Grado en Ingeniería de Electrónica Industrial

Universitat de València

Trabajo Fin de Grado

Conteo de peatones mediante detección segmentada con entrenamiento de redes neuronales aplicado a un sistema Edge.

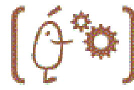
Autor: Francisco Sánchez Faubel

Tutores: Vicent Girbés Juan

Valero Laparra Pérez-Muelas

Septiembre 2022

VNIVERSITAT DE VALÈNCIA



Escola Tècnica Superior d'Enginyeria ETSE-UV

Trabajo Fin de Grado

Conteo de peatones mediante detección segmentada con entrenamiento de redes neuronales aplicado a un sistema Edge.

Autor: Francisco Sánchez Faubel

Tutores: Vicent Girbés Juan

Valero Laparra Pérez-Muelas

TRIBUNAL

PRESIDENTE/A: VOCAL 1:

VOCAL 2:

FECHA DE DEFENSA:

CALIFICACIÓN:

Declaración de autoría:

Yo, Francisco Sánchez Faubel, declaro la autoría del Trabajo Fin de Grado titulado "Conteo de peatones mediante detección segmentada con entrenamiento de redes neuronales aplicado a un sistema Edge." y que el citado trabajo no infringe las leyes en vigor sobre propiedad intelectual. El material no original que figura en este trabajo ha sido atribuido a sus legítimos autores.

Valencia, 14 de julio de 2022

Fdo: Francisco Sánchez Faubel

Resumen:

La finalidad es realizar un conteo de personas únicamente con visión, sin sensores, mediante el entrenamiento de una red neuronal y ver como esta mejora las predicciones, cómo lo hace, la eficacia y error que puede presentar ésta.
Se pretende construir la funcionalidad en un sistema embebido, un ordenador de bajos recursos y consumo reducido perfecto para instalaciones en entornos externos o incluso alimentados por placas, pues se pretenden instalar en un entorno aislado.
El siguiente TFG pretende plantear una solución más potente y compleja que sensores simples de paso o esquemas de nueves de puntos. Pues se podrá extrapolar el sistema a adquisición de datos biométricos tales como, género o edad aproximada, todo ello sin almacenar ningún tipo de imagen, únicamente con el procesado de la misma. Una posible finalidad sería la realización de estudios en establecimientos, tales como marketing, estudios de mercado o simplemente un control.

Resum:

La finalitat és realitzar un comptatge de persones únicament amb visió, sense sensors, mitjançant léntrenament dúna xarxa neuronal i veure com aquesta millora les prediccions, com ho fa, léficàcia i error que pot presentar aquesta. Es pretén construir la funcionalitat en un sistema embegut, un ordinador de baixos recursos i consum reduït perfecte per a instal·lacions en entorns externs o fins i tot alimentats per plaques, perquè es pretenen instal·lar en un entorn aïllat.
El següent TFG pretén plantejar una solució més potent i complexa que sensors simples de pas o esquemes de núvols de punts. Perquè es podrà extrapolar el sistema a adquisició de dades biomètriques com ara, gènere o edat aproximada, tot això sense emmagatzemar cap mena dímatge, únicament amb el processament dáquesta. Una possible finalitat seria la realització déstudis en establiments, com ara màrqueting, estudis de mercat o simplement un control.

Summary:

The aim is to count people only with vision, without sensors, by training a neural network and see how it improves predictions, how it does so, and the efficiency and error it can present.
It is intended to build the functionality in an embedded system, a computer with low resources and reduced consumption, perfect for installations in external environments or even powered by boards, as they are intended to be installed in an isolated environment.
The following TFG aims to propose a more powerful and complex solution than simple pitch sensors or nine-point schemes. It will be possible to extrapolate the system to the acquisition of biometric data such as gender or approximate age, all without storing any type of image, only by processing it. A possible purpose would be to carry out studies in establishments, such as marketing, market studies, or simply a control.

Agradecimientos:

En primer lugar me gustaría agradecer a todas las personas que han hecho este TFG posible, en primer lugar a mis padres, pilares y apoyo fundamental. En segundo lugar a mi pareja Ainoa por ser parte de este TFG, pues su apoyo y ánimos han sido fundamentales.

Índice general

1. Introducción **21**

1.1. Introducción 21

1.2. Motivación 21

1.3. Objetivos 21

1.4. Sistemas Edge 22

1.5. Uso y futuro 23

2. Estudio de producto **25**

2.1. Análisis de aplicaciones similares 25

3. Teoría sobre redes neuronales **29**

3.1. Introducción a las redes neuronales artificiales (RNA) 29

3.2. Comparación entre IA y Aprendizaje autónomo 30

3.3. Bloques de construcción de las redes neuronales 32

3.4. Implementación de la propagación feedforward 34

3.5. Cálculos básicos teóricos 34

3.5.1. Valores de la capa oculta 35

3.5.2. Función de activación 36

3.5.3. Valores de la capa de salida 37

3.5.4. Valores de las pérdidas 38

3.6. Código 40

3.6.1. Instalaciones 40

3.6.2. Propagación de avance A.2 41

3.6.3. Implementación de la propagación hacia atrás 42

3.7. Tasa de aprendizaje 42

3.8. Descenso gradual en código 42

3.9. Impacto del ratio de aprendizaje 44

3.10. Conclusión del proceso de entrenamiento de una red neuronal 45

3.11. Introducción a las redes neuronales convolucionales 45

3.12. CNN 45
3.12.1. Generación de CAMs 47
3.12.2. Convolución 48
3.12.3. Filtros 50

4. PyTorch **51**
4.1. CUDA B.1 51
4.2. Instalación de PyTorch 51
4.3. Tensores de PyTorch 52
4.4. Inicialización del tensor 52
4.5. Conjunto de datos 53
4.6. Construyendo una red neuronal profunda con Pytorch 53
4.7. Problemas y soluciones 57

5. Aplicación práctica **59**
5.1. Clasificación de imágenes mediante CNN profundas C.1 59
5.2. Impacto en el número de imágenes utilizadas 60
5.3. Detectron2 61
5.4. Uso de un modelo preentrenado D.1 61
5.5. Detección de la postura humana 66
5.6. Aplicaciones Extras 67
5.7. Obtención y preparación de datos D.3 68
5.8. Entrenamiento 69
5.8.1. Etiquetado Manual(Labelme) 72
5.8.2. Comparación de modelos 76
5.8.3. Problemas y soluciones 78

6. Conclusiones **79**
6.1. Conclusión 79
6.2. Mejora y futuro 80

A. Teoría de una red neuronal **81**
A.1. Instalación de librerias 81
A.2. Propagación de avance 82
A.3. Descenso gradual 83
A.4. Impacto del ratio de aprendizaje 84
A.5. Generando CAMs 85

B. Pytorch **89**

B.1. CUDA 89
B.2. Instalación de PyTorch 90

C. Clasificación y detección de objetos **91**
C.1. Clasificación de imágenes mediante CNN profundas 91

D. Aplicación práctica **95**
D.1. Uso de un modelo preentrenado 95
D.2. Estimación postura humana 96
D.3. Obtención y preparación de datos 98
D.4. Labeltome2COCO 100

Bibliografía **101**

Índice de figuras

1.1. Nvidia Jetson Nano . . . 22
1.2. Nvidia Jetson Xavier NX . . . 22
1.3. Google Coral . . . 23
1.4. Rendimientos empresariales derivados de la IA por regiones (2015-2024) (en millones de dólares) [1] ***Fuente: Tractica*** . . . 23
1.5. Un ejemplo típico de segmentación con etiquetas verdaderas y predichas superpuestas sobre cortes axiales, sagitales y coronales de IRM T1c. [2]. . . 24
1.6. Detección de peatones [3] . . . 24

2.1. Contador de personas de tipo barrera por infrarrojos para puertas y entradas 25
2.2. La luz solar en sus distintas longitudes de onda . . . 26
2.3. Sistema de control de aforo CA6826-ACR . . . 26
2.4. Sistema de control de aforo Twix Aforo-ACR . . . 27
2.5. Sistema de control de aforo semáforo pase/espere . . . 27

3.1. Tasas de error en el reto de reconocimiento visual a gran escala de ImageNet. La precisión mejoró drásticamente con la introducción del aprendizaje profundo en 2012 y siguió mejorando a partir de entonces. Los humanos actúan con una tasa de error de aproximadamente el 5 por ciento . . . 30
3.2. Diagrama explicativo de una RNA . . . 30
3.3. Ejemplo de extracción de características . . . 31
3.4. Ejemplo de extracción de características errónea . . . 31
3.5. Esquema de una red neuronal completa . . . 32
3.6. Estructura típica de una red neuronal [4] . . . 33
3.7. Neurona artificial [5] . . . 33
3.8. Red neuronal con una capa oculta con cuatro nodos en ella . . . 34
3.9. Representación de pesos y posibles valores aleatorios . . . 35
3.10. Esquema con los valores de la capa oculta (antes de la activación) . . . 36
3.11. Representaciones de las funciones de activación mencionadas anteriormente. [6] . . . 37
3.12. Cálculo de los valores finales de la capa oculta después de aplicar la activación sigmoidea. . . . 38

3.13. [7] 38
3.14. [8] 39
3.15. [9] 39
3.16. Ejemplo CAM con persona 47
3.17. Mapa de activación de clases ponderado por el gradiente 48

4.1. Elección de CUDA 51
4.2. Imagen representada como una matriz con valores de píxeles [10] 54
4.3. Histograma de luminosidad 55
4.4. Bordes y esquinas que se pueden encontrar en la imagen de ejemplo 55
4.5. Descomposición por colores 56
4.6. Menú de instalación pythorch 57
4.7. Detectron2 versiones 57

5.1. Entrenamiento con 2000 imágenes 60
5.2. Precisión de entrenamiento y validación 500 imágenes 61
5.3. Resultado del procesado de una red preentrenada 63
5.4. Resultado del procesado de una red preentrenada con segmentación th= 0.7 64
5.5. Resultado del procesado de una red preentrenada con segmentación th= 0.8 65
5.6. Resultado del procesado de una red preentrenada con segmentación th= 0.1 65
5.7. Estimación de pose con Detectron2 y keypoint_rcnn_R_101_FPN_3x.yaml y th=0.5 66
5.8. Estimación de pose con Detectron2 y keypoint_rcnn_R_101_FPN_3x.yaml y th=0.5 66
5.9. Resultado de red neuronal de R3 encargada de la seguridad y la estimación de plazas 67
5.10. Peso del datasheed 69
5.11. Primeros pasos de las iteraciones 69
5.12. Parámetros térmicos del PC mientras se procesa el entrenamiento 70
5.13. Uso de la memoria y CPU mientras se procesa el entrenamiento 70
5.14. Últimos pasos de las iteraciones 70
5.15. Resultado del entrenamiento 70
5.16. Resultado del entrenamiento 71
5.17. Resultado del entrenamiento 71
5.18. Etiquetado de datos en labelme 72
5.19. Resultado del entrenamiento 73
5.20. Resultado del entrenamiento 73
5.21. Resultado del entrenamiento 73

5.22. Resultado del entrenamiento 74
5.23. Resultado del entrenamiento 74
5.24. Resultado del entrenamiento 74
5.25. Resultado del entrenamiento 75
5.26. Ejecución del modelo 76
5.27. Ejecución del modelo Preentrenado COCO 77
5.28. Error en la ejecución de nuestra red creada a partir de pesos propios 78

6.1. Uso del tracking en detección; imágenes cedidas por R3 Recymed 80

Capítulo 1

Introducción

1.1. Introducción

La inteligencia artificial poco a poco va ganando terreno en aplicaciones complejas o nuevas necesidades que se requieren cubrir. Un claro ejemplo de esto es el control de aforos en un recinto, en muchos establecimientos se pretende contabilizar el número de personas de manera mecánica, es decir, mediante tornos de paso, barreras, tickets, incluso conteo mediante contadores manuales gestionados por una intervención de tipo humana.

1.2. Motivación

La motivación viene dada en primer lugar por las tendencias de la industria y la gran versatilidad que presentan las tecnologías de visión artificial, como estas pueden mejorar nuestras vidas exponencialmente. El concepto de SmartCity va ligado íntimamente con este tipo de tecnologías, por lo que el potencial y futuro de las aplicaciones planteadas con sistemas de inteligencia artificial está a la orden del día.

1.3. Objetivos

El Objetivo del TFG es plantear una solución a una problemática compleja. En primer lugar, se construirá una red neuronal desde cero utilizando tanto NumPy como PyTorch, conceptos que desarrollaremos con más extensión a medida que avancemos el TFG. Posteriormente, se estudiarán las mejores prácticas para ajustar los hiperparámetros de una red neuronal. A medida que avancemos, hablaremos sobre las CNN, el aprendizaje por transferencia con un enfoque en la clasificación de imágenes y los aspectos prácticos que hay que tener en cuenta al construir un modelo de NN. Finalmente, usaremos esto para implementarlo en detección de imágenes en secuencia, extracción de resultados y el posterior tratamiento de los puntos.

Una vez obtenido este motor, se trabajará la programación para establecer ID entre "frames", con el objetivo de ver la trayectoria de los puntos detectados y poder conocer la dirección de paso de la persona, contabilizando y distinguiendo el flujo de paso para poder sumar o restar el aforo de un edificio. Posteriormente, se planteará cómo se trabajan estos datos para una posible comercialización del sistema en el mercado.

1.4. Sistemas Edge

Existen infinidad de máquinas preparadas para hacer procesado en el borde en este caso llamado sistema "(Edge)", estos sistemas están optimizados para procesar estos tipos de datos con una eficiencia muy alta. Nvidia presenta muchas opciones, las más económicas serían Nvidia Jetson Nano y Nvidia Jetson Xavier NX (Ver figuras 1.1 y 1.2), no obstante el precio se puede elevar exponencialmente y con ello, su poder de computación. Una de las ventajas de usar este tipo de placas es que las podemos usar para la realización de sistemas embebidos. Las placas referenciadas 1.1 y 1.2 de Nvidia pertenecen a la familia de placas para desarrollo, ya que existe una versión del mismo sistema listo para la producción, esto se denomina módulo (SOM). Esta cualidad le confiere al diseño del sistema una agilidad requerida para la adaptabilidad del producto a la producción y a las características que se busquen en la industria. La Nvidia Jetson Xavier NX es una versión más potente, preparadas para cargas de trabajo más elevadas. Google también presenta una placa propia llamada Google coral, que se puede observar en la figura 1.3

Figura 1.1: Nvidia Jetson Nano

Figura 1.2: Nvidia Jetson Xavier NX

Figura 1.3: Google Coral

1.5. Uso y futuro

La Inteligencia Artificial (IA) es una realidad, y se ha convertido en una herramienta muy utilizada en infinidad de campos. La inteligencia artificial está siendo protagonista en algunas de las aplicaciones modernas que se utilizan a diario. Al igual que el descubrimiento/invención del fuego, la rueda, el petróleo, la electricidad y la electrónica, la Inteligencia Artificial está dando un giro a nuestro mundo de una manera asombrosa. La inteligencia artificial ha sido históricamente una herramienta de ciencias de la computación de uso minoritario, ofrecida por una minoría de desarrolladores. Pero debido al auge de excelentes teorías, aumento de la potencia de cálculo y a la disponibilidad de datos, el campo empezó a crecer exponencialmente desde la década de 2000 y no ha mostrado signos de desaceleración a corto plazo, tal y como se aprecia en las estimaciones de de la figura 1.6.

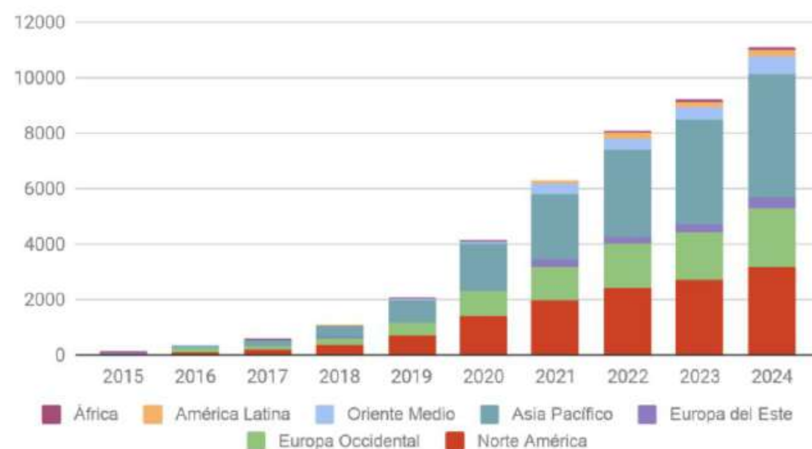


Figura 1.4: Rendimientos empresariales derivados de la IA por regiones (2015-2024) (en millones de dólares) [1] ***Fuente: Tractica***

La IA ha demostrado, que si se le da el algoritmo adecuado y una cantidad suficiente de datos, puede aprender la tarea por sí misma, con una intervención humana mínima y producir resultados que rivalizan con el juicio humano y a veces incluso lo superan.

Tanto si se trata de una persona inexperta que está aprendiendo una determinada tarea como si es un experto, hay muchas razones para entender cómo funciona la IA. Las redes neuronales son una de las clases más flexibles de algoritmos de Inteligencia Artificial que se han adaptado a una amplia gama de aplicaciones que incluyen los datos estructurados, el texto y los dominios de visión.

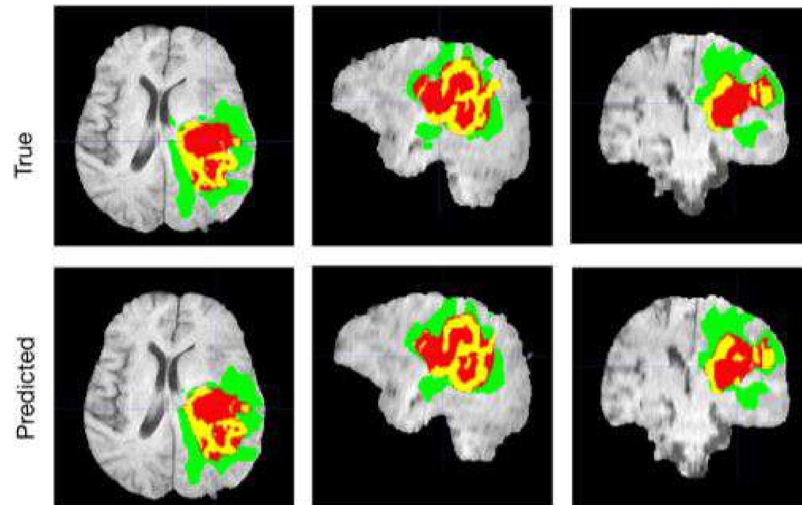


Figura 1.5: Un ejemplo típico de segmentación con etiquetas verdaderas y predichas superpuestas sobre cortes axiales, sagitales y coronales de IRM T1c. [2].

La clase de tumor completo (WT) incluye todas las etiquetas visibles (una unión de etiquetas verdes, amarillas y rojas), la clase de núcleo tumoral (TC) es una unión de rojo y amarillo, y la clase de núcleo tumoral potenciador (ET) se muestra en amarillo (una parte tumoral hiperactiva). Los usos que se le pueden dar son prácticamente infinitos, de ahí su enorme potencial a futuras aplicaciones y nuevas necesidades.

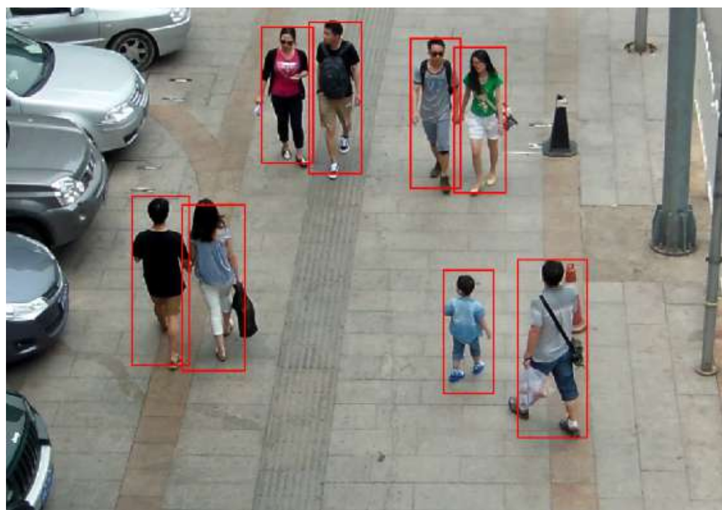


Figura 1.6: Detección de peatones [3]

Capítulo 2

Estudio de producto

2.1. Análisis de aplicaciones similares

En el TFG se pretende plantear una solución con posible comercialización que intenta solventar la problemática de contabilizar aforos de forma automática.

Figura 2.1: Contador de personas de tipo barrera por infrarrojos para puertas y entradas

El mercado ofrece soluciones basadas barreras IR, como la mostrada en la Figura 2.1. El IR es una tecnología muy económica y fácil de implementar, pero su baja sofisticación hace que presente unas limitaciones. En primer lugar, el IR no es recomendado para exteriores pues la luz solar en su descomposición, posee naturalmente luz infrarroja que podría dar errores si incide el sol de forma directa, tal y como podemos ver en la Figura 2.2

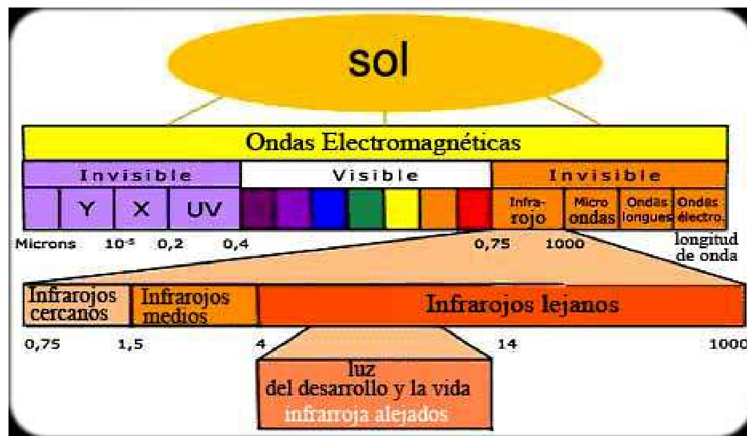


Figura 2.2: La luz solar en sus distintas longitudes de onda

La luz solar consta de distintas longitudes de onda que se combinan para producir "la luz del día"son lo que se ve cada mañana. La luz infrarroja se sitúa en las longitudes de ondas inferiores del espectro [11].

Existen diferentes controles de aforo utilizados en lugares donde es necesario saber la cantidad de personas que se encuentran dentro permitiendo que se bloquee la entrada cuando se alcanza la ocupación máxima permitida. Seguidamente se muestran algunos ejemplos de estos controles de aforo [12]:

Primeramente el control de aforo CA6826-ACR es un sensor que permite vigilar la ocupación de un espacio mediante la configuración virtual de una linea de conteo que separa las zonas de dentro/fuera. Este se instala en el techo y a través de una doble lente puede distinguir las personas que entran, incluso si entran simultáneamente, distinguiéndolas de objetos que no serán contabilizados, como por ejemplo un carro de compra. Dispone de una alarma para bloquear la entrada al llegar al máximo aforo permitido.

Figura 2.3: Sistema de control de aforo CA6826-ACR

Seguidamente, otro tipo de control de aforo son los tornos trípodes. El sistema Twix Aforo-ACR permite realizar una integración idónea como solución para limitar el acceso de personas a un lugar. El torno realiza la función de autorización de entrada a las personas abonadas o que dispongan entrada, además de la función del conteo de personas dentro del recinto. Este sistema también cuenta con una alarma que en el momento que se alcanza la ocupación máxima bloquea el torno y alerta mediante un indicador luminoso. Este tipo de control se suele implantar en espacios como piscinas, gimnasios o parques de atracciones.

