aumentando el tamaño de la imagen hasta alcanzar el original mediante convoluciones traspuestas.

7.4. Segmentación mediante U-Net

7.4.1. Descripción

Primero cargamos el fichero de texto que contiene la descripción de los valores de los píxeles de las máscaras y lo mostramos para comprobar que lo ha leído bien:

```
codes = np.loadtxt(path_codes, dtype=str)
codes
```

Fragmento de código 7.1: Cargar fichero de valores de máscara

A continuación debemos definir la función de etiquetado.A partir de la ruta de la imagen de la galaxia, esta función devuelve la ruta a la máscara correspondiente.

```
def label_func(fn):
nombre=fn.stem
extension=fn.suffix
return path_mascaras+"/"+nombre+"_M"+extension
```

Fragmento de código 7.2: Función de etiquetado

Después definimos el DataBlock. El bloque de entrada es un ImageBlock y el bloque de salida es un MaskBlock al que se le pasa como parámetro el fichero de los códigos para los píxeles de la máscara que se ha leído anteriormente. Los datos de entrada se obtienen mediante la ruta que se especificará en el DataLoaders, y la salida se

```
galaxias = DataBlock(blocks=(ImageBlock, MaskBlock(codes)),
                    get_items = get_image_files,
                    get_y = label_func,
                    splitter=RandomSplitter()
                    )
```

Fragmento de código 7.3: DataBlock para segmentación de galaxias

Luego, se crea el DataLoaders, especificando el path a las imágenes de las galaxias.

```
dls = galaxias.dataloaders(path_imagenes, path=".", bs=8, num_workers=0)
```

Fragmento de código 7.4: DataLoaders para segmentación de galaxias

Por último, creamos el learner y entrenamos:

```
learn = unet_learner(dls, resnet18, metrics=foreground_acc, cbs=CSVLogger(
    fname=rutaAlCsv, append=true))
learn.fine_tune(10)
```

Fragmento de código 7.5: Learner para segmentación de galaxias

La función de loss que estamos usando es la que se pone por defecto para este tipo de entrenamientos, que es FlattenedLoss of CrossEntropyLoss().La métrica que estamos usando es `foreground_acc`, que mide el porcentaje de píxeles que la red ha acertado al intentar segmentar las imágenes. Cuando hemos terminado el entrenamiento podemos hacer que nos muestre algunos resultados con la línea de código del listing 7.6. En el parámetro *max_n* especificamos el número de imágenes que queremos, aunque solo puede mostrar como máximo un número de imágenes igual al tamaño del batch con el que hayamos entrenado. Esta función muestra una imagen y su máscara superpuesta, y a su lado, la misma imagen pero con la máscara que la red neuronal ha predicho, de forma que podemos hacernos una idea aproximada de cómo de bien ha aprendido la red a segmentar.

```
learn.show_results(max_n=5)
```

Fragmento de código 7.6: Mostrar resultados de segmentación

Otra función interesante que podemos usar es la que se muestra en el listing 7.7. Al igual que en clasificación podíamos obtener un ClassificationInterpretation para crear una matriz de confusión, aquí podemos obtener un SegmentationInterpretation y obtener las imágenes con mayor loss del entrenamiento, es decir, las que peor ha predicho la red. El parámetro k de la función `plot_top_losses` sirve para indicar que queremos que muestre las k primeras imágenes con mayor loss.

```
interp = SegmentationInterpretation.from_learner(learn)
interp.plot_top_losses(k=3)
```

Fragmento de código 7.7: Mostrar top losses de segmentación

Se ha entrenado siempre durante 20 epochs y realizando todas las combinaciones de los siguientes parámetros:

- Sin transformaciones y con transformaciones.

- Con los modelos ResNet18 y ResNet34.

- Con las técnicas de `fine_tune` y `fit_one_cycle`.

A continuación se muestran los resultados de las pruebas realizadas y se comentan las conclusiones que se pueden sacar a partir de estos resultados.

7.4.2. Sin transformaciones

- ResNet18 con `fine_tune`:

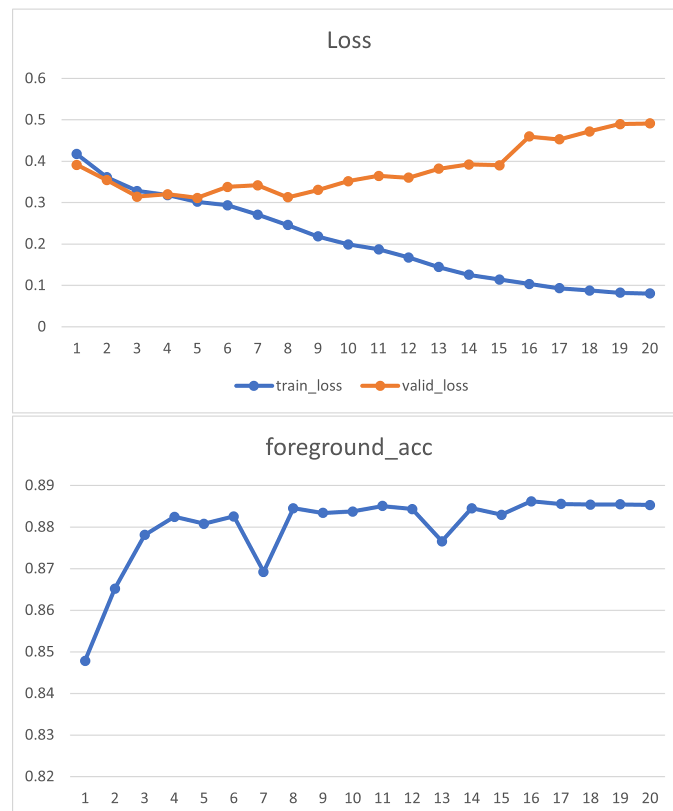


Figura 7.4: Segmentación ResNet18, fine_tune, sin transformaciones

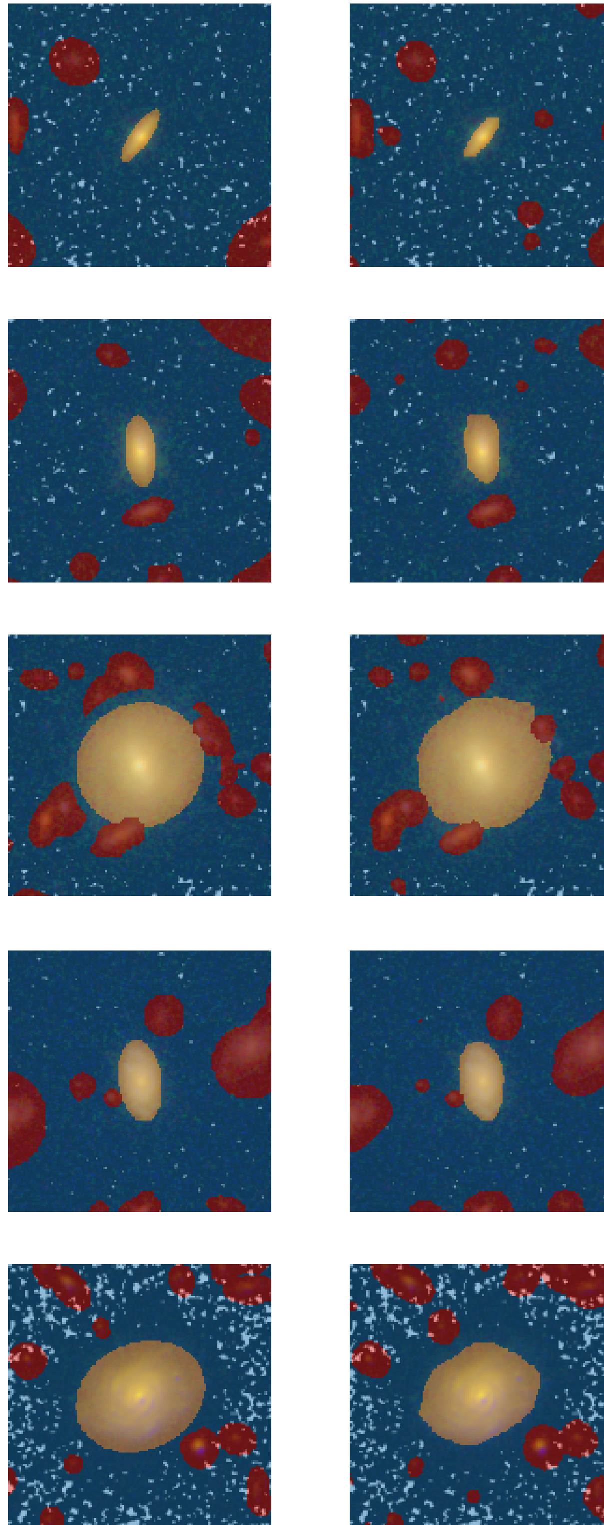


Figura 7.5: Resultado de show_results(), segmentación ResNet18, fine_tune, sin transformaciones

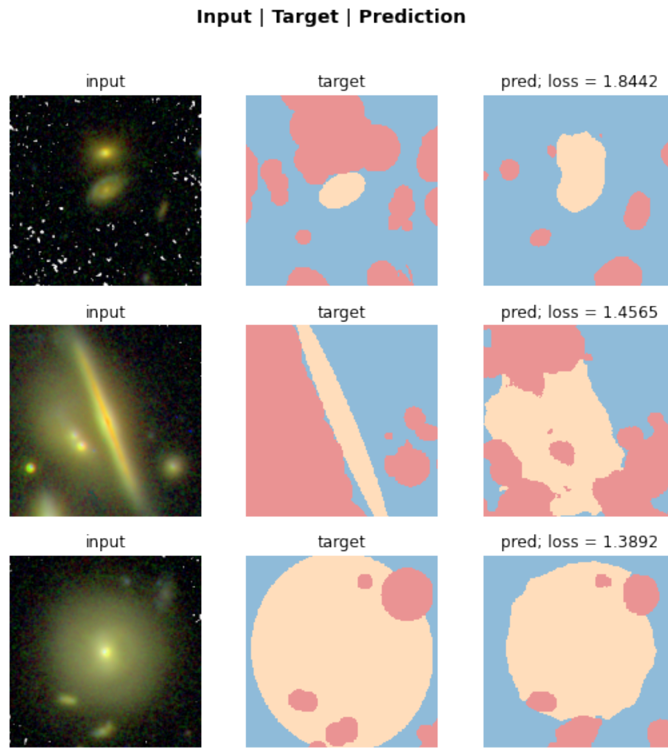


Figura 7.6: Resultado de top_losses(), segmentación ResNet18, fine_tune, sin transformaciones

- ResNet18 con `fit_one_cycle`:

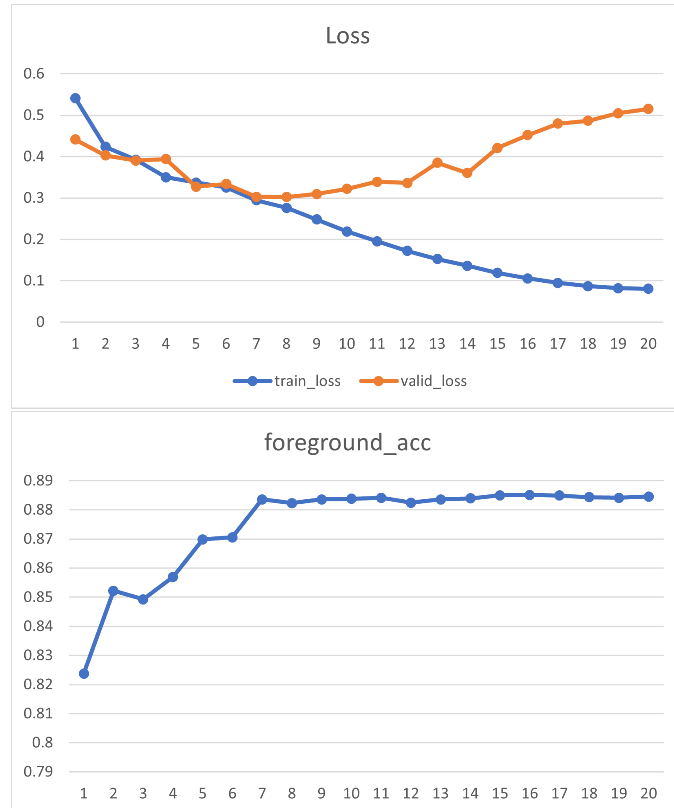


Figura 7.7: Segmentación ResNet18, fit_one_cycle, sin transformaciones

- ResNet34 con `fine_tune`:

