Reconocimiento de edificios en imágenes aéreas mediante redes neuronales

Francisco de Asís González García-Tizón
Màster Universitari en Enginyeria Informàtica
Intel.ligència Artificial

Antoni Burguera Burguera
Carles Ventura Royo

A Coruña, 24 de diciembre de 2021

Aquesta obra està subjecta a una llicència de Reconeixement-NoComercial-SenseObraDerivada 3.0 Espanya de Creative Commons

Agradecimientos:

A mi director de TFM y consultor de la asignatura de *Intel·ligència artificial avançada*, el doctor Antoni Burguera Burguera, que se dejó fichar sin oponer resistencia. Gracias por tus buenos consejos, los *tocho mails* y sobre todo por encarnar mi modelo ideal de profesor universitario.

A tots els altres consultors del màster, que em van ajudar amb el repte d'estudiar en una llengua nova per a mi.

A Roberto Iniesta Ojea, por haber puesto la banda sonora a todas las noches que dediqué a realizar este TFM.

Dedicatorias:

A mi abuelita, Sofía Iglesias de Torres, ser humano excepcional e irrepetible. Te echo de menos cada día.

A mi abuelo, Francisco González Biedma, por ser ejemplo permanente de esfuerzo, trabajo duro y superación en mi vida. Sit tibi terra levis.

Al doctor Jose María García-Tizón Iglesias, que me animó a estudiar esta carrera, evitando que me convirtiera en un físico sin vocación.

A mis padres, que fueron mecenas necesarios de esta aventura.

A mi esposa, Nuria González Abeal, que siempre quiso casarse con un ingeniero guapo y que con este trabajo, al menos, cumple la mitad de sus sueños.

Y por encima de todos ellos a mis hijos, Rui Pablo y Alba, las dos personas más importantes de mi vida. No hay palabras en el diccionario para describir cuánto os quiero. Esto es por y para vosotros.

FITXA DEL TREBALL FINAL

Títol del treball:	*Reconocimiento de edificios en imágenes aéreas mediante redes neuronales*
Nom de l'autor:	*Francisco de Asís González García-Tizón*
Nom del consultor:	*Antoni Burguera Burguera*
Nom del PRA:	*Carles Ventura Royo*
Data de lliurament (mm/aaaa):	*12/2021*
Titulació:	*Màster Universitari en Enginyeria Informàtica*
Àrea del Treball Final:	*Intel.ligència Artificial*
Idioma del treball:	*Castellà*
Paraules clau	*Deep learning, computer vision, aerial images*

Resum del Treball:

La segmentación de imágenes aéreas para reconocer edificios de forma automatizada puede tener multitud de aplicaciones prácticas en distintos campos. El presente estudio ejemplifica cómo podría llevarse a cabo esta tarea de forma empírica.

En este TFM se parte de un dataset limitado en el que se conoce para cada foto, qué puntos se corresponden con edificaciones. Como primer paso se desarrollaron diferentes técnicas que amplían el conjunto de datos de partida hasta hacerlo prácticamente ilimitado, descartando aquellas que no son significativas. Posteriormente, se configuraron y parametrizaron distintos modelos de redes neuronales convolucionales que, tomando como entrada esos datos, se entrenaron y evaluaron hasta lograr predicciones con suficiente calidad acerca de las construcciones que se escondían dentro de imágenes inéditas.

Finalmente se compararon entre sí los resultados. Se observó que, aunque se consiguieron modelos secuenciales capaces de obtener resultados satisfactorios, en el presente trabajo fueron batidos por modelos preentrenados. También se obtuvieron medidas de la precisión de las predicciones y visualizaciones tanto de los aciertos como de los errores que comete la red en un lote de imágenes de ejemplo.

Como conclusión se extrae que una red bien entrenada puede competir con un humano realizando este trabajo y que el desarrollo de esta tecnología tiene un indudable interés científico y social. No se pretende con este proyecto dar una

solución general al problema, sino constatar que las redes neuronales son muy capaces de resolverlo de forma satisfactoria y que puede ampliarse el estudio a casuísticas reales.

Abstract:

The segmentation of aerial images to recognize buildings in an automated way can have a multitude of practical applications in different fields. The present study exemplifies how this task could be carried out empirically.

In this TFM we start from a limited dataset in which it is known for each photo which pixels correspond to buildings. As a first step, different techniques were developed to expand the starting dataset until it is practically unlimited, discarding those that are not significant. Subsequently, different models of convolutional neural network models were configured and parameterized, which, taking these data as input, were trained and evaluated to achieved predictions with enough quality about the buildings hidden inside unknown images.

Finally, the results were compared with each other. It was observed that, although sequential models capable of obtaining satisfactory results were achieved, in the current work they were beaten by pre-trained models. We also obtained measures of the accuracy of the predictions and visualizations of both the hit and misses made by the network on a batch of example images.

The conclusion is that a well-trained network can fight with a human being doing this job and that the development of this technology is of undoubted scientific and social interest. The aim of this project is not to provide a general solution to the problem, but to prove that neural networks are very capable of solving it successfully and that the study can be extended to real cases.

Índice

1. Introducción....................1
1.1 Contexto y justificación del trabajo....................1
1.2 Objetivos del trabajo....................2
1.3 Método seguido....................3
1.4 Herramientas y planificación del trabajo....................4
1.5 Breve sumario de productos obtenidos....................10
1.6 Breve descripción del resto de capítulos de la memoria....................10
2. Preprocesamiento y generación de datos....................12
2.1 Preprocesamiento del dataset....................12
2.2 Generación de datos....................18
3. Estudio de modelos secuenciales....................22
3.1 Breve descripción....................22
3.2 Principales configuraciones analizadas....................22
3.3 Persistencia del entrenamiento y gráficas de precisión y pérdida....................25
3.4 Evaluación de la alternativa seleccionada....................27
3.5 Predicción: representación gráfica y medidas....................27
4. Estudio de modelos preentrenados....................31
4.1 Breve descripción....................31
4.2 Principales configuraciones analizadas....................32
4.3 Persistencia del entrenamiento y gráficas de precisión y pérdida....................33
4.4 Evaluación de la alternativa seleccionada....................34
4.5 Predicción: representación gráfica y medidas....................34
5. Comparativa entre modelos....................36
6. Conclusiones....................38
7. Líneas de trabajo futuras....................40
8. Glosario....................42
9. Bibliografía....................44
10. Anexos....................45

Lista de figuras

Figura 1: Meme relativo a Google Colab. Autor: desconocido............................5
Figura 2: Planificación inicial..7
Figura 3: Planificación final..9
Figura 4: Función generateTiles(). Fuente: cuaderno de Jupyter......................13
Figura 5: Imagen aérea de Viena. Fuente: dataset del INRIA...........................13
Figura 6: Ground truth de imagen aérea de Viena. Fuente: dataset del INRIA. 13
Figura 7: Cortes disjuntos practicados a una imagen aérea de Viena. Fuente: dataset del INRIA...14
Figura 8: Ground truths de cortes disjuntos practicados a una imagen aérea de Viena. Fuente: dataset del INRIA...14
Figura 9: Ejemplo de llamada a la función flip. Fuente: cuaderno de Jupyter…14
Figura 10: Ejemplo de flip horizontal. Fuente: dataset del INRIA......................15
Figura 11: Ejemplo de llamada a la función rotate. Fuente: cuaderno de Jupyter 15
Figura 12: Ejemplo de una rotación de 180º. Fuente: dataset del INRIA..........16
Figura 13: Extracto de la función isGroundTruthValid(). Fuente: cuaderno de Jupyter...17
Figura 14: Extracto del generador de datos. Bloque que implementa los flips. Fuente: cuaderno de Jupyter..19
Figura 15: Extracto del generador de datos. Bloque que implementa las rotaciones. Fuente: cuaderno de Jupyter...20
Figura 16: Bloque que ejecuta la división del dataset y la creación de los generadores de datos. Fuente: cuaderno de Jupyter.......................................21
Figura 17: Modelo secuencial inicial. Fuente: cuaderno de Jupyter..................23
Figura 18: Modelo secuencial inicial con aumento de parámetros. Fuente: cuaderno de Jupyter...24
Figura 19: Modelo secuencial final. Fuente: cuaderno de Jupyter.....................25
Figura 20: Bloque de entrenamiento del modelo secuencial. Fuente: cuaderno de Jupyter...25
Figura 21: Función save_trained_model(). Fuente: cuaderno de Jupyter.........26
Figura 22: Función load_trained_model(). Fuente: cuaderno de Jupyter..........26
Figura 23: Gráfica de pérdida. Fuente: cuaderno de Jupyter............................27
Figura 24: Gráfica de precisión. Fuente: cuaderno de Jupyter.........................27
Figura 25: Bloque de evaluación del modelo secuencial. Fuente: cuaderno de Jupyter...27
Figura 26: Bloque de predicción de un lote. Fuente: cuaderno de Jupyter…….28
Figura 27: Imágenes de referencia del lote inédito. Fuente: cuaderno de Jupyter 28
Figura 28: Ground truths del lote inédito. Fuente: cuaderno de Jupyter............29
Figura 29: Predicciones por probabilidades del lote inédito. Fuente: cuaderno de Jupyter...29
Figura 30: Predicciones redondeadas a las clases del lote inédito. Fuente: cuaderno de Jupyter...29