Figura 2.4: Sistema de control de aforo Twix Aforo-ACR

Por último, otro tipo de control de aforo es el semáforo pase/espere. Este tipo es un sistema simple que a través de un semáforo LED verde/rojo indica si se puede pasar o debe esperar. El control de semáforo se realiza mediante un emisor de dos pulsadores donde primeramente activa el rojo y después el verde. Si la puerta de entrada/salida al espacio es automática se permite bloquear la puerta cuando el semáforo esté en rojo. Este tipo de control de aforo se puso en funcionamiento en algunos establecimientos durante el confinamiento por el COVID-19.

Figura 2.5: Sistema de control de aforo semáforo pase/espere

Capítulo 3

Teoría sobre redes neuronales

3.1. Introducción a las redes neuronales artificiales (RNA)

Una red neuronal artificial (RNA) es un algoritmo de aprendizaje supervisado inspirado en el funcionamiento del cerebro humano. Al igual que las neuronas se conectan entre sí mismas estableciendo una red. Una red neuronal toma la entrada y la pasa a través de una función, lo que da lugar a que se activen ciertas neuronas subsiguientes y, en consecuencia, se produzca la salida. Existen numerosas arquitecturas de RNA. El teorema de la aproximación universal dice que siempre podemos encontrar una arquitectura de red neuronal lo suficientemente grande con el conjunto correcto de pesos que pueda predecir exactamente cualquier salida para cualquier entrada dada. Esto significa que, para un conjunto de datos/tarea determinado, podemos crear una arquitectura y seguir ajustando sus pesos hasta que la RNA prediga lo que queremos que prediga. Este ajuste de los pesos hasta que la red pueda predecir lo que se necesite para el proceso, se llama entrenar la red neuronal.

Gracias a la realización de entrenamientos exitosos en grandes conjuntos de datos y la posibilidad del uso de arquitecturas personalizadas es la forma en que las RNA han ganado protagonismo en la resolución de diversas tareas relevantes. Una de las tareas destacadas en visión por ordenador es reconocer la clase del objeto presente en una imagen.

Un claro ejemplo de cómo ha evolucionado esta tecnología podemos verlo en una competición llamada ImageNet. En esta competencia, los participantes competían para identificar una clase de objetos en una imagen. A lo largo de los años la tasa de error se ha reducido de forma drástica.

En el año 2012 se utilizó una red neuronal (AlexNet) en la solución ganadora del concurso. Como se puede ver en el gráfico de la Figura 3.1, hubo una reducción considerable de los errores del año 2011 al año 2012 al aprovechar las redes neuronales. Desde entonces, con redes neuronales más profundas y complejas, el error de clasificación siguió reduciéndose y ha superado el rendimiento a nivel humano.

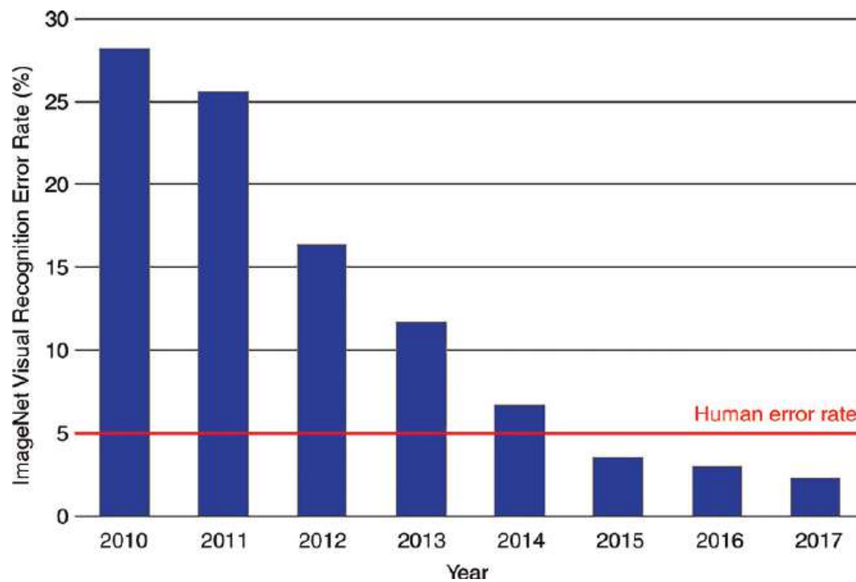


Figura 3.1: Tasas de error en el reto de reconocimiento visual a gran escala de ImageNet. La precisión mejoró drásticamente con la introducción del aprendizaje profundo en 2012 y siguió mejorando a partir de entonces. Los humanos actúan con una tasa de error de aproximadamente el 5 por ciento
[13]

3.2. Comparación entre IA y Aprendizaje autónomo

Tradicionalmente, los sistemas se planteaban utilizando sofisticados algoritmos. Supongamos que nos interesa reconocer si una foto contiene una bicicleta o no. En el entorno tradicional del aprendizaje automático, un profesional en la materia identifica primero las características que deben extraerse, es decir, los rasgos característicos y las pasa por un algoritmo que descifra las características para saber si la imagen es de una bicicleta o no. El siguiente diagrama ilustra la misma idea:

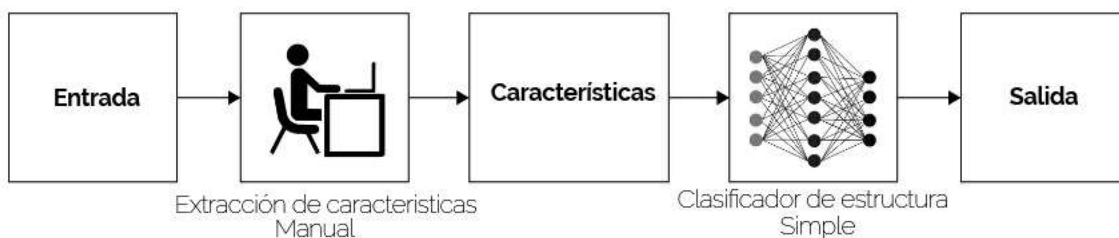


Figura 3.2: Diagrama explicativo de una RNA

Supongamos un ejemplo de que se entiende por extraer las características de una imagen según el siguiente ejemplo:

A partir de la figura 3.3, una regla sencilla podría ser que si una imagen contiene dos anillos negros alineados, puede clasificarse como una bicicleta. Sin embargo, esta regla fallaría ante un vehículo que presente estos rasgos como podría ser una motocicleta (ver Figura 3.4).

Figura 3.3: Ejemplo de extracción de características

Figura 3.4: Ejemplo de extracción de características errónea

Por supuesto, esta regla también falla cuando se muestra una imagen con cualquier cosa que no sea una bicicleta de perfil. Naturalmente, por tanto, el número de reglas manuales que tendríamos que crear para la clasificación precisa de múltiples tipos puede ser exponencial, especialmente a medida que las imágenes se vuelven más complejas. Por lo tanto, el enfoque tradicional funciona bien en un entorno restringido y muy controlado (por ejemplo, una foto de un documento de identidad, en la que todas las dimensiones están limitadas en milímetros) y funciona mal en un entorno no restringido ya que cada imagen variará mucho.

Anteriormente, si alguien estaba interesado en programar para resolver una tarea que respondía a un elemento físico del mundo real, era necesario que entendiera todo sobre los datos de entrada y escribiera tantas reglas como fuera posible para cubrir cada escenario. Esto es complicado y no hay garantía de que todas las situaciones sigan dichas reglas. Sin embargo, aprovechando las redes neuronales artificiales, podemos hacerlo en un solo paso.

Las redes neuronales ofrecen la ventaja única de combinar la extracción de características (ajuste manual) y utilizar esas características para la clasificación/regresión en una sola toma con poca ingeniería manual de características.

Estas dos subtareas sólo requieren datos etiquetados (por ejemplo, qué imágenes contienen una bicicleta y cuáles no) y una arquitectura de red neuronal.

No es necesario que un humano elabore reglas para clasificar una imagen, lo que elimina la mayor parte de la carga que las técnicas tradicionales imponen al programador. El principal requisito es proporcionar una gran cantidad de ejemplos para la tarea que necesita una solución. Por ejemplo, en el caso anterior, tenemos que proporcionar montones y montones de imágenes de bicicletas y no bicicletas al modelo para que aprenda las características. A continuación, se plantea un esquema de cómo se aprovechan las redes neuronales para la tarea de clasificación:

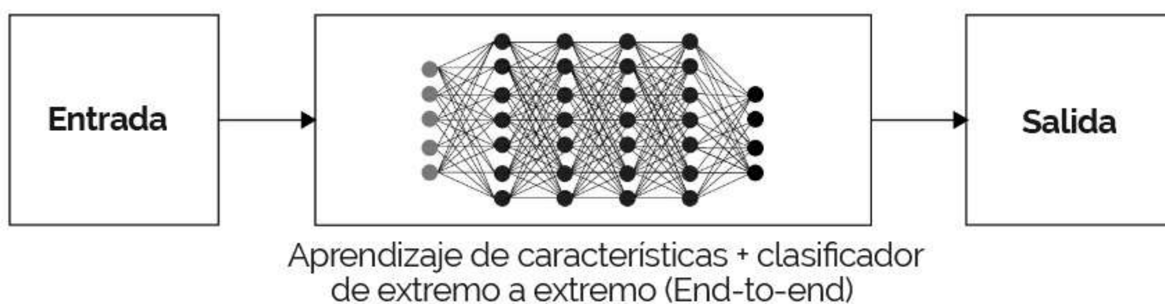


Figura 3.5: Esquema de una red neuronal completa

3.3. Bloques de construcción de las redes neuronales

Una RNA es un conjunto de tensores, también llamado (pesos) y operaciones matemáticas, dispuestas de forma que reproduzcan de una forma muy simple y de manera superficial el funcionamiento de un cerebro humano. Puede verse como una función matemática que toma uno o más tensores como entradas y predice uno o más tensores como salidas. La estructura de las operaciones que conectan estas entradas con las salidas se denominan arquitectura de la red neuronal, que podemos personalizar en función de la tarea a realizar, es decir, en función de si el problema contiene datos estructurados (tabulares) o no estructurados (imagen, texto, audio) (que es la lista de tensores de entrada y salida). Una RNA se compone de lo siguiente:

- **Capas de entrada:** Estas capas toman como entrada las variables independientes.
- **Capas ocultas (intermedias):** Estas capas conectan las capas de entrada y salida mientras realizan cambios sobre los datos de entrada. Además, las capas ocultas contienen nodos, como muestra la Figura 3.6, para modificar sus valores de entrada en valores de mayor o menor dimensión. La funcionalidad para lograr una representación más compleja se consigue utilizando diversas funciones de activación que modifican los valores de los nodos de las capas intermedias.
- **Capa de salida:** Contiene los valores que se espera obtengan las variables de entrada.

Teniendo esto en cuenta, la estructura típica de una red neuronal es la siguiente:

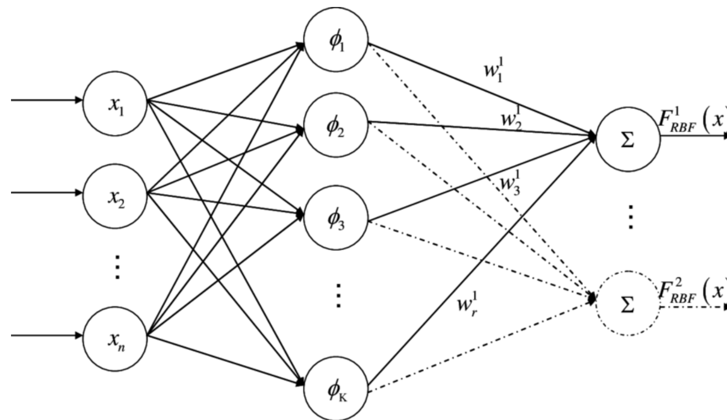


Figura 3.6: Estructura típica de una red neuronal [4]

El número de nodos (círculos en el diagrama que muestra la Figura 3.6) en la capa de salida, depende de la tarea en cuestión y de si estamos tratando de predecir una variable continua o una variable categórica. Si la salida es una variable continua, la salida tiene un nodo. Si la salida es categórica con m posibles clases, habrá m nodos en la capa de salida. Pongamos un ejemplo de uno de los nodos/neuronas y veamos lo que ocurre. Un nodo transforma sus entradas de la siguiente manera:

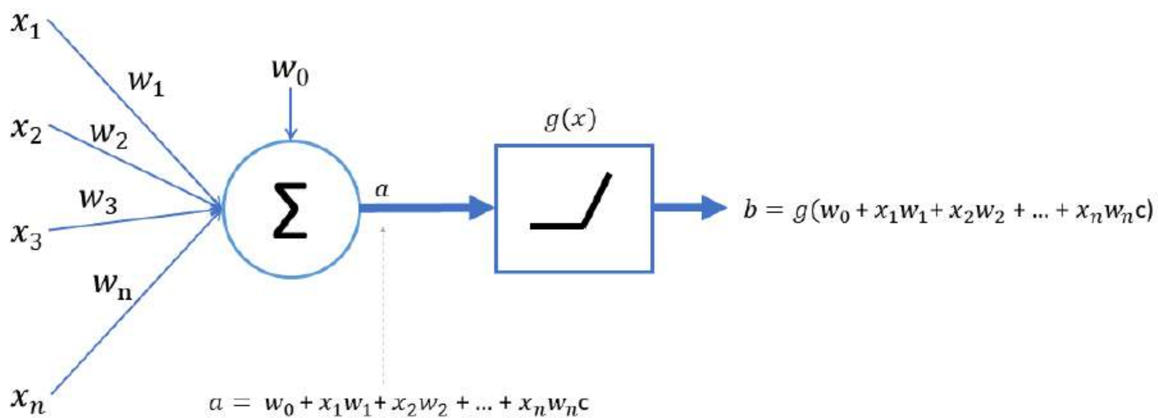


Figura 3.7: Neurona artificial [5]

En el diagrama anterior, x1,x2, ..., xn son las variables de entrada, y w0 es el término de sesgo (similar a la forma en que tenemos un sesgo en la regresión lineal/logística). W1,w2, ..., wn son las ponderaciones dadas a cada una de las variables de entrada y w0 es el término de sesgo. El valor de salida a se calcula como sigue [14]:

$$a = f(w_0 + \sum_{i=1}^{n} w_i x_i) \tag{3.1}$$

Como puede ver, es la suma de los productos de los pesos y entradas seguida de una función adicional f (el término de sesgo + la suma de productos). La función f es la función de activación que se utiliza para aplicar la no linealidad sobre esta suma de productos.

Se puede conseguir una mayor no linealidad teniendo más de una capa oculta, apilando multitud de nodos. A alto nivel, una red neuronal es una colección de nodos donde cada nodo tiene un valor flotante ajustable y los nodos están interconectados como un gráfico para devolver salidas en un formato que está dictado por la arquitectura de la red. Hay que tener en cuenta que se puede tener un número mayor (n) de capas ocultas, y el término aprendizaje profundo se refiere al mayor número de capas ocultas. Normalmente, se necesitan más capas ocultas cuando la red neuronal tiene que comprender algo complicado, como el reconocimiento de imágenes. Una vez comprendida la arquitectura de una red neuronal, es necesario entender el concepto de la propagación feedforward, que ayuda a estimar la cantidad de error (pérdida) que tiene la arquitectura de la red.

3.4. Implementación de la propagación feedforward

Para comprender bien cómo funciona la propagación feedforward, vamos a ver un ejemplo teórico simple de entrenamiento de una red neuronal. La entrada a la red neuronal es (0, 0), como se puede observar en la Figura 3.8 y la salida correspondiente (esperada) es 1. Aquí vamos a encontrar los pesos óptimos de la red neuronal basándonos en este único par de entrada-salida. Sin embargo, hay que tener en cuenta que, en realidad, habrá miles de puntos de datos sobre los que se entrena una red neuronal.

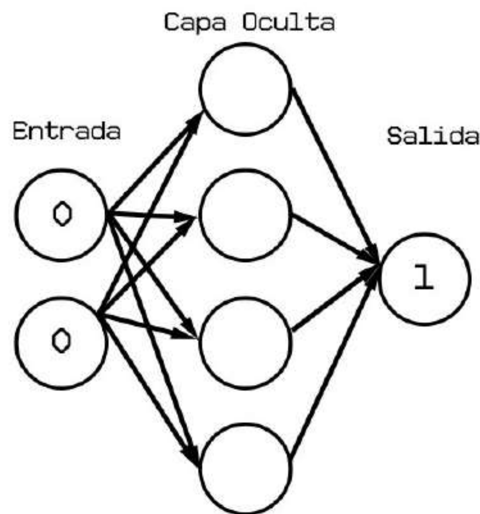


Figura 3.8: Red neuronal con una capa oculta con cuatro nodos en ella

Cada flecha del diagrama anterior contiene exactamente un valor flotante (peso) que es ajustable. Hay 10 (8 en la primera capa oculta y 4 en la segunda) valores que tenemos que encontrar, para que cuando la entrada sea (0,0), la salida sea lo más cercana posible a (1). Esto es lo que entendemos por entrenar la red neuronal.

3.5. Cálculos básicos teóricos

En los siguientes apartados, se pretende de forma teórica como calcular:

- **Valores de la capa oculta.**

- **Activaciones no lineales.**
- **Cálculo del valor de la capa de salida.**
- **Calcular el valor de pérdida correspondiente al valor esperado.**

Esto se planteará para el esquema planteado en la Figura 3.8.

3.5.1. Valores de la capa oculta

A continuación, asignaremos pesos a todas las conexiones. En el primer paso, asignamos los pesos al azar en todas las conexiones. Lo más común en las redes neuronales, es la inicialización con pesos aleatorios antes de comenzar el entrenamiento. Comencemos con los pesos iniciales que se inicializan aleatoriamente entre 0 y 1, es necesario tener en cuenta que los pesos finales después del proceso de entrenamiento de una red neuronal no necesitan estar entre un conjunto específico de valores. En el siguiente diagrama que muestra la Figura 3.9 se ofrece una representación de los pesos y valores en la red (mitad izquierda) y en la mitad derecha se ofrecen los pesos inicializados aleatoriamente en la red.

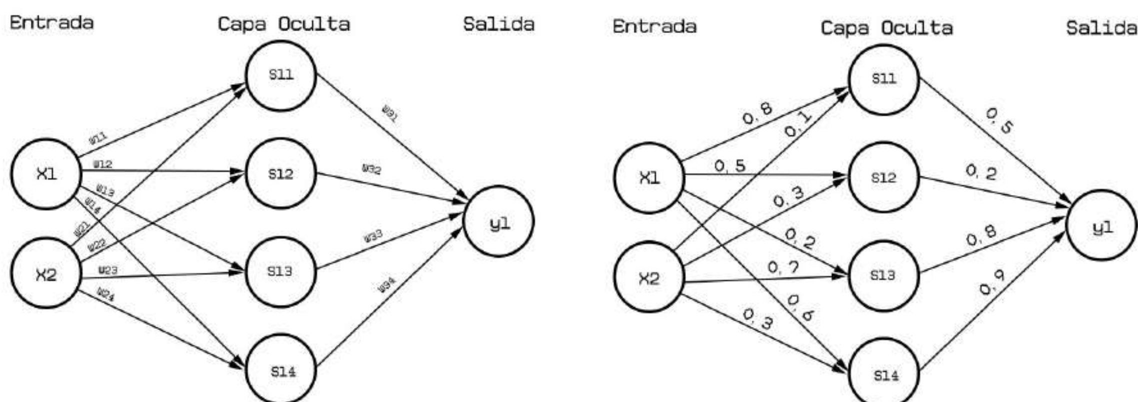


Figura 3.9: Representación de pesos y posibles valores aleatorios

En el siguiente paso, realizamos la multiplicación de la entrada con los pesos aleatorios para calcular los valores de la capa oculta. Los valores de la capa oculta antes de la activación se obtienen como sigue:

$$s_{11} = X_1 * w_{11} + X_2 * w_{21} = 1 * 0{,}8 + 1 * 0{,}1 = 0{,}9$$
$$s_{12} = X_1 * w_{12} + X_2 * w_{22} = 1 * 0{,}5 + 1 * 0{,}3 = 0{,}8$$
$$s_{13} = X_1 * w_{13} + X_2 * w_{23} = 1 * 0{,}2 + 1 * 0{,}7 = 0{,}9$$
$$s_{14} = X_1 * w_{14} + X_2 * w_{24} = 1 * 0{,}6 + 1 * 0{,}3 = 0{,}9$$

Los valores de la capa oculta (antes de la activación) que se calculan, se pueden observar en la figura 3.10:

En este ejemplo los valores son muy parejos, esto no tiene porqué ser así, los valores pueden ser mayores a la unidad y muy distinto entre ellos.

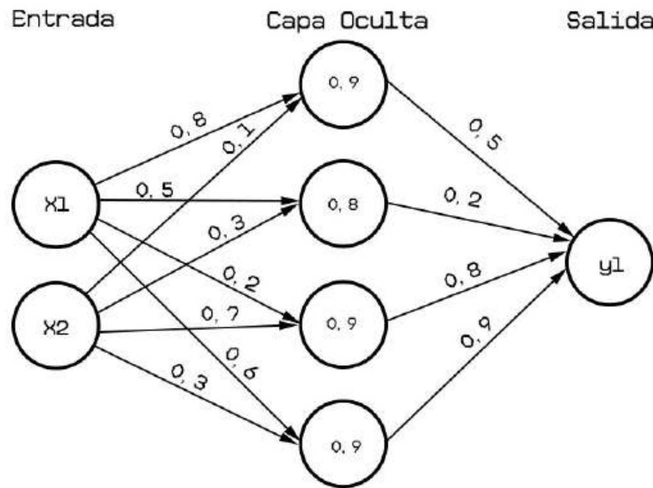


Figura 3.10: Esquema con los valores de la capa oculta (antes de la activación)

Ahora, pasaremos los valores de la capa oculta por una activación no lineal. Es de crucial importancia pasar los valores de la capa oculta por una activación no lineal, si no aplicamos una función de activación no lineal en la capa oculta, la red neuronal se convierte en una gigantesca conexión lineal de la entrada a la salida, sin importar cuántas capas ocultas existan.

3.5.2. Función de activación

Las funciones de activación ayudan a construir las conexiones matemáticamente muy complejas entre la entrada y la salida [15].