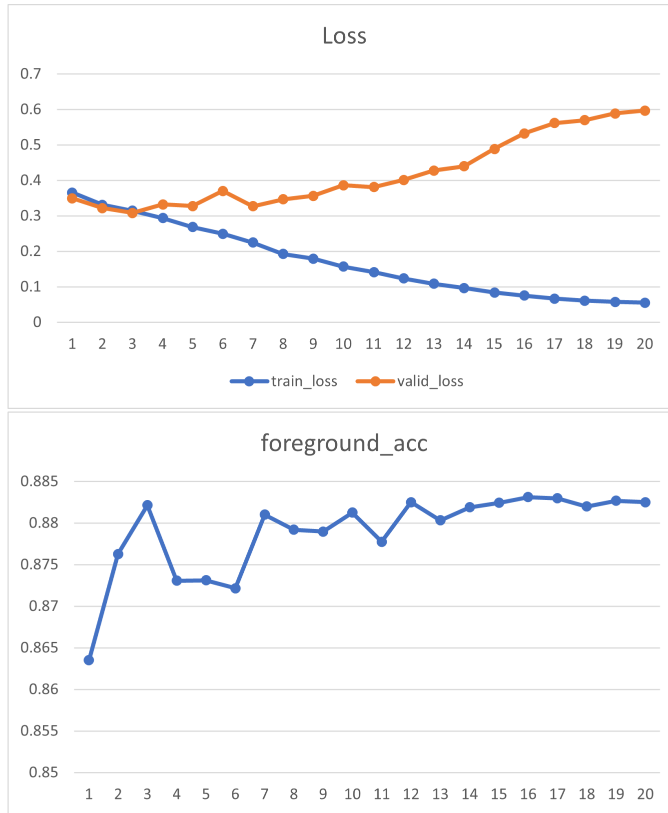


Figura 7.8: Segmentación ResNet34, fine_tune, sin transformaciones

- ResNet34 con `fit_one_cycle`:

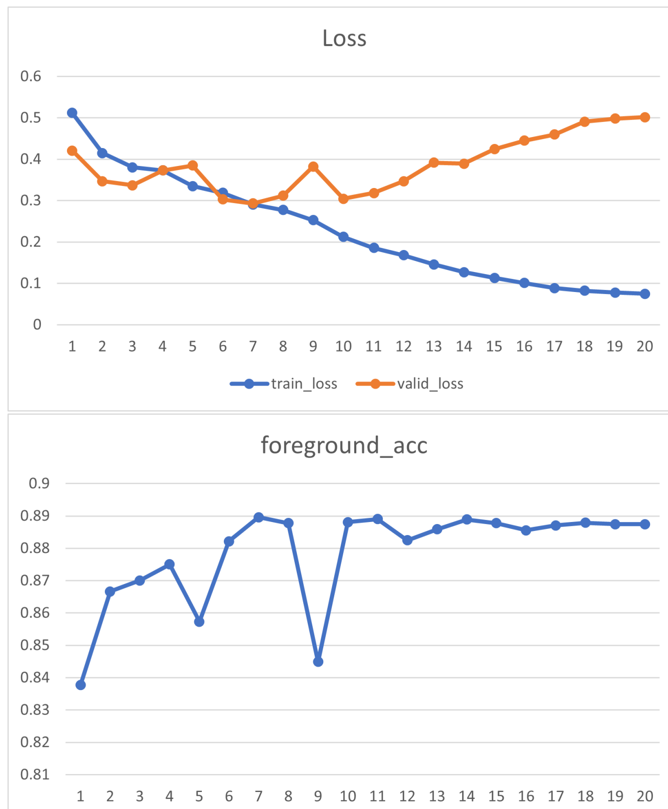


Figura 7.9: Segmentación ResNet34, fit_one_cycle, sin transformaciones

7.4.3. Con transformaciones

Las transformaciones aplicadas son muy similares a las aplicadas en el capítulo de clasificación. La primera está descrita en el listing 6.8, y la segunda está descrita en el listing 6.9, pero con una única diferencia: Hay que cambiar `TensorImage` por `PILImage`, porque ahora el ruido Gaussiano solo debe aplicarlo a la imagen de la galaxia y no a su máscara. La máscara hay que mantenerla intacta. Si pusiéramos `TensorImage`, aplicaría las transformaciones tanto a la imagen como a la máscara, por lo que estaríamos entrenando de forma errónea.

- ResNet18 con `fine_tune`:

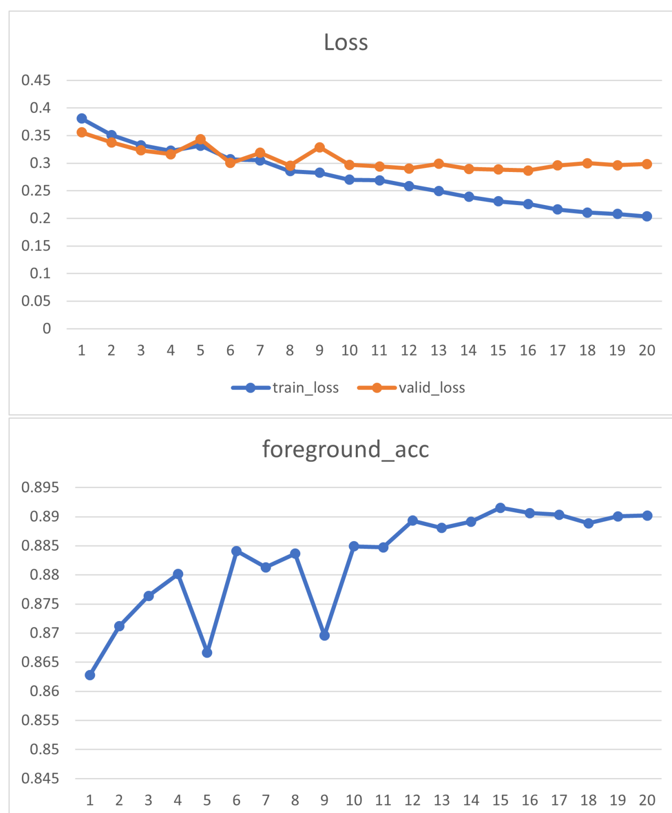


Figura 7.10: Segmentación ResNet18, fine_tune, con transformaciones

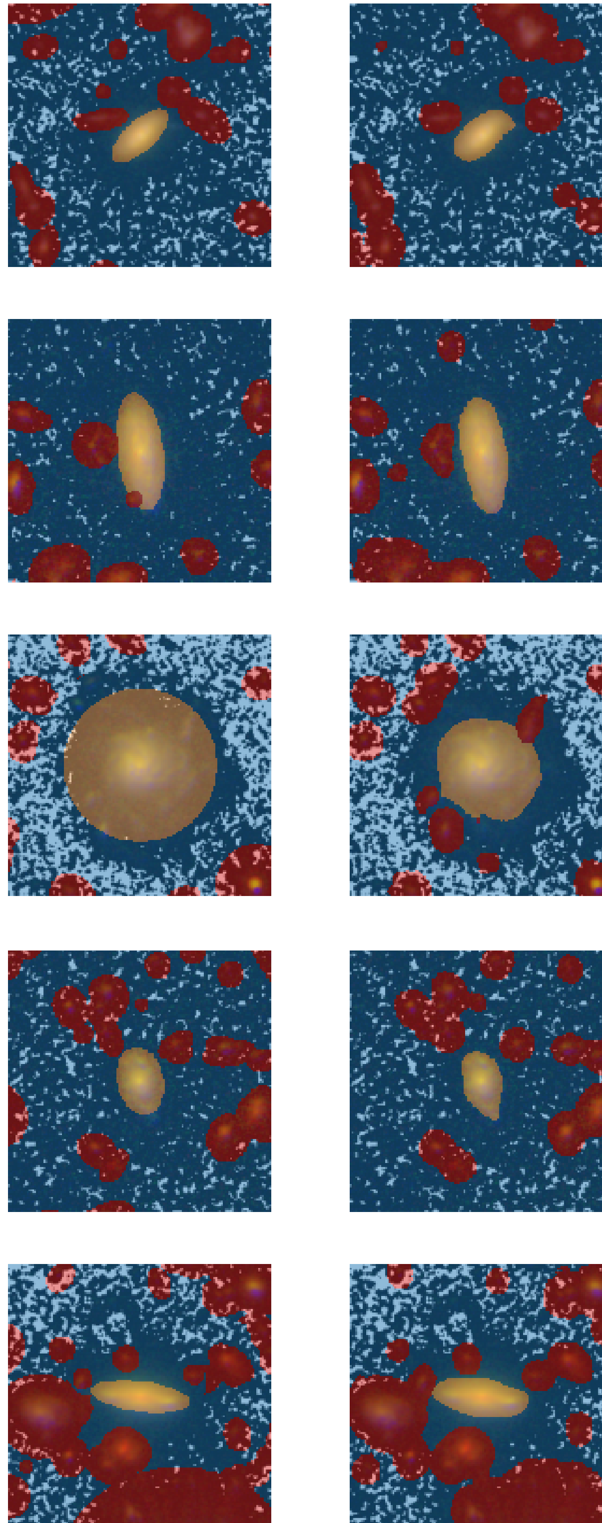


Figura 7.11: Resultado de show_results(), segmentación ResNet18, fine_tune, con transformaciones

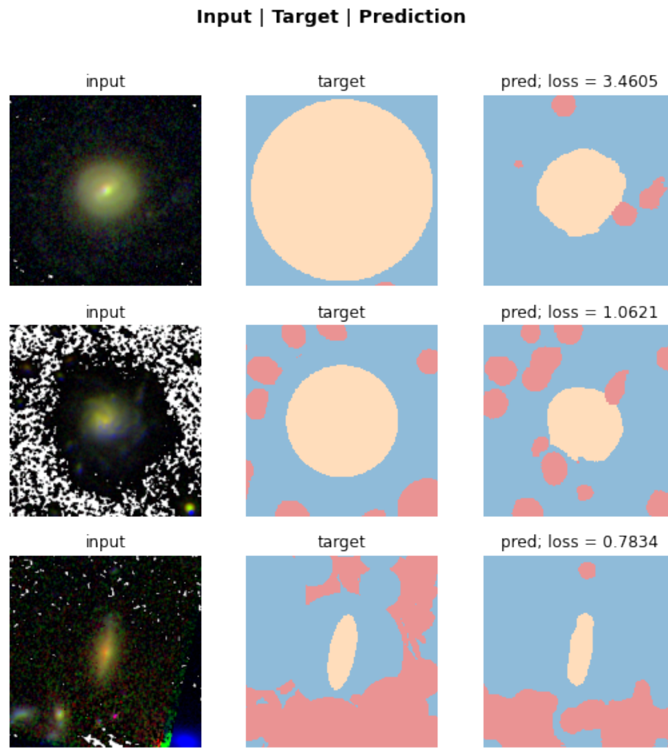


Figura 7.12: Resultado de top_losses(), segmentación ResNet18, fine_tune, con transformaciones

- ResNet18 con `fit_one_cycle`:

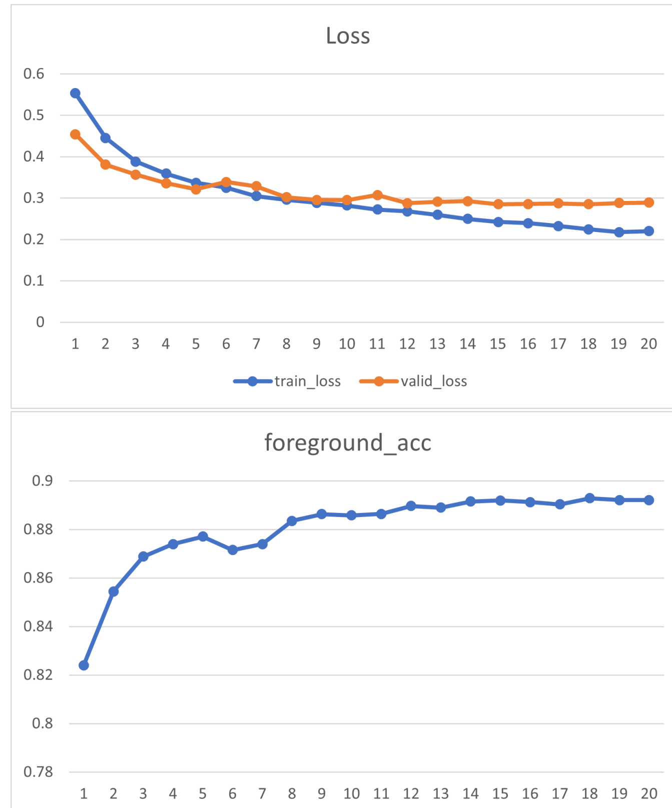


Figura 7.13: Segmentación ResNet18, fit_one_cycle, con transformaciones

- ResNet34 con `fine_tune`:

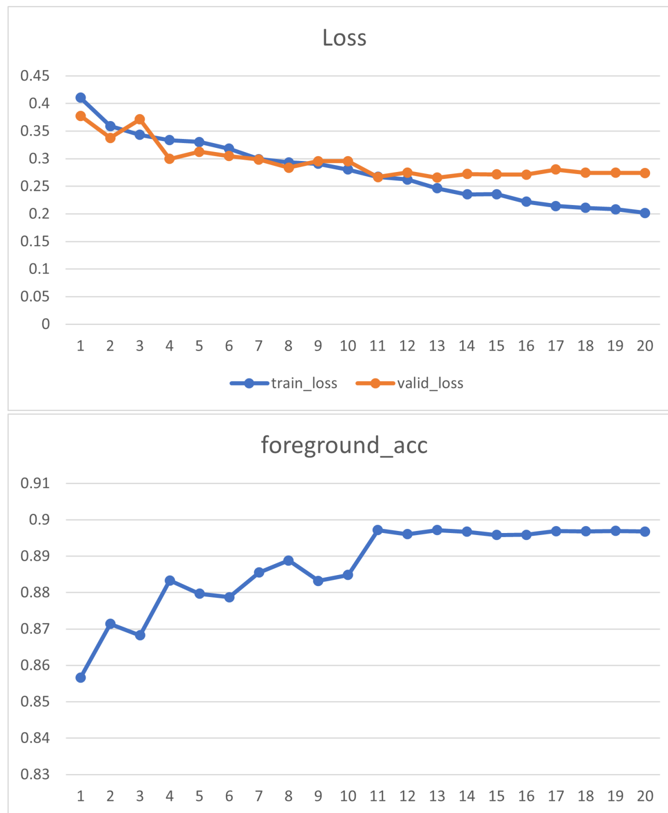


Figura 7.14: Segmentación ResNet34, fine_tune, con transformaciones

- ResNet34 con `fit_one_cycle`:

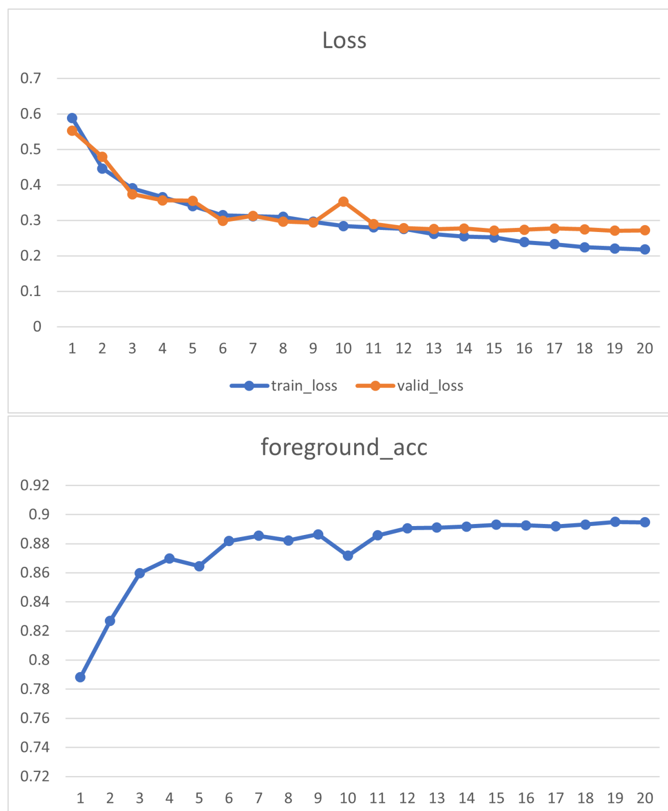


Figura 7.15: Segmentación ResNet34, fit_one_cycle, con transformaciones

7.4.4. Conclusiones

Como se puede observar todas las pruebas dan resultados similares. Ninguna consigue llegar a un $0,9$ de `foreground_accuracy`, y al llegar al epoch 11 o 12, comienza a producirse sobreajuste, ya que el valor de la función de coste en validación comienza a superar al valor en entrenamiento. Esto significa que la red no puede aprender más con este conjunto de datos y habría que modificarlo, ya sea reduciéndolo, quitando imágenes y máscaras que puedan confundir a la red y/o aumentándolo, añadiendo nuevas imágenes y máscaras que puedan ayudar a la red a aprender mejor.

También se puede interpretar estos resultados de otra forma, y hay que fijarse en las figuras 7.5 y 7.6. En la primera figura se ve en la columna derecha las máscaras reales, y en la izquierda, las máscaras que la red ha creado. Las máscaras reales están hechas con programas externos y parece que a las galaxias centrales y a la mayoría de las vecinas las remarca con círculos y elipses perfectas, mientras que la red neuronal nunca las remarca así, si no que hace formas irregulares, ajustándose lo más posible a la forma de las galaxias que detecta. Por lo tanto, en realidad la red funciona mejor de lo que indica la métrica `foreground_acc`. En la segunda figura también se observa esto con la tercera fila de fotos, aunque es cierto que en la primera y segunda filas, ha confundido una galaxia vecina con la galaxia central y eso sí que es completamente erróneo.

7.5. Segmentación mediante aprendizaje de redes federadas

En esta sección vamos a aplicar la técnica de *federated learning* o aprendizaje federado a nuestro problema de segmentación y analizar los resultados que se obtienen al aplicarla

7.5.1. Definición

El aprendizaje federado consiste en entrenar varias redes por separado y después combinar los pesos de cada una en otra red. Los modelos de las redes necesitan tener la misma arquitectura para que se puedan combinar y la forma de combinar los pesos puede variar: media aritmética, una media armónica, etc.

El concepto fue introducido por primera vez por Google en su blog de inteligencia artificial [40], donde explica que esta técnica es la solución para poder aplicar deep learning sin comprometer la privacidad de los datos. El resto de técnicas de deep learning necesitan tener todos los datos centralizados para poder entrenar a las redes neuronales, pero el aprendizaje federado consiste en entrenar cada red en un lugar diferente con sus datos locales y después enviar el modelo entrenado. De esta forma, los datos de entrenamiento no se envían a ningún servidor externo centralizado. Esto es muy útil en el campo de la medicina, donde los datos con los que se entrenan son confidenciales y no deben salir al exterior para respetar la privacidad de los pacientes.

Desde entonces, esta técnica ha sido reconocida como muy útil y varias organizaciones han creado su software para realizar esta técnica, por ejemplo, el sistema Nvidia Clara [41] o el paquete PySyft, para aplicar estas técnicas en PyTorch.

7.5.2. Exportar y combinar modelos

Vamos a realizar el proceso más básico posible para aplicar federated learning mediante fastai. Primero debemos cargar tres modelos que hayamos previamente entrenado y exportado como fichero .pkl. Después obtenemos todos sus pesos mediante el método `state_dict` como se puede observar en el fragmento de código 7.8.

```
learn1 = load_learner("SegmentationResNet18NoAug.pkl")
learn2 = load_learner("SegmentationResNet18AugRotationYGauss.pkl")
learn3 = load_learner("SegmentationResNet18AugZoom.pkl")
sd1 = learn1.model.state_dict()
sd2 = learn2.model.state_dict()
sd3 = learn3.model.state_dict()
```

Fragmento de código 7.8: Cargar learners y obtener pesos

Esta función devuelve un diccionario donde la clave es el nombre de cada capa del modelo y el valor es el tensor con los valores de los parámetros de la capa. De esta forma, podemos modificar como queramos los parámetros de la red que hemos entrenado. Por ejemplo, podemos crear un modelo nuevo cuyos pesos sean la media de los pesos de otros tres modelos, como se ve en el fragmento de código 7.9

```
for key in sd_combinado:
    sd_combinado[key] = (1/2)*sd1[key]+(1/2)*sd2[key]
learn_combinado.model.load_state_dict(sd_combinado)
```

Fragmento de código 7.9: Combinar pesos

Una vez que tenemos el modelo federado a partir de otros debemos medir su precisión con un conjunto de imágenes de test con las que la red no haya entrenado previamente. Para ello, se ha diseñado la siguiente función:

```
def get_precision_media(learner):
    lista_accuracies=[]
    for fichero in os.listdir(path_images_rgb_test):
        path = os.path.join(path_images_rgb_test, fichero)
        tensor_prediccion_en_varios_canales = learner.predict(path)[2]
        tensor_mascara=get_tensor_mascara_correspondiente(fichero)
        accuracy=foreground_acc(tensor_prediccion_en_varios_canales,
   tensor_mascara, axis=0)
        lista_accuracies.append(accuracy)
    return np.asarray(lista_accuracies).mean()
```

Fragmento de código 7.10: Calcular precisión de red federada

Esta función obtiene todas las imágenes de un directorio y para cada imagen hace lo siguiente:

1. Obtiene la predicción que hace el modelo para esa imagen mediante la función `predict`, que es un tensor en varios canales.

2. Obtiene el tensor que corresponde a la máscara real de la imagen, mediante otra función hecha específicamente para esto(fragmento 7.11).

3. Compara la predicción obtenida por el modelo con la máscara real mediante la función de fastai `foreground_acc`, que devuelve un número entre 0 y 1 que representa el porcentaje de píxeles acertados.

4. Añade esta precisión a una lista.

Por último, devuelve la media de las precisiones obtenidas. De esta forma estamos midiendo la efectividad de la red directamente, sin entrenarla.

```
def get_tensor_mascara_correspondiente(fichero):
    nombre_con_extension=fichero.split('/')[-1]
    nombre_sin_extension=nombre_con_extension.split('.')[0]
    nombre_mascara=nombre_sin_extension+"_M.png"
    mascara = Image.open(path_labels_test+nombre_mascara)
    return torch.LongTensor(np.asarray(mascara))
```

Fragmento de código 7.11: Obtener tensor de la máscara real

7.5.3. Resultados

Se han realizado dos formas diferentes crear los modelos usados para crear otro federado:

- Entrenar tres modelos con el mismo dataset pero con diferentes transformaciones cada uno.
- Entrenar tres modelos sin ninguna transformación pero con diferentes datasets.

Para cada variante se han medido las precisiones de los modelos individuales con un dataset que ninguno ha visto, que corresponde al 10 % del dataset entero que se ha usado en la primera parte de este capítulo. Y después, con este mismo dataset también se han medido las precisiones medias de diferentes combinaciones de estos learners para, que son:

- Media de los dos primeros modelos
- Media de los tres modelos
- Media ponderada los dos primeros modelos: $(1/10) * pesos_modelo_1 + (9/10) * pesos_modelo_2$
- Media ponderada los tres modelos: $(8/10) * pesos_modelo_1 + (1/10) * pesos_modelo_2 + (1/10) * pesos_modelo_3$
- Multiplicar por un escalar uno de los modelos: $0{,}999 * pesos_modelo_1$

En el apéndice A.7 se describe, mediante un Jupyter Notebbok, el proceso para crear estos modelos.