Figura 31: Diferencias entre los ground truths y las predicciones redondeadas del lote inédito. Fuente: cuaderno de Jupyter..30
Figura 32: Arquitectura del modelo VGG-16. Fuente: [13]................................31
Figura 33: Llamada a la función save_trained_model. Fuente: cuaderno de Jupyter..33
Figura 34: Gráfica de precisión. Fuente: cuaderno de Jupyter..........................33
Figura 35: Gráfica de pérdida. Fuente: cuaderno de Jupyter............................33
Figura 36: Llamada a la función evaluate. Fuente: cuaderno de Jupyter..........34
Figura 37: Predicciones por probabilidades del lote inédito. Fuente: cuaderno de Jupyter..34
Figura 38: Predicciones redondeadas a las clases del lote inédito. Fuente: cuaderno de Jupyter..35
Figura 39: Diferencias entre los ground truths y las predicciones redondeadas del lote inédito. Fuente: cuaderno de Jupyter..35
Figura 40: Primera foto del lote 1 del modelo secuencial. Fuente: cuaderno de Jupyter..36
Figura 41: Primera foto del lote 1 del modelo preentrenado. Fuente: cuaderno de Jupyter..36
Figura 42: Séptima foto del lote 1 del modelo secuencial. Fuente: cuaderno de Jupyter..37
Figura 43: Séptima foto del lote 1 del modelo preentrenado. Fuente: cuaderno de Jupyter..37
Figura 44: Instanciación y compilación de un modelo preentrenado. Fuente: cuaderno de Jupyter..38
Figura 45: Ejemplo de piscina en forma de playa. Fuente: Piscinas Natursand40

Lista de tablas

Tabla 1: Fecha de entrega de las PACs....................6
Tabla 2: Métricas del modelo secuencial seleccionado....................30
Tabla 3: Métricas del modelo preentrenado seleccionado....................35

1. Introducción

1.1 Contexto y justificación del trabajo

Aunque el origen de la fotografía aérea se remonta a mediados del siglo XIX [1], el acceso a este tipo de instantáneas de todo el planeta, por parte del gran público, es un fenómeno muy reciente. Pensemos que hasta abril de 2006, prácticamente anteayer, *Google maps* [2] no estaba disponible en España. Cuando un particular o un ayuntamiento deseaba conseguir una foto aérea de una finca que decorara el salón o una ortofoto del término municipal para incluir en el *Plan General de Ordenación Municipal* (PGOM) respectivamente, debían contratar un vuelo ligero (avioneta, helicóptero o incluso un globo aerostático) y a un fotógrafo que capturara las instantáneas. Pasado el tiempo, si querían una toma de la nueva construcción anexa o de un nuevo polígono residencial, debían repetir el trámite. Hoy en día, basta abrir un ordenador conectado a internet, acceder a cualquiera de las múltiples webs disponibles (*Bing maps* [3], *wego.here.com* [4], etcétera) y se pueden visualizar imágenes periódicas de cualquier lugar del mundo al momento. También satélites como Sentinel [5], capturan fotos aéreas diarias. Además de esto, cualquier particular puede, mediante un dron dotado de una cámara 4K, conseguir las fotos del sitio de su interés instantáneamente (desplazándose hasta ese lugar con el equipo adecuado, claro está).

Pero, ¿para qué puede necesitarse una foto aérea de un determinado terreno, aparte de para una finalidad lúdica? Pensemos en una administración pública gallega, por ejemplo, con competencias en urbanismo: los concellos o la Xunta. En esa comunidad autónoma se sufre un problema endémico conocido como *feísmo* (sic). Influenciado o no por el minifundismo, es cierto que rara es la familia que no es propietaria de al menos un terreno, y muchas de ellas se lanzan a construir de forma irregular, sin permiso, de cualquier manera, destruyendo y afeando el paisaje, en una cultura denominada como el «*ti vai facendo*» (tú vete construyendo).

Hasta hace relativamente poco, los alcaldes comprometidos recorrían su territorio a pie persiguiendo estas ilegalidades, aunque es cierto que el método ha sufrido una enorme mejora con la fotografía aérea ya comentada. Pese a todo, una vez disponen de las tomas, la forma de detectar las irregularidades es manual. El funcionario del departamento de urbanismo de turno debe conocer las construcciones legales de cada cuadrante, y comprobar visualmente en las instantáneas que le lleguen si hay alguna edificación ilegal nueva desconocida. Automatizar este ingente trabajo parece una necesidad acuciante.

Es en este punto donde se propone aplicar la inteligencia artificial. En concreto las redes neuronales convolucionales o *Convolutional Neural Networks* (CNN). La finalidad de usar estas redes sería la de poder

segmentar esas imágenes aéreas y descubrir qué edificios contienen de forma automática: éste será el ámbito del dominio del presente proyecto.

Las CNN [6] son un tipo de red neuronal específica que se usa para trabajar (sobre todo) con imágenes y están formadas, generalmente, por combinaciones de dos bloques básicos en secuencia: bloques de convolución y bloques de agrupación. Estos bloques tienen funciones de activación no lineales.

En los bloques convolucionales se cogen grupos de píxeles vecinos de la imagen de entrada (que forman una matriz) y se va haciendo el producto escalar de ella con otra pequeña matriz llamada kernel (en concreto no solo un kernel sino un conjunto de ellos que se conocen como filtros) hasta barrer toda la matriz de entrada. El resultado es una nueva matriz que constituye una capa oculta.

Por su parte, los bloques de agrupación tienen como objetivo, o bien reducir la dimensionalidad (*Max pooling*), o bien aumentarla (*Upsampling*).

1.2 Objetivos del trabajo

Este TFM no pretende dar una solución general al problema descrito, pues la segmentación puede conseguirse a través de distintos caminos (modelos, parametrizaciones, etcétera) y con diferentes grados de éxito.

La tarea que se llevará a cabo consistirá en estudiar algunos modelos de CNNs, especificando la arquitectura de cada una: capas y filtros elegidos. Se entrenarán, evaluarán, y medirán para poder compararlas entre sí. También se visualizarán gráficamente sus predicciones para poder extraer resultados cualitativos y obtener las conclusiones.

Para conseguir el objetivo general, que no es otro que el de lograr segmentar con éxito las imágenes, es preceptivo alcanzar distintas metas parciales imprescindibles para llevar la tarea a cabo, a saber:

- Preprocesar un pequeño dataset dado de imágenes grandes, dividiéndolo en un subconjunto de fotos más pequeñas practicándoles cortes disjuntos, y después mediante técnicas de aumentación de datos conseguir un conjunto virtualmente ilimitado. Las imágenes finales con las que se harán los entrenamientos de la red deben ser significativas para que sean eficaces en el aprendizaje, para ello se descartarán aquellas que tengan demasiados, o por el contrario, muy pocos edificios.