Existen numerosas funciones de activación, como ejemplo:

- **Función sigmoidea**

$$f(x) = \frac{1}{1+e^{-x}} \tag{3.2}$$

- **Función Tanh**

$$f(x) = \frac{sinh(x)}{cosh(x)} = \frac{1-e^{-2x}}{1+e^{-2x}} = \frac{e^{x}-e^{-x}}{e^{x}+e^{-x}} = \frac{e^{2x}-1}{e^{2x}+1} = 2f(x)_{sigmoid}(2x)-1 \tag{3.3}$$

- **Función ReLU**

$$f(x) = max(0,x) = \begin{cases} 0 & (x \leq 0) \\ x & (x > 0) \end{cases} \tag{3.4}$$

- **Función Lineal**

$$f(x) = x \tag{3.5}$$

Las visualizaciones de cada una de las activaciones anteriores para varios valores de entrada se pueden observar en la figura 3.11:

Como ejemplo, vamos a utilizar la función sigmoide. Aplicando la activación sigmoidea , Q(x), a las cuatro sumas de la capa oculta, obtenemos los siguientes valores tras la activación:

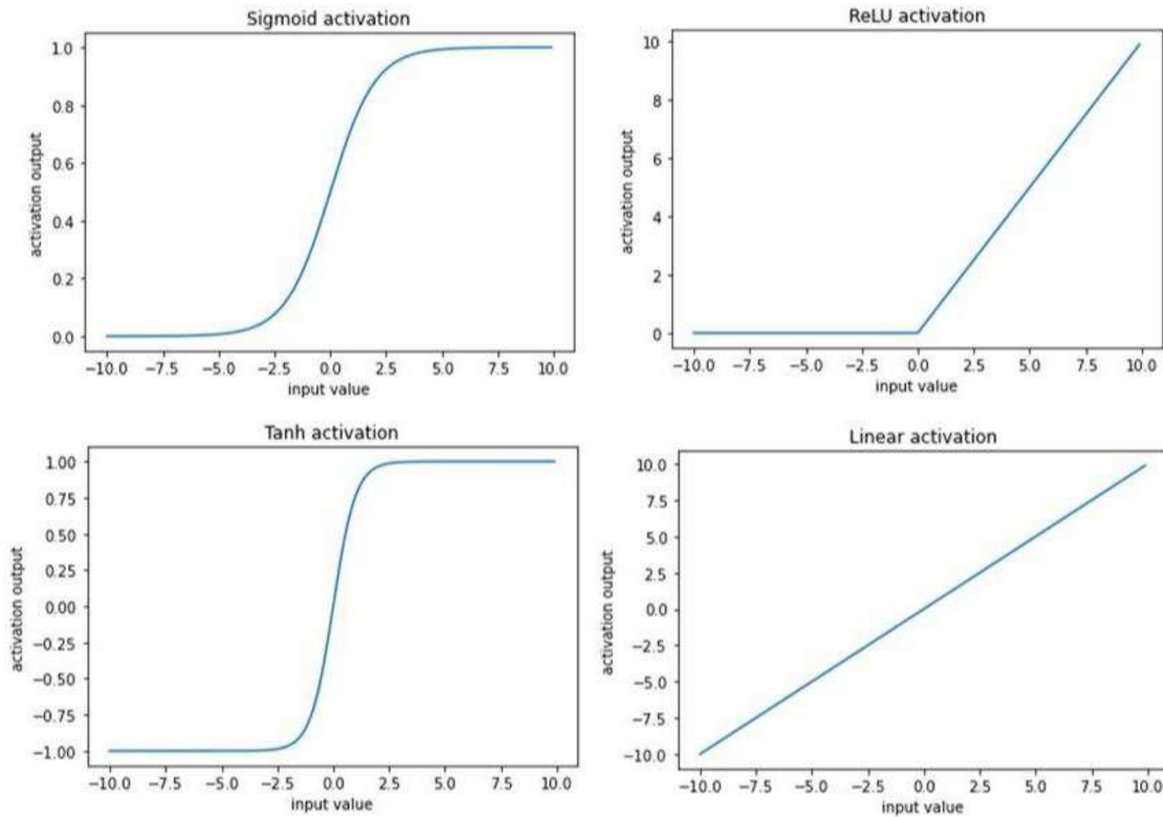


Figura 3.11: Representaciones de las funciones de activación mencionadas anteriormente. [6]

$$S_{11} = Q(0{,}9) = \frac{1}{1+e^{-0{,}9}} = 0{,}7109$$
$$S_{12} = Q(0{,}8) = \frac{1}{1+e^{-0{,}8}} = 0{,}6899$$
$$S_{13} = Q(0{,}9) = \frac{1}{1+e^{-0{,}9}} = 0{,}7109$$
$$S_{14} = Q(0{,}9) = \frac{1}{1+e^{-0{,}9}} = 0{,}7109$$

Ahora que están calculados los valores de la capa oculta después de la activación, lo siguiente es la obtención los valores de la capa de salida.

3.5.3. Valores de la capa de salida

Hasta ahora, se ha realizado el cálculo de los valores finales de la capa oculta después de aplicar la activación sigmoidea. Utilizando los valores de la capa oculta después de la activación, y los valores de los pesos (inicializados aleatoriamente en la primera iteración), el siguiente paso es obtener el valor de la salida $y1$ (Ver la Figura 3.10).

Realizamos la suma de productos de los valores de la capa oculta y los valores de los pesos para calcular el valor de salida. Esto lo consideraremos una aproximación ya que excluimos los términos de sesgo que tendrían que añadirse en cada nodo, cuando se realice la programación, (implementación en código) esto lo tendremos en cuenta y lo incluiremos en la propagación hacia adelante y en la retro propagación.

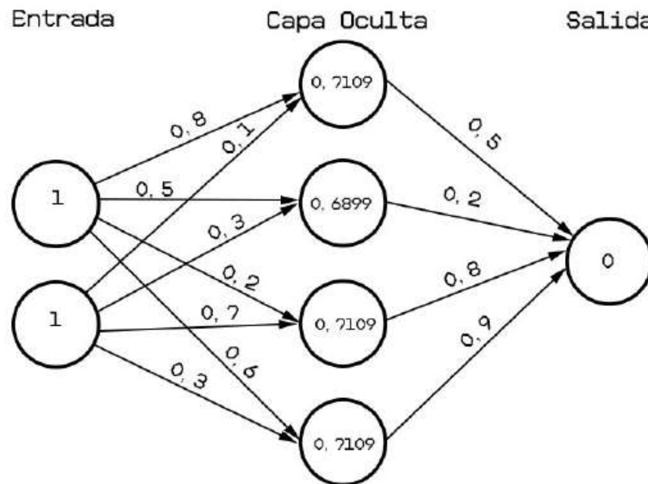


Figura 3.12: Cálculo de los valores finales de la capa oculta después de aplicar la activación sigmoidea.

$$y_1 = s_{11} * w_{31} + s_{12} * w_{32} + s_{13} * w_{33} + s_{14} * w_{34} \tag{3.6}$$

$$y_1 = 0{,}7109 * 0{,}5 + 0{,}6899 * 0{,}2 + 0{,}7109 * 0{,}8 + 0{,}7109 * 0{,}9 = 1{,}7020$$

Como empezamos con un conjunto aleatorio de pesos, el valor del nodo de salida es muy diferente del objetivo. En este caso, la diferencia es de 1.7020 (el objetivo es 0, tal y como se plantea en 3.12 y 3.9). Es necesario el calculo del valor de perdida asociado a la red en su estado actual.

3.5.4. Valores de las pérdidas

Los valores de pérdida son los valores que optimizamos en una red neuronal. Para entender cómo se calculan los valores de pérdida, veamos dos posibles situaciones:

- **Pérdida durante la predicción de la variable continua**

  Normalmente, cuando la variable es continua, el valor de la pérdida se calcula como la media del cuadrado de la diferencia entre los valores reales y las predicciones, es decir, se intenta minimizar el error medio al cuadrado variando los valores de los pesos asociados a la red neuronal. El valor del error medio al cuadrado se calcula de la siguiente manera:

$$J_\theta = \frac{1}{m} \sum_{i=1}^{m} (y_i - \hat{y}_i)^2 \tag{3.7}$$

$$\hat{y}_i = \eta_{\theta(x_i)} \tag{3.8}$$

Figura 3.13: [7]

  En la ecuación anterior, y_i es la salida real. $\hat{y}_i$ es la predicción calculada por la red neuronal η cuyos pesos se almacenan en forma de θ, donde su entrada es x_i , y m es el número de filas del conjunto de datos. El objetivo es que para cada conjunto único

de pesos, la red neuronal plantearía una pérdida diferente y tenemos que encontrar el conjunto clave de pesos para el que la pérdida es cero, o en escenarios realistas, lo más cercano a cero posible. Siguiendo con el ejemplo, supongamos que el resultado que estamos prediciendo es continuo. En ese caso, el valor de la función de pérdida es el error cuadrático medio.

$$Error(perdida) = 1{,}7020^2 = 2{,}8968$$

- **Pérdida durante la predicción de una variable categórica.**

  Cuando la variable a predecir es discreta (es decir, sólo hay unas pocas categorías en la variable), solemos utilizar una función de pérdida de entropía cruzada categórica. Cuando la variable a predecir tiene dos valores distintos, la función de pérdida es la entropía cruzada binaria. La entropía cruzada binaria se calcula como:

$$-\frac{1}{m}\sum_{i=1}^{m}(y_i log(p_i) + (1 - y_i) log(1 - p_i)) \tag{3.9}$$

Figura 3.14: [8]

y es el valor real de la salida, p es el valor predicho de la salida y m es el número total de puntos de datos.

La entropía cruzada categórica se calcula como sigue:

$$-\frac{1}{m}\sum_{j=1}^{C}\sum_{i=1}^{m} y_i log(p_i) \tag{3.10}$$

Figura 3.15: [9]

y_i es el valor real de la salida, p_i es el valor predicho de la salida, m es el número total de puntos de datos y C es el número total de clases, entendiendo en este caso como clases, aquellas cosas que queremos detectar. Una forma sencilla de visualizar la pérdida de entropía cruzada es observar la propia matriz de predicción. Supongamos que en un problema de reconocimiento de imágenes se predicen cinco clases: perro, gato y persona. La red neuronal tendría necesariamente tres neuronas en la última capa con activación Softmax, la función Softmax calcula la distribución de probabilidades del evento sobre 'n' eventos diferentes, es un tipo de función sigmoidea [16]. Así, se verá obligada a predecir una probabilidad para cada clase, para cada punto de datos. Supongamos que hay tres imágenes y que las probabilidades de predicción se ven así (la celda resaltada en cada fila corresponde a la clase objetivo:

Clases	**Probabilidades de predicción**			**Pérdida de entropía cruzada**	
	Perro	**Gato**	**Humano**		
Perro	0.91	0.01	0.08	-log(0.91)	=0.094
Gato	0.15	0.62	0.23	-log(0.62)	=0.478
Humano	0.35	0.09	0.56	-log(0.56)	=0.58

Cuadro 3.1: Ejemplo de probabilidades de una detección con eventos diferentes

Cada fila suma la unidad. En la primera fila, cuando el objetivo es Perro y la probabilidad de predicción es 0,91, la pérdida correspondiente es 0,094 (que es el negativo del logaritmo de 0,91). Del mismo modo, se calculan las demás pérdidas. Como se puede apreciar en 3.5.4 el valor de la pérdida es menor cuando la probabilidad de la clase correcta mayor. Las probabilidades pueden variar entre 0 y 1. De esta forma se deduce que, la pérdida mínima posible es 0 (cuando la probabilidad es 1) y la pérdida máxima puede ser infinita cuando la probabilidad es 0. La pérdida final dentro de un conjunto de datos es la media de todas las pérdidas individuales en todas las filas.

3.6. Código

Se va a codificar la propagación feedforward en Python utilizando matrices NumPy para solidificar los detalles de funcionamiento. El sistema operativo será Linux.

3.6.1. Instalaciones

Todos los comandos empleados para la instalación que se nombran a continuación, se pueden ver en el apéndice A.1.

- **Linux (Ubuntu 20.04.3 LTS)** Para esta instalación se ha destinado una partición del disco duro totalmente limpia. No se han utilizado maquinas virtuales ejecutando el SO desde un segundo sistema operativo.

  La finalidad de esto es simple, para las tareas que vamos a realizar, el ordenador requiere de toda su potencia, en el caso de ejecutarlo mediante una maquina virtual, limitaríamos los recursos a los definidos en la maquina virtual, pues no podemos dejar al sistema que ejecuta la maquina virtual sin recursos, esto haría que funcionara mal también el SO ejecutado en la máquina virtual.

- **Visual Studio Code 1.63.2 [17]**

  Actualizar el índice de paquetes e instalar las dependencias necesarias.

  Importar la clave GPG de Microsoft utilizando wget.

  Habilitar el repositorio de VS Code.

  Iniciar la instalación del paquete.

- **Python 3.8.3 [18]**

  Actualizar y recargar las listas de repositorios.

  Instalar el software de apoyo.

  Añadir Deadsnakes PPA Deadsnakes es un PPA con versiones más nuevas que los repositorios por defecto de Ubuntu.

  Instalar Python 3.8

- **Pip 20.1.1**

  Pip es una herramienta muy útil para instalar paquetes de Python. Con pip, se puede buscar, descargar e instalar paquetes desde Python Package Index (PyPI) y otros índices de paquetes.

- **Numpy 1.22.1**

  NumPy es una biblioteca que se puede utilizar en Python, utilizada para trabajar con matrices. También tiene funciones para trabajar en el dominio del álgebra lineal, la transformada de Fourier y las matrices.

3.6.2. Propagación de avance A.2

- **Introducir los valores**

  El primer paso es introducir los valores de las variables de entrada (inputs), los pesos (inicializados aleatoriamente si se trata de la primera iteración), y las salidas reales en el conjunto de datos proporcionado como parámetros de la función de realimentación:

```
import numpy as np
def feed_forward(entradas, salidas, pesos):
```

  Para este ejemplo, tendremos un valor de referencia asociado a cada nodo. Así, la matriz de pesos contendrá no sólo los pesos que conectan los diferentes nodos, sino también el valor asociado a los nodos de las capas ocultas y de salida.

- **Capa oculta**

  Calcular los valores de la capa oculta realizando la multiplicación de la matriz multiplicación *np.dot* de las entradas y los valores de los pesos ($pesos[0]$ que conectan la capa de entrada con la capa oculta y añadir los términos de sesgo $pesos[1]$ asociados a los nodos de la capa oculta:

```
pre_capa_oculta = np.dot(entradas,pesos[0])+ pesos[1]
```

- **Función de activación**

  El siguiente paso es aplicar la función de activación sigmoidea (en este caso) sobre los valores de la capa oculta obtenidos teniendo en cuenta el sesgo asociado a los nodos de la capa oculta:

```
capa_oculta = 1/(1+np.exp(-pre_capa_oculta))
```

- **Capa de salida**

  Para el cálculo de la capa de salida se realiza una multiplicación matricial con *np.dot* de los valores de activación de la capa oculta $Capa_oculta$ y los pesos que conectan la capa oculta con la capa de salida de salida $pesos[2]$ y sumando la salida con el sesgo asociado al nodo de la capa de salida - pesos[3] :

```
prediccion_salida = np.dot(capa_oculta, pesos[2]) + pesos[3]
```

- **Cálculo de error**

  Por último el cálculo del error.

```
error_medio_cuadratico = np.mean(np.square(prediccion_salida -
    salidas))
```

3.6.3. Implementación de la propagación hacia atrás

En la propagación hacia adelante, conectamos la capa de entrada a la capa oculta, que a su vez se conecta a la capa de salida. En la primera iteración, inicializamos los pesos al azar y luego calculamos la pérdida resultante de esos valores de peso. En la propagación hacia atrás, adoptamos el enfoque inverso. Comenzamos con el valor de pérdida obtenido en la propagación hacia adelante y actualizamos los pesos de la red de de tal manera que el valor de la pérdida se minimice al máximo.

Para reducir este error se puede hacer las siguientes acciones:

- Variar los pesos dentro de la red neuronal.
- Medir el cambio en la pérdida.
- Obtener el valor de dividir el valor de los pesos entre la pérdida y aplicar una variable que multiplica a este valor, es un valor positivo y es llamado un hiper parámetro conocido como tasa de aprendizaje.

  En este caso la actualización realizada en un peso concreto es proporcional a la cantidad de pérdida que se reduce al cambiarla en una pequeña cantidad. Intuitivamente, si el cambio de un peso reduce la pérdida en un valor grande, entonces podemos actualizar el peso en una gran cantidad. Sin embargo, si la reducción de la pérdida es pequeña al cambiar el peso, entonces actualizarlo sólo en una pequeña cantidad.

Si los pasos anteriores se realizan n veces en todo el conjunto de datos (donde hemos realizado tanto la propagación hacia delante como la retro propagación o propagación hacia atrás), se obtiene esencialmente un entrenamiento de n fases. Como una red neuronal típica contiene miles/millones (si no miles de millones) de pesos, cambiar el valor de cada peso y comprobar si la pérdida aumenta o disminuye no es óptimo. El paso central en la lista anterior es la medición del çambio de pérdidaçuando se cambia el peso. Medir esto es lo mismo que calcular el gradiente de pérdida respecto al peso.

3.7. Tasa de aprendizaje

Intuitivamente, la tasa de aprendizaje ayuda a crear confianza en el algoritmo. Por ejemplo, al decidir la magnitud de la actualización del peso, no cambiaríamos el valor del peso en una gran cantidad de una sola vez, sino que lo actualizaríamos más lentamente.

Esto permite obtener estabilidad en nuestro modelo.

3.8. Descenso gradual en código

Todo este proceso por el que actualizamos los pesos para reducir los errores se llama descenso de gradiente. El descenso de gradiente estocástico es la forma en que se minimizan los errores. El gradiente significa la diferencia (que es la diferencia en los valores de pérdida cuando el valor del peso se varía en una pequeña cantidad) y descenso significa reducir. Estocástico significa la selección de muestras aleatorias basadas en una decisión.

Aparte del descenso de gradiente estocástico, muchos otros optimizadores similares ayudan a minimizar los valores de pérdida.

Una vez definido la red feedforward y calculado el valor de pérdida del error medio cuadrático, se aumentará cada valor de los pesos y sesgos en una cantidad muy pequeña, a continuación se procederá al calculo del valor global del error cuadrático para cada una de las actualizaciones de peso y sesgo.

Creamos una función llamada $actualizacion_{pesos}$, esta función es la encargada de actualizar los valores de los pesos mediante el descenso de gradiente. Las entradas de la función son las variables de entrada a la red son tanto los pesos introducidos en primera iteración de forma aleatoria como la salida que se espera.

```
def actualizacion_pesos(entradas, salidas, pesos, lr):
```

A continuación haremos unas copias de los pesos. Como dichos pesos se van a manipular es importante utilizar la librería:

```
from copy import deepcopy
```

El funcionamiento de python no hace duplicado de memoria cuando planteamos una igualdad, más bien asigna ese espacio de memoria a la nueva variable, deepcopy asegura que podamos trabajar con múltiples copias de los pesos sin alterar los pesos reales, por lo que es de crucial importancia el uso de *deepcopy*.

```
original_pesos = deepcopy(pesos)
temp_pesos = deepcopy(pesos)
pesos_actualizados = deepcopy(pesos)
```

Calcular el valor de la pérdida con el conjunto original de pesos a través de la función feed forward planteada anteriormente:

```
perdida_real = feed_forward(entradas, salidas, original_pesos)
```

Haremos un bucle a través de todas las capas de la red: Hay un total de cuatro listas de parámetros dentro de nuestra red neuronal dos listas para los parámetros de peso y sesgo que conectan la entrada con la capa oculta y otras dos listas para los parámetros de peso y sesgo que conectan la capa oculta a la capa de salida. Dentro de ese bucle planteamos otro bucle a través de todos los parámetros individuales y como cada lista tiene una forma diferente, utilizamos numpy para recorrer cada parámetro de la lista:

```
for i, layer in enumerate(original_pesos):
    for index, weight in np.ndenumerate(layer):
```

Ahora almacenamos el conjunto original de pesos en temp pesos. Seleccionamos su peso en la capa y lo aumentamos en un pequeño valor. Por último, calculamos la nueva pérdida con el nuevo conjunto de pesos para la red neuronal:

```
temp_pesos = deepcopy(pesos)
temp_pesos[i][index] += 0.0001
_loss_plus = feed_forward(entradas, salidas, temp_pesos)
```

Calculamos el cambio en el valor de la pérdida.

```
grad = (_loss_plus - perdida_real)/(0.0001)
pesos_actualizados[i][index] -= grad*lr
```

El código quedaría: A.3

3.9. Impacto del ratio de aprendizaje

Para entender cómo repercute el ritmo de aprendizaje en el entrenamiento de un modelo se plantea un ejemplo básico, en el que tratamos de ajustar una ecuación muy sencilla, tal como:

$$y = \frac{7x}{2} \tag{3.11}$$

Planteamos y como la salida y x como la entrada. Con un conjunto de valores de entrada y con un conjunto de valores de entrada y salida esperados, intentaremos ajustar la ecuación para entender el impacto de la tasa de aprendizaje.

- **Conjunto de datos.**

```
x =   [[2],[4],[6],[8]]
y =   [[7],[14],[21],[28]]
```

- **Función de propagación hacia delante.**

  Definimos la función de propagación hacia delante. Además, en este caso modificaremos la red de tal manera que no tengamos una capa oculta para simplificar lo máximo posible.

- **Función de actualización de pesos.**

- **Valores de peso y sesgo.**

  Inicializar los valores de peso y sesgo a un valor aleatorio

```
W = [np.array ([[0]],  dtype=np.float32), np.array  ([[0]],   dtype=np.float32)]
```

- **Tasa de aprendizaje.**

  Aprovechemos la función de actualización de pesos con una tasa de aprendizaje de 0,001, repasemos 2.000 iteraciones y se visualiza cómo varía el valor de peso (W) a lo largo de épocas crecientes.

```
weight_value = []
for epx in range(2000):
    W = actualizacion_pesos(x,y,W,0.001)
    valor_peso.append(W[0][0][0])
```

  Teniendo en cuenta que, en el código anterior, estamos utilizando una tasa de aprendizaje de 0,001 y repitiendo la función de actualización de pesos para obtener el peso modificado al final de cada iteración. En cada iteración, damos el peso actualizado más reciente como entrada para para obtener el peso actualizado en la siguiente iteración.

- **Instalación matplotlib**

```
pip  install  matplotlib
```

Matplotlib es una biblioteca para crear visualizaciones estáticas, animadas e interactivas en Python. Se usarán para poder visualizar información de forma gráfica. La API de pyplot tiene una interfaz al estilo de MATLAB. De hecho, matplotlib se escribió originalmente como una alternativa de código abierto para MATLAB. La API y su interfaz son más personalizables y potentes que pyplot, pero se consideran más difíciles de usar. Como resultado, la interfaz de pyplot es más comúnmente utilizada.

- **Representar el valor.**

  En la salida anterior, el valor del peso aumentó gradualmente en la dirección correcta y luego se saturó en el valor óptimo de $\frac{7}{2}$. El código quedaría: A.4

3.10. Conclusión del proceso de entrenamiento de una red neuronal

El entrenamiento de una red neuronal es un proceso que consiste en obtener los pesos óptimos para una arquitectura de red neuronal repitiendo los dos pasos clave, la propagación hacia delante y la retro propagación con una tasa de aprendizaje determinada.

En la propagación hacia delante, aplicamos un conjunto de pesos a los datos de entrada, los hacemos pasar por las capas ocultas definidas, realizamos la activación no lineal definida en la salida de las capas ocultas y, a continuación, conectamos la capa oculta a la capa de salida multiplicando los valores de los nodos de la capa oculta por otro conjunto de pesos para estimar el valor de salida. Por último, calculamos la pérdida global correspondiente al conjunto de pesos dado. En la primera propagación hacia delante, los valores de los pesos se inicializan de forma aleatoria.

En la retro propagación, disminuimos el valor de la pérdida (error) ajustando los pesos en una dirección que reduzca la pérdida global. Además, la magnitud de la actualización de los pesos es el gradiente multiplicado por la tasa de aprendizaje.

El proceso de propagación hacia delante y de retro propagación se repite hasta que consigamos la menor pérdida posible. Esto implica que, al final del entrenamiento, la red neuronal ha ajustado sus pesos de forma que predice la salida prevista.

3.11. Introducción a las redes neuronales convolucionales

Para la tarea que queremos desarrollar, contabilizar personas humanas, necesitamos aplicar la detección de objetos y para ello aplicaremos redes neuronales convolucionales. La finalidad de ello es establecer como entrada de la red una imagen o conjunto de imágenes y determinar el número de personas que se encuentran en la misma.

3.12. CNN

Las CNN son las arquitecturas más complejas que se utilizan para trabajar con imágenes, es decir, ya no tenemos un resultado numérico puro, sino más bien se descompone

en más parámetros de salida.

Las CNN abordan las principales limitaciones de las redes neuronales que hemos visto anteriormente. Además de la clasificación de imágenes, también ayudan a la detección de segmentación de imágenes, CAMs, y muchos más, básicamente se les puede aplicar un uso a todo lo que requiera de imagen. Existen diferentes formas de construir una red neuronal convolucional, y existen múltiples modelos preentrenados que aprovechan las CNN para realizar diversas funciones.

3.12.1. Generación de CAMs

Se debe tener en cuenta un escenario en el que se ha construido un modelo que es capaz de hacer buenas predicciones. Sin embargo, la parte interesada a la que presenta el modelo quiere entender la razón por la que las predicciones del modelo son como son. Los CAM son muy útiles en este caso. Un ejemplo de CAM es el siguiente, donde tenemos la imagen de entrada a la izquierda y los píxeles que se utilizaron para llegar a la predicción de la clase resaltados a la derecha.