Mismo dataset, diferentes transformaciones

Se han entrenado tres modelos con un dataset formado por 953 imágenes y sus respectivas máscaras. El primero ha sido entrenado sin transformaciones, el segundo ha sido entrenado con transformaciones de ruido gaussiano y rotaciones aleatorias en múltiplos de 90º y el tercero ha sido entrenado con transformaciones de zoom aleatorio entre x1 y x2. Las precisiones medias de estos tres modelos por separado son :

- Sin transformaciones: 0.88791823
- Ruido gaussiano y rotaciones: 0.893509

- Zoom: 0.8911084

Después, se han combinado de las 5 formas explicadas en el apartado anterior y los resultados han sido los siguientes:

- Media de los dos primeros modelos: 0.7416855

- Media de los tres modelos: 0.74702615

- Media ponderada los dos primeros modelos: 0.8707591

- Media ponderada los tres modelos: 0.8868992

- Multiplicar por un escalar uno de los modelos: 0.8380975

Por último, hemos *reentrenado* este modelo federado con el dataset entero y sin transformaciones para ver cómo se comporta, figura 7.16

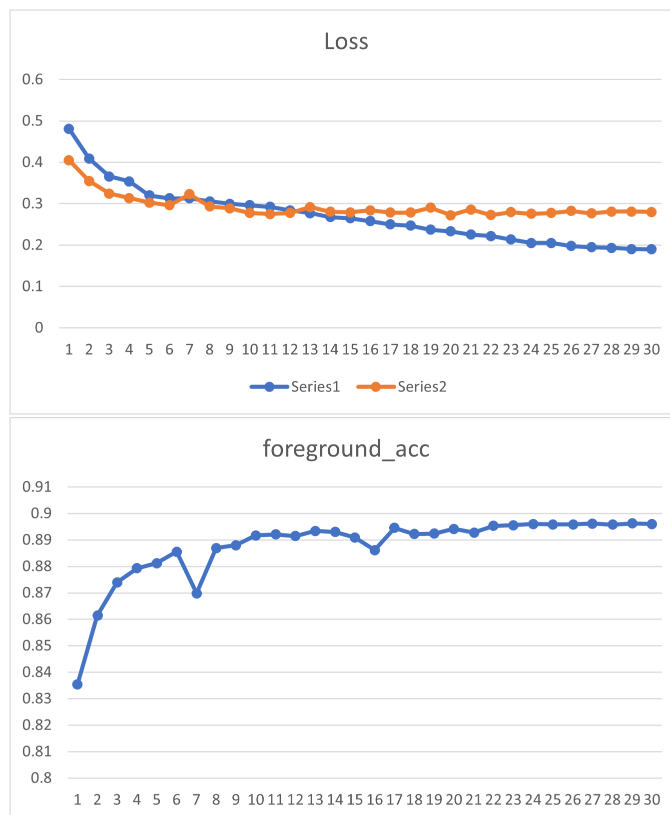


Figura 7.16: Loss y precisión de la red federada a partir de tres con diferentes transformaciones pero con el mismo dataset

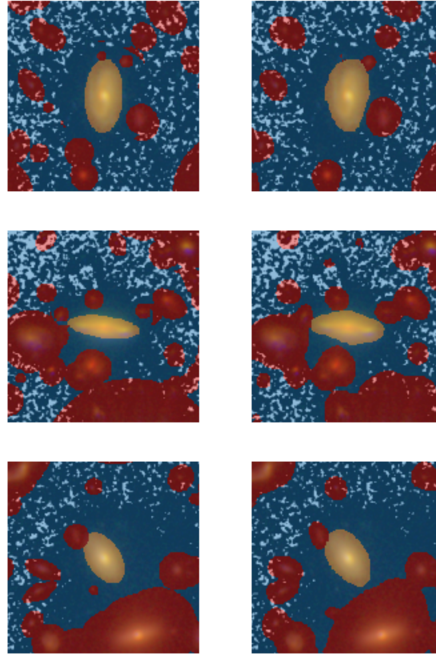


Figura 7.17: Resultado de show_results() de la red federada a partir de tres con diferentes transformaciones pero con el mismo dataset

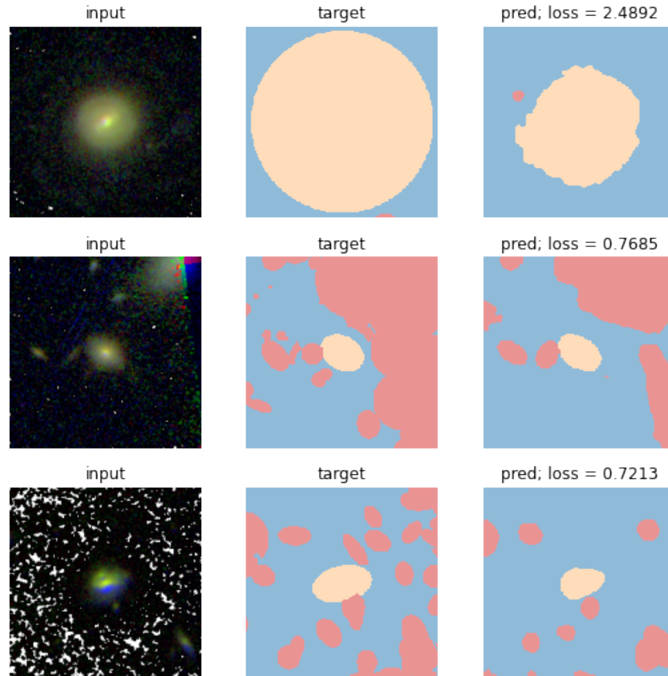


Figura 7.18: Resultado de top_losses() de la red federada a partir de tres con diferentes transformaciones pero con el mismo dataset

Distinto dataset, sin transformaciones

Se han entrenado tres modelos sin ninguna transformación y con datasets distintos. Cada dataset está formado por 318 imágenes y sus respectivas máscaras. Las precisiones medias de

estos tres modelos por separado son :

- Primer modelo: 0.8775515
- Segundo modelo: 0.8771863
- Tercer modelo: 0.8768469

Después, se han combinado de las mismas 5 formas que en el apartado anterior y los resultados han sido los siguientes:

- Media de los dos primeros modelos: 0.7413885
- Media de los tres modelos: 0.7419647
- Media ponderada los dos primeros modelos: 0.84995466
- Media ponderada los tres modelos: 0.87265795
- Multiplicar por un escalar uno de los modelos: 0.8301922

Por último, hemos *reentrenado* también este modelo federado con el dataset entero y sin transformaciones para ver cómo se comporta, figura 7.19

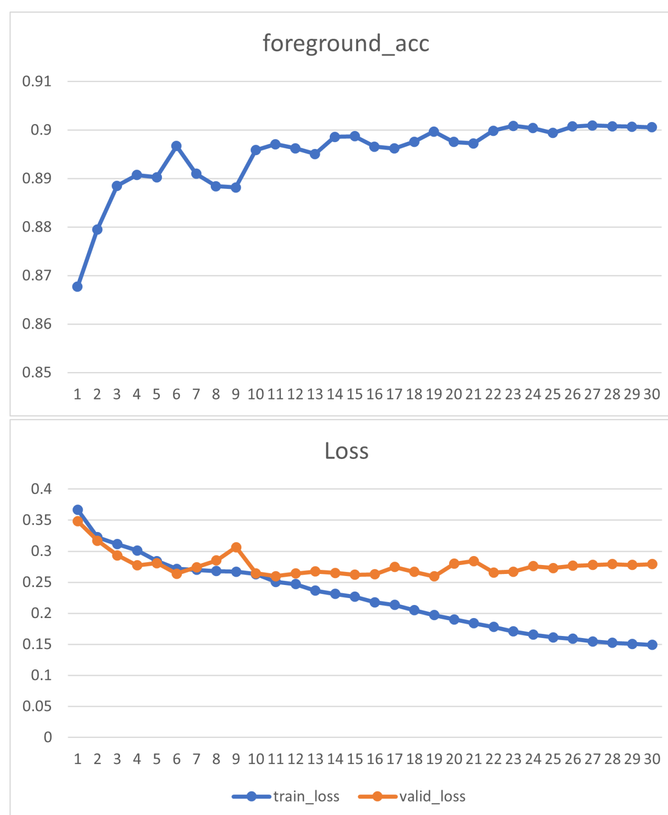


Figura 7.19: Loss y precisión de la red federada a partir de tres con diferentes datasets sin transformaciones

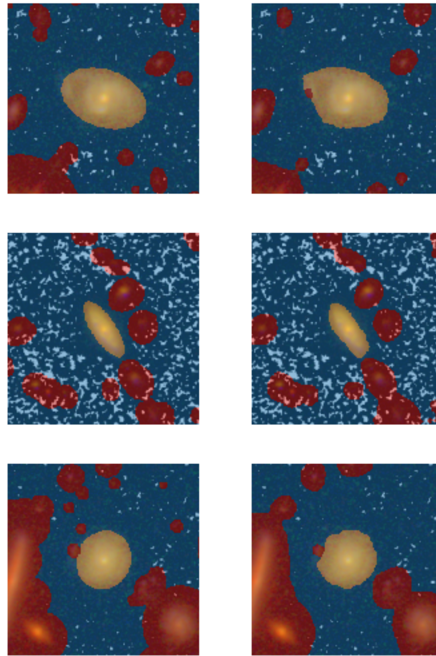


Figura 7.20: Resultado de show_results() de la red federada a partir de tres con diferentes datasets sin transformaciones

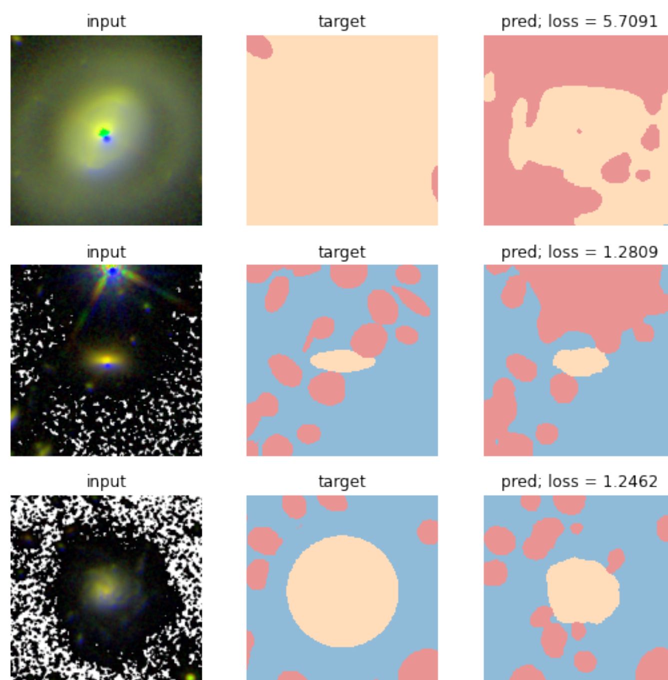


Figura 7.21: Resultado de top_losses() de la red federada a partir de tres con diferentes datasets sin transformaciones

7.5.4. Conclusiones

A partir de los resultados obtenidos aplicando esta técnica se pueden sacar varias conclusiones.

Crear una red cuyos pesos son la media de otros tres no hace que mejore la red, ya que cada red converge a un mínimo local, por lo que obtener el valor intermedio no tiene que significar un mejor rendimiento. Esto se puede observar en las precisiones obtenidas con los pesos medios de 2 o de 3 modelos, que tan solo llega a 0,74

Cuanto más nos acerquemos a los pesos originales de una de las redes, mejor precisión se obtiene, pero nunca sin llegar a alcanzar a la precisión de cualquiera de las redes individuales. Esto se puede observar con las precisiones obtenidas al hacer medias ponderadas en la que los pesos de una de las redes valen mucho más que los de las otras dos.

Entrenando redes con el mismo dataset pero con diferentes transformaciones hemos obtenido mejores resultados (precisión de 0,89 que entrenando redes con el dataset dividido y sin transformaciones (precisión de 0,87). Esto tiene sentido, ya que en el segundo caso, las redes han entrenado con tres veces menos imágenes, por lo que no han podido aprender tanto como las del otro caso.

Reentrenar la red federada da buenos resultados, aunque son casi iguales a los de las redes individuales, por lo que esta técnica, por lo menos en este problema concreto, no ha supuesto ninguna mejora sustancial.

7.6. Segmentación mediante un modelo personalizado

En esta sección vamos a realizar segmentación como en la sección 7.4 con una diferencia. En vez de utilizar un modelo preentrenado que se importa automáticamente con una sola línea de código en fastai, vamos a crear nosotros el modelo. Este modelo está sacado del capítulo 14 del libro de fastai[3], que a su vez se inspira en el paper de Tong He et al. *Bag of Tricks for Image Classification with Convolutional Neural Networks*[42].

7.6.1. Creación del modelo

El objetivo es definir un modelo manualmente que sea equivalente a una ResNet. La ventaja de hacer esto es que se puede modificar muy fácilmente según nuestras necesidades. Primero definimos los bloques que tendrá nuestra red, cada bloque contendrá varias capas:

```
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)))
```

Fragmento de código 7.12: Bloques del modelo personalizado

A continuación definimos los primeros niveles de la red, que se llaman *stem*. Se comienza con unos niveles directamente convolucionales, finalizando con maxpool.

```python
def _resnet_stem(*sizes):
    return [
        ConvLayer(sizes[i], sizes[i+1], 3, stride = 2 if i==0 else 1)
            for i in range(len(sizes)-1)
    ] + [nn.MaxPool2d(kernel_size=3, stride=2, padding=1)]
```

Fragmento de código 7.13: Primeros niveles del modelo personalizado

Después definimos la clase de la red:

```python
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)
        ])
```

Fragmento de código 7.14: Modelo personalizado

El constructor de la clase recibe varios parámetros:

- `n_out`: Int que indica el número de salidas de la red
- `layers`: Array unidimensional de int que indica el número de bloques convolucionales que se crearán
- `expansion`: Int que indica el factor por el que se multiplica el número de parámetros base de las capas. Cuanto mayor sea este número mayor número de parámetros tendrá la red.