- Estudiar modelos secuenciales, variando el tipo y el número de capas, los filtros de cada una, entrenarlos con el dataset del punto anterior (con más o menos imágenes) hasta detectar el overfitting, evaluarlos y hacer predicciones, que serán evaluadas indicando si aciertan o no.

- Llevar a cabo un estudio similar pero con modelos preentrenados.
- Comparar ambos modelos y extraer conclusiones.

1.3 Método seguido

Aunque el autor utiliza profesionalmente la metodología Agile para el desarrollo de software, se ha elegido el ciclo de vida clásico en cascada (waterfall) que está estructurado como es conocido en varias fases:

a) Fase de análisis:

Durante esta fase se realizan las siguientes tareas:

- *Preprocesado del dataset*: El preprocesado consistirá en distintas subtareas como la definición de la estructura de carpetas de entrada/salida, el estudio de los algoritmos de disección de imágenes, junto al formato y tamaño adecuado final (de cada foto y de la cantidad de éstas). También se establecerán los márgenes de aceptación de las imágenes (% de edificios), el tamaño de cada conjunto de datos (entrenamiento, test y validación) y se decidirá cómo implementar y estructurar los constructores de datos y la selección de técnicas de aumentación.

- *Estudio modelos secuenciales*: se centrará en la selección de la arquitectura y la elección de la precisión de la predicción aceptable.

- *Estudio modelos preentrenados*: consisitirá en la selección de las distintas alternativas de modelos preentrenados. Estos modelos cuentan con arquitecturas de red predefinidas y se han entrenado previamente con datos diferentes a los de este proyecto. Cada uno de los modelos cuenta con una *input shape* que habrá que contemplar. Por último habrá que definir la precisión de la predicción que se considera aceptable.

b) Fase de diseño:

Durante esta fase se realizan las siguientes tareas:

- *Preprocesado del dataset*: se implementarán funciones auxiliares y se establecerá el orden de las fases constitutivas del proceso.

- *Estudio modelos secuenciales*: se programarán las funciones de persistencia y recuperación de modelos ya entrenados. Se mostrarán las gráficas de precisión y pérdida. Finalmente, se establecerá la estructura de visualización de datos en filas, dicha visualización permitirá comparar cualitativamente los datos de

entrada con la predicción obtenida y la respuesta esperada (*ground truth*).

- *Estudio modelos preentrenados*: se harán subtareas análogas a los secuenciales.

c) Fase de implementación:

En esta fase se codificarán los algoritmos que plasmen las fases previas en el lenguaje Python.

d) Fase de ejecución:

En esta fase se lanzarán los entrenamientos, se ejecutarán las evaluaciones y se mostrarán las predicciones.

e) Fase de evaluación:

En esta fase se realizará un análisis cualitativo y cuantitativo de los resultados obtenidos. Una vez interpretados, se variarán los parámetros y se relanzará la fase de ejecución.

f) Fase final de comparación entre modelos:

En esta fase se compararán los modelos secuenciales y preentrenados seleccionados, lo que permitirá extraer las conclusiones del proyecto.

Varias son las razones que han llevado a decantarse por la metodología de desarrollo clásica. Por ejemplo, esta metodología permite sin problemas, (sobre todo en el caso concreto de un solo recurso, con una finalidad y un horizonte temporal tasado en las dos primeras PACs), llegar a afinar el proceso de selección y evaluación de modelos de forma exitosa ya que su núcleo es la constante iteración entre fases. Esto posibilitará rectificar los parámetros elegidos, hasta encontrar aquella configuración que consiga la precisión en la identificación de clases deseada. No es necesario usar el método ágil ya que no se prevé que se presenten eventualmente nuevos requisitos que haya que implementar en ciclos rápidos y no hay equipos que gestionar.

1.4 Herramientas y planificación del trabajo

Para implementar el proyecto se usará el lenguaje de programación Python (versión 3), que gracias a bibliotecas como *Keras* se considera muy cómodo para construir una CNN, entrenarla y probarla.
El entorno de trabajo será la herramienta de Google: COLAB. Entre sus ventajas se puede citar que evita problemas de dependencias de bibliotecas, a diferencia de una instalación local y permite el uso de unidades de procesamiento gráfico o *Graphics Processing Units* (GPU)

lo que acelera las ejecuciones. En el debe, hay que destacar que es interactiva, aunque también puede usarse para ejecuciones pesadas como es el caso, si el terminalista pone de su parte confirmando su presencia de forma periódica (mecanismo *CAPTCHA*). De todas formas, aún contando con un humano pendiente constantemente de la ejecución, llega un momento que el entorno de ejecución se desconecta. Es un evento tan común que incluso se han creado memes, como el mostrado en la figura 1, al respecto.

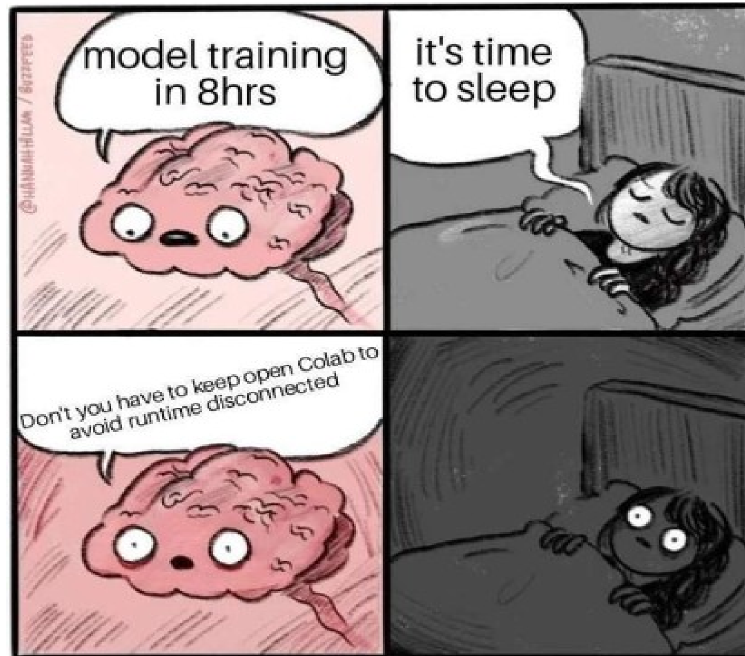

Figura 1: Meme relativo a Google Colab. Autor: desconocido

En el plan docente de la asignatura se especifican las fechas de entrega de los PACs, que se muestran en la tabla 1.

Pac 0	22/09/2021
Pac 1	04/10/2021
Pac 2	08/11/2021
Pac 3	09/12/2021

Tabla 1: Fecha de entrega de las PACs

Las PACs 0 y 1 son de definición de contenidos y plan de trabajo respectivamente.

Teniendo en cuenta que el desarrollo del TFM se divide en 3 grandes bloques:

- Preprocesamiento y generación de datos
- Estudio de modelos secuenciales
- Estudio de modelos preentrenados

se optó por asignar la mayor carga de trabajo a la PAC 2 (los dos primeros bloques) y dejar a la PAC 3 solo el tercero. Así (como efectivamente sucedió) si acontecía algún imprevisto, había margen de maniobra para llegar en plazo a la redacción de la presente memoria.

La primera planificación se muestra en un diagrama de Gantt representado en la figura 2.