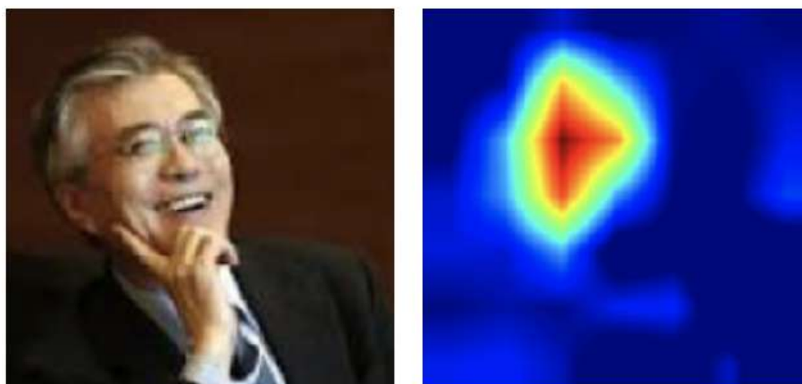


Figura 3.16: Ejemplo CAM con persona

La generación de CAM una vez entrenado el modelo se realiza de la siguiente forma. Los mapas de características son activaciones intermedias que vienen después de una operación de convolución. Normalmente, la forma de estos mapas de activación es de n canales x altura x anchura. Si tomamos la media de todas estas activaciones, muestran los puntos calientes de todas las clases de la imagen. Pero si nos interesan las localizaciones que sólo son importantes para una clase concreta (por ejemplo, la persona), tenemos que averiguar sólo aquellas características mapas entre n canales que son responsables de esa clase. Para la capa de convolución que generó estos mapas de características, podemos calcular sus gradientes con respecto a la clase persona. Se puede observar que sólo los canales responsables de la predicción de persona tendrán un gradiente alto. Esto significa que podemos utilizar la información del gradiente para dar peso a cada uno de los n canales y obtener un mapa de activación exclusivamente para la persona.

Hay una serie de pasos prácticos para llevar a cabo la generación de CAMS, primeramente se decide para qué clase quiere calcular el CAM y para qué capa convolucional de la red neuronal quiere calcular el CAM; se calculan las activaciones que surgen de cualquier capa convolucional - digamos que la forma de la característica en una capa convolucional aleatoria es 512 x 7 x 7; seguidamente se obtienen los valores de gradiente que surgen de esta capa con respecto a la clase de interés. La forma del gradiente de salida es 256 x 512 x 3 x 3 (que es la forma del tensor convolucional - es decir, canales de entrada x canales de salida x tamaño del núcleo x tamaño del núcleo); se calculan la media de los gradientes dentro de cada canal de salida. La forma de salida es 512; se calcula el mapa de activación ponderado, que es la multiplicación de las medias de los 512 gradientes por los 512 canales de activación. La forma de salida es 512 x 7 x 7; se calcula la media (a través de 512 canales) del mapa de activación ponderado para obtener una salida de la forma 7 x 7; se redimensionan (upscale) las salidas del mapa de activación ponderado para obtener una imagen del mismo tamaño que la de entrada (esto se realiza para que obtener

un mapa de activación que sea similar a la imagen original); finalmente se superpone el mapa de activación ponderado a la imagen de entrada.

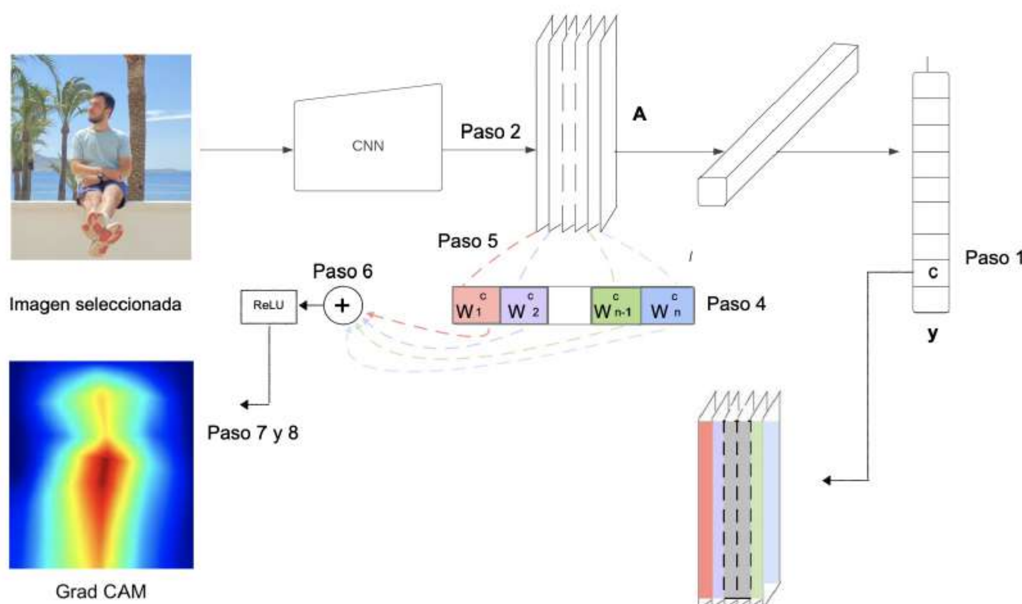


Figura 3.17: Mapa de activación de clases ponderado por el gradiente

Se puede decir que la parte más importante de todo el proceso de generación de CAMs se encuentra en el cálculo del mapa de activación ponderado (la multiplicación de las medias de los 512 gradientes por los 512 canales de activación). En este paso, podemos considerar dos aspectos: -Si un determinado píxel es importante, entonces la CNN tendrá una gran activación en esos píxeles. -Si un determinado canal convolucional es importante con respecto a la clase requerida, los gradientes en ese canal serán muy grandes. Si se multiplican estos dos, se obtendría un mapa de importancia en todos los píxeles.

El código ejecutado para poder extraer el resultado, puede encontrarse en A.5

3.12.2. Convolución

Una convolución es básicamente la multiplicación entre dos matrices. La multiplicación de matrices es un recurso clave del entrenamiento de una red neuronal. (Realizamos la multiplicación de matrices cuando calculamos los valores de la capa oculta, que es una multiplicación matricial de los valores de entrada y los valores de peso que conectan la entrada con la capa oculta. Del mismo modo, realizamos una multiplicación matricial para calcular los valores de la capa de salida).

Supongamos que tenemos dos matrices que podemos utilizar para realizar la convolución.

1	2	3	4
5	6	7	8
9	10	11	12

Cuadro 3.2: Matriz A

1	2
3	4

Cuadro 3.3: Matriz B

Mientras se realiza la operación de convolución, se desliza la matriz B (la matriz más pequeña) sobre la matriz A (la matriz más grande). Además, estamos realizando una multiplicación elemento a elemento entre la Matriz A y la Matriz B: 1. Multiplicar 1,2,5,6 de la matriz mayor por 1,2,3,4 de la matriz menor:

$$1 * 1 + 2 * 2 + 5 * 3 + 6 * 4 = 44$$

2. Multiplica 2,3,6,7 de la matriz mayor por 1,2,3,4 de la matriz menor:

$$2 * 1 + 3 * 2 + 6 * 3 + 7 * 4 = 54$$

3. Multiplica 3,4,7,8 de la matriz mayor por 1,2,3,4 de la matriz menor:

$$3 * 1 + 4 * 2 + 7 * 3 + 8 * 4 = 64$$

4. Multiplica 5,6,9,10 de la matriz mayor por 1,2,3,4 de la matriz menor:

$$5 * 1 + 6 * 2 + 9 * 3 + 10 * 4 = 84$$

5. Multiplica 6,7,10,11 de la matriz mayor por 1,2,3,4 de la matriz menor:

$$6 * 1 + 7 * 2 + 10 * 3 + 11 * 4 = 94$$

6. Multiplica 7,8,11,12 de la matriz mayor por 1,2,3,4 de la matriz menor:

$$7 * 1 + 8 * 2 + 11 * 3 + 12 * 4 = 104$$

7. Multiplica 9,10,13,14 de la matriz mayor por 1,2,3,4 de la matriz menor:

$$9 * 1 + 10 * 2 + 13 * 3 + 14 * 4 = 124$$

8. Multiplica 10,11,14,15 de la matriz mayor por 1,2,3,4 de la matriz :

$$10 * 1 + 11 * 2 + 14 * 3 + 15 * 4 = 134$$

9. Multiplica 11,12,15,16 de la matriz mayor por 1,2,3,4 de la matriz menor:

$$11 * 1 + 12 * 2 + 15 * 3 + 16 * 4 = 144$$

El resultado:

La matriz más pequeña suele llamarse filtro o núcleo, mientras que la matriz mayor es la imagen original. Entendiéndose como imagen original, una matriz en la que a cada píxel se le asigna un número.

44	54	64
84	94	104
124	134	144

Cuadro 3.4: Matriz resultado

3.12.3. Filtros

Un filtro es una matriz de pesos que se inicia aleatoriamente al principio. El modelo aprende los valores de peso óptimos de un filtro a lo largo de las de iteraciones. En general, cuantos más filtros tengamos en una CNN, más características de de una imagen que el modelo se puede aprender. Por ejemplo, un determinado filtro puede aprender sobre un rasgo característico como podría ser la lengua de un perro y proporcionar una alta activación (un valor de multiplicación de la matriz) cuando la parte de la imagen con la que convoluciona contiene la lengua del perro.

Si 10 filtros (2 x 2) diferentes multiplican la matriz más grande imagen original (4 x 4), el resultado son 10 conjuntos de matrices de salida de 3 x 3. En el caso anterior, una imagen de 4 x 4 se convoluciona con 10 filtros de 2 x 2, lo que da como resultado 3 x 3 x 10 valores de salida. Es decir, cuando una imagen es convolucionada por múltiples filtros, la salida tiene tantos canales como filtros con los que la imagen se ha convolucionado. Además, en un escenario en el que se trata de imágenes en color donde hay tres canales, el filtro que convoluciona con la imagen original también tendría tres canales, lo que daría como resultado una única salida escalar por convolución.

Capítulo 4

PyTorch

4.1. CUDA B.1

Para la instalación de CUDA es necesario disponer de un dispositivo compatible, en este caso se utilizará una tarjeta de vídeo compatible con CUDA (GeForce GTX 1050).

Introduciendo el código:

```
cat /etc/*release
```

Podremos ver el SO y la versión, esta información será necesaria para la instalación de CUDA.

```
(base) francisco@francisco-GL62M-7RDX:~$ cat /etc/*release
DISTRIB_ID=Ubuntu
DISTRIB_RELEASE=20.04
DISTRIB_DESCRIPTION="Ubuntu 20.04.3 LTS"
NAME="Ubuntu"
VERSION="20.04.3 LTS (Focal Fossa)"
```

Figura 4.1: Elección de CUDA

4.2. Instalación de PyTorch

PyTorch proporciona múltiples funcionalidades que ayudan a construir una red neuronal abstrayendo los distintos componentes utilizando un método de alto nivel y también

nos proporciona objetos tensoriales que aprovechan las GPU para entrenar una red neuronal más rápidamente. El código quedaría: B.2

Uno de los aspectos importantes de trabajar con modelos de redes neuronales es la necesidad de guardar y cargar de nuevo un modelo después de entrenarlo.

Es común realizar inferencias de un modelo ya entrenado, por lo que se piensa en un escenario en el que se cargaría el modelo entrenado en lugar de volver a entrenarlo. De ahí sale la necesidad de guardar y poder cargar el modelo.

Como vamos a trabajar con CUDA, es una buena práctica transferir el modelo a la CPU antes de llamar al script encargado de almacenar los tensores, ya que esto guardará los tensores como tensores de la CPU y no como tensores de CUDA. Esto ayudará a cargar el modelo en cualquier máquina tanto si contiene capacidades CUDA como si no.

La idea inicial es que el modelo que creemos se pueda cargar en otro dispositivo, o incluso prepararlo para embeberlo en un servidor y mediante una rest API, podamos procesar una imagen o conjunto de ellas.

4.3. Tensores de PyTorch

Los tensores son los tipos de datos fundamentales de PyTorch. Un tensor es una matriz multidimensional similar a los número de arays de NumPy: Un escalar puede ser representado como un tensor de cero dimensiones. Un vector puede ser representado como un tensor unidimensional. Una matriz bidimensional puede representarse como un tensor bidimensional. Una matriz multidimensional puede representarse como un tensor multidimensional.

Por ejemplo, podemos considerar una imagen en color como un tensor tridimensional de valores de píxeles, ya que una imagen en color consta de altura x anchura x 3 píxeles, donde los tres canales corresponden a los canales RGB. Del mismo modo, una imagen en escala de grises puede considerarse un tensor bidimensional, ya que está formada por píxeles de altura x anchura.

4.4. Inicialización del tensor

Los tensores son útiles de múltiples maneras. Aparte de utilizarlos como estructuras de datos base para las imágenes, un uso más destacado es cuando los tensores se aprovechan para inicializar los pesos que conectan las diferentes capas de una red neuronal.

Además, de forma similar a NumPy, se pueden realizar varias operaciones básicas sobre objetos tensoriales. Los paralelos a las operaciones de redes neuronales son la multiplicación matricial de la entrada con los pesos, la adición de términos de sesgo y la remodelación de los valores de entrada o de peso cuando sea necesario.

Es importante saber que se pueden hacer casi todas las operaciones de NumPy en PyTorch con casi la misma sintaxis que NumPy. Las operaciones matemáticas estándar, como abs, add, argsort, ceil, floor, sin, cos, tan, cumsum, cumprod, diag, eig, exp, log, log2, log10, mean, median, mode, resize, round, sigmoid, softmax, square, sqrt, svd, y transpose, pueden ser llamadas directamente sobre cualquier tensor con o sin ejes donde sea aplicable. Se puede ejecutar dir(torch.Tensor) para ver todos los métodos posibles para

un tensor de Torch y help(torch.Tensor.<method>) para ir a través de la ayuda oficial y la documentación para ese método.

En ciertos casos, es posible que tengamos que implementar una función de pérdida personalizada para el problema que estamos resolviendo - especialmente en casos de uso complejos que involucran la detección de objetos/redes adversas generativas (GANs). PyTorch proporciona las funcionalidades para que podamos construir una función de pérdida personalizada escribiendo una función propia.

4.5. Conjunto de datos

Un hiperparámetro de una red neuronal que aún no hemos considerado es el tamaño del lote. El tamaño del lote se refiere al número de puntos de datos considerados para calcular el valor de la pérdida o actualizar los pesos. Este hiperparámetro resulta especialmente útil en escenarios en los que hay millones de puntos de datos, y utilizarlos todos para una instancia de actualización de pesos no es óptimo, ya que la memoria no está disponible para contener tanta información. Además, una muestra puede ser suficientemente representativa de los datos. El tamaño del lote ayuda a obtener múltiples muestras de datos que son lo suficientemente representativas, pero no necesariamente el 100

4.6. Construyendo una red neuronal profunda con Pytorch

Hay una gran cantidad de entradas que se deben ajustar en una red mientras se entrena. Normalmente, estas entradas se conocen como hiperparámetros. A diferencia de los parámetros de una red neuronal (que se aprenden durante el entrenamiento), estas entradas son hiperparámetros que proporciona la persona que construye la red. Cambiar diferentes aspectos de cada hiperparámetro puede afectar a la precisión o a la velocidad de entrenamiento de una red neuronal. Además, algunas técnicas adicionales como el escalado, la normalización de lotes y la regularización ayudan a mejorar el rendimiento de una red neuronal.

Para representar una imagen, se selecciona un archivo de imagen digital (normalmente asociado a la extensión "JPEG.º "PNG") compuesto por una matriz de píxeles. Un píxel es el elemento constitutivo más pequeño de una imagen. En una imagen en escala de grises, cada píxel es un valor escalar (único) entre 0 y 255 - 0 es negro, 255 es blanco, y todo lo que está en medio es gris (cuanto más pequeño es el valor del píxel, más oscuro es). En cambio, los píxeles de las imágenes en color son vectores tridimensionales que corresponden a los valores escalares que se encuentran en sus canales rojo, verde y azul. Una imagen tiene altura x anchura x c píxeles, donde la altura es el número de filas de píxeles, la anchura es el número de columnas de píxeles y c es el número de canales. c es 3 para las imágenes en color (un canal para cada una de las intensidades de rojo, verde y azul de la imagen) y 1 para las imágenes en escala de grises, tal y como podemos ver en la Figura 4.2.

De nuevo, un valor de píxel de 0 significa que es negro total, mientras que 255 significa que es luminancia pura (es decir, blanco puro para la escala de grises y rojo/verde/azul puro en el canal respectivo para una imagen en color).

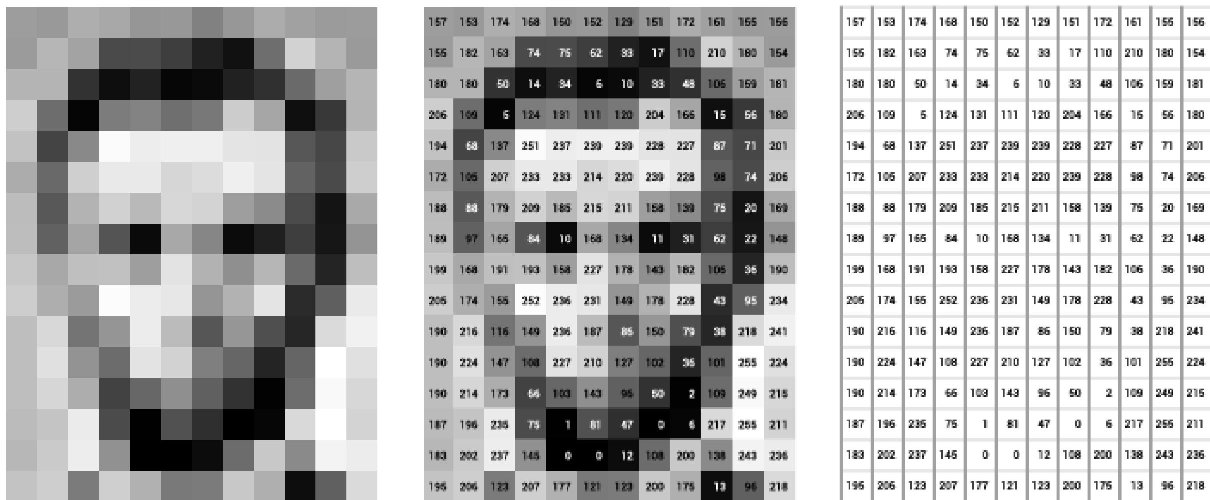


Figura 4.2: Imagen representada como una matriz con valores de píxeles [10]

Phyton puede convertir imágenes en arrays estructurados y escalares, siguiendo una serie de pasos: Se descarga una imagen; se importan las bibliotecas cv2 (para leer una imagen del disco) y matplotlib (para trazar la imagen cargada) y se lee la imagen descargada en el entorno de Python; se recorta la imagen entre las filas 50-250 y las columnas 40-240; se convierte la imagen en escala de grises y la representamos gráficamente; se transforma la imagen en una matriz de 25 x 25 y se traza (hay que tener en cuenta que debido a que se dispone de menos píxeles para representar la imagen, el resultado es más borroso); Finalmente se inspeccionan los valores de los píxeles.

Como se ha mencionado anteriormente, los píxeles con un valor escalar más cercano a 255 aparecen más claros, mientras que los más cercanos a 0 aparecen más oscuros. Esta regla también se aplica a las imágenes en color, que se representan como vectores tridimensionales. El píxel rojo más brillante se denota como (255,0,0), y del mismo modo, un píxel blanco puro en una imagen vectorial tridimensional se representa como (255,255,255).

Teniendo esto en cuenta, se crea una matriz estructurada de valores de píxeles para una imagen de color: se descarga una imagen a color; se importan los paquetes correspondientes y cargamos la imagen; se recorta la imagen (En el código debemos reordenar los canales utilizando el método cv2.cvtcolor, se realiza esto porque cuando se importan imágenes usando cv2, los canales se ordenan como Azul primero, Verde después, y finalmente Rojo ya que es habitual ver las imágenes en canales RGB, donde la secuencia es Rojo, Verde, y finalmente Azul) ; se traza la imagen obtenida; se obtiene la matriz de píxeles, se imprime y se trazan los valores de píxeles.

Convertir una imagen en una matriz estructurada de números (es decir, leer una imagen en la memoria de Python) permite realizar operaciones matemáticas sobre las imágenes. Se puede aprovechar esta estructura de datos para realizar varias tareas, como la clasificación, la detección y la segmentación.

Se debe hacer uso de las redes neuronales para el análisis de imágenes, ya que de esta forma se evita un esfuerzo al entrenar una red neuronal. En la visión por ordenador tradicional, se crearían unas cuantas características para cada imagen antes de utilizarlas como entrada.

Algunas de las características son:

- **Función de histograma:** Para algunas tareas, como el auto brillo o la visión nocturna, es importante conocer la iluminación de la imagen; es decir, la fracción de píxeles que son brillantes u oscuros. El siguiente gráfico muestra un histograma de la imagen de ejemplo:

Figura 4.3: Histograma de luminosidad

- **Característica de bordes y esquinas:** Para tareas como la segmentación de imágenes, en las que es importante encontrar el conjunto de píxeles correspondientes a cada persona, tiene sentido extraer primero los bordes, ya que el borde de una persona no es más que un conjunto de bordes. En otras tareas, como el registro de imágenes, es fundamental detectar los puntos de referencia clave. Estos puntos de referencia serán un subconjunto de todas las esquinas de una imagen. La siguiente imagen representa los bordes y esquinas que se pueden encontrar en nuestra imagen de ejemplo:

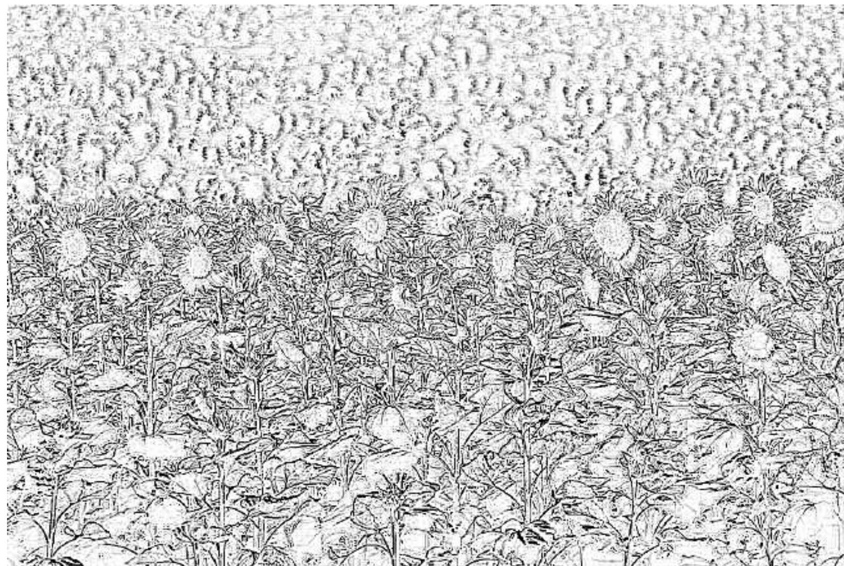


Figura 4.4: Bordes y esquinas que se pueden encontrar en la imagen de ejemplo

- **Función de separación de colores:** En tareas como la detección de semáforos para un coche teledirigido, es importante que el sistema entienda de qué color son los semáforos. La siguiente imagen (que se ve mejor en color) muestra los píxeles exclusivamente rojos, verdes y azules de la imagen de ejemplo:

Figura 4.5: Descomposición por colores

- **Función de gradientes de imagen:** Podría ser importante entender cómo los colores están cambiando a nivel de píxel. Diferentes texturas pueden ofrecer diferentes gradientes, lo que significa que pueden utilizarse como detectores de textura. De hecho, encontrar gradientes es un requisito previo para la detección de bordes.