El constructor llamará durante la creación de la red a la función `make_layer` varias veces, que es la que se encarga de crear las capas de la red a partir de los bloques definidos anteriormente. Por último definimos una función para obtener el modelo con los parámetros que queramos:

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

Fragmento de código 7.15: Obtener modelo personalizado

El primer parámetro 2 indica que queremos una red con 2 salidas, aunque después en segmentación vamos a recortar esa última capa y la penúltima. El segundo parámetro, que es un array de números, son los parámetros para conseguir una red equivalente a una ResNet18. Si quisiéramos una red equivalente a una ResNet50 debemos poner `ResNet(2,[3,4,6,3],4)`. Este último

parámetro que para crear una ResNet18 no lo hemos usado, es la expansión de la red, que significa que los bloques de la red tendrán muchos más parámetros. Por último creamos el Learner de fastai:

```
learn = unet_learner(dls, get_model, cut=-2, metrics = foreground_acc)
```

Fragmento de código 7.16: Crear Learner con modelo personalizado

El modelo lo obtenemos mediante nuestra función `get_model`, y como se va convertir en una red en U gracias al método de fastai `unet_learner` debemos quitar las dos últimas capas con el parámetro `cut=-2`.

7.6.2. Resultados

Vamos a entrenar este modelo sin preentrenar con 50 epochs, tamaño de batch de 8, con transformaciones de ruido gaussiano y rotaciones en múltiplos de 90º y con el mismo dataset que en la sección 7.4. Los resultados son los siguientes:

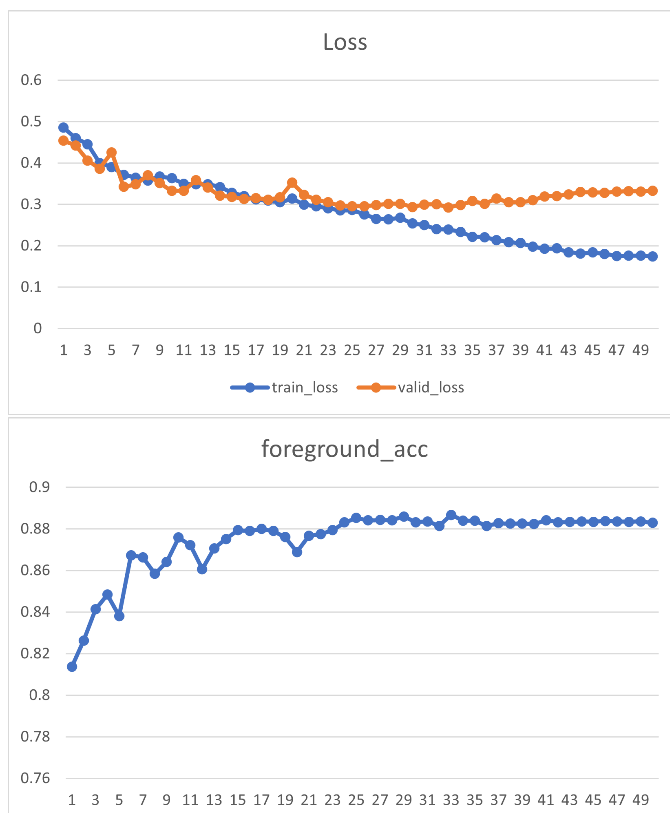


Figura 7.22: Loss y precisión de la red personalizada

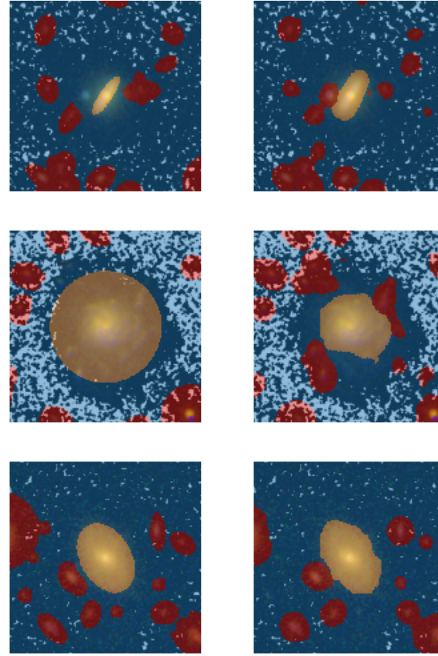


Figura 7.23: Resultado de show_results() de la red personalizada

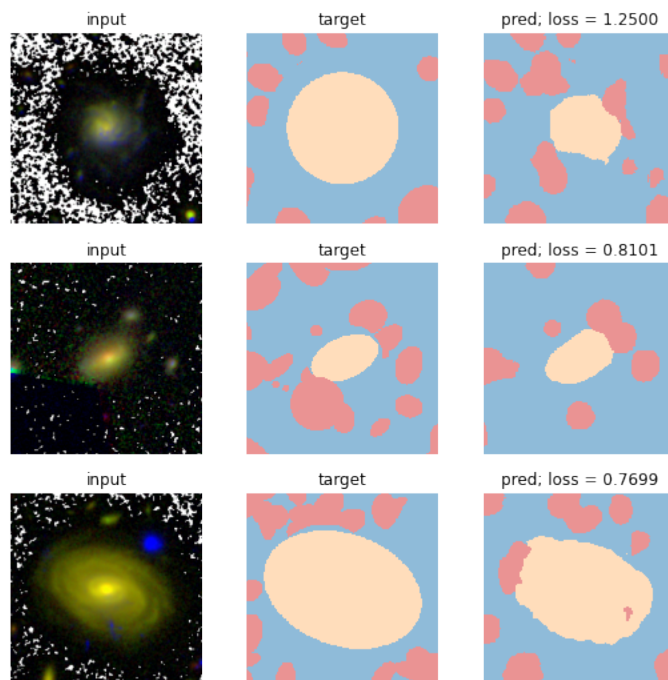


Figura 7.24: Resultado de top_losses() de la red personalizada

7.6.3. Conclusiones

Hemos obtenido unos resultados casi iguales a los obtenidos en la sección 7.4.3 con una ResNet18 preentrenada, lo que significa que la red que hemos construido *a mano* funciona correctamente: alcanza una precisión del 0.88 y se estanca en ese valor porque no puede aprender más. Además, esta red que hemos creado no está preentrenada, por lo que con este conjunto de datos y estas transformaciones ha podido entrenarse y ajustarse igual que una preentrenada.

Capítulo 8

Conclusiones finales

8.0.1. Consecución de objetivos

- Aprender los conceptos fundamentales del machine learning y del deep learning, y más en profundidad, de las redes neuronales.
- Conocer los frameworks principales de Deep Learning y las diferencias entre ellos: TensorFlow, Keras, PyTorch y Fastai.
- Aprender a utilizar herramientas informáticas usadas por los astrónomos, como DS9, Source Extractor, Topcat o Aladin.
- Aprender sobre el tratamiento de imágenes astronómicas de galaxias u otros objetos del espacio profundo. En concreto, aprender a utilizar el formato .fits o los diferentes filtros de colores en los que los telescopios espaciales como el Hubble o el Euclid, toman las fotografías.
- Aprender a utilizar Fastai y PyTorch para resolver distintos tipos problemas.
- Utilizar Fastai para resolver problemas reales de visión artificial en el campo de la astronomía, en concreto:
    - Clasificación morfológica de galaxias.
    - Regresión morfológica de galaxias.
    - Detección de bordes de galaxias mediante segmentación.

8.0.2. Futuras mejoras

Para mejorar la precisión de las redes en sus predicciones habría que empezar por modificar los datasets. En cada una de las tres partes principales de este trabajo: clasificación, regresión y segmentación, hemos usado conjuntos de datos fijos, variando los modelos y los hiperparámetros.

En clasificación no se ha conseguido superar el 70 % de precisión y en segmentación se ha conseguido superar el 90 % de precisión. En regresión se conseguían muy buenos resultados, por lo que no sería un objetivo prioritario, pero en clasificación y segmentación sería conveniente revisar y limpiar el dataset de imágenes que puedan confundir a la red o de máscaras que estén mal generadas. Después habría que conseguir más imágenes de este tipo y en el caso de segmentación, crear, además sus respectivas máscaras.

También se podría seguir profundizando en el aprendizaje de redes federadas para intentar conseguir mejores resultados.

Otra posible línea de mejora sería seguir con la ResNet creada manualmente mediante parámetros en la última sección del capítulo anterior para experimentar con modelos diferentes a las ResNet con las que hemos obtenido los resultados descritos.

Apéndice A

Jupyter Notebooks usados

A.1. Tutorial de segmentación

```
from fastbook import *
from fastai.vision import *

#Obtener dataset
path = untar_data(URLs.CAMVID_TINY)
path.ls()

#Obtener la correspondencia entre int y classes
codes = np.loadtxt(path/'codes.txt', dtype=str)
codes

#Definir funcion para etiquetar la imagen a partir de su mascara
def label_func(fn): return path/"labels"/f"{fn.stem}_P{fn.suffix}"

camvid = DataBlock(blocks=(ImageBlock, MaskBlock(codes)),
                   get_items = get_image_files,
                   get_y = label_func,
                   splitter=RandomSplitter(),
                   batch_tfms=aug_transforms(size=(120,160)))

dls = camvid.dataloaders(path/"images", path=path, bs=8, num_workers=0)
dls.show_batch(max_n=6)

#Definir Learner y entrenar. En segmentacion debe usarse un unet_learner
learn = unet_learner(dls, resnet34)
learn.fine_tune(6)

#Mostrar algunos resultados. A la izquierda la mascara real, a la derecha la
    mascara predicha por la red.
learn.show_results(max_n=6, figsize=(7,8))
```

Fragmento de código A.1: Código del tutorial de segmentación

A.2. Ejemplo con .fits. Sección 4.3

```
#Importamos las bibliotecas necesarias:
import numpy as np
from astropy.io import fits
from astropy.table import Table
import matplotlib.pyplot as plt

#Leemos los campos del catalogo de objetos que ha creado SExtractor:
tt = Table.read("first.cat",names=["id","ra","dec","mag_aper","magerr_aper","x_image","y_image",\
                                   "flux_radius","ellipticity","theta","kron","a"],format="ascii.commented_header")

#Abrimos la imagen astronomica de la que queremos obtener fragmentos:
hdu = fits.open("hlsp_candels_hst_wfc3_cos-tot_f160w_v1.0_drz.fits")
img = hdu[0].data
hdr = hdu[0].header
radio=100

#Creamos un vector de dos dimensiones con la rad y la dec de diez galaxias que elijamos. Estos valores los obtenemos a mano del Aladin:
galax = [[150.1143535, 2.356457],
         [150.1138174, 2.3607032],
         [150.1124122, 2.3549241],
         [150.0909541, 2.3912195],
         [150.1056863, 2.3991092],
         [150.1189451, 2.4571717],
         [150.1203373, 2.4715464],
         [150.0907101, 2.2056509],
         [150.0951051, 2.1937415],
         [150.1018265, 2.2009676]]

#Despues obtenemos los objetos de la tabla cuyo rad y dec coinciden con los de la tabla:
ListaGalaxias = []
contadorGalax = 0
for gal in galax:
    valorRa = galax[contadorGalax][0]
    valorDec = galax[contadorGalax][1]
    for linea in tt:
        if valorRa == float(linea["ra"]):
            if  valorDec == float(linea["dec"]):
                contadorGalax = contadorGalax+1
                ListaGalaxias.append(int(linea["id"]))
print(contadorGalax)

#Por ultimo, imprimimos los recortes de la imagen junto con otros datos, por ejemplo, el angulo de posicion de la galaxia:
#Muestra las galaxias junto con su angulo de posicion
!mkdir galaxias
contadorGalax = 0
```

```
46 for gal in ListaGalaxias:
47     linea = tt[gal-1]
48     contadorGalax=contadorGalax+1
49     #Ahora obtenemos las coordenadas de los pixeles en la imagen
50     x=int(linea["x_image"])
51     y=int(linea["y_image"])
52     plt.axis('off')
53     nuevoA=np.log10(img[y-radio:y+radio,x-radio:x+radio])
54     plt.imshow(nuevoA, origin = "lower", cmap = "cubehelix")
55     plt.savefig(r"galaxias\galaxia"+str(contadorGalax)+".png",bbox_inches='
    tight')
56     plt.show()
57     print("Angulo de posicion: "+str(float(linea["theta"])))
```