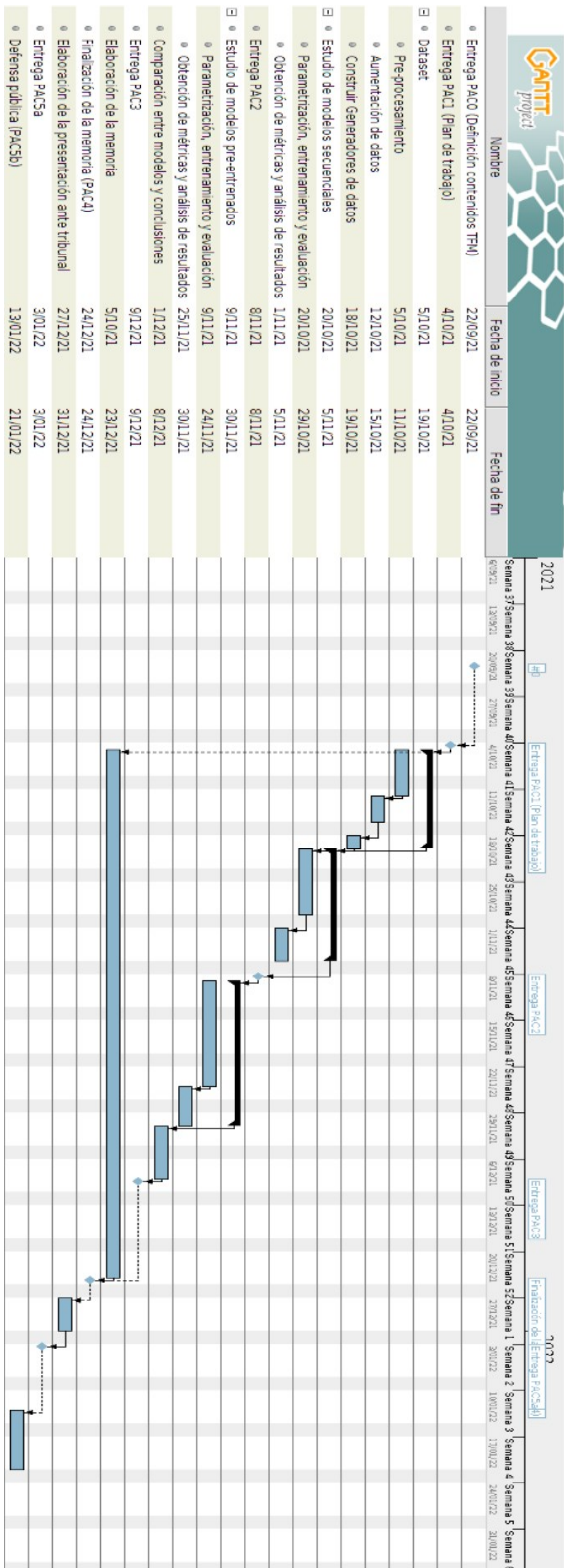

Figura 2: Planificación inicial

Llegado el momento de entregar la PAC 2 no se había completado correctamente el estudio del modelo secuencial, así que se replanificó el proyecto. Se incluyó en la PAC 3 la parte del estudio de los modelos secuenciales que había quedado pendiente. En la figura 3 aparece la planificación final, seguida con éxito hasta la fecha de finalización de la presente memoria.

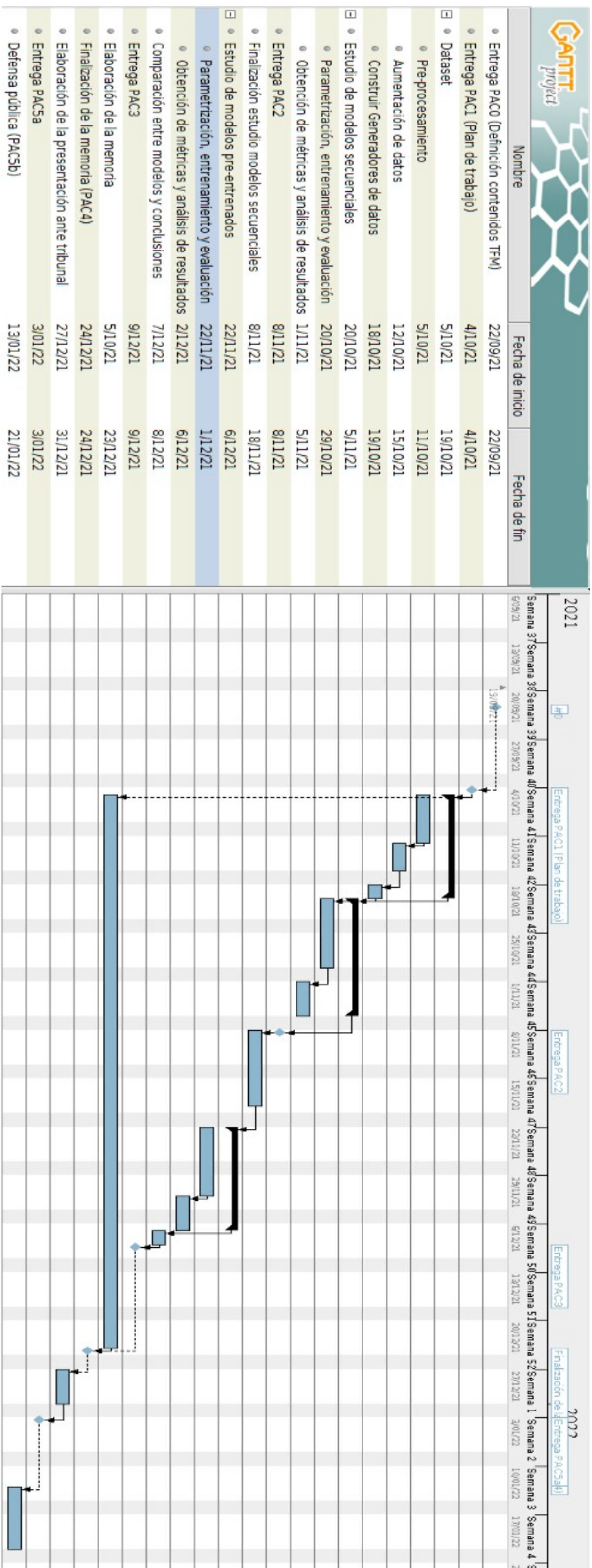

Figura 3: Planificación final

1.5 Breve sumario de productos obtenidos

El producto final del trabajo se concreta en un único cuaderno de Jupyter denominado *PAC3.ipynb*, que está disponible dentro del material suministrado. En cada celda de dicho cuaderno se implementa una funcionalidad que es ejecutada (secuencial o alternativamente) y cuyo resultado queda almacenado para poder ser visualizado a posteriori.

A modo de resumen se describe brevemente cada celda:

- Rutas de entrada/salida de las fotos y función que realiza los cortes disjuntos
- Función que determina si cada corte es aceptable
- Listado de los cortes eliminados
- Listado final con el número de cortes dados por buenos
- Funciones auxiliares
- Generador de datos (incluye aumentación de datos bajo demanda)
- División del dataset óptimo en trainSet, testSet y validationSet
- Ejecuciones no definitivas del modelo secuencial
- Definición del modelo secuencial seleccionado
- Resultado del entrenamiento
- Funciones para cargar y guardar el resultado del entrenamiento
- Gráficas de pérdida y precisión del entrenamiento
- Resultado de la evaluación
- Predicción de un lote de ejemplo (con resultados gráficos)
- Funciones auxiliares para el cálculo de medidas
- Cálculo de medidas
- Instalación de paquetes necesarios para ejecutar modelos preentrenados
- Ejecuciones no definitivas del modelo preentrenado
- Definición del modelo preentrenado seleccionado
- Resultado del entrenamiento
- Persistencia del resultado del entrenamiento
- Resultado de la evaluación
- Gráficas de pérdida y precisión del entrenamiento
- Predicción de un lote de ejemplo (con resultados gráficos)
- Cálculo de medidas

1.6 Breve descripción del resto de capítulos de la memoria

El resto de capítulos de la memoria se corresponden principalmente con los tres grandes bloques en que se dividió el desarrollo del proyecto:

- Capítulo 2. Preprocesamiento y generación de datos: se describirá el proceso de elección y optimización del dataset, el generador de datos y la división en los conjuntos de entrenamiento, validación y test.