Éstas son sólo algunas de estas características, ya que hay una gran cantidad de ellas. El principal inconveniente de la creación de estas características es que es necesario ser un experto en análisis de imágenes y señales y entender perfectamente qué características son las más adecuadas para resolver un problema. Incluso si se cumplen ambas restricciones, no hay garantía de que ese experto sea capaz de encontrar la combinación correcta de entradas, e incluso si lo hace, sigue sin haber garantía de que esa combinación funcione en escenarios nuevos e inéditos. Debido a estos inconvenientes, la comunidad se ha decantado en gran medida por los modelos basados en redes neuronales. Estos modelos no sólo encuentran las características adecuadas de forma automática, sino que también aprenden a combinarlas de forma óptima para realizar el trabajo.

4.7. Problemas y soluciones

Para la puesta a punto de todos los módulos se han encontrado unos problemas:

- **Conda:** Con Conda se puede crear, exportar, listar, eliminar y actualizar entornos que tienen diferentes versiones de Python y/o paquetes instalados en ellos. Esto es muy útil cuando se están desarrollando proyectos paralelamente que requieren dependencias muy concretas de ciertas versiones. En este caso para la instalación de Detectron 2, la versión de CUDA instalada no era compatible con entornos Conda, o por lo que se comprobó el sistema no funcionaba correctamente. Tal y como se observa en la Figura 4.6 .

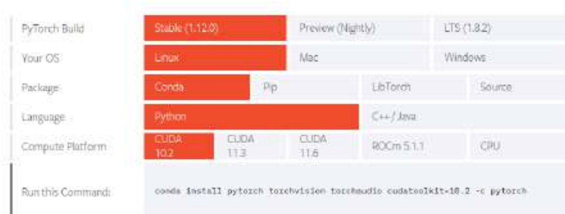


Figura 4.6: Menú de instalación pythorch

Para poder usar nuestra GPU Nvidia compatible con CUDA solo estaban disponibles de forma nativa en la web de Pythorch las versiones 10.2 , 11.3 y 11.6. En este caso se instaló CUDA 11.1, ya que para Detectron 2 es la que menos restricciones de versiones presenta para versiones de Torch, como muestra la Figura 4.7.

CUDA	torch 1.10	torch 1.9	torch 1.8
11.3	▸ install		
11.1	▸ install	▸ install	▸ install
10.2	▸ install	▸ install	▸ install
10.1			▸ install
cpu	▸ install	▸ install	▸ install

Figura 4.7: Detectron2 versiones

- **Driver NVIDIA** Otro de los grandes problemas, por error, se pretendía desinstalar CUDA 11.6 para instalar CUDA 11.1 como se ha comentado anteriormente, erróneamente también se desinstaló el driver de la controladora. La solución a este problema, fue reinstalar el SO, con la copia de seguridad creada anteriormente.

Capítulo 5

Aplicación práctica

5.1. Clasificación de imágenes mediante CNN profundas C.1

Como pequeño test de cómo se realiza la clasificación de imágenes real realizaremos una pequeña prueba para ver unos aspectos teóricos a tener en cuenta para poder estimar parámetros como la cantidad de imágenes que usaremos y como estas afectan en el rendimiento y precisión. Entrenaremos algo sencillo, distinguir un perro de un humano.

Para simplificar las cosas, las imágenes con las que trabajaremos nuestro entrenamiento las descargaremos desde una plataforma donde podemos conseguir gran cantidad de datos, kaggle [19].

Basándose en el directorio al que corresponde la imagen, proporcionamos una etiqueta de 1 para las imágenes (perro) y 0 para las imágenes (humano). Además hay que asegurarse de que la imagen obtenida ha sido normalizada a una escala entre 0 y 1 (ya que los modelos de PyTorch esperan que los canales se especifiquen primero, antes de la altura y la anchura de la imagen).

Especificamos donde se encuentran nuestras carpetas con las imagenes.

```
train_data_dir = '/home/francisco/Escritorio/Code_TFG/data_kaggle/data/training_set'
test_data_dir = '/home/francisco/Escritorio/Code_TFG/data_kaggle/data/test_set'
from torch.utils.data import DataLoader, Dataset
```

Proporcionamos la etiqueta de 1 para las imágenes (perro) y una etiqueta de 0 para las imágenes (humano).

```
class human_dogs(Dataset):
    def __init__(self, folder):
        human = glob(folder+'/human/*.jpg')
        dogs = glob(folder+'/dogs/*.jpg')
        self.fpaths = human + dogs
```

Se cambian de manera aleatoria las rutas de los archivos y se crean variables de destino basadas en la carpeta correspondiente a estas rutas de archivos:

```
        from random import shuffle, seed; seed(10); shuffle( self .fpaths)
        self . targets = [fpath. split ('/')[−1].startswith('dog') for fpath in self .fpaths] # dog=1 &
            humano=0

    def __len__(self): return len(self.fpaths)
```

Se define el método que utilizamos para especificar una ruta de archivo aleatoria de la lista de rutas, leemos la imagen, y redimensionamos todas las imágenes para que tengan un tamaño de 224 x 224. Dado que nuestra CNN espera que las entradas del canal sean de cada imagen, permutaremos la imagen redimensionada para que los canales se proporcionen primero antes de que devolvamos la imagen escalada y el valor objetivo correspondiente.

Definimos la función que entrenará el modelo en un lote de datos.

Definimos las funciones para calcular la precisión y la pérdida de validación.

Se crea la función get data, que crea un objeto de la clase human dogs y un DataLoader con un batch size de 16 para las carpetas de entrenamiento y validación. El tamaño del batch size lo va a determinar el espacio que disponga la máquina con la que trabajamos, en este caso trabajamos con un msiGL62M 7RDX con una gráfica (Trabajamos con CUDA) Nvidia GTX 1050 de 4GB, en este caso un tamaño del batch size de 32 es excesivo pues obtendríamos un error de memoria, por lo que usamos 16.

Entrenamos modelo durante 5 épocas y miramos la precisión de los datos de prueba, dichos datos, son imágenes que hemos separado en un directorio aparte para poder comprobar la precisión, dichas imágenes no tienen que estar en el conjunto de imágenes, de lo contrario no haríamos una comparación real. Definimos el modelo y obtenemos los DataLoaders necesarios.

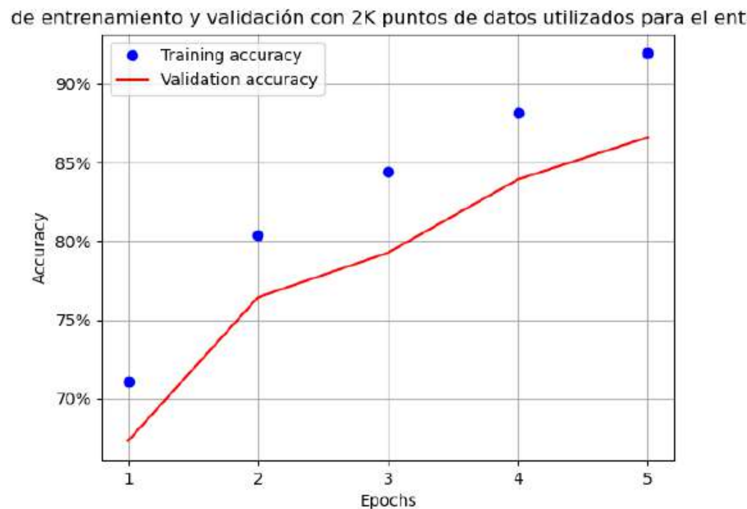


Figura 5.1: Entrenamiento con 2000 imágenes

5.2. Impacto en el número de imágenes utilizadas

Como se puede observar, contra más imágenes tenga nuestro modelo a entrenar más preciso será época tras época, en el caso de la utilización de 2000 imágenes en la época

5, tenemos una precisión de un 87/100 mientras que la utilización de las 500 imágenes en la época 5, tenemos una precisión de un 67/100 es decir, mucho menor. De ahí la importancia de utilizar una buena base de datos para construir nuestra red.

Las redes neuronales tradicionales fallan cuando se introducen nuevas imágenes muy similares a imágenes previamente vistas.

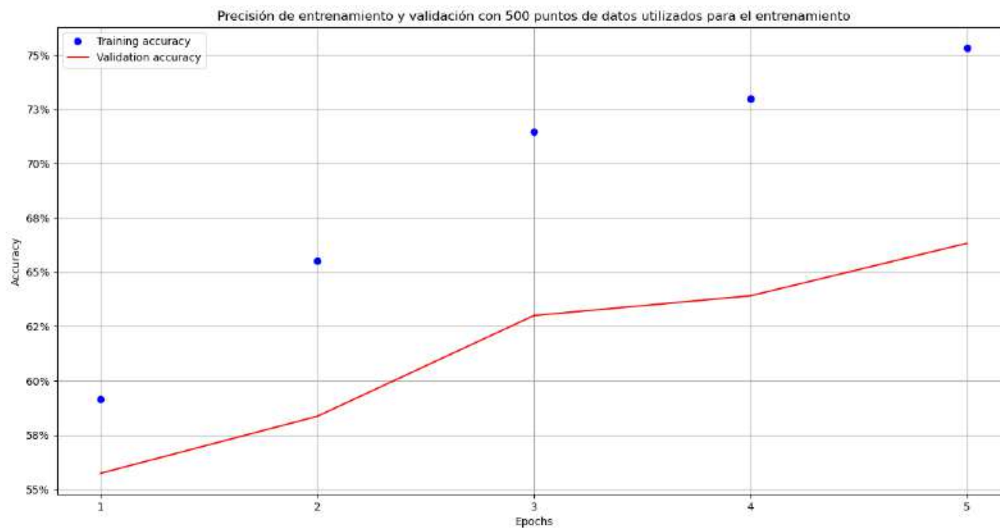


Figura 5.2: Precisión de entrenamiento y validación 500 imágenes

5.3. Detectron2

Ahora dejamos a parte aspectos más teóricos y nos centramos en escenarios más realistas, usaremos redes optimizadas para resolver problemas de detección y segmentación. Usando el marco Detectron2 para entrenar y detectar objetos presentes en una imagen.

Detectron2 soporta una serie de funciones relacionadas con la detección de objetos. Facebook AI Research (FAIR) ideó esta biblioteca avanzada, que dio resultados sorprendentes en problemas de detección y segmentación de objetos. Detectron2 se basa en el benchmark maskr cnn. Su implementación está en PyTorch. Requiere CUDA debido a la gran cantidad de cálculos que implica.

Al igual que Detectron original, admite la detección de objetos con cajas, detección de puntos clave, máscaras de segmentación de instancias y predicción de la postura humana. Además, Detectron2 añade soporte para la segmentación semántica y la segmentación panóptica (una tarea que combina la segmentación por instancias). Proporciona modelos pre-entrenados que se pueden cargar fácilmente y utilizar en nuevas imágenes.

5.4. Uso de un modelo preentrenado D.1

Se puede acceder a muchos modelos preentrenados de Detectron2 en model zoo. Estos modelos han sido entrenados en diferentes conjuntos de datos y están listos para ser utilizados.

Incluso cuando la gente entrena su conjunto de datos personalizado, es común utilizar estos pesos preentrenados para inicializar su modelo. Se ha demostrado que reduce el tiempo de entrenamiento y mejora el rendimiento. El modelo que vamos a utilizar está preentrenado en el conjunto de datos COCO. Tal y como veremos en el código de los apéndices, Detectron2 presenta una herramientas y términos utilizados que son necesarias conocer:

- **Visualizador**: El visualizador es la forma que tiene Detectron2 de representar las instancias de los objetos. Dado que las predicciones (presentes en la variable de salida) son un diccionario de tensores, el visualizador los convierte en información de píxeles y los dibuja en una imagen. Veamos qué significa cada entrada:

  im: La imagen que queremos visualizar.

  scale: El tamaño de la imagen cuando se dibuja.

  metadata: Necesitamos información de clase del conjunto de datos, principalmente el mapeo índice-clase para que cuando enviemos los tensores en bruto como entrada para ser trazados, la clase los decodificará en clases reales legibles para el ser humano.

  instance_mode: Le pedimos al modelo que sólo resalte los píxeles segmentados. Por último, una vez creada la clase, podemos pedirle que dibuje las predicciones de las instancias procedentes del modelo y que muestre la imagen.

En primer lugar, tenemos que definir la configuración completa del modelo de detección de objetos. Hemos importado la función 'get-cfg'del módulo detectron2.config, que utilizaremos ahora. Se ha elegido la configuración de segmentación de Instancia Coco (archivo YAML). Hay otras opciones disponibles también. También hay que establecer el umbral de puntuación del modelo (normalmente se establece entre 0,4 y 0,6). Una vez terminada la parte de configuración, se inicializa el DefaultPredictor con la configuración, y ya estaría listo para comenzar a predecir en las imágenes. Para hacer la detección se pasa la imagen de entrada al predictor inicializado anteriormente. Seguidamente, se utiliza herramienta Visualizer de Detectron2 para ver cómo se ha realizado la detección. Visualizer tiene una función para dibujar las predicciones de las instancias. El código ejecutado para poder obtener el resultado que se muestra en la Figura 5.5 se puede ver en el apéndice D.1.

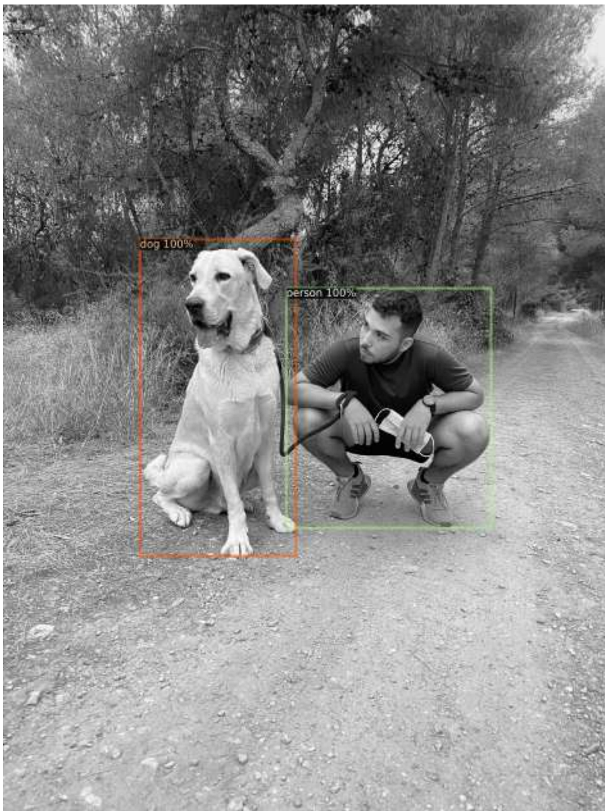


Figura 5.3: Resultado del procesado de una red preentrenada

Detectron2 permite la utilización de segmentación, lo que se consigue es establecer una zona donde la detección encuentre el objeto encontrado. La idea y el objetivo, es poder trabajar con áreas y puntos clave. Se plantea el siguiente ejemplo con la red preentrenada Detectron2:

```
self . cfg . merge\_from\_file(model\_zoo.get\_config\_file("COCO−InstanceSegmentation/mask\
    _rcnn\_R\_50\_FPN\_3x.yaml"))
self . cfg . MODEL.WEIGHTS = model\_zoo.get\_checkpoint\_url("COCO−InstanceSegmentation/
    mask\_rcnn\_R\_50\_FPN\_3x.yaml")
```

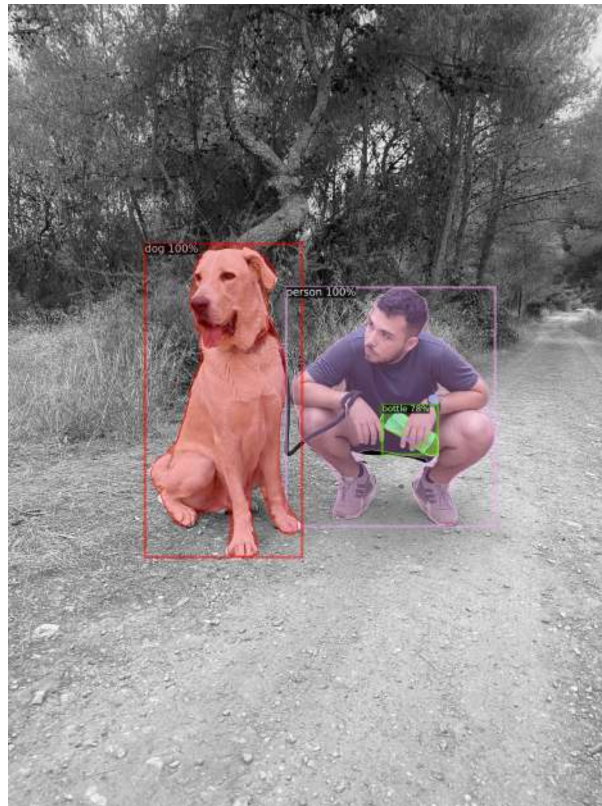


Figura 5.4: Resultado del procesado de una red preentrenada con segmentación th= 0.7

Tal y como podemos ver en la Figura 5.4, existe un error pues detecta como una botella lo que en realidad es una mascarilla. Para poder eliminar este error, se le puede exigir al script que sea más selectivo y que únicamente detecte todo aquello que sobrepase un umbral. self.cfg.MODEL.ROI_HEADS.SCORE_THRESH_TEST = 0.8

No obstante, esto no es recomendable ya que un script demasiado restrictivo conlleva a falta de detección, mientras un script excesivamente laxo, para falsos positivos. Hagamos un test con un threshold, poco restrictivo, th= 0.1

Tal y como se puede ver en la Figura 5.6 cuando disminuimos el threshold, se hacen más notables los errores, pero también es capaz de detectar más, como en el caso del reloj.

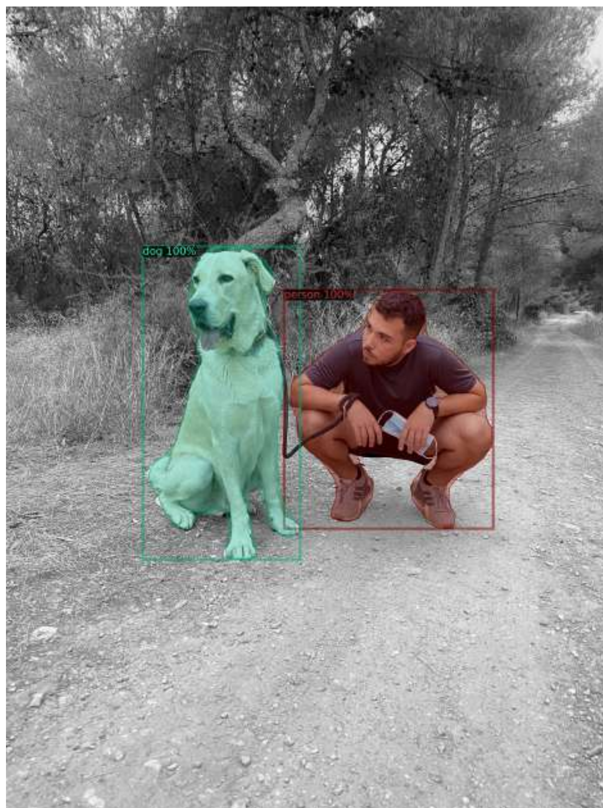


Figura 5.5: Resultado del procesado de una red preentrenada con segmentación th= 0.8

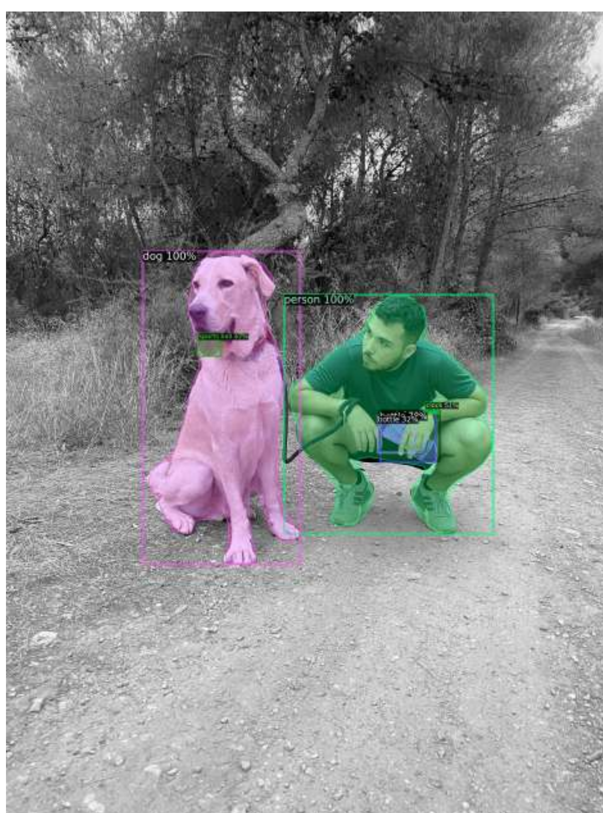


Figura 5.6: Resultado del procesado de una red preentrenada con segmentación th= 0.1

5.5. Detección de la postura humana

Detectar los puntos clave de partes del cuerpo de las personas presentes en la imagen utilizando Detectron2. La detección de puntos clave es muy útil en múltiples casos de uso, cómo en la analítica deportiva y la seguridad. Este tipo de detección de Detectron2 devuelve una serie de puntos que forman vectores. Extrapolando y explotando estos puntos se pueden obtener infinidad de información, aplicándola al sistema o la necesidad que requiera.

Figura 5.7: Estimación de pose con Detectron2 y keypoint_rcnn_R_101_FPN_3x.yaml y th=0.5

El código ejecutado para poder extraer el resultado, puede encontrarse en C.1

Figura 5.8: Estimación de pose con Detectron2 y keypoint_rcnn_R_101_FPN_3x.yaml y th=0.5

5.6. Aplicaciones Extras

Una funcionalidad en la que actualmente se ha podido trabajar es la detección de vehículos para parkings públicos sin la necesidad de sensores enterrados y sin la necesidad de delimitar plazas exactas, la empresa R3 ha cedido imágenes de un parking en Ontinyent, mediante la cual se estiman la ocupación, el procesamiento insitu mediante un sistema edge. (Ver Figura 5.9).

Figura 5.9: Resultado de red neuronal de R3 encargada de la seguridad y la estimación de plazas

5.7. Obtención y preparación de datos D.3

Por razones obvias, en este TFG no disponemos de los medios necesarios para poder realizar una base de imágenes suficientes para tener una red neuronal eficiente que pueda usarse para hacer las demostraciones. Por ese motivo trabajaremos con las imágenes disponibles en el conjunto de datos Open Images (que contiene millones de imágenes junto con sus anotaciones) proporcionado por Google) [20].

Los datos son muy grandes, por lo que es necesario plantear una criba, de lo contrario sería inutilizable, el conjunto de datos elevado es un limitante, pues es necesario tener muchos recursos para construir un modelo.

Se descarga el subconjunto de datos, Open Images tiene 16 archivos ZIP para el entrenamiento de máscaras. Cada archivo ZIP tendrá sólo unas pocas máscaras, así que borraremos el resto después de mover las máscaras necesarias en una carpeta.

Por razones de espacio, se comprimen todas las imágenes, máscaras y se guardan. Una vez creado el archivo ZIP. El tamaño del archivo acaba siendo de unos 1,6 GB. Por último, movemos los archivos a un único directorio. Dado que hay tantos componentes en movimiento en el código de detección de objetos, como forma de estandarización, Detectron acepta un formato de datos rígidos para el entrenamiento. Si bien es posible escribir una definición del conjunto de datos y usar Detectron, es más fácil guardar todos los datos de entrenamiento en COCOformat, de este modo, se pueden aprovechar otros algoritmos de entrenamiento como los transformadores de Detectron (DETR), sin modificar los datos en absoluto.