Fragmento de código A.2: Código completo para recortar imágenes de galaxias

A.3. Creación de imágenes RGB. Sección 5.3

A.3.1. Create_images_RGB_lupton

```
1 import numpy as np
2 from astropy.io import fits
3 from astropy.table import Table
4 import matplotlib.pyplot as plt
5 import os
6 from astropy.visualization import make_lupton_rgb
7 from scipy.ndimage import gaussian_filter
8 from functs_for_truncations import find_right_image, find_right_mask
9
10 def createRGB(b_im, g_im, r_im, stretch=1, Q=10, scale=None):
11     """
12     Crea an RGB color composite image.
13
14     Parameters
15     ----------
16     image : `MultibandImage`
17         `MultibandImage` to display.
18     bgr : sequence
19         A 3-element sequence of filter names (i.e. keys of the exps dict)
    indicating what band
20         to use for each channel. If `image` only has three filters then this
    parameter is ignored
21         and the filters in the image are used.
22     stretch: int
23         The linear stretch of the image.
24     Q: int
25         The Asinh softening parameter.
26     scale: list of 3 floats, each less than 1. (default: None)
27         Re-scales rgb channels
28
29     Returns
```

```
30      -------
31      rgb: ndarray
32          RGB (integer, 8-bits per channel) colour image as an NxNx3 numpy array
        .
33      """
34
35      if scale is None:
36          pass
37      else:
38          b_im = b_im*scale[0]
39          g_im = g_im*scale[1]
40          r_im = r_im*scale[2]
41
42      rgb = make_lupton_rgb(image_r=r_im,
43                            image_g=g_im,
44                            image_b=b_im,
45                            stretch=stretch, Q=Q)  # parameters used to stretch
        and scale the pixel values
46
47      return rgb
48
49  def selecting_disks_according_Huertas15(f_sph,f_disk,f_irr):
50
51      cond1 = f_sph > 0.667
52      cond2 = f_disk < 0.667
53      cond3 = f_irr < 0.1
54      pure_bulges = np.logical_and.reduce((cond1,cond2,cond3),dtype=bool)
55      cond1 = f_sph < 0.667
56      cond2 = f_disk > 0.667
57      cond3 = f_irr < 0.1
58      pure_disks = np.logical_and.reduce((cond1,cond2,cond3),dtype=bool)
59      cond1 = f_sph > 0.667
60      cond2 = f_disk > 0.667
61      cond3 = f_irr < 0.1
62      disks_sph = np.logical_and.reduce((cond1,cond2,cond3),dtype=bool)
63      cond1 = f_sph < 0.667
64      cond2 = f_disk > 0.667
65      cond3 = f_irr > 0.1
66      irr_disks = np.logical_and.reduce((cond1,cond2,cond3),dtype=bool)
67      cond1 = f_sph < 0.667
68      cond2 = f_disk < 0.667
69      cond3 = f_irr > 0.1
70      irregulars_mergers = np.logical_and.reduce((cond1,cond2,cond3),dtype=bool)
71
72      morph = np.full(f_sph.size,"NONE",dtype="U100")
73      morph[pure_bulges] = "SPH"
74      morph[pure_disks] = "DISK"
75      morph[disks_sph]  = "DISKSPH"
76      morph[irr_disks]  = "DISKIRR"
77      morph[irregulars_mergers] = "IRR"
78
79      return(morph)
```

```
80
81 def main():
82
83     # Se crean los directorios de salida organizados para poder entrenar con
    fastai.
84     # Cada directorio con el nombre de su cateogoria
85     path_output = "./rgb_images/"
86     if not os.path.exists(path_output):
87         os.mkdir(path_output)
88
89     for i in ["SPH","DISK","DISKSPH","DISKIRR","IRR","NONE"]:
90       if not os.path.exists(path_output+i):
91         os.mkdir(path_output+i)
92
93 #Se lee el fichero .cat que contiene la lista de galaxias
94
95     tt = Table.read("
    galaxies_main_props_with_struct_params_and_no_bad_objects_and_hudf_flags
    _and_morphs_and_z_flags6.cat",
96                     names=["gal_id","field","ra","dec","mass","z",
97                            "flag_fit","mag","dmag","re","dre","n","dn",
98                            "q","dq","pa","dpa","sn",
99                            "IAU_name","f_sph","f_disk","f_irr","f_ps",
100                           "f_unc","max_f","delta1-2","dom_class","entropy",
101                           "zspec","quality_zspec","zphot","zspec_legac","
    zspec_zcosmos",
102                           "confidence_class_zcosmos","zspec_legac_dr3","
    zspec_hathi","zspec_damjanov","zspec_damjanov_quality",
103                           "hudf_for_B","hudf_for_V","hudf_for_I","hudf_for_Y"
    ,"hudf_for_J","hudf_for_H"],format="ascii.commented_header")
104     gals = np.array(tt["gal_id"],dtype="U100")
105     ra = np.array(tt["ra"])
106     dec = np.array(tt["dec"])
107     field = np.array(tt["field"])
108     ar = np.array(tt["q"])
109     pa  = np.array(tt["pa"])
110     zz_old = np.array(tt["z"])
111     zspec = np.array(tt["zspec"])
112     quality_zspec = np.array(tt["quality_zspec"])
113     zphot = np.array(tt["zphot"])
114     zspec_legac = np.array(tt["zspec_legac"])
115     zspec_zcosmos = np.array(tt["zspec_zcosmos"])
116     confidence_class_zcosmos = np.array(tt["confidence_class_zcosmos"])
117     zspec_hathi = np.array(tt["zspec_hathi"])
118     zspec_legac_dr3 = np.array(tt["zspec_legac_dr3"])
119     zspec_damjanov = np.array(tt["zspec_damjanov"])
120
121     f_sph = np.array(tt["f_sph"])
122     f_disk = np.array(tt["f_disk"])
123     f_irr = np.array(tt["f_irr"])
124
125 #Se ejecuta el proceso del bucle para cada galaxia que aparece en el fichero .
```

```
    cat
126     for ii,gal in enumerate(gals):
127 
128         scale = [1.,0.7,0.7]
129         stretch = 0.05 #Lower the stretch if you want to highlight better the
    galaxy outer parts
130         QQ = 10 #Q darkens the image
131         level_of_gray = 10 #this factor controls the level of grey outside of
    the galaxy
132         kernel_sigma = 1 #[pix]
133 
134 #Se obtiene la morfologia y se pasa a cadena de caracteres
135         morfo = selecting_disks_according_Huertas15(f_sph[ii],f_disk[ii],f_irr
    [ii])
136         morfo =str(morfo[0])
137 
138 #Se obtienen los ficheros de las imagenes en los 3 filtros: I, J y H
139         filename_img_b,filename_rms_b,flag_use_hudf = find_right_image(morfo,
    gal,field[ii],"I")
140         filename_img_g,filename_rms_g,flag_use_hudf = find_right_image(morfo,
    gal,field[ii],"J")
141         filename_img_r,filename_rms_r,flag_use_hudf = find_right_image(morfo,
    gal,field[ii],"H")
142         if filename_img_b is not None:
143             hdu = fits.open(filename_img_b)
144             img_b = hdu[0].data
145             hdr_b    = hdu[0].header
146             hdu.close()
147         else:
148             print("Image B (speaking about RGB) not available", filename_img_b
    )
149         if filename_img_g is not None:
150             hdu = fits.open(filename_img_g)
151             img_g = hdu[0].data
152             hdr_g    = hdu[0].header
153             hdu.close()
154         else:
155             print("Image G (speaking about RGB) not available", filename_img_g
    )
156         if filename_img_r is not None:
157             hdu = fits.open(filename_img_r)
158             img_r = hdu[0].data
159             hdr_r    = hdu[0].header
160             hdu.close()
161         else:
162             print("Image R (speaking about RGB) not available", filename_img_r
    )
163 
164 
165 # Se obtienen los ficheros de mascaras
166         mask_filename_b=find_right_mask(morfo,gal,field[ii],"I")
167         mask_filename_g=find_right_mask(morfo,gal,field[ii],"J")
```

```
168         mask_filename_r=find_right_mask(morfo,gal,field[ii],"H")
169 
170         if mask_filename_b is not None:
171             hdu = fits.open(mask_filename_b)
172             mask_b = hdu[0].data
173             mask_hdr_b    = hdu[0].header
174             hdu.close()
175         else:
176             print("Mask B (speaking about RGB) not available", mask_filename_b
    )
177         if mask_filename_g is not None:
178             hdu = fits.open(mask_filename_g)
179             mask_g = hdu[0].data
180             mask_hdr_g    = hdu[0].header
181             hdu.close()
182         else:
183             print("Image G (speaking about RGB) not available",
    mask_filename_g)
184         if mask_filename_r is not None:
185             hdu = fits.open(mask_filename_r)
186             mask_r = hdu[0].data
187             mask_hdr_r    = hdu[0].header
188             hdu.close()
189         else:
190             print("Image R (speaking about RGB) not available",
    mask_filename_r)
191 
192 # Se aplica la mascara. Se usa siempre la mascara del filtro R porque de los
    otros filtros no siempre hay.
193         img_b=mask_r*img_b
194         img_g=mask_r*img_g
195         img_r=mask_r*img_r
196 
197         #Recortar imagen para pasar de 200x200 a 100x100 si se quisiera
198         #img_b_recortada=img_b[49:150, 49:150]
199         #img_g_recortada=img_g[49:150, 49:150]
200         #img_r_recortada=img_r[49:150, 49:150]
201 
202         #Se crea la imagen RGB propiamenete dicha
203         rgb_original = createRGB(img_b, img_g, img_r, scale=scale, stretch=
    stretch, Q=QQ)
204         #rgb_original = createRGB(img_b_recortada, img_g_recortada,
    img_r_recortada, scale=scale, stretch=stretch, Q=QQ)
205         rgb_smoothed = gaussian_filter(rgb_original, sigma = kernel_sigma)#Se
    aplica un filtro gaussiano para suavizar los bordes
206 
207 
208         fig = plt.figure(figsize=(2,2)) #2 pulgadas de ancho y alto
209         ax = fig.add_subplot(1,1,1)
210         fig.subplots_adjust(0,0,1,1) #Quitar cualquier tipo de margen o borde
    alrededor de la imagen
211 
```

```
        rgb_alpha = np.stack([rgb_original[:,:,0],
                              rgb_original[:,:,1],
                              rgb_original[:,:,2],
                              np.sum(rgb_smoothed, axis=2)*level_of_gray],
                             axis=2)
        ax.imshow(rgb_alpha, origin='lower')
        ax.set_axis_off()

# Se podria guardar en distintos formatos
#       plt.savefig(path_output+morfo+"/"+morfo+"_"+gal+"_"+field[ii]+"
    _rgb_image.pdf", format='pdf', dpi=100)
#       plt.savefig(path_output+morfo+"/"+morfo+"_"+gal+"_"+field[ii]+"
    _rgb_image.jpg", format='jpg', dpi=100)
        plt.savefig(path_output+morfo+"/"+morfo+"_"+gal+"_"+field[ii]+"
    _rgb_image.png", format='png', dpi=100)
        plt.clf()
        plt.close(fig)


if __name__ == "__main__":
    main()
```

Fragmento de código A.3: Crear imágenes en RGB para clasificación

A.4. Clasificación morfológica de galaxias. Sección 5.4

A.4.1. Clasificación simple ResNet18 No Augmentation

```
from fastbook import *
from fastai.vision import *
path_input="rgb_images_200x200"

#Creamos el Datablock
galaxias=DataBlock(
    blocks=(ImageBlock, CategoryBlock),
    get_items=get_image_files,
    splitter=RandomSplitter(valid_pct=0.2, seed=1),
    get_y=parent_label,
)

#Creamos el DataLoaders
dls=galaxias.dataloaders(path_input, num_workers=0)

#Creamos el learner con el modelo de red que queramos y con el Callback que
    muestra un grafico del loss
learn = cnn_learner(dls, resnet18, metrics=error_rate, cbs=CSVLogger(fname="
    ClasificacionSimpleResNet18NoAug.csv", append=true))

#Entrenamos la red
learn.fine_tune(20)

```

```
#Mostramos la matriz de confusion
interp=ClassificationInterpretation.from_learner(learn)
interp.plot_confusion_matrix()
```

Fragmento de código A.4: Código para entrenar una ResNet18 sin data augmentation en clasificación