- Capítulo 3. Estudio de modelos secuenciales: arquitectura seleccionada como ejemplo, alternativas de parametrización, elección de la seleccionada, entrenamiento, evaluación y predicción de un lote de ejemplo.

- Capítulo 4. Estudio de modelos preentrenados: modelos estudiados, modelo seleccionado, entrenamiento, evaluación y predicción de un lote de ejemplo.

Los tres últimos capítulos abarcan aspectos transversales de los bloques mencionados y son los siguientes:

- Capítulo 5. Comparativa entre modelos.

- Capítulo 6. Conclusiones.

- Capítulo 7. Líneas de trabajo futuras.

2. Preprocesamiento y generación de datos

2.1 Preprocesamiento del dataset

Para poder entrenar con éxito una CNN, se necesita un dataset adecuado. Lo suficientemente numeroso y preciso para que la red pueda aprender y predecir la segmentación.
Tras descartar varios [7], se seleccionó uno del Instituto Nacional de Investigación en Informática y Automática de Francia (INRIA) [8].

Dicho dataset consta de 180 imágenes aéreas de las ciudades de Austin (Texas, EEUU), Chicago (Illinois, EEUU), Lienz (Tirol, Austria), Viena (capital federal de Austria) y del condado de Kitsap (Washington, EEUU), en formato TIFF y con una resolución espacial de tan solo 30 centímetros por píxel. Cada una de esas imágenes de referencia cuenta con su propio ground truth, que solo distingue dos clases (edificio o fondo), que es precisamente el objetivo de este TFM. Fueron las características de los ground truths las que facilitaron la elección de este conjunto de datos en particular.

Por el contrario, 180 fotos no son muchas para entrenar una CNN ya que, como se ha dicho, el conjunto debe ser lo suficientemente extenso (en el rango de miles de fotos idealmente). Por suerte las imágenes aéreas eran inmensas, cada una de 5000*5000 píxeles. Este tamaño no es adecuado para el entrenamiento debido a la enorme cantidad de matrices que implicaría, pero practicándoles cortes disjuntos se consiguió un doble objetivo: por un lado disponer de más fotos y por otro reducir el tamaño de cada una lo que redundó en la calidad del entrenamiento. El único punto que parecía problemático fue un reto que finalmente se convirtió en fortaleza.

Respecto al tamaño de los cortes se opta por una potencia de 2 (512), que es conocido que es aceptada, por ejemplo, como entrada en los modelos preentrenados Resnet. Así cada imagen de referencia se convertirá en 81 de 512*512 píxeles. Hay que dividir de igual forma los ground truths, de tal manera que, esos nuevos trozos sigan estando relacionados con sus correspondientes cortes de referencia.

Por último se cambia el formato de TIFF a PNG, sin pérdida, de 3 canales (RGB) ya que, por ejemplo, los pesos preentrenados en aplicaciones Keras requieren una entrada de 3 canales. En los ground truths los 3 canales tendrán el mismo valor o (255, 255, 255) píxel blanco correspondiente a edificio o (0, 0, 0) píxel negro relativo al fondo.

Los cortes disjuntos y el cambio de formato de las imágenes está implementado en la función *generateTiles()* del cuaderno de Python, y se reproduce a modo ilustrativo en la figura 4.

```python
def generateTiles(inputPath, outputPath):

  # Se lista el contenido de la ruta
  # que llega como parámetro de entrada
  files = os.listdir(inputPath)

  # Se recorre la lista de nombres de las imágenes
  for file in files:

    # Se captura el nombre de cada imagen sin su extensión
    imageName = file.split('.')
    filename = imageName[0]

    # Se lee cada imagen TIFF
    # El segundo parámetro es el parámetro del número de canal y la profundidad de bits,
    # IMREAD_COLOR = 1
    # Para poder convertir a imagen RGB de tres canales, la profundidad de la imagen se convierte a 8 bits
    img = cv2.imread(os.path.join(inputPath,file),1)

    # Se recorta cada imágen y se graba en el directorio de salida
    # en un formato más adecuado para que puedan ser tratadas por la red neuronal: PNG (sin pérdida)
    for i in range(0, 4608, 512):
      for j in range(0, 4608, 512):

        cropImg = img[i:i+512, j:j+512]
        cv2.imwrite(os.path.join(outputPath,filename + "_" + str(i) + "_" + str(j) + ".png"), cropImg)
```

Figura 4: Función generateTiles(). Fuente: cuaderno de Jupyter

La figura 5 muestra una de las imágenes del dataset, en concreto la imagen aérea nº 5 de Viena. Su ground truth correspondiente puede verse en la figura 6.

Figura 5: Imagen aérea de Viena. Fuente: dataset del INRIA

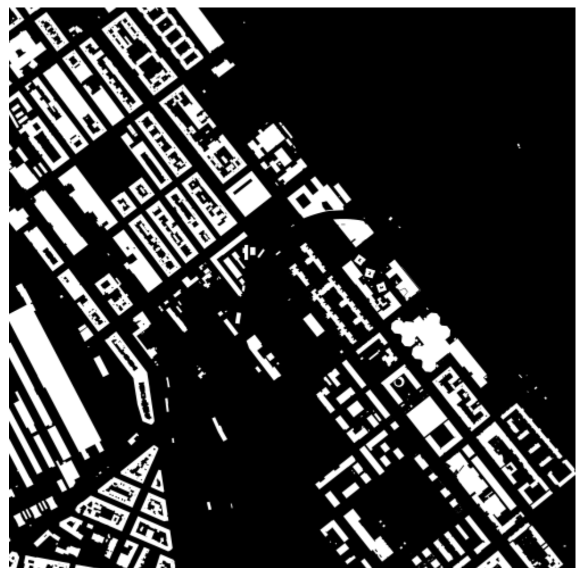

Figura 6: Ground truth de imagen aérea de Viena. Fuente: dataset del INRIA

La figura 7 muestra los cuatro primeros cortes disjuntos practicados a la figura 5 en su esquina superior izquierda, con tamaño unitario de 512*512 píxeles.

Figura 7: Cortes disjuntos practicados a una imagen aérea de Viena. Fuente: dataset del INRIA

En la figura 8 se pueden apreciar los ground truths de la figura 7.

Figura 8: Ground truths de cortes disjuntos practicados a una imagen aérea de Viena. Fuente: dataset del INRIA

Para obtener aún más imágenes distintas a partir de los nuevos cortes, se ha usado la técnica conocida como aumentación de datos (*data augmentation*), que se puede definir como la generación artificial de nuevos datos mediante modificaciones de los datos originales.
En concreto en el proyecto se han usado dos subtipos, los flips y las rotaciones [9], debido a que son transformaciones disponibles en la biblioteca OpenCV y que se consiguen fácilmente escribiendo una sola línea de código.

El primer subtipo de aumentación de datos utilizado en el proyecto son los flips. Este efecto consiste en invertir las imágenes de forma vertical, horizontal o ambas a la vez. Como se ha comentado, hacer la inversión de una imagen es muy sencillo. Basta con llamar a la función flip de la biblioteca OpenCV pasándole como parámetros la imagen que se desea alterar y el tipo de inversión. Como resultado se obtiene la imagen invertida. En la figura 9 se muestra un ejemplo de llamada a la función en Python.

```
flipImg = cv2.flip(curImage, flipMode)
```

Figura 9: Ejemplo de llamada a la función flip. Fuente: cuaderno de Jupyter

Los parámetros son: curImage (imagen original) y flipMode (tipo de inversión) cuyos posibles valores son:

0 → flip vertical
>0 → flip horizontal
<0 → flip vertical y horizontal

La función flip devuelve flipImg, que es la imagen invertida del modo deseado.

En la figura 10 se puede ver el resultado de aplicar un flip a uno de los cortes de tamaño 512*512 píxeles.