- **Definir las categorías de las clases**.
- **Definir las categorías necesarias en formato COCO**.
- **Se importan los paquetes**. Se crea un diccionario vacío con las claves necesarias para guardar el archivo JSON de COCO Se establecen algunas variables en su lugar que contienen la información sobre el directorio de las imágenes y de los archivos con las anotaciones.
- **Recorrer cada nombre de archivo de imagen y rellenar las imágenes en el diccionario coco output**.
- **Recorrer cada anotación de segmentación y rellenar la clave anotaciones en el diccionario coco output**.
- **Guardar coco output en un archivo JSON**. Con esto, tenemos nuestros archivos en formato COCO, que puede ser fácilmente para entrenar nuestro modelo con el framework Detectron2.

Dado que hay tantos componentes en el código de detección de objetos, como forma de estandarización, Detectron acepta un formato de datos rígido para la formación. Aunque es posible escribir una definición del conjunto de datos para usar Detectron2, es más fácil (y más rentable) guardar todos los datos de entrenamiento en COCOformat. De este modo, se pueden aprovechar otros algoritmos de entrenamiento como los transformadores de Detectron (DETR), sin modificar los datos.

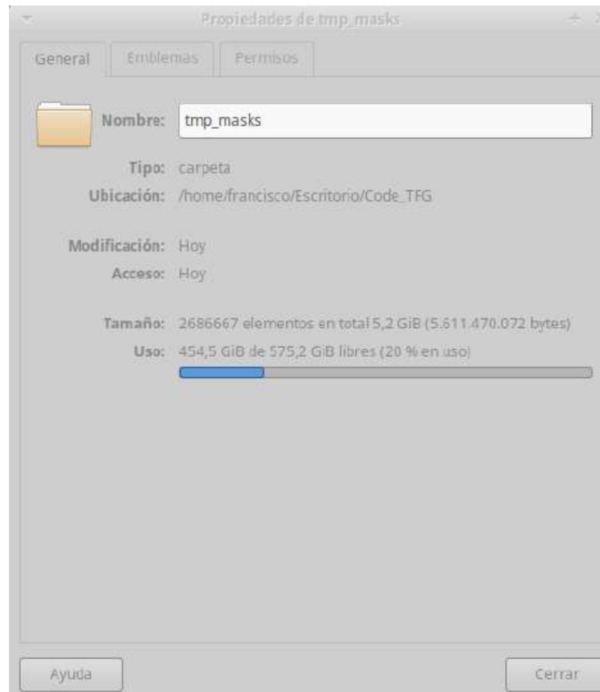


Figura 5.10: Peso del datasheed

5.8. Entrenamiento

El proceso de entrenamiento tardó aproximadamente unas 16 horas. Los ajustes del entrenamiento son:

```
cfg .DATASETS.TRAIN = ("dataset_train",)
cfg .SOLVER.IMS_PER_BATCH = 2
cfg .SOLVER.BASE_LR = 0.00025
cfg .SOLVER.MAX_ITER = 5000
cfg .MODEL.ROI_HEADS.BATCH_SIZE_PER_IMAGE = 512
```

Figura 5.11: Primeros pasos de las iteraciones

Como se puede observar el error va disminuyendo por cada iteración. Se van a realizar 5000 iteraciones.

Tal y como se observa en las Figuras 5.12 y 5.13, quién se encarga de procesar la información es la parte gráfica del sistema, gracias a la compatibilidad con CUDA. Graficando los valores del entrenamiento, podemos ver como tras las primeras iteraciones, el valor del error decrece de forma exponencial, no obstante no tarda en estabilizar estabilizarse poco después, esto es causado por la acción cfg.SOLVER.BASE_LR = 0.00025, en la que se busca dicha tasa de aprendizaje, esto ocurre aproximadamente en la iteración 1000, tal y como podemos observar en la gráfica 5.17.

Una de las opciones por lo que un entrenamiento podría no ser eficaz es por la falta de data o unos parámetros poco exigentes, extraemos como conclusión que se requiere de

Sensor	Value	Min	Max	Color	Graph
temp1	52°C	51°C	53°C		
Package id 0	92°C	82°C	93°C		
Core 0	81°C	72°C	93°C		
Core 1	68°C	68°C	87°C		
Core 2	92°C	70°C	93°C		
Core 3	74°C	73°C	92°C		
temp1	56°C	56°C	56°C		
temp1	92°C	88°C	94°C		
temp2	28°C	28°C	28°C		
temp3	30°C	30°C	30°C		
NVIDIA GeForce GTX 1050 0 temp	76°C	73°C	78°C		
NVIDIA GeForce GTX 1050 0 graphics	100%	93%	100%		
NVIDIA GeForce GTX 1050 0 video	0%	0%	0%		
NVIDIA GeForce GTX 1050 0 memory	69%	50%	93%		
NVIDIA GeForce GTX 1050 0 PCIe	0%	0%	1%		
CPU usage	32%	22%	45%		
free memory	3%	3%	3%		
HGST HTS721010A9E630	45°C	45°C	46°C		

Figura 5.12: Parámetros térmicos del PC mientras se procesa el entrenamiento

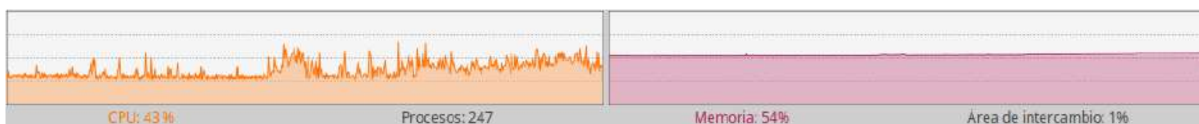


Figura 5.13: Uso de la memoria y CPU mientras se procesa el entrenamiento

gran capacidad de procesamiento y una base de datos muy extensa. Estos datos se han obtenido para 5000 iteraciones y el PC muestra el aviso de que la memoria se ha excedido.

```
eta: 0:02:20  iter: 4899  total_loss: 0.6864  loss_cls: 0.1367  loss_box_reg: 0.118  loss_mask: 0.3426  loss_rpn_cls: 0.02358  loss_rpn_loc: 0.01353  time: 1.4419  data_time: 0.0038  lr: 0.00025  max_mem: 3029M
eta: 0:01:52  iter: 4919  total_loss: 0.8105  loss_cls: 0.1382  loss_box_reg: 0.1101  loss_mask: 0.4339  loss_rpn_cls: 0.02763  loss_rpn_loc: 0.01226  time: 1.4419  data_time: 0.0035  lr: 0.00025  max_mem: 3029M
eta: 0:01:24  iter: 4939  total_loss: 0.7244  loss_cls: 0.1517  loss_box_reg: 0.1401  loss_mask: 0.4067  loss_rpn_cls: 0.01599  loss_rpn_loc: 0.01284  time: 1.4421  data_time: 0.0033  lr: 0.00025  max_mem: 3029M
eta: 0:00:55  iter: 4959  total_loss: 0.8215  loss_cls: 0.1723  loss_box_reg: 0.15  loss_mask: 0.4895  loss_rpn_cls: 0.02615  loss_rpn_loc: 0.01204  time: 1.4419  data_time: 0.0033  lr: 0.00025  max_mem: 3029M
eta: 0:00:28  iter: 4979  total_loss: 0.7808  loss_cls: 0.1682  loss_box_reg: 0.1338  loss_mask: 0.3829  loss_rpn_cls: 0.02862  loss_rpn_loc: 0.01246  time: 1.4421  data_time: 0.0035  lr: 0.00025  max_mem: 3029M
eta: 0:00:00  iter: 4999  total_loss: 0.8021  loss_cls: 0.1154  loss_box_reg: 0.09302  loss_mask: 0.4272  loss_rpn_cls: 0.02336  loss_rpn_loc: 0.01285  time: 1.4421  data_time: 0.0038  lr: 0.00025  max_mem: 3029M
Overall training speed: 4998 iterations in 2:00:07 (1.4421 s / it)
Total training time: 2:00:32 (0:00:24 on hooks)
```

Figura 5.14: Últimos pasos de las iteraciones

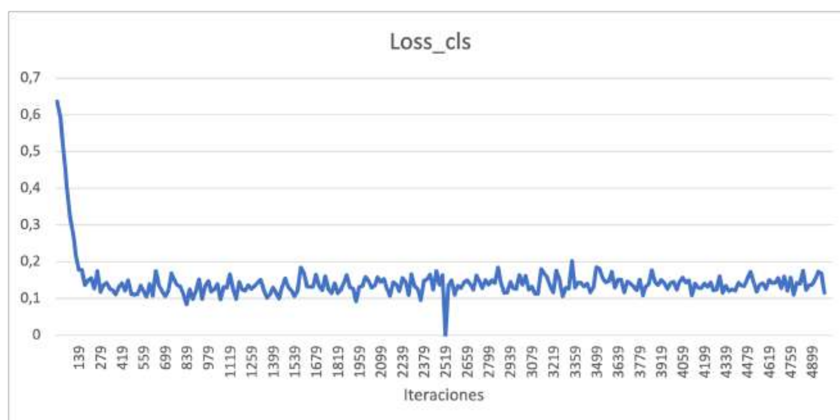


Figura 5.15: Resultado del entrenamiento

Pérdida de clasificación del ROI. Mide la pérdida por clasificación, es decir, lo bueno que es el modelo a la hora de etiquetar una zona predicha con la clase correcta. Es una

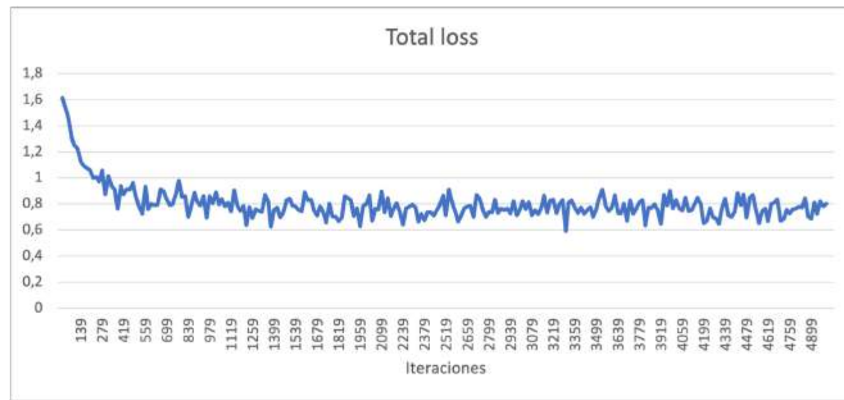


Figura 5.16: Resultado del entrenamiento

suma ponderada de las siguientes pérdidas individuales calculadas durante la iteración. Por defecto, los pesos son todos uno.

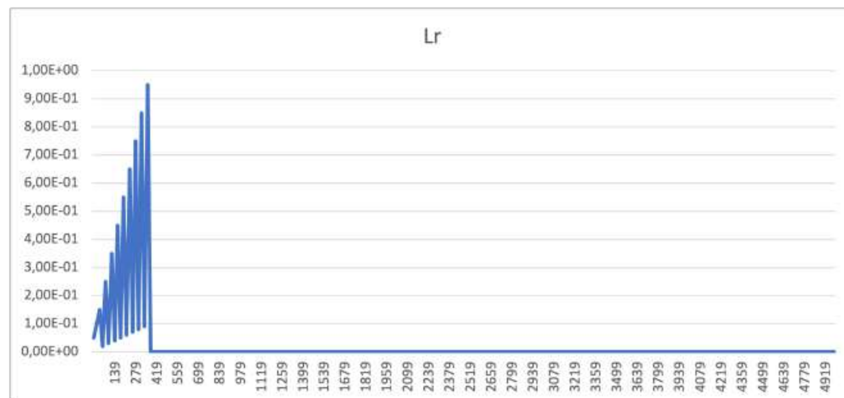


Figura 5.17: Resultado del entrenamiento

Con cada lote de descenso de gradiente, idealmente, la red debería acercarse más y más al valor mínimo global de pérdida. Por lo tanto, tiene sentido reducir la tasa de aprendizaje a medida que avanza el entrenamiento, de modo que el algoritmo no se exceda y se asiente lo más cerca posible del mínimo. SGDR es una variante del descenso de gradiente estocástico con reinicios. En esta técnica, aumentamos la tasa de aprendizaje repentinamente de vez en cuando.

La finalidad es aumentar repentinamente la tasa de aprendizaje, al hacerlo, el descenso del gradiente no se atasca en ningún mínimo local y puede saltar fuera de él en su camino hacia un mínimo global.

5.8.1. Etiquetado Manual(Labelme)

Ya que no se ha podido obtener resultados concluyentes, planteamos una data inferior y un etiquetado manual, esto se realiza con labelme, para intentar simplificar, para ello preparamos unas imágenes y las etiquetamos. Labelme es una herramienta de anotación gráfica de imágenes escrita en Python, utiliza Qt para su interfaz gráfica. En ella se puede etiquetar por puntos unas áreas manualmente, etiquetándolas para el entrenamiento.

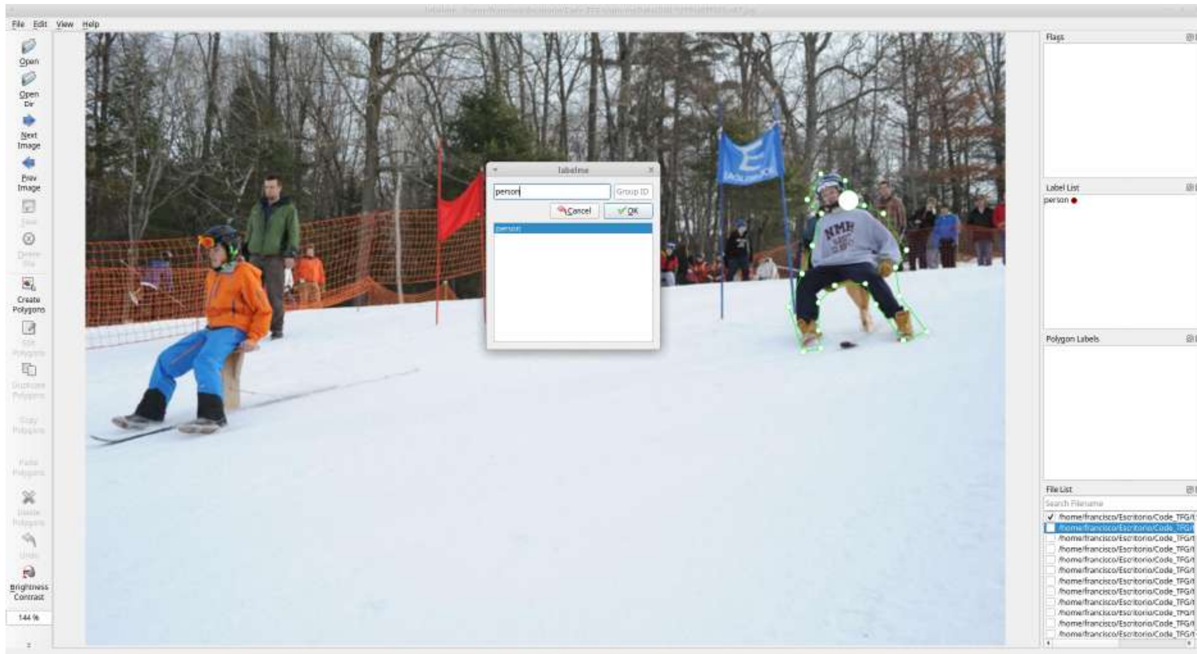


Figura 5.18: Etiquetado de datos en labelme

Una vez disponemos de las imágenes etiquetadas, dispondremos de una carpeta con imágenes y un archivo Json, dicho Json lo separaremos en do bloques, el primero se destinará a usarse como data para el entrenamiento, mientras que el segundo bloque lo usaremos para comprobar el abance y posterior test. Los archivos que acompañan a la foto tienen la información de cada etiqueta individualmente. No obstante igual que anteriormente, necesitamos los datos en COCO format para facilitarnos las cosas, por eso usaremos un script que podemos ver en D.3 para poder unificar todos los archivos json en un fichero con estructura COCO format.. Una vez tengamos el archivo en COCO format, de la misma forma que hemos visto en el apartado anterior procedemos a entrenar. En este caso el entrenamiento se va a realizar con 55 imágenes, 1000 iteraciones y un lr de 0.00025.

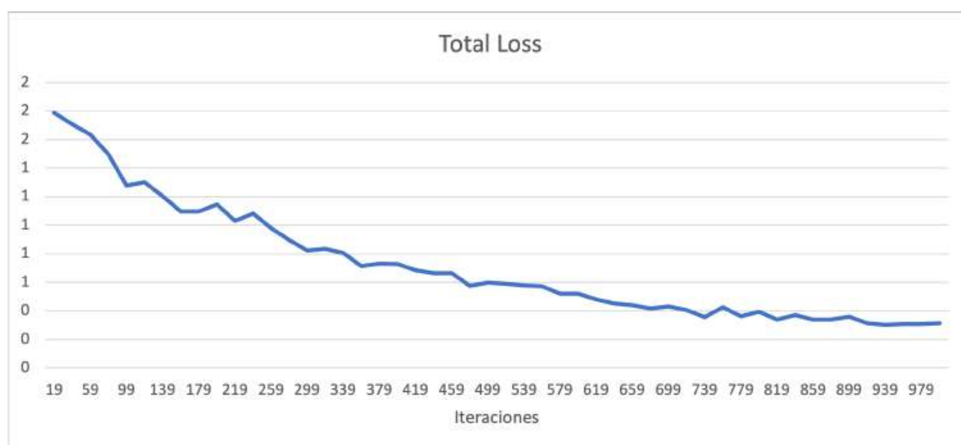


Figura 5.19: Resultado del entrenamiento

Figura 5.20: Resultado del entrenamiento

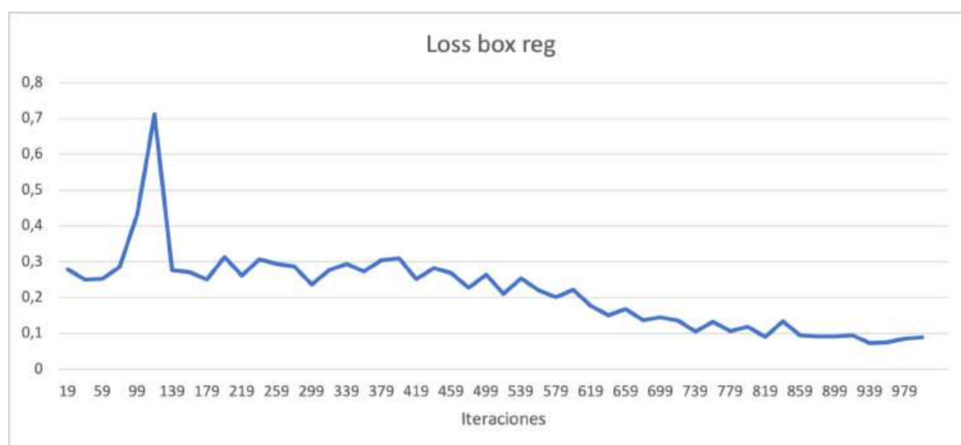


Figura 5.21: Resultado del entrenamiento

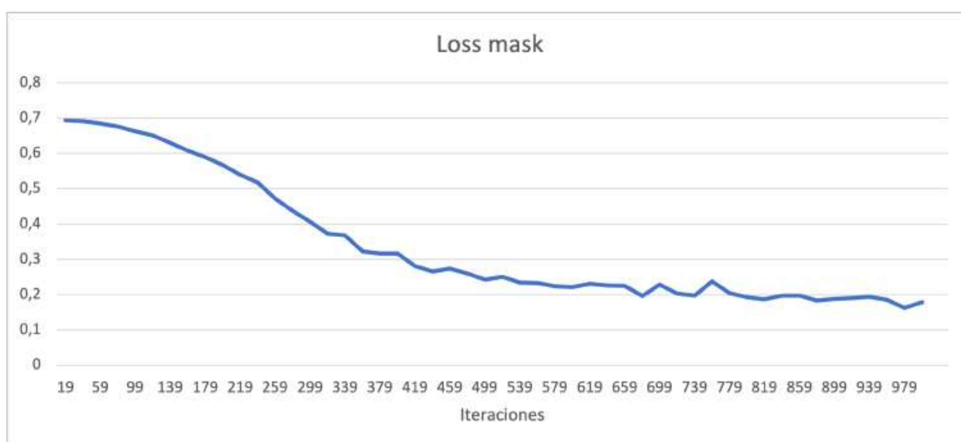


Figura 5.22: Resultado del entrenamiento

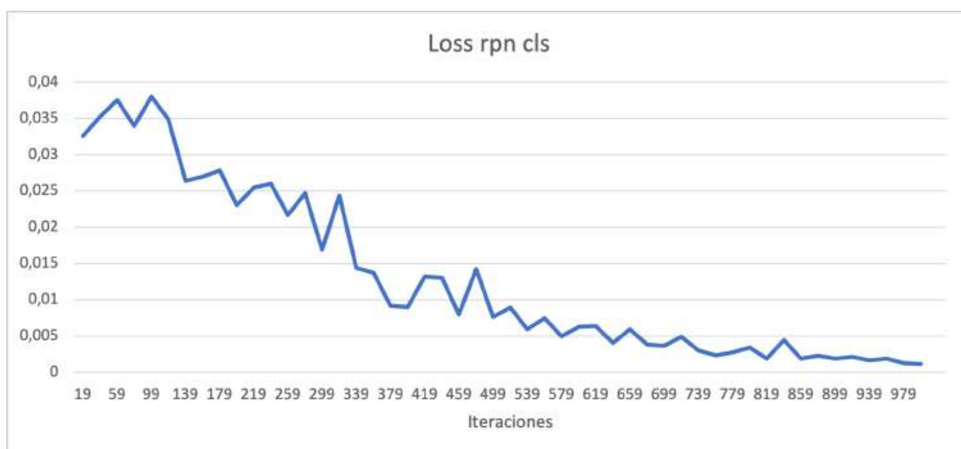


Figura 5.23: Resultado del entrenamiento

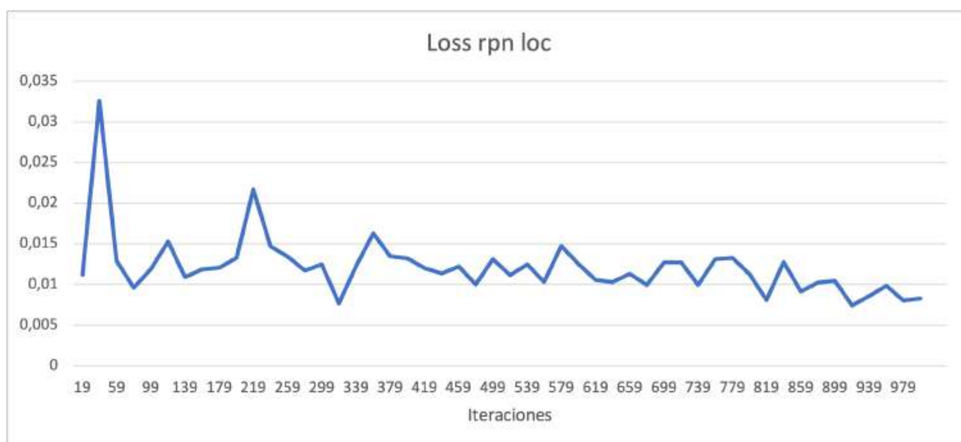


Figura 5.24: Resultado del entrenamiento

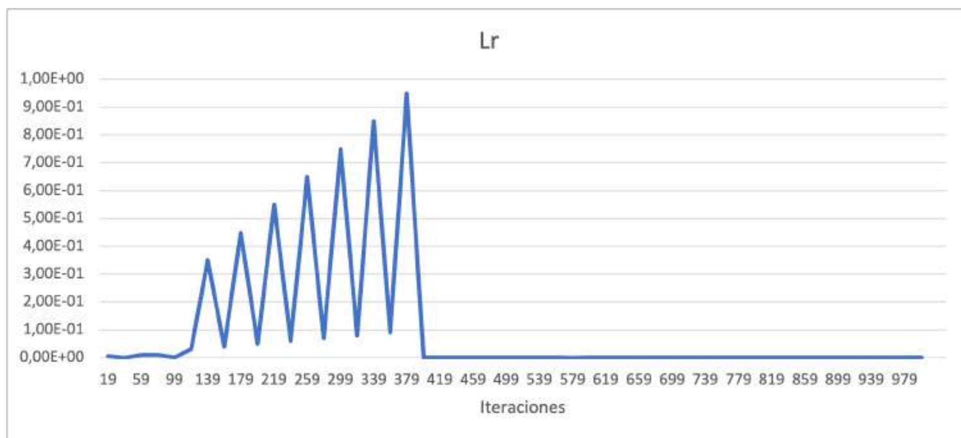


Figura 5.25: Resultado del entrenamiento

5.8.2. Comparación de modelos

Una vez ejecutamos el modelo creado obtenemos la salida.