A.4.2. Clasificación simple ResNet18 Augmentation

```
from fastbook import *
from fastai.vision import *
path_input="rgb_images_200x200"

# Devuelve un tensor float de batch-size elementos con las rotaciones
    aleatorias multiplos de 90
# x es el tensor con el batch completo.
def RotacionAleatoria(x):
    batchsize=x.size(0)
    result=torch.zeros(([batchsize]), dtype = torch.float, device='cuda')
    for i in range(batchsize):
        valor=torch.rand(1)
        if(valor<0.25):
           result[i]=0.0
        elif(valor<0.5):
           result[i]=90.0
        elif(valor<0.75):
           result[i]=180.0
        else:
           result[i]=270.0
    return result

#Ruido gaussiano media 0 y std 0.005. Se pone despues de normalizar asi que
    std es 0,5%
class RuidoGaussiano(Transform):
    order=100 #After normalize
    def encodes(self, x:TensorImage):
        return x+torch.normal(mean=0.0, std=0.005, size=x.shape, device='cuda'
    )

#Definimos las transformaciones que vamos a utilizar
tfms1=[Normalize.from_stats(mean=0.0, std=1.0), Rotate(p=1.0, draw=
    RotacionAleatoria), RuidoGaussiano]

#Creamos el Datablock
galaxias=DataBlock(
    blocks=(ImageBlock, CategoryBlock),
    get_items=get_image_files,
    splitter=RandomSplitter(valid_pct=0.2, seed=1),
    get_y=parent_label,
    batch_tfms=tfms1
)

```

```
40 #Creamos el DataLoaders
41 dls=galaxias.dataloaders(path_input, num_workers=0)
42 
43 #Creamos el learner con el modelo de red que queramos y con el Callback que
      muestra un grafico del loss
44 learn = cnn_learner(dls, resnet18, metrics=error_rate, cbs=CSVLogger(fname="
      ClasificacionSimpleResNet18Aug.csv", append=true))
45 
46 #Entrenamos la red
47 learn.fine_tune(20)
48 
49 #Mostramos la matriz de confusion
50 interp=ClassificationInterpretation.from_learner(learn)
51 interp.plot_confusion_matrix()
```

Fragmento de código A.5: Código para entrenar una ResNet18 con data augmentation en clasificación

A.4.3. AumentarNumeroImagenes

```
1 from PIL import Image
2 import os, os.path, sys
3 
4 #Definimos en el directorio en el que estan las imagenes que se quieren
      multiplicar.
5 path_imgs="rgb_images_200x200_duplicar"
6 
7 def main(pathCarpetas, numCopias):
8     with os.scandir(pathCarpetas) as lista_ficheros:
9         for fichero in lista_ficheros:
10             copiarImgs(fichero.name, numCopias)
11 
12 #Copia cada imagen el numero de veces especificado en numCopias.#
13 def copiarImgs(pathCarpeta, numCopias):
14     with os.scandir(path_imgs+"/"+pathCarpeta) as lista_ficheros:
15         for fichero in lista_ficheros:
16             im = Image.open(path_imgs+"/"+pathCarpeta+"/"+fichero.name)
17             ficheronuevo=fichero.name.split('.')
18             for numCopia in range(numCopias):
19                 im.save(path_imgs+"/"+pathCarpeta+"/"+ficheronuevo[0]+"__"+str
      (numCopia)+".png",'PNG')
20 
21 main(path_imgs, 1)
```

Fragmento de código A.6: Código para multiplicar el número de imágenes del dataset

A.5. Regresión morfológica de galaxias. Sección 5.5

A.5.1. CrearCsv

```python
from PIL import Image
import os, os.path, sys
import pandas as pd
import csv

path_imgs="rgb_images_200x200"
path_viejo_csv="galaxies_main_props_with_struct_params_and_no_bad_objects_and_hudf_flags_and_morphs_and_z_flags6.cat"
path_nuevo_csv="csvGalaxias.csv"
numCopias=0 #Veces que se copia cada fila del csv
df=pd.read_csv(path_viejo_csv, delim_whitespace=True)

#Parsea el nombre de la galaxia para obtener el numero
def getNumeroGalaxia(pathImagen):
    nombreImagen=str(pathImagen).split("\\")
    #print(pathImagen)
    nombreImagenSeparado=str(nombreImagen[1]).split('_')
    return int(nombreImagenSeparado[1])

#Devuelve el valor de una celda a partir del numero de galaxia y del nombre de la columna
def getFormaGalaxia(numeroGalaxia, nombreColumna):
    numFila=df[df['gal'] == numeroGalaxia].index[0]
    valor= df.at[numFila, nombreColumna]
    return float(valor)

#Devuelve un tensor con los valores morfologicos de una galaxia a partir de su path
def getProbCategories(pathImagen):
    numeroGalaxia=getNumeroGalaxia(pathImagen)
    f_spheroid=getFormaGalaxia(numeroGalaxia, 'f_spheroid')
    f_disk=getFormaGalaxia(numeroGalaxia, 'f_disk')
    f_irr=getFormaGalaxia(numeroGalaxia, 'f_irr')
    f_ps=getFormaGalaxia(numeroGalaxia, 'f_ps')
    f_unc=getFormaGalaxia(numeroGalaxia, 'f_unc')
    return [f_spheroid, f_disk, f_irr, f_ps, f_unc]

#Crea el fichero CSV
def crearCSV(path):
    with open(path, mode='w') as galaxiasCSV:
        galaxias_writer = csv.writer(galaxiasCSV, delimiter=' ', quotechar='"', quoting=csv.QUOTE_MINIMAL)
        galaxias_writer.writerow(["galPath", "f_spheroid", "f_disk", "f_irr", "f_ps", "f_unc" ])

#Inserta una nueva fila de valores en el fichero
def insertarValoresEnCSV(path, valores):
    with open(path_nuevo_csv, mode='a') as galaxiasCSV:
        galaxias_writer = csv.writer(galaxiasCSV, delimiter=' ', quotechar='"', quoting=csv.QUOTE_MINIMAL)
        galaxias_writer.writerow([path, valores[0], valores[1], valores[2], valores[3], valores[4]])
```

```
46
47 #numeroCopias indica el numero de veces que se copia cada fila del csv
48 def main(numeroCopias):
49     with os.scandir(path_imgs) as lista_ficheros:
50         for fichero in lista_ficheros:
51             pathImagen=path_imgs+"\\"+fichero.name
52             ficheronuevo=fichero.name.split('.')
53             valoresImg = getProbCategories(pathImagen)
54             insertarValoresEnCSV(pathImagen, valoresImg)
55             for numCopia in range(numeroCopias):
56                 insertarValoresEnCSV(pathImagen, valoresImg)
57
58 crearCSV(path_nuevo_csv)
59 main(numCopias)
```

Fragmento de código A.7: Código para crear un CSV

A.5.2. Regresión Simple ResNet18 sin data augmentation

```
1 from fastbook import *
2 from fastai.vision.all import *
3 import pandas as pd
4
5 path_csv='csvGalaxias.csv'
6
7 df = pd.read_csv(path_csv, delim_whitespace=True)
8 df.head()
9
10 def get_x(r): return r['galPath']
11 def get_y(r): return (tensor([r['f_spheroid'], r['f_disk'], r['f_irr'], r['
     f_ps'], r['f_unc']]))
12
13 galaxias= DataBlock(blocks=(ImageBlock, RegressionBlock),
14                  splitter = RandomSplitter(valid_pct=0.2, seed=1),
15                  get_x=get_x,
16                  get_y=get_y)
17
18 dls = galaxias.dataloaders(df, num_workers=0)
19
20 def DistanciaLoss(preds, target):
21     return (preds-target).abs().mean()
22
23 #Aplico label smoothing aleatorio del 10% maximo (+-0.5)
24 def DistanciaLossLabelSmoothing(preds, target):
25     target = target+(torch.rand(target.size(1), device='cuda')-0.5)*0.1
26     return (preds-target).abs().mean()
27
28 learn = cnn_learner(dls, resnet18, loss_func=DistanciaLossLabelSmoothing,
     metrics=DistanciaLoss, cbs=CSVLogger(fname="RegresionSimpleResNet18NoAug.
     csv", append=true))
29
```

```
30 learn.fine_tune(20)
```

Fragmento de código A.8: Código para entrenar una ResNet18 sin data augmentation en regresión

A.6. Detección de bordes de galaxias. Sección 6.2

A.6.1. Creating_rgb_images_lupton

```
1 import numpy as np
2 from astropy.io import fits
3 from astropy.table import Table
4 import matplotlib.pyplot as plt
5 import os
6 from astropy.visualization import make_lupton_rgb
7 from scipy.ndimage import gaussian_filter
8 from PIL import Image
9
10 def createRGB(b_im, g_im, r_im, stretch=1, Q=10, scale=None):
11     """
12     Crea an RGB color composite image.
13
14     Parameters
15     ----------
16     image : `MultibandImage`
17         `MultibandImage` to display.
18     bgr : sequence
19         A 3-element sequence of filter names (i.e. keys of the exps dict)
    indicating what band
20         to use for each channel. If `image` only has three filters then this
    parameter is ignored
21         and the filters in the image are used.
22     stretch: int
23         The linear stretch of the image.
24     Q: int
25         The Asinh softening parameter.
26     scale: list of 3 floats, each less than 1. (default: None)
27         Re-scales rgb channels
28
29     Returns
30     -------
31     rgb: ndarray
32         RGB (integer, 8-bits per channel) colour image as an NxNx3 numpy array
    .
33     """
34
35     if scale is None:
36         pass
37     else:
38         b_im = b_im*scale[0]
39         g_im = g_im*scale[1]
40         r_im = r_im*scale[2]
```

```
41
42      rgb = make_lupton_rgb(image_r=r_im,
43                            image_g=g_im,
44                            image_b=b_im,
45                            stretch=stretch, Q=Q)  # parameters used to stretch
    and scale the pixel values
46
47      return rgb
48
49  def main():
50      path_input = "./3abr21/images_low_21mar21/"
51      path_input_mascaras = "./3abr21/segmentation_low_21mar21/"
52      path_output_images="./images_rgb/"
53      path_output_labels="./labels/"
54      if not os.path.exists(path_output_images):
55          os.mkdir(path_output_images)
56      if not os.path.exists(path_output_labels):
57          os.mkdir(path_output_labels)
58
59  #Se lee el fichero .cat que contiene la lista de galaxias
60      tt = Table.read("benja_ppales_params_low_v4.csv",
61                      names=["id_unique","q","pa","r_edge","xcH","ycH"])
62      gals = np.array(tt["id_unique"])
63  #Se ejecuta el proceso del bucle para cada galaxia que aparece en el fichero .
    cat
64      for ii,gal in enumerate(gals):
65          scale = [1.,0.7,0.7]
66          stretch = 0.05 #Lower the stretch if you want to highlight better the
    galaxy outer parts
67          QQ = 10 #Q darkens the image
68          level_of_gray = 10 #this factor controls the level of grey outside of
    the galaxy
69          kernel_sigma = 1 #[pix]
70  #Se obtienen los ficheros de las imagenes en los 3 filtros: I, J y H
71          filename_img_b=path_input+gal+"_I.fits"
72          filename_img_g=path_input+gal+"_J.fits"
73          filename_img_r=path_input+gal+"_H.fits"
74
75          if os.path.isfile(filename_img_b)==False:
76              print("Fichero imagen no existe:", filename_img_b)
77              filename_img_b=None
78          if os.path.isfile(filename_img_g)==False:
79              print("Fichero imagen no existe:", filename_img_g)
80              filename_img_g=None
81          if os.path.isfile(filename_img_r)==False:
82              print("Fichero imagen no existe:", filename_img_r)
83              filename_img_r=None
84
85          if filename_img_b is not None:
86              hdu = fits.open(filename_img_b)
87              img_b = hdu[0].data
88              hdr_b   = hdu[0].header
```

```
89              hdu.close()
90          else:
91              print("Image B (speaking about RGB) not available", filename_img_b
    )
92          if filename_img_g is not None:
93              hdu = fits.open(filename_img_g)
94              img_g = hdu[0].data
95              hdr_g   = hdu[0].header
96              hdu.close()
97          else:
98              print("Image G (speaking about RGB) not available", filename_img_g
    )
99          if filename_img_r is not None:
100             hdu = fits.open(filename_img_r)
101             img_r = hdu[0].data
102             hdr_r   = hdu[0].header
103             hdu.close()
104         else:
105             print("Image R (speaking about RGB) not available", filename_img_r
    )
106 # Se obtienen los ficheros de mascaras
107         mask_filename=path_input_mascaras+gal+"_segmentation.fits"
108         if os.path.isfile(mask_filename) == False:
109             print("Fichero imagen no existe", mask_filename)
110             mask_filename=None
111         if mask_filename is not None:
112             hdu = fits.open(mask_filename)
113             mask = hdu[0].data
114             mask_hdr = hdu[0].header
115             hdu.close()
116         else:
117             print("Mask not available", mask_filename)
118 #Se crea la imagen RGB propiamente dicha
119         rgb_original = createRGB(img_b, img_g, img_r, scale=scale, stretch=
    stretch, Q=QQ)
120         rgb_smoothed = gaussian_filter(rgb_original, sigma = kernel_sigma)#Se
    le aplica un filtro Gaussiano para suavizar los bordes.
121         fig = plt.figure(figsize=(2,2)) #2 pulgadas de ancho y alto
122         ax = fig.add_subplot(1,1,1)
123         fig.subplots_adjust(0,0,1,1) #Quitar cualquier tipo de margen o borde
    alrededor de la imagen
124         rgb_alpha = np.stack([rgb_original[:,:,0],
125                               rgb_original[:,:,1],
126                               rgb_original[:,:,2],
127                               np.sum(rgb_smoothed, axis=2)*level_of_gray],
128                              axis=2)
129         ax.imshow(rgb_alpha)
130         ax.set_axis_off()
131 #Se guarda la imagen en formato png
132         plt.savefig(path_output_images+gal+"_rgb_image.png", format='png', dpi
    =100)
133         plt.clf()
```

```
134         plt.close(fig)
135 #Para evitar que se reescale el valor de cada punto de la mascara, hay que
    generar el png con PIL.
136 #Sumo 1 a la mascara porque los colores se ven mucho mejor en fastai. Tambien
    cambio
137         mask = mask+1
138         img=Image.fromarray(np.uint8(mask),mode='L')
139         img.save(path_output_labels+gal+"_rgb_image_M.png", dpi=(100,100),
    figsize=(2,2))
140 
141 if __name__ == "__main__":
142     main()
```