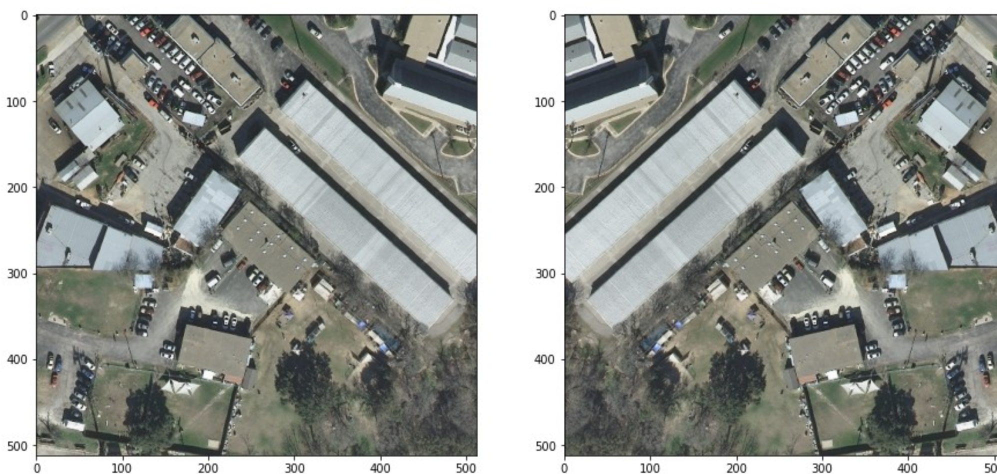

Figura 10: Ejemplo de flip horizontal. Fuente: dataset del INRIA

El segundo subtipo de aumentación de datos utilizado en el proyecto son las rotaciones. Esta vez el efecto se consigue rotando las imágenes 180º, 90º en el sentido de las agujas del reloj o 90º en sentido contrario. Para rotar una imagen en Python, solo hay que llamar a la función rotate de la biblioteca OpenCV pasándole como parámetros la imagen que se desea alterar y el tipo de rotación. Como resultado se obtiene la imagen una vez rotada. En la figura 11 se muestra un ejemplo de llamada a la función en Python.

```
rotationImg = cv2.rotate(curImage, rotationMode)
```

Figura 11: Ejemplo de llamada a la función rotate. Fuente: cuaderno de Jupyter

Los parámetros son: curImage (imagen original) y rotationMode (tipo de rotación) cuyos posibles valores son:

cv2.ROTATE_180
cv2.ROTATE_90_COUNTERCLOCKWISE
cv2.ROTATE_90_CLOCKWISE

La función rotate devuelve rotationImg, que es la imagen rotada del modo deseado.

En la figura 12 se muestra ver el resultado de rotar 180º uno de los cortes de tamaño 512*512 píxeles.

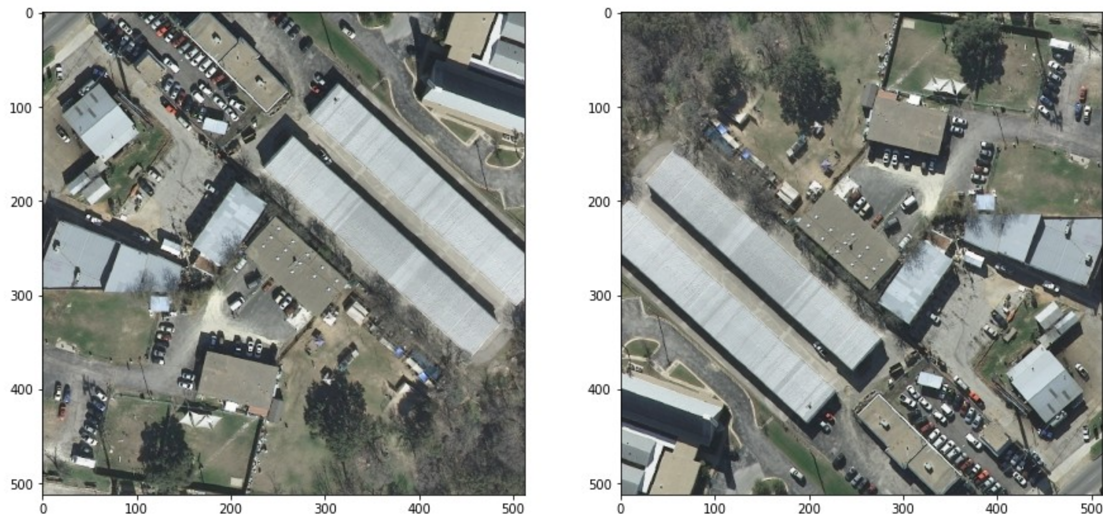

Figura 12: Ejemplo de una rotación de 180º. Fuente: dataset del INRIA

La aumentación de datos se aplica bajo demanda en el método __getitem__ de los generadores de datos como se verá posteriormente. Así en caso de ser necesario, el data generator puede producir imágenes nuevas sin necesidad de tenerlas persistidas en un soporte de almacenamiento masivo.

Cuando ya se dispone de un conjunto de imágenes mayor, una vez realizados los cortes y ajustado el formato, hay que comprobar si este dataset está balanceado. Se entiende por balanceado que las imágenes que van a servir de entrenamiento a la CNN sean eficaces para el aprendizaje. Por ello se eliminarán aquellas (tanto imágenes de referencia como sus ground truths) que no contengan un número de píxeles correspondientes a edificios significativo. Por reducción al absurdo, de nada valdría entrenar la red con una zona de parque sin ningún edificio o con panorámicas donde solo se vea un edificio en cada una. Se establecen los límites inferiores y superiores en el 10% y el 90% de píxeles de edificios respecto al total respectivamente. Se consideran adecuados estos valores para que la red pueda reconocer y segmentar las construcciones con calidad.

Para realizar esta tarea se recorren todos los ground truths y se llama a la función *isGroundTruthValid()* del cuaderno para cada uno. En la figura 13 se muestra la lógica principal de esa función.

```
for i in range(data.shape[0]):
  for j in range(data.shape[1]):

    if (data[i][j] > 127):

      whitePixels = whitePixels + 1

      if (whitePixels > tenPercent):
        isValid = True

      if (whitePixels > ninetyPercent):
        return False

return isValid
```

Figura 13: Extracto de la función isGroundTruthValid(). Fuente: cuaderno de Jupyter

El funcionamiento de la función es bastante sencillo. Se recorren los píxeles que forman la imagen y se van contando los que se corresponden con edificios (valor 255). Por **seguridad** y en coherencia con el resto del código, se toman como edificios (píxeles blancos) aquellos mayores de 127. Si son menores, se considerarán píxeles negros, es decir correspondientes al fondo. Como ya se ha explicado, si el número de píxeles blancos está entre el 10% y el 90% del total, la imagen se da por válida.

Las imágenes nuevas obtenidas mediante la aumentación de datos no necesitan ser revisadas, ya que tendrán exactamente la misma cantidad de píxeles de edificios que la original que ya ha sido validada.

2.2 Generación de datos

Las funciones de Keras que realizan el entrenamiento (fit) y evaluación (evaluate) requieren un «generador de datos» [10]. Un generador de datos es un objeto de Keras que hereda de Sequence y que se utiliza para alimentar con datos en tiempo real a los modelos Keras. Tienen que implementar al menos dos métodos: __len__() que proporciona el número de lotes y __getitem__(i) que proporciona el i-ésimo lote de datos y ground truths.