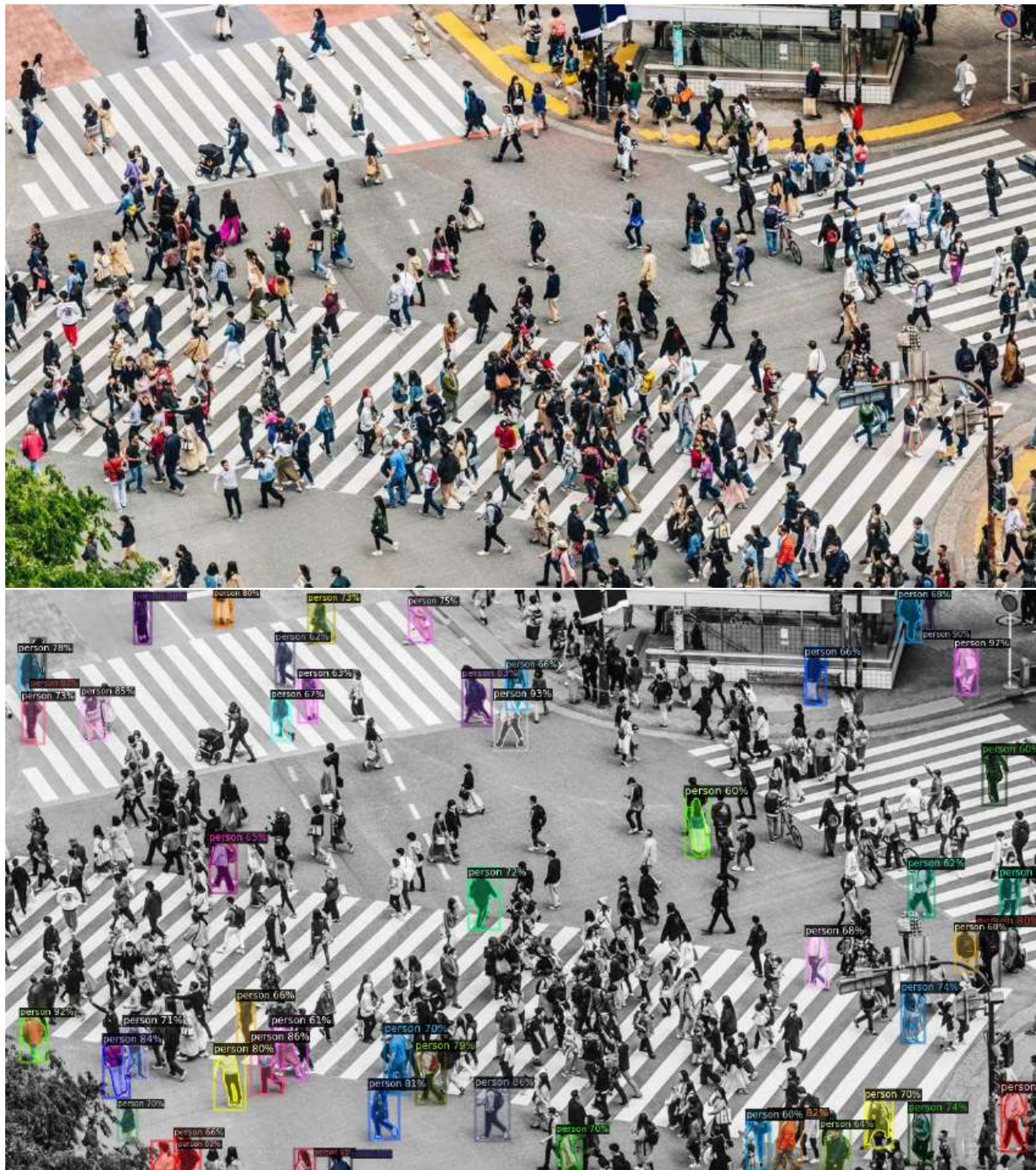


Figura 5.26: Ejecución del modelo

Figura 5.27: Ejecución del modelo Preentrenado COCO

5.8.3. Problemas y soluciones

Por la limitación que presenta el PC, el ordenador se ha quedado bloqueado en numerosas ocasiones, la cual cosa ha ralentizado de enormemente este apartado. La memoria RAM llegaba a su límite, esto hacía bloquear al ordenador, una solución rápida fue aumentar la memoria Swap (memoria mucho más lenta) pero de esta forma se consiguió terminar el entrenamiento satisfactoriamente sin bloquearse. El ordenador tendía a calentarse en exceso, se solucionó con la instalación de dos ventiladores de servidor en su base para aumentar el flujo de aire. En la ejecución del modelo entrenado con nuestra base de datos, se obtenía como resultado "The checkpoint state_dict contains keys that are not used by the model: z la imagen analizada sin ninguna detección. La solución es añadir cfg.MODEL.ROI_HEADS.NUM_CLASSES = num_of_classes, pues es necesario especificar el numero de clases.

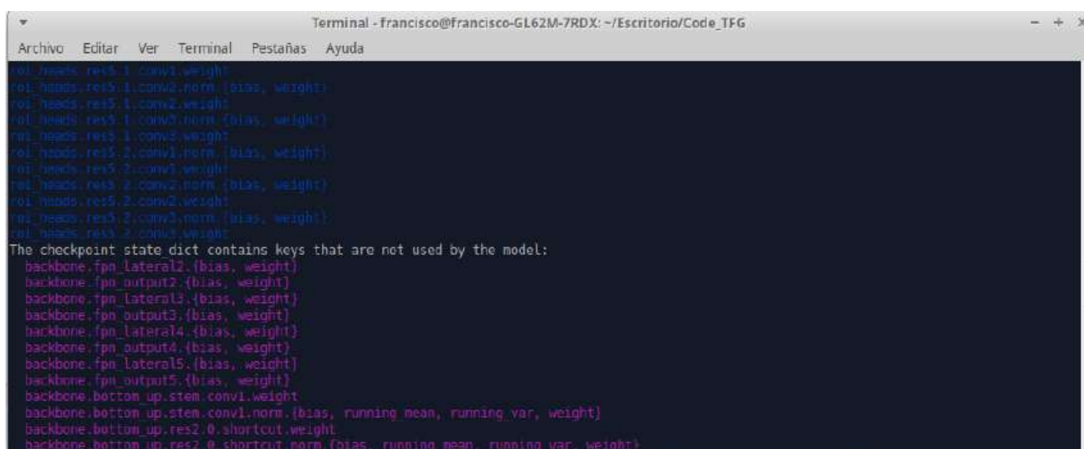


Figura 5.28: Error en la ejecución de nuestra red creada a partir de pesos propios

Capítulo 6

Conclusiones

6.1. Conclusión

Como conclusiones podemos destacar.

- **Potencial del procesado de imagen frente a posibles sensores**. Hemos podido comprobar como de una imagen y un post procesado, es posible extraer una gran cantidad de información tanta como sea necesario para cada aplicación.

- **Viabilidad de entrenamiento de una RNA**. Entrenar una RNA decente requiere de muchos recursos, de ahí que muchas veces decidamos utilizar redes preentrenadas que han sido procesadas en servidores destinados a ello. Entrenar una RNA es la mejor opción a nivel de resultados, sobre todo a largo plazo y la posibilidad de personalizar cualquier tipo de detección y aplicarla al ámbito que sea necesario (medicina, SmartCity, conducción autónoma y un sin fin de posibles aplicaciones). Sin embargo no debemos olvidar las redes preentrenadas, muchas redes están dotadas de infinidad de data, la cual cosa permite ser muy efectivas y resolutivas, tal y como se ha visto en este TFG.

- **Posibles aplicaciones de la tecnología**. En este TFG nos hemos centrado en el factor humano, la detección en personas. Por lo que la aplicación más directa sería saber de forma automatizada si una imagen tiene la presencia de una persona o no. Se podría usar por ejemplo como una alarma frente a intrusos, o contar personas en una imagen estática. Los datos que obtenemos se sitúan en la imagen mediante puntos, estos puntos pueden ser tratados para cualquier otra funcionalidad como por ejemplo, establecer zonas donde el paso está prohibido o contar únicamente las personas que se encuentren en una zona, lo que llamaríamos ROI.

- **Potencia de la máquina para ejecutar los modelos**. En muchos casos como en la conducción autónoma, sistemas que necesitan procesar información en tiempo real, es necesario una potencia considerable para procesar muchas imágenes por segundo conformadas por el vídeo. Es cierto que podemos usar modelos muy ligeros, pero perderemos efectividad en la detección. Por lo que existe una relación muy estrecha entre potencia y ciertas aplicaciones. En el caso que el tiempo de procesado no sea un limitante, será posible utilizar tanto un sistema edge como un procesado en la nube.

6.2. Mejora y futuro

Como posible mejoras se puede plantear la realización de un script para establecer un tracking a partir de los puntos obtenidos frame a frame de un vídeo. Esta mejora permitiría contabilizar las personas que están pasando por un roi determinado, e incluso llegando a conocer el sentido de paso que realiza. Este tipo de tecnología se aplica a conteos de peatones y conteo en vehículos de uso público sincronizándolos con su SAE para corroborar el ticaje. Otra mejora posible es la utilización de un servidor para aumentar las capacidades de procesamiento e incluso programar un servicio a modo de API que devuelva los puntos de interés de una imagen o en el caso de necesitar procesar un vídeo la devolución de la información que se desee obtener. En las imágenes de la Figura 6.1 podemos observar la mejora mencionada anteriormente dotando a la aplicación con la posibilidad de tratar vídeos en lugar de imágenes estáticas.

Figura 6.1: Uso del tracking en detección; imágenes cedidas por R3 Recymed

Apéndice A

Teoría de una red neuronal

A.1. Instalación de librerias

```
sudo apt update; sudo apt install software-properties-common apt-transport-https wget
```

```
wget -q https://packages.microsoft.com/keys/microsoft.asc -O- | sudo apt-key add -
```

```
sudo add-apt-repository "deb [arch=amd64] https://packages.microsoft.com/repos/vscode
    stable main"
```

```
sudo apt install code
```

```
sudo apt update
```

```
sudo apt install software-properties-common
```

```
sudo add-apt-repository ppa:deadsnakes/ppa
```

```
sudo apt install python3.8
```

```
sudo apt update
sudo apt install python3-pip
```

```
sudo python3.8 -m pip install numpy
```

A.2. Propagación de avance

```
import numpy as np
def feed_forward(entradas, salidas, pesos):
    pre_capa_oculta = np.dot(entradas,pesos[0])+ pesos[1]
    capa_oculta = 1/(1+np.exp(-pre_capa_oculta))
     prediccioniccioniccion_salida  = np.dot(capa_oculta, pesos[2]) + pesos[3]
    error_medio_cuadratico = np.mean(np.square(prediccioniccioniccion_salida - salidas))
    retorno error_medio_cuadratico
```

A.3. Descenso gradual

```
import numpy as np
from copy import deepcopy
import matplotlib.pyplot as plt
x = np.array ([[1,1]])
y = np.array ([[0]])

def feed_forward(entradas, salidas, pesos):
    pre_capa_oculta = np.dot(entradas,pesos[0])+ pesos[1]
    hidden = 1/(1+np.exp(-pre_capa_oculta))
    salida = np.dot(hidden, pesos[2]) + pesos[3]
    mean_squared_error = np.mean(np.square(salida - salidas))
    retorno mean_squared_error

def actualizacion_pesos(entradas, salidas, pesos, lr):
    original_pesos = deepcopy(pesos)
    temp_pesos = deepcopy(pesos)
    pesos_actualizados = deepcopy(pesos)
    perdida_real = feed_forward(entradas, salidas, original_pesos)
    for i, layer in enumerate(original_pesos):
        for index, weight in np.ndenumerate(layer):
            temp_pesos = deepcopy(pesos)
            temp_pesos[i][index] += 0.0001
            _perdida_plus = feed_forward(entradas, salidas, temp_pesos)
            grad = (_perdida_plus - perdida_real)/(0.0001)
            pesos_actualizados[i][index] -= grad*lr
    retorno pesos_actualizados, perdida_real
```

A.4. Impacto del ratio de aprendizaje

```
from copy import deepcopy
import numpy as np
import matplotlib.pyplot as plt

x =   [[2],[4],[6],[8]]
y =   [[7],[14],[21],[28]]

def feed_forward(entradas, salidas, pesos):
    out = np.dot(entradas,pesos[0])+ pesos[1]
    media_cuadratica_error = np.mean(np.square(out - salidas))
    retorno media_cuadratica_error

def actualizacion_pesos(entradas, salidas , pesos, lr):
    original_pesos = deepcopy(pesos)
    org_perdida = feed_forward(entradas, salidas, original_pesos)
    actualizaciond_pesos = deepcopy(pesos)
    for i, layer in enumerate(original_pesos):
        for index, weight in np.ndenumerate(layer):
            temp_pesos = deepcopy(pesos)
            temp_pesos[i][index] += 0.0001
            _perdida_plus = feed_forward(entradas, salidas, temp_pesos)
            grad = (_perdida_plus - org_perdida)/(0.0001)
            actualizaciond_pesos[i][index] -= grad*lr
    retorno actualizaciond_pesos

W = [np.array([[0]], dtype=np.float32), np.array([[0]], dtype=np.float32)]

Valor_peso = []
for epx in range(2000):
    W = actualizacion_pesos(x,y,W,0.001)
    Valor_peso.append(W[0][0][0])

plt.plot(Valor_peso)
plt.title('Valor del peso a lo largo de las iteraciones')
plt.xlabel('Iteraciones')
plt.ylabel('Valor del peso')
plt.show()
```

A.5. Generando CAMs

```
import os
if not os.path.exists('cell_images'):
from torch_snippets import *
id2int = {'Parasitized': 0, 'Uninfected': 1}
from torchvision import transforms as T
        trn_tfms = T.Compose([
                        T.ToPILImage(),
                        T.Resize(128),
  T.CenterCrop(128),
T.ColorJitter(brightness=(0.95,1.05),
              contrast=(0.95,1.05),
              saturation=(0.95,1.05),
              hue=0.05),
T.RandomAffine(5, translate=
T.ToTensor(),
T.Normalize(mean=[0.5, 0.5, 0.5],
            std=[0.5, 0.5, 0.5]),

            val_tfms = T.Compose([
                    T.ToPILImage(),
                    T.Resize(128),
                    T.CenterCrop(128),
                    T.ToTensor(),
                    T.Normalize(mean=[0.5, 0.5, 0.5],
 ])
std=[0.5, 0.5, 0.5]),

class Imagen_(Dataset):
           def __init__(self, files, transform=None):
  self.files = files
               self.transform = transform
               logger.info(len(self))
           def __len__(self):
               retorno len(self.files)
           def __getitem__(self, ix):
               fpath = self.files[ix]
               clss = fname(parent(fpath))
               img = read(fpath, 1)
               retorno img, clss
           def choose(self):
               retorno self[randint(len(self))]
           def collate_fn(self, batch):
               _imgs, classes = list(zip(*batch))
               if self.transform:
                   imgs = [self.transform(img)[None] for img in _imgs]
               classes = [torch.tensor([id2int[clss]]) for class in classes]
               imgs, classes = [torch.cat(i).to(device) for i in [imgs, classes]]
               retorno imgs, classes, _imgs

device = 'cuda' if torch.cuda.is_available() else 'cpu'
       all_files = Glob('cell_images/*/*.png')
       np.random.seed(10)
       np.random.shuffle(all_files)
       from sklearn.model_selection import train_test_split
       trn_files, val_files = train_test_split(all_files, random_state=1)
       trn_1 = Imagen_(trn_files, transform=trn_tfms)
       val_1 = Imagen_(val_files, transform=val_tfms)
```

```
        trn_2 = DataLoader(trn_ds, 32, shuffle=True,
                            collate_fn=trn_ds.collate_fn)
        val_dl = DataLoader(val_1, 32, shuffle=False,
collate_fn=val_1.collate_fn

ImagesClassifier :
def convBlock(ni, no):
            retorno nn.Sequential(
                nn.Dropout(0.2),
                nn.Conv2d(ni, no, kernel_size=3, padding=1),
                nn.ReLU(inplace=True),
                nn.BatchNorm2d(no),
                nn.MaxPool2d(2),
)
        class ImagesClassifier (nn.Module):
            def __init__(self):
                super().__init__()
                self .model = nn.Sequential(
                    convBlock(3, 64),
                    convBlock(64, 64),
                    convBlock(64, 128),
                    convBlock(128, 256),
                    convBlock(256, 512),
                    convBlock(512, 64),
                    nn.Flatten(),
                    nn.Linear(256, 256),
                    nn.Dropout(0.2),
                    nn.ReLU(inplace=True),
                    nn.Linear(256, len(id2int))
                )
                self .perdida_fn = nn.CrossEntropyperdida()
            def forward(self, x):
                retorno self .model(x)
            def compute_metrics(self, prediccions, targets):
                perdida = self.perdida_fn(prediccions, targets)
                accion =(torch.max(prediccions, 1)
        [1]==targets).float().mean()
                retorno perdida, accion

def train_batch(model, data, optimizer, criterion):
            model.train()
            ims, labels, _ = data
            _prediccions = model(ims)
            optimizer.zero_grad()
            perdida, accion = criterion(_prediccions, labels)
            perdida.backward()
            optimizer.step()
            retorno perdida.item(), accion.item()

        def validate_batch(model, data, criterion):
            model.eval()
            ims, labels, _ = data
            _prediccions = model(ims)
            perdida, accion = criterion(_prediccions, labels)
            retorno perdida.item(), accion.item()

            model = ImagesClassifier().to(device)
        criterion = model.compute_metrics
        optimizer = optim.Adam(model.parameters(), lr=1e-3)
```

```
        n_epochs = 2
        log = Report(n_epochs)
        for ex in range(n_epochs):
            N = len(trn_2)
            for bx, data in enumerate(trn_2):
 \
perdida, accion = train_batch(model, data, optimizer,
 criterion )
                log.record(ex+(bx+1)/N,trn_perdida=perdida,trn_accion=accion, \
end='\r')
            N = len(val_dl)
            for bx, data in enumerate(val_dl):
                perdida, accion = validate_batch(model, data,
        criterion )
                log.record(ex+
        (bx+1)/N,val_perdida=perdida,val_accion=accion, \
end='\r')
            log.report_avgs(ex+1)

 im2fmap = nn.Sequential(*
        ( list (model.model[:5].children ())+ \
  [:2]. children ())))
 list (model.model[5]

def im2gradCAM(x):
            model.eval()
            logits = model(x)
            heatmaps = []
            activaciones = im2fmap(x)
            print( activaciones .shape)
            prediccion = logits.max(-1)[-1]
            model.zero_grad()
            logits [0, prediccion ]. backward(retain_graph=True)
            pooled_grads = model.model[-7][1]\
                                .weight.grad.data.mean((0,2,3))

            for i in range(activaciones.shape[1]) :
                activaciones [:, i ,:,:]  *= pooled_grads[i]

        fmap)
            heatmap =torch.mean(activaciones, dim=1)
        [0]. cpu().detach()
            retorno heatmap, 'Uninfected' if prediccion.item() \
else 'Parasitized'

 SZ = 128
        def upsampleHeatmap(map, img):
   m,M = map.min(), map.max()
            map = 255 * ((map-m) / (M-m))
            map = np.uint8(map)
            map = cv2.resize(map, (SZ,SZ))
            map = cv2.applyColorMap(255-map, cv2.COLORMAP_JET)
            map = np.uint8(map)
            map = np.uint8(map*0.7 + img*0.3)
            retorno map

N = 20
        _val_dl = DataLoader(val_1, batch_size=N,
  shuffle =True, \
```

```
collate_fn=val_1.collate_fn)
  x,y,z = next(iter(_val_dl))
  for i in range(N):
      image = resize(z[i], SZ)
      heatmap, prediccioniccion = im2gradCAM(x[i:i+1])
      if(prediccioniccion=='Uninfected'):
          continue
      heatmap = upsampleHeatmap(heatmap, image)
      subplots([image, heatmap], nc=2, figsize=(5,3), \
                  suptitle=prediccioniccion)
```

Apéndice B

Pytorch

B.1. CUDA

```
wget https://developer.download.nvidia.com/compute/cuda/repos/ubuntu2004/x86_64/cuda-
    ubuntu2004.pin
sudo mv cuda-ubuntu2004.pin /etc/apt/preferences.d/cuda-repository-pin-600
wget https://developer.download.nvidia.com/compute/cuda/11.3.0/local_installers/cuda-repo-
    ubuntu2004-11-3-local_11.3.0-465.19.01-1_amd64.deb
sudo dpkg -i cuda-repo-ubuntu2004-11-1-local_11.1.0-465.19.01-1_amd64.deb
sudo apt-key add /var/cuda-repo-ubuntu2004-11-1-local/7fa2af80.pub
sudo apt-get update
sudo apt-get -y install cuda
```

```
| NVIDIA-SMI 465.19.01 Driver Version: 465.19.01      CUDA Version: 11.1     |
|                                   |                          |        MIG M. |
|   NVIDIA GeForce GTX 1050 On | 00000000:01:00.0 Off |                  N/A |
```

B.2. Instalación de PyTorch

```
(base) francisco@francisco-GL62M-7RDX:~$ pip3 install torch torchvision torchaudio --extra-
    index-url https://download.pytorch.org/whl/cu113
Looking in indexes: https://pypi.org/simple, https://download.pytorch.org/whl/cu113
Collecting torch
  Downloading https://download.pytorch.org/whl/cu113/torch-1.11.0%2Bcu113-cp38-cp38-
      linux_x86_64.whl (1637.0 MB)

Collecting torchvision
  Downloading https://download.pytorch.org/whl/cu113/torchvision-0.12.0%2Bcu113-cp38-cp38-
      linux_x86_64.whl (22.3 MB)
Collecting torchaudio
  Downloading https://download.pytorch.org/whl/cu113/torchaudio-0.11.0%2Bcu113-cp38-cp38-
      linux_x86_64.whl (2.9 MB)
Requirement already satisfied: typing-extensions in ./anaconda3/lib/python3.8/site-packages (from
    torch) (3.7.4.2)
Requirement already satisfied: requests in ./anaconda3/lib/python3.8/site-packages (from
    torchvision) (2.24.0)
Requirement already satisfied: numpy in ./anaconda3/lib/python3.8/site-packages (from torchvision)
    (1.18.5)
Requirement already satisfied: pillow!=8.3.*,>=5.3.0 in ./anaconda3/lib/python3.8/site-packages (
    from torchvision) (7.2.0)
Requirement already satisfied: idna<3,>=2.5 in ./anaconda3/lib/python3.8/site-packages (from
    requests->torchvision) (2.10)
Requirement already satisfied: chardet<4,>=3.0.2 in ./anaconda3/lib/python3.8/site-packages (from
    requests->torchvision) (3.0.4)
Requirement already satisfied: certifi>=2017.4.17 in ./anaconda3/lib/python3.8/site-packages (from
     requests->torchvision) (2020.6.20)
Requirement already satisfied: urllib3!=1.25.0,!=1.25.1,<1.26,>=1.21.1 in ./anaconda3/lib/python3
    .8/site-packages (from requests->torchvision) (1.25.9)
Installing collected packages: torch, torchvision, torchaudio
Successfully installed torch-1.11.0+cu113 torchaudio-0.11.0+cu113 torchvision-0.12.0+cu113
```