Fragmento de código A.9: Crear imágenes en RGB para segmentación

A.7. Segmentación mediante aprendizaje de redes federadas. Sección 6.5

```
1 from fastai.vision.all import *
2 import numpy as np
3 import torch
4 
5 def label_func(fn):
6     nombre=fn.stem
7     extension=fn.suffix
8     return path_mascaras+"/"+nombre+"_M"+extension
9 
10 def RotacionAleatoria(x):
11     batchsize=x.size(0)
12     result=torch.zeros(([batchsize]), dtype = torch.float, device='cuda')
13     for i in range(batchsize):
14         valor=torch.rand(1)
15         if(valor<0.25):
16             result[i]=0.0
17         elif(valor<0.5):
18             result[i]=90.0
19         elif(valor<0.75):
20             result[i]=180.0
21         else:
22             result[i]=270.0
23     return result
24 
25 class RuidoGaussiano(Transform):
26     order=100 #After normalize
27     def encodes(self, x:PILImage):
28         return x+torch.normal(mean=0.0, std=0.005, size=x.shape, device='cuda'
    )
29 
30 learn1 = load_learner("SegmentationResNet18NoAug.pkl")
31 learn2 = load_learner("SegmentationResNet18AugRotationYGauss.pkl")
```

```
learn3 = load_learner("SegmentationResNet18AugZoom.pkl")
path_images_rgb_test="images_rgb_test/"
path_labels_test="labels_test/"

sd1 = learn1.model.state_dict()
sd2 = learn2.model.state_dict()
sd3 = learn3.model.state_dict()

def get_tensor_mascara_correspondiente(fichero):
    nombre_con_extension=fichero.split('/')[-1]
    nombre_sin_extension=nombre_con_extension.split('.')[0]
    nombre_mascara=nombre_sin_extension+"_M.png"
    mascara = Image.open(path_labels_test+nombre_mascara)
    return torch.LongTensor(np.asarray(mascara))

def get_precision_media(learner):
    lista_accuracies=[]
    for fichero in os.listdir(path_images_rgb_test):
        path = os.path.join(path_images_rgb_test, fichero)
        tensor_prediccion_en_varios_canales = learner.predict(path)[2]
        tensor_mascara=get_tensor_mascara_correspondiente(fichero)
        accuracy=foreground_acc(tensor_prediccion_en_varios_canales,
   tensor_mascara, axis=0)
        lista_accuracies.append(accuracy)
    return np.asarray(lista_accuracies).mean()

learn_combinado = load_learner("SegmentationResNet18NoAug.pkl")
sd_combinado = learn_combinado.model.state_dict()

for key in sd_combinado:
    sd_combinado[key] = (1/2)*sd1[key]+(1/2)*sd2[key]
learn_combinado.model.load_state_dict(sd_combinado)
precision_media_dos_learners=get_precision_media(learn_combinado)

for key in sd_combinado:
    sd_combinado[key] = (1/3)*sd1[key]+(1/3)*sd2[key]+(1/3)*sd3[key]
learn_combinado.model.load_state_dict(sd_combinado)
precision_media_tres_learners=get_precision_media(learn_combinado)

for key in sd_combinado:
    sd_combinado[key] = sd1[key] * 0.9
learn_combinado.model.load_state_dict(sd_combinado)
precision_escalado_un_learner=get_precision_media(learn_combinado)

for key in sd_combinado:
    sd_combinado[key] = (1/10)*sd1[key]+(9/10)*sd2[key]
learn_combinado.model.load_state_dict(sd_combinado)
precision_media_ponderada_dos_learners=get_precision_media(learn_combinado)

for key in sd_combinado:
    sd_combinado[key] = (8/10)*sd1[key]+(1/10)*sd2[key]+(1/10)*sd3[key]
learn_combinado.model.load_state_dict(sd_combinado)
```

```
83 precision_media_ponderada_tres_learners=get_precision_media(learn_combinado)
84 
85 print(precision_media_dos_learners)
86 print(precision_media_tres_learners)
87 print(precision_escalado_un_learner)
88 print(precision_media_ponderada_dos_learners)
89 print(precision_media_ponderada_tres_learners)
```

Fragmento de código A.10: Combinar modelos mediante aprendizaje federado

Bibliografía

[1] Michael A. Nielsen. *Neural Networks and Deep Learning*. Determination Press, 2015. URL: http://www.neuralnetworksanddeeplearning.com.

[2] 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/.

[3] 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.

[4] Bob Hughes y Mike Cotterell. *Software Project Management*. McGraw-Hill Education, 2009. ISBN: 978-0077122799.

[5] Warren Mcculloch y Walter Pitts. «A Logical Calculus of Ideas Immanent in Nervous Activity». En: *Bulletin of Mathematical Biophysics* 5 (1943), págs. 127-147.

[6] Stanley Farlow. «The GMDH algorithm of Ivakhnenko». En: *American Statistician - AMER STATIST* 35 (nov. de 1981), págs. 210-215. DOI: 10.1080/00031305.1981.10479358.

[7] *¿Qué es el Descenso del Gradiente? Algoritmo de Inteligencia Artificial — DotCSV*. Feb. de 2018. URL: https://www.youtube.com/watch?v=A6FiCDoz8_4.

[8] Ben Frederickson. *An Interactive Tutorial on Numerical Optimization*. URL: http://www.benfrederickson.com/numerical-optimization/.

[9] Yann LeCun, Corinna Cortes y Christopher J.C. Burges. *THE MNIST DATABASE of handwritten digits*. URL: http://yann.lecun.com/exdb/mnist/.

[10] Jin-Cheng Li y col. «Bi-firing deep neural networks». En: *International Journal of Machine Learning and Cybernetics* 5 (feb. de 2014). DOI: 10.1007/s13042-013-0198-9.

[11] Shiva Sankeerth Reddy. «Understanding the Universal Approximation Theorem». En: (jul. de 2020). URL: https://towardsai.net/p/deep-learning/understanding-the-universal-approximation-theorem.

[12] Victor Powell. «Image Kernel Explained visually». En: (2017). URL: https://setosa.io/ev/image-kernels/.

[13] Quoc V. Le y col. *Building high-level features using large scale unsupervised learning*. 2012. arXiv: 1112.6209 [cs.LG].

[14] Alex Krizhevsky, Ilya Sutskever y Geoffrey E. Hinton. «ImageNet Classification with Deep Convolutional Neural Networks». En: *Proceedings of the 25th International Conference on Neural Information Processing Systems - Volume 1*. NIPS'12. Lake Tahoe, Nevada: Curran Associates Inc., 2012, págs. 1097-1105.

[15] Anaconda Inc. *Anaconda Individual Edition*. URL: https://www.anaconda.com/products/individual.

[16] Anaconda Inc. «Understanding Conda and Pip». En: (nov. de 2018). URL: https://www.anaconda.com/blog/understanding-conda-and-pip.

[17] Project Jupyter. *Project Jupyter*. URL: https://jupyter.org/.

[18] Wikipedia. *Scale-invariant feature transform*. Mar. de 2021. URL: https://en.wikipedia.org/wiki/Scale-invariant_feature_transform.

[19] Wikipedia. *Speeded up robust features*. Dic. de 2019. URL: https://en.wikipedia.org/wiki/Speeded_up_robust_features.

[20] Wikipedia. *k-d Tree*. Nov. de 2020. URL: https://en.wikipedia.org/wiki/K-d_tree.

[21] Wikipedia. *k-d Tree*. Dic. de 2020. URL: https://en.wikipedia.org/wiki/Ball_tree.

[22] Spotify. *Annoy*. URL: https://github.com/spotify/annoy.

[23] Yahoo. *NGT*. URL: https://github.com/yahoojapan/NGT.

[24] Facebook. *Faiss*. URL: https://github.com/facebookresearch/faiss.

[25] MIT. *Falconn*. URL: https://github.com/falconn-lib/falconn.

[26] Wikipedia. *Locality-sensitive hashing*. Dic. de 2020. URL: https://en.wikipedia.org/wiki/Locality-sensitive_hashing.

[27] Leslie N. Smith y Nicholay Topin. *Super-Convergence: Very Fast Training of Neural Networks Using Large Learning Rates*. 2018. arXiv: 1708.07120 [cs.LG].

[28] Wikipedia. *Astroinformatics*. Dic. de 2020. URL: https://en.wikipedia.org/wiki/Astroinformatics.

[29] Microsoft. *WSL*. URL: https://docs.microsoft.com/es-es/windows/wsl/install-win10.

[30] Wikipedia. *FITS*. Mayo de 2020. URL: https://en.wikipedia.org/wiki/FITS.

[31] IPAC. *Science Collaborations*. URL: https://www.ipac.caltech.edu/science/collaborations.

[32] *CANDELS, Cosmic Assembly Near-infrared Deep Extragalactic Legacy Survey*. 2010. URL: http://arcoiris.ucolick.org/candels/.

[33] Ryan Hausen y Brant E. Robertson. «Morpheus: A Deep Learning Framework for the Pixel-level Analysis of Astronomical Image Data». En: *The Astrophysical Journal Supplement Series* 248.1 (mayo de 2020), pág. 20. ISSN: 1538-4365. DOI: 10.3847/1538-4365/ab8868. URL: http://dx.doi.org/10.3847/1538-4365/ab8868.

[34] Olaf Ronneberger, Philipp Fischer y Thomas Brox. *U-Net: Convolutional Networks for Biomedical Image Segmentation*. 2015. arXiv: 1505.04597 [cs.CV].

[35] M. Huertas-Company y col. «A CATALOG OF VISUAL-LIKE MORPHOLOGIES IN THE 5 CANDELS FIELDS USING DEEP LEARNING». En: *The Astrophysical Journal Supplement Series* 221.1 (oct. de 2015), pág. 8. ISSN: 1538-4365. DOI: 10.1088/0067-0049/221/1/8. URL: http://dx.doi.org/10.1088/0067-0049/221/1/8.

[36] Wikipedia. *Hubble sequence*. Jun. de 2021. URL: https://en.wikipedia.org/wiki/Hubble_sequence.

[37] Kaiming He y col. *Deep Residual Learning for Image Recognition*. 2015. arXiv: 1512.03385 [cs.CV].

[38] Pablo Ruiz. «Understanding and visualizing ResNets». En: (oct. de 2018). URL: https://towardsdatascience.com/understanding-and-visualizing-resnets-442284831be8.

[39] Jeremy Zhang. «UNet — Line by Line Explanation». En: (oct. de 2019). URL: https://towardsdatascience.com/unet-line-by-line-explanation-9b191c76baf5.

[40] Brendan McMahan y Daniel Ramage. «Federated Learning: Collaborative Machine Learning without Centralized Training Data». En: (abr. de 2017). URL: https://ai.googleblog.com/2017/04/federated-learning-collaborative.html.

[41] Holger Roth Yuhong Wen Wenqi Li y Prerna Dogra. «Federated Learning powered by NVIDIA Clara». En: (dic. de 2019). URL: https://developer.nvidia.com/blog/federated-learning-clara/.

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