A continuación se explican las particularidades del data generator utilizado:

a) Al instanciar el objeto además de definir el tamaño del lote, (10 en el código del cuaderno de Python), se asigna el valor a dos flags:

- applyFlips : Si es **True**, hacemos flips de las imágenes (y de los ground truths)
- applyRotation: Si es **True**, se rotan las imágenes (y los ground truths)

Si no aplica ninguno de los 2 flags, se utilizan los recortes dados por buenos, persistidos en las carpetas OUTPUT (imágenes de referencia) y OUTPUT_GT (sus ground truths).

b) Dentro de la función __getitem__ se llama a una función auxiliar *_load_image_pair_* que recupera la imagen de referencia y el ground truth para cada elemento del lote. Los ground truths se categorizan inicialmente a 3 clases para que sea más sencillo rotar o invertir la imagen en caso de ser necesario. El generador de datos debe devolver finalmente una imagen con tan solo 2 canales de los 3 iniciales (RGB) (que se corresponderán con las 2 clases objetivo: edificio y fondo), pero si aplica algún tipo de aumentación de datos, tal y como se vio antes, las funciones necesitan como parámetro de entrada una imagen completa (3 canales), por lo tanto en principio se conservan los 3 canales.

c) La función __getitem__ devuelve todas las imágenes de referencia y sus correspondientes groud truths de un lote. Chequea los flags explicados en el apartado a), y si aplican, se devuelve la imagen invertida o rotada. Para conseguir imágenes distintas en cada ejecución, se procede de la siguiente forma: se genera un nº entero aleatorio entre -1 y 1, y dependiendo del valor se hace una determinada inversión o rotación. Finalmente se devuelven solo 2 clases (correspondientes a edificio o fondo). Como se ha explicado en el apartado b) se debe devolver una imagen con solo 2 canales, es por ello que se elimina el canal B del ground truth.

En la figura 14 se muestra el bloque del generador de datos si se desea aplicar los flips.

```
#########################
# FLIP
#
# Para hacer un tipo de flip aleatorio del lote
# Se genera un entero aleatorio entre -1 y 1:
# Posibles valores: -1, 0 y 1.
#
# flipMode =  0: flip vertical
# flipMode =  1: flip horizontal
# flipMode = -1: flip vertical y horizontal
#########################

if self.applyFlips:

    flipMode = random.randint(-1,1)

    flipImg = cv2.flip(curImage, flipMode)
    X.append(flipImg)
    flipGT = cv2.flip(curGT, flipMode)
    # Se borra el canal B, para tener 2 clases a la salida, fondo y edificio
    flipGTwoB =flipGT[:,:,:2]

    y.append(flipGTwoB)
```

Figura 14: Extracto del generador de datos. Bloque que implementa los flips. Fuente: cuaderno de Jupyter

En la figura 15 se muestra el bloque del generador de datos si se desea aplicar las rotaciones.

```
#########################
# ROTACIÓN
#
# Para crear una rotación aleatoria del lote
# Se genera un entero aleatorio entre -1 y 1:
# Posibles valores: -1, 0 y 1.
#
# rotationMode =  0: rota 180 grados
# rotationMode =  1: rota  90 grados
# rotationMode = -1: rota 270 grados
#########################

elif self.applyRotation:

    indice = random.randint(-1,1)

    if indice==0:
        rotationMode = cv2.ROTATE_180
    elif indice==1:
        rotationMode = cv2.ROTATE_90_CLOCKWISE
    else:
        rotationMode = cv2.ROTATE_90_COUNTERCLOCKWISE

    rotationImg = cv2.rotate(curImage, rotationMode)
    X.append(rotationImg)
    rotationGT = cv2.rotate(curGT, rotationMode)
    # Se borra el canal B, para tener 2 clases a la salida, fondo y edificio
    rotationGTwoB=rotationGT[:,:,:2]
    y.append(rotationGTwoB)
```

Figura 15: Extracto del generador de datos. Bloque que implementa las rotaciones. Fuente: cuaderno de Jupyter

Una vez definido el generador de datos, hay que partir el dataset.
Esta tarea se lleva a cabo con la función *splitData()* del cuaderno.
El dataset quedará dividido en 3 partes: trainset (conjunto de entrenamiento), testset (conjunto de prueba) y valset (conjunto de validación).
Respecto al tamaño apropiado de cada conjunto, existen diferentes corrientes. Existe consenso en que el conjunto de entrenamiento debe ser el mayor pues es el que usa la red para aprender. Con el conjunto de validación se testea el entrenamiento y se suele evaluar el modelo con el conjunto de test, para medir la calidad de la red. Estos dos últimos conjuntos son los menores.
Se ha elegido una de las dos proporciones típicas (la del 70-20-10%) que sugieren numerosos artículos [11]. La otra alternativa es la que prefiere un 80% para el entrenamiento, inspirada en el principio de Pareto.

Finalmente, se crean los generadores de datos de entrenamiento, test y validación con su respectivo conjunto de datos. En la figura 16 se muestra cómo se divide el conjunto de datos y se crean los 3 generadores en el cuaderno.

```
# Se dividen los PNG
trainSet, testSet, valSet = splitData(aerialImagesOutputPath)

# Se crea el trainGenerator
trainGenerator = DataGenerator(trainSet, True, False, False)
# Prueba con aumentación de datos "FLIPS"
# trainGenerator = DataGenerator(trainSet, True, True, False)
# Prueba con aumentación de datos "ROTACIÓN"
# trainGenerator = DataGenerator(trainSet, True, False, True)

#Se crea el testGenerator
testGenerator=DataGenerator(testSet, True, False, False)
#Se crea el valGenerator
valGenerator=DataGenerator(valSet, True, False, False)
```

Figura 16: Bloque que ejecuta la división del dataset y la creación de los generadores de datos. Fuente: cuaderno de Jupyter

3. Estudio de modelos secuenciales

3.1 Breve descripción

Los modelos secuenciales [12] se pueden definir como una secuencia de capas. Cada una de ellas tiene un tensor de entrada y uno de salida. A su vez, un tensor es un array multidimensional de tipo uniforme.
Dentro de los modelos secuenciales se propone utilizar, para ilustrarlos, la arquitectura codificador-decodificador que es muy utilizada para segmentar imágenes.

El codificador transforma la imagen de entrada, gracias a una secuencia de capas, en un conjunto menor de información. En la implementación propuesta, se utilizarán capas convolucionales y de agrupamiento pues son las más comunmente usadas.

Las capas convolucionales se ocuparán de la convolución de fragmentos 3*3 y se aplicará la operación no lineal *ReLU* y las de agrupamiento reducirán la dimensión de la matriz de entrada. Se consigue convirtiendo cada submatriz de entrada 2*2 en una única celda cuyo valor será el máximo de entre los cuatro precedentes.

El decodificador por su parte, toma como entrada la salida del codificador y la convierte de nuevo en su entrada inicial, pero representada por las dos clases del ground truth. En la implementación considerada también se usan las clásicas capas: convolucionales y las de upsampling (que duplica las dimensiones de su entrada).

3.2 Principales configuraciones analizadas

La primera parametrización del modelo, representada en la figura 17, está formada por solo 9 capas y pocos parámetros: 1.358.

```
# FASE ENCODER
Conv2D(4, (3, 3), activation='relu', padding='same',input_shape=(512,512,3)),
MaxPooling2D((2, 2), padding='same'),

Conv2D(8, (3, 3), activation='relu', padding='same'),
MaxPooling2D((2, 2), padding='same'),

# FASE DECODER
Conv2D(8, (3, 3), activation='relu', padding='same'),
UpSampling2D((2, 2)),

Conv2D(4, (3, 3), activation='relu', padding='same'),
UpSampling2D((2, 2)),

Conv2D(2, (3, 3), activation='softmax', padding='same')
```

Figura 17: Modelo secuencial inicial. Fuente: cuaderno de Jupyter

Se intenta entrenar varias veces, pero ninguna de las ejecuciones converge, obteniéndose siempre a partir de epochs muy tempranas una precisión de la categorización constante de entre un 82-83%.