Apéndice C

Clasificación y detección de objetos

C.1. Clasificación de imágenes mediante CNN profundas

```
#Importamos todo lo necesario

import torch
import torchvision
import torch.nn as nn
import torch.nn.functional as F
from torchvision import transforms,models,datossets
from PIL import Image
from torch import optim
import matplotlib.pyplot as plt
from glob import glob
device = 'cuda' if torch.cuda.is_available() else 'cpu'
import cv2, glob, numpy as np, pandas as pd
```

```
    def __getitem__(self, ix):
        f = self.fpaths[ix]
        objetivo = self.objetivos[ix]
        im = (cv2.imread(f) [:,:,::-1])
        im = cv2.resize(im, (224,224))
        return torch.tensor(im/255).permute(2,0,1).to(device).float(), torch.tensor([objetivo]) .
            float().to(device)

datos = human_dogs(entrenado_datos_dir)
im, label = datos[200]

plt.imshow(im.permute(1,2,0).cpu())
print(label)

def conv_capa(ni,no,tamaño_ker,stride=1):
    return nn.Sequential(
    nn.Conv2d(ni, no, tamaño_ker, stride),
    nn.ReLU(),
    nn.BatchNorm2d(no),
    nn.MaxPool2d(2)
    )
def get_modelo():
    model = nn.Sequential(
```

```
    conv_capa(3, 64, 3),
    conv_capa(64, 512, 3),
    conv_capa(512, 512, 3),
    conv_capa(512, 512, 3),
    conv_capa(512, 512, 3),
    conv_capa(512, 512, 3),
    nn.Flatten(),
    nn.Linear(512, 1),
    nn.Sigmoid(),
    ).to(device)
    loss_fn = nn.BCELoss()
    optimizer = torch.optim.Adam(model.parameters(), lr= 1e-3)
    return model, loss_fn, optimizer

from torchsummary import summary
model, loss_fn, optimizer = get_modelo()
summary(model, input_size=(3, 224, 224))
```

```
def entrenado_batch(x, y, model, opt, loss_fn):
    prediction = model(x)
    batch_loss = loss_fn(prediction, y)
    batch_loss.backward()
    optimizer.step()
    optimizer.zero_grad()
    return batch_loss.item()
```

```
@torch.no_grad()
def precision_(x, y, model):
    prediction = model(x)
    is_correcto = (prediction > 0.5) == y
    return is_correcto.cpu().numpy().tolist()
```

```
def get_datos():
    entrenado = human_dogs(entrenado_datos_dir)
    trn_dl = datosLoader(entrenado, tamaño_batch=16, barajar=True, elimin_ultimo = True)
    val = human_dogs(test_datos_dir)
    val_dl = datosLoader(val, tamaño_batch=16, barajar=True, elimin_ultimo = True)
    return trn_dl, val_dl

@torch.no_grad()

def val_perdida(x, y, model):
    prediction = model(x)
    val_perdida = loss_fn(prediction, y)
    return val_perdida.item()
```

```
trn_dl, val_dl = get_datos()
model, loss_fn, optimizer = get_modelo()

entrenado_perdida, entrenado_precision = [], []
val_perdidaes, val_precisions = [], []
for epocas in range(5):

    print(epocas)
    entrenado_epocas_perdidas, entrenado_epocas_precision = [], []
```

```
    val_epocas_precision = []
    for ix, batch in enumerate(iter(trn_dl)):
        x, y = batch
        batch_loss = entrenado_batch(x, y, model, optimizer, loss_fn)
        entrenado_epocas_perdidas.append(batch_loss)
    entrenado_epocas_loss = np.array(entrenado_epocas_perdidas).mean()

    for ix, batch in enumerate(iter(trn_dl)):
        x, y = batch
        is_correcto = precision_(x, y, model)
        entrenado_epocas_precision.extend(is_correcto)
    entrenado_epocas_precision_ = np.mean(entrenado_epocas_precision)

    for ix, batch in enumerate(iter(val_dl)):
        x, y = batch
        val_is_correcto = precision_(x, y, model)
        val_epocas_precision.extend(val_is_correcto)
    val_epocas_precision_ = np.mean(val_epocas_precision)

    entrenado_perdida.append(entrenado_epocas_loss)
    entrenado_precision.append(entrenado_epocas_precision_)
    val_precisions.append(val_epocas_precision_)

len(entrenado_epocas_precision)
len(entrenado_epocas_precision[0])

epocass = np.arange(5)+1
import matplotlib.ticker as mtick
import matplotlib.pyplot as plt
import matplotlib.ticker as mticker
plt.plot(epocass, entrenado_precision, 'bo', label='entrenadoing precision_')
plt.plot(epocass, val_precisions, 'r', label='Validation precision_')
plt.gca().xaxis.set_major_locator(mticker.MultipleLocator(1))
plt.title('entrenadoing and validation precision_ with 4K datos points used for entrenadoing')
plt.xlabel('epocass')
plt.ylabel('precision_')
plt.gca().set_yticklabels(['{:.0f}%'.format(x*100) for x in plt.gca().get_yticks()])
plt.legend()
plt.grid('off')
plt.show()
```

Apéndice D

Aplicación práctica

D.1. Uso de un modelo preentrenado

```
from detectron2.engine import DefaultPredictor
from detectron2.config import get_cfg
from detectron2.data import MetadataCatalog
from detectron2.utils. visualizer import ColorMode, Visualizer
from detectron2 import model_zoo

import cv2
import numpy as np


class Detector_1:
    def __init__(self):
        self.cfg = get_cfg()

        #Cargamos la configuración del modelo preentrenado
        self.cfg.merge_from_file(model_zoo.get_config_file("COCO-Detection/
            faster_rcnn_R_101_FPN_3x.yaml"))
        self.cfg.MODEL.WEIGHTS = model_zoo.get_checkpoint_url("COCO-Detection/
            faster_rcnn_R_101_FPN_3x.yaml")
        self.cfg.MODEL.ROI_HEADS.SCORE_THRESH_TEST = 0.6
        self.cfg.MODEL.DEVICE = "cuda"
        self.predictor = DefaultPredictor(self.cfg)

    def onImage(self, imagePath):
        image = cv2.imread(imagePath)
        predictions = self.predictor(image)
        viz = Visualizer(image [:,:,::-1], metadata = MetadataCatalog.get(self.cfg.DATASETS.
            entrenar[0]),
        instance_mode = ColorMode.IMAGE_BW)
        output = viz.draw_instance_predictions(predictions["instances"].to("cpu"))
        cv2.imshow("Result", output.get_image()[:,:,::-1])
        cv2.waitKey(0)
```

```
from detectron_99 import *

Detector_1 = Detector_1()

Detector_1.onImage("/home/francisco/Escritorio/Code_TFG/foto_1.jpg")
```

D.2. Estimación postura humana

```
from torch_snippets import *
import detectron2
from detectron2.utils.logger import setup_logger

setup_logger()
import cv2
from detectron2 import model_zoo
from detectron2.engine import DefaultPredictor
from detectron2.config import get_cfg
from detectron2.utils.visualizer import Visualizer
from detectron2.data import MetadataCatalog,DatasetCatalog


cfg = get_cfg()
cfg.merge_from_file(model_zoo.get_config_file("COCO-Keypoints/
    keypoint_rcnn_R_101_FPN_3x.yaml"))
cfg.MODEL.ROI_HEADS.SCORE_THRESH_TEST = 0.5
cfg.MODEL.WEIGHTS = model_zoo.get_checkpoint_url("COCO-Keypoints/
    keypoint_rcnn_R_101_FPN_3x.yaml")
predictor = DefaultPredictor(cfg)
from torch_snippets import read, resize
im = read('foto_1.jpg',1)
im = resize(im, 0.4)
outputs = predictor(im)

s = Visualizer(im [:,:,::-1],  MetadataCatalog.get(cfg.DATASETS.entrenar[0]), scale=1.3)
out = s.draw_instance_predictions(outputs["instances"].to("cpu"))
cv2.imshow("Result", out.get_image() [:,:,::-1])
cv2.waitKey(0)
```

```
from detectron2.engine import DefaultPredictor
from detectron2.config import get_cfg
from detectron2.data import MetadataCatalog
from detectron2.utils.visualizer import ColorMode, Visualizer
from detectron2 import model_zoo

import cv2
import numpy as np


class Detector_1:
    def __init__(self, model_type = "OD"):
        self.cfg = get_cfg()

        #Cargamos la configuración del modelo preentrenado
        if model_type == "OD":
            self.cfg.merge_from_file(model_zoo.get_config_file("COCO-Detection/
                faster_rcnn_R_101_FPN_3x.yaml"))
            self.cfg.MODEL.WEIGHTS = model_zoo.get_checkpoint_url("COCO-Detection/
                faster_rcnn_R_101_FPN_3x.yaml")
        elif model_type == "IS":
            self.cfg.merge_from_file(model_zoo.get_config_file("COCO-InstanceSegmentation/
                mask_rcnn_R_50_FPN_3x.yaml"))
            self.cfg.MODEL.WEIGHTS = model_zoo.get_checkpoint_url("COCO-
                InstanceSegmentation/mask_rcnn_R_50_FPN_3x.yaml")
        self.cfg.MODEL.ROI_HEADS.SCORE_THRESH_TEST = 0.6
        self.cfg.MODEL.DEVICE = "cuda"
```

```
        self.predictor = DefaultPredictor(self.cfg)

    def onImage(self, imagePath):
        image = cv2.imread(imagePath)
        predictions = self.predictor(image)

        viz = Visualizer(image[:,:,::-1], metadata = MetadataCatalog.get(self.cfg.DATASETS.
            TRAIN[0]),
        instance_mode = ColorMode.IMAGE_BW)

        output = viz.draw_instance_predictions(predictions["instances"].to("cpu"))

        cv2.imshow("Result", output.get_image()[:,:,::-1])
        cv2.waitKey(0)
```

```
from detectron_999 import *

Detector_1 = Detector_1(model_type = "IS")

Detector_1.onImage("/home/francisco/Escritorio/Code_TFG/foto_1.jpg")
```

D.3. Obtención y preparación de datos

```
(base) francisco@francisco-GL62M-7RDX:~/Escritorio/Code_TFG$ wget -q https://storage.
    googleapis.com/openimages/v5/train-masks/train-masks-{0}.zip
(base) francisco@francisco-GL62M-7RDX:~/Escritorio/Code_TFG$ wget -q https://storage.
    googleapis.com/openimages/v5/train-masks/train-masks-0.zip
(base) francisco@francisco-GL62M-7RDX:~/Escritorio/Code_TFG$ wget -q https://storage.
    googleapis.com/openimages/v5/train-masks/train-masks-1.zip
(base) francisco@francisco-GL62M-7RDX:~/Escritorio/Code_TFG$ wget -q https://storage.
    googleapis.com/openimages/v5/train-masks/train-masks-2.zip
(base) francisco@francisco-GL62M-7RDX:~/Escritorio/Code_TFG$ wget -q https://storage.
    googleapis.com/openimages/v5/train-masks/train-masks-3.zip
(base) francisco@francisco-GL62M-7RDX:~/Escritorio/Codewget -q https://storage.googleapis.com
    /openimages/v5/train-masks/train-masks-4.zip
(base) francisco@francisco-GL62M-7RDX:~/Escritorio/Code_TFG$ wget -q https://storage.
    googleapis.com/openimages/v5/train-masks/train-masks-5.zip
(base) francisco@francisco-GL62M-7RDX:~/Escritorio/Code_TFG$ wget -q https://storage.
    googleapis.com/openimages/v5/train-masks/train-masks-6.zip
(base) francisco@francisco-GL62M-7RDX:~/Escritorio/Code_TFG$ wget -q https://storage.
    googleapis.com/openimages/v5/train-masks/train-masks-7.zip
(base) francisco@francisco-GL62M-7RDX:~/Escritorio/Code_TFG$ wget -q https://storage.
    googleapis.com/openimages/v5/train-masks/train-masks-8.zip
(base) francisco@francisco-GL62M-7RDX:~/Escritorio/Code_TFG$ wget -q https://storage.
    googleapis.com/openimages/v5/train-masks/train-masks-9.zip
(base) francisco@francisco-GL62M-7RDX:~/Escritorio/Code_TFG$ wget -q https://storage.
    googleapis.com/openimages/v5/train-masks/train-masks-10.zip
(base) francisco@francisco-GL62M-7RDX:~/Escritorio/Code_TFG$ wget -q https://storage.
    googleapis.com/openimages/v5/train-masks/train-masks-a.zip
(base) francisco@francisco-GL62M-7RDX:~/Escritorio/Code_TFG$ wget -q https://storage.
    googleapis.com/openimages/v5/train-masks/train-masks-b.zip
(base) francisco@francisco-GL62M-7RDX:~/Escritorio/Code_TFG$ wget -q https://storage.
    googleapis.com/openimages/v5/train-masks/train-masks-c.zip
(base) francisco@francisco-GL62M-7RDX:~/Escritorio/Code_TFG$ wget -q https://storage.
    googleapis.com/openimages/v5/train-masks/train-masks-d.zip
(base) francisco@francisco-GL62M-7RDX:~/Escritorio/Code_TFG$ wget -q https://storage.
    googleapis.com/openimages/v5/train-masks/train-masks-e.zip
(base) francisco@francisco-GL62M-7RDX:~/Escritorio/Code_TFG$ wget -q https://storage.
    googleapis.com/openimages/v5/train-masks/train-masks-f.zip
(base) francisco@francisco-GL62M-7RDX:~/Escritorio/Code_TFG$ wget -q https://storage.
    googleapis.com/openimages/v5/train-masks/train-masks-g.zip
```

```
from torch_snippets import *
required_classes = 'person'
required_classes = [c.lower() for c in required_classes.lower().split(',')]

 classes = pd.read_csv('classes.csv', header=None)
 classes.columns = ['class','class_name']
 classes = classes[classes['class_name'].map(lambda x: x in required_classes)]

from detectron2 import model_zoo
from detectron2.engine import DefaultPredictor
from detectron2.config import get_cfg
from detectron2.utils.visualizer import Visualizer
from detectron2.data import MetadataCatalog, DatasetCatalog
from detectron2.engine import Defaultentrenador

from detectron2.data.datasets import register_coco_instances
register_coco_instances("dataset_train", {}, "images.json", "train/myData2020")
```

```
cfg = get_cfg()
cfg.merge_from_file(model_zoo.get_config_file("COCO-InstanceSegmentation/
    mask_rcnn_R_50_FPN_3x.yaml"))
cfg.DATASETS.TRAIN = ("dataset_train",)
cfg.DATASETS.TEST = ()
cfg.DATALOADER.NUM_WORKERS = 2
cfg.MODEL.WEIGHTS = model_zoo.get_checkpoint_url("COCO-InstanceSegmentation/
    mask_rcnn_R_50_FPN_3x.yaml")
cfg.SOLVER.IMS_PER_BATCH = 2
cfg.SOLVER.BASE_LR = 0.00025
cfg.SOLVER.MAX_ITER = 5000
cfg.MODEL.ROI_HEADS.BATCH_SIZE_PER_IMAGE = 512
cfg.MODEL.ROI_HEADS.NUM_CLASSES = len(classes)

os.makedirs(cfg.OUTPUT_DIR, exist_ok=True)
entrenador = Defaultentrenador(cfg)
entrenador.resume_or_load(resume=False)
entrenador.train()
```

```
import os
from detectron2 import model_zoo
from detectron2.engine import DefaultPredictor
from detectron2.config import get_cfg
from detectron2.utils.visualizer import Visualizer
from detectron2.data import MetadataCatalog, DatasetCatalog
from detectron2.engine import DefaultTrainer
from torch_snippets import *
import datetime
from pycococreatortools import pycococreatortools
from os import listdir
from os.path import isfile, join
from PIL import Image
import json

from detectron2.data.datasets import register_coco_instances
register_coco_instances("dataset_train", {}, "images.json", "train/myData2020")

cfg = get_cfg()
cfg.MODEL.WEIGHTS = os.path.join(cfg.OUTPUT_DIR, "trained_model.pth")
cfg.MODEL.ROI_HEADS.SCORE_THRESH_TEST = 0.25
predictor = DefaultPredictor(cfg)
from detectron2.utils.visualizer import ColorMode
 files = Glob('entrenar/myData2020')

im = cv2.imread('entrenar/myData2020/0ad7a3553c8b736a.jpg')
outputs = predictor(im)
v = Visualizer(im[:, :, ::-1],
               scale=0.5,
               metadata=MetadataCatalog.get("dataset_entrenar"),
               instance_mode=ColorMode.IMAGE_BW
# remove the colors of unsegmented pixels.
# This option is only available for segmentation models
)

out = v.draw_instance_predictions(outputs["instances"].to("cpu"))
show(out.get_image())
```

D.4. Labeltome2COCO

```
from pathlib import Path
from typing import List

import numpy as np
from PIL import Image
from sahi.utils.coco import Coco, CocoAnnotation, CocoCategory, CocoImage
from sahi.utils.file import list_files_recursively, load_json, save_json
from tqdm import tqdm


class labelme2coco:
    def __init__(self):
        raise RuntimeError(
            "Use labelme2coco.convert() or labelme2coco.get_coco_from_labelme_folder() instead."
        )


def get_coco_from_labelme_folder(
    labelme_folder: str, coco_category_list: List = None
) -> Coco:
    """
    Args:
        labelme_folder: folder that contains labelme annotations and image files
        coco_category_list: start from a predefined coco cateory list
    """
    # get json list
    _, abs_json_path_list = list_files_recursively(labelme_folder, contains=[".json"])
    labelme_json_list = abs_json_path_list

    # init coco object
    coco = Coco()

    if coco_category_list is not None:
        coco.add_categories_from_coco_category_list(coco_category_list)

    # parse labelme annotations
    category_ind = 0
    for json_path in tqdm(
        labelme_json_list, "Converting labelme annotations to COCO format"
    ):
        data = load_json(json_path)
        # get image size
        image_path = str(Path(labelme_folder) / data["imagePath"])
        width, height = Image.open(image_path).size
        # init coco image
        coco_image = CocoImage(file_name=data["imagePath"], height=height, width=width)
        # iterate over annotations
        for shape in data["shapes"]:
            # set category name and id
            category_name = shape["label"]
            category_id = None
            for (
                coco_category_id,
                coco_category_name,
            ) in coco.category_mapping.items():
                if category_name == coco_category_name:
                    category_id = coco_category_id
```

```
                break
            # add category if not present
            if category_id is None:
                category_id = category_ind
                coco.add_category(CocoCategory(id=category_id, name=category_name))
                category_ind += 1
            # parse bbox/segmentation
            if shape["shape_type"] == "rectangle":
                x1 = shape["points"][0][0]
                y1 = shape["points"][0][1]
                x2 = shape["points"][1][0]
                y2 = shape["points"][1][1]
                coco_annotation = CocoAnnotation(
                    bbox=[x1, y1, x2 - x1, y2 - y1],
                    category_id=category_id,
                    category_name=category_name,
                )
            elif shape["shape_type"] == "polygon":
                segmentation = [np.asarray(shape["points"]).flatten().tolist()]
                coco_annotation = CocoAnnotation(
                    segmentation=segmentation,
                    category_id=category_id,
                    category_name=category_name,
                )
            else:
                raise NotImplementedError(
                    f'shape_type={shape["shape_type"]} not supported.'
                )
            coco_image.add_annotation(coco_annotation)
        coco.add_image(coco_image)

    return coco


if __name__ == "__main__":
    labelme_folder = "tests/data/labelme_annot"
    coco = get_coco_from_labelme_folder(labelme_folder)
```

Bibliografía

[1] Javier Plaza Penedés. María Iranzo Cabrera. Andrés Pedreño Muñoz., Luis Moreno Izquierdo. *Big Data e Inteligencia Artificial. Una visión económica y legal de estas tecnologías disruptivas.* Universidad de Valencia, Fundació Parc Científic Universitat de València c/ Catedrático Agustín Escardino, 9 46980 Paterna (España), 2018.

[2] Agarwal A. Barham P. Brevdo E. Chen Z. Citro C. Corrado G.S. Davis A. Dean J. Devin M. Ghemawat S. Goodfellow I. Harp A. Irving G. Isard M. Jia Y. Jozefowicz R. Kaiser L. Kudlur M. Levenberg J. Man´e D. Monga R. Moore S. Murray D. Olah C. Schuster M. Shlens J. Steiner B. Sutskever I. Talwar K. Tucker P. Vanhoucke V. Vasudevan V. Vi´egas F. Vinyals O. Warden P. Wattenberg M. Wicke M. Yu Y. Zheng Abadi, M.

[3] V Kishore Ayyadevara. Hands-on machine learning on google cloud platform: Implementing smart and efficient analytics using cloud ml engine.

[4] Mouncef Filali Bouami. Desarrollo y optimización de nuevos modelos de redes neuronales basadas en funciones de base radial. 01 2022.

[5] Redes neuronales. https://interactivechaos.com/es/manual/tutorial-de-machine-learning/redes-neuronales. Accessed: 2022-01-08.

[6] Tomasz Szandała. Review and comparison of commonly used activation functions for deep neural networks. 10 2020.

[7] Mengye Ren, Wenyuan Zeng, Bin Yang, and Raquel Urtasun. Learning to reweight examples for robust deep learning. In Jennifer Dy and Andreas Krause, editors, *Proceedings of the 35th International Conference on Machine Learning*, volume 80 of *Proceedings of Machine Learning Research*, pages 4334–4343. PMLR, 10–15 Jul 2018.

[8] Darwin Patiño Pérez, Miguel Botto-Tobar, Celia Munive Mora, et al. Predicción del composite requerido en el diseño de un recipiente toroidal mediante una red neuronal artificial. *Investigación, Tecnología e Innovación*, 13(13):45–53, 2021.

[9] Zhilu Zhang and Mert R Sabuncu. Generalized cross entropy loss for training deep neural networks with noisy labels. In *32nd Conference on Neural Information Processing Systems (NeurIPS)*, 2018.

[10] stanford. https://ai.stanford.edu/ syyeung/cvweb/tutorial1.html, 26 de Enero de 2021.

[11] 2008 Biothermic. Longitudes de onda de la luz solar.

[12] Accesor. https://www.accesor.com/-/sistemas-complementarios/sistemas-control-de-aforo/, 27 de Febrero de 2022.

[13] PhD. Bibb Allen MD. Bradley J. Erickson MD PhD. Jayashree Kalpathy-Cramer PhD. Keith Bigelow BA. Tessa S. Cook MD PhD. Adam E. Flanders MD. Matthew P. Lungren MD MPH. David S. Mendelson MD. Jeffrey D. Rudie MD PhD. Ge Wang PhD. Krishna Kandarpa MD PhD Curtis P. Langlotz, MD. *A Roadmap for Foundational Research on Artificial Intelligence in Medical Imaging*. Stanford University, Stanford, CA 94305 (C.P.L., M.P.L.), e-mail: langlotz@stanford.edus, 2018.

[14] Yaoshiang Ho and Samuel Wookey. The real-world-weight cross-entropy loss function: Modeling the costs of mislabeling. *IEEE Access*, 8:4806–4813, 2019.

[15] Qingshan Liu and Jun Wang. A one-layer recurrent neural network with a discontinuous hard-limiting activation function for quadratic programming. *IEEE Transactions on Neural Networks*, 19(4):558–570, 2008.

[16] Fran Ramírez. Las matemáticas del machine learning: Funciones de activación. *Ingeniero/Grado en Informática de Sistemas, Técnico Superior en Electrónica Digital y Máster en Seguridad de las TIC. Investigador de seguridad informática en el equipo de Ideas Locas CDCO de Telefónica.*, 2020.

[17] Damián Amoedo. https://ubunlog.com/visual-studio-code-editor-codigo-abierto-ubuntu-20-04/comments, 22 de Agosto de 2020.

[18] Sofija Simic. https://phoenixnap.com/kb/how-to-install-python-3-ubuntu, 12 de Diciembre de 2019.

[19] SchubertSlySchubert. https://www.kaggle.com/, 7 de Febrero de 2022.

[20] SchubertSlySchubert. https://storage.googleapis.com/openimages/web/index.html, 7 de Febrero de 2022.