Se piensa entonces en aumentar el número de parámetros con vistas a, por un lado a lograr un modelo convergente, y por otro a mejorar la precisión.
La segunda parametrización también está formada por 9 capas, pero en cambio el número de parámetros crece hasta los 298.178. Como se puede observar en la figura 18, esto se consigue añadiendo más filtros a cada capa convolucional.

```
# FASE ENCODER
Conv2D(64, (3, 3), activation='relu', padding='same',input_shape=(512,512,3)),
MaxPooling2D((2, 2), padding='same'),

Conv2D(128, (3, 3), activation='relu', padding='same'),
MaxPooling2D((2, 2), padding='same'),

# FASE DECODER
Conv2D(128, (3, 3), activation='relu', padding='same'),
UpSampling2D((2, 2)),

Conv2D(64, (3, 3), activation='relu', padding='same'),
UpSampling2D((2, 2)),

Conv2D(2, (3, 3), activation='softmax', padding='same')
```

Figura 18: Modelo secuencial inicial con aumento de parámetros. Fuente: cuaderno de Jupyter

Con esta cantidad tan enorme de parámetros, el modelo sí converge hasta una precisión aceptable de alrededor del 89%. El problema es que el entrenamiento es muy lento y no se consigue que Colab pase de la epoch nº 20 ya que el entorno de ejecución se desconecta en los distintas pruebas llevadas a cabo (véase figura 1).

En este punto, se intenta un cambio de estrategia consistente en aumentar el nº de capas (17), pero con un nº de parámetros intermedio (21.614) para hacer manejable el entrenamiento. Esta estructura se visualiza en la figura 19.

```
# FASE ENCODER
Conv2D(4, (3, 3), activation='relu', padding='same',input_shape=(512,512,3)),
MaxPooling2D((2, 2), padding='same'),

Conv2D(8, (3, 3), activation='relu', padding='same'),
MaxPooling2D((2, 2), padding='same'),

Conv2D(16, (3, 3), activation='relu', padding='same'),
MaxPooling2D((2, 2), padding='same'),

Conv2D(32, (3, 3), activation='relu', padding='same'),
MaxPooling2D((2, 2), padding='same'),

# FASE DECODER
Conv2D(32, (3, 3), activation='relu', padding='same'),
UpSampling2D((2, 2)),

Conv2D(16, (3, 3), activation='relu', padding='same'),
UpSampling2D((2, 2)),

Conv2D(8, (3, 3), activation='relu', padding='same'),
UpSampling2D((2, 2)),

Conv2D(4, (3, 3), activation='relu', padding='same'),
UpSampling2D((2, 2)),

Conv2D(2, (3, 3), activation='softmax', padding='same')
```

Figura 19: Modelo secuencial final. Fuente: cuaderno de Jupyter

Los distintos entrenamientos realizados sobre este modelo, se lanzaron mediante la función que muestra la figura 20.

```
epochs = 50
trainHistory = theModel.fit(trainGenerator, epochs=epochs, validation_data=valGenerator)
```

Figura 20: Bloque de entrenamiento del modelo secuencial. Fuente: cuaderno de Jupyter

Estos entrenamientos consiguen llegar sin problemas a la epoch 50 con una precisión mejorada de alrededor del 91% (dependiendo de la ejecución), lo que se considera más que aceptable. Se da esta configuración por buena, y se toma como ejemplo del estudio del modelo secuencial.

3.3 Persistencia del entrenamiento y gráficas de precisión y pérdida

Los entrenamientos de los modelos pueden llegar a durar una cantidad de horas considerable. Es por ello que es conveniente guardarlos en disco, lo que permite ser recuperados a voluntad tiempo después y

conseguir hacer predicciones sin necesidad de repetir el entrenamiento previamente cada vez.
Las funciones de Python, implementadas en el cuaderno, que permiten persistir y recuperar el entrenamiento del modelo son respectivamente: *save_trained_model()*, figura 21, y *load_trained_model()*, figura 22.

```
def save_trained_model(fileName, theModel, trainHistory):

  # Se guarda el trainHistory
  with open(os.path.join(modelsPath,fileName) + '.pkl', 'wb') as historyFile:
        pickle.dump(trainHistory.history, historyFile)

  # Se salva el modelo
  theModel.save(os.path.join(modelsPath,fileName) + '.hd5')
```

Figura 21: Función save_trained_model(). Fuente: cuaderno de Jupyter

```
def load_trained_model(fileName):

  # Se carga el modelo
  new_model = models.load_model(os.path.join(modelsPath,fileName) + '.hd5')

  # Se carga el trainHistory
  historyStream = open(os.path.join(modelsPath,fileName) + '.pkl','rb')
  historyFile = pickle.load(historyStream)
  historyStream.close()

  return new_model, historyFile
```

Figura 22: Función load_trained_model(). Fuente: cuaderno de Jupyter

Una característica de esos entrenamientos persistidos es que distintas magnitudes de su historia pueden ser representadas gráficamente, lo que permite descubrir algunos fenómenos interesantes.

En la figura 23 se muestra la gráfica de la *pérdida* que permite observar cómo se comporta ésta durante el entrenamiento y la validación con el paso de las epochs.

Alrededor de la epoch 20, se puede ver cómo la pérdida del entrenamiento continúa bajando mientras que la de la validación empieza a fluctuar. Este comportamiento se denomina *overfitting* y significa que el modelo en vez de continuar aprendiendo, memoriza los datos del entrenamiento.

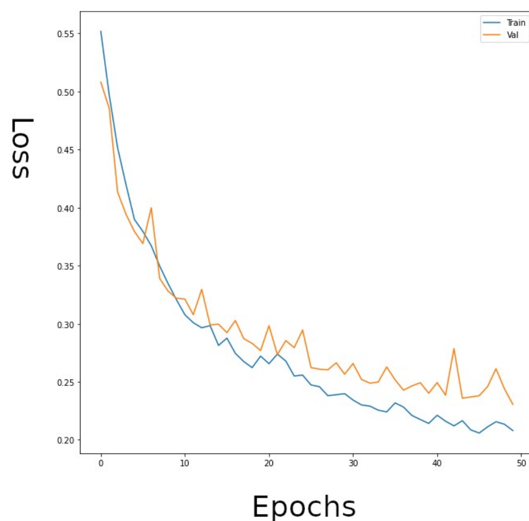

Figura 23: Gráfica de pérdida. Fuente: cuaderno de Jupyter

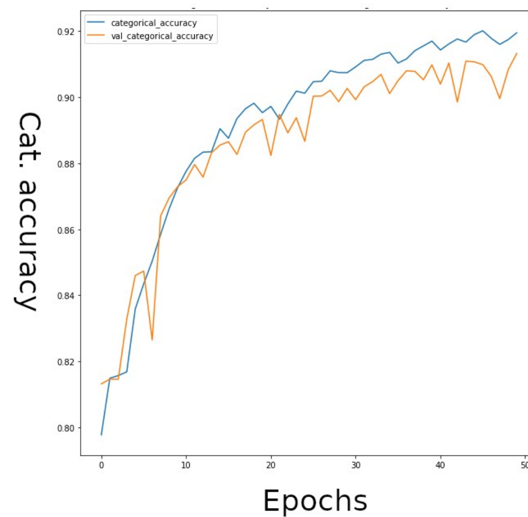

Figura 24: Gráfica de precisión. Fuente: cuaderno de Jupyter

En la figura 24 se representa la gráfica de la precisión. Se aprecia también cómo alrededor de la epoch 20, la precisión del entrenamiento y de la validación se separan y esta última empieza a fluctuar y ya no llega a mucho más de ese 90%. Es decir, el modelo llega a un punto en el que ya no aprende más y por tanto no consigue mejorar su precisión.

3.4 Evaluación de la alternativa seleccionada

Una vez definido el modelo, se evalúa con el conjunto de test, con la llamada mostrada en la figura 25, para comprobar si la precisión obtenida es semejante a la del entrenamiento.

```
results = theModel.evaluate(testGenerator, verbose=1)

11/11 [==============================] - 167s 13s/step - loss: 0.2180 - categorical_accuracy: 0.9165
```

Figura 25: Bloque de evaluación del modelo secuencial. Fuente: cuaderno de Jupyter

Efectivamente es así (91%) y se concluye que el entrenamiento fue exitoso.

3.5 Predicción: representación gráfica y medidas

Como colofón al estudio de los modelos secuenciales se le presenta al modelo, entrenado y evaluado, un lote de imágenes inéditas (nunca

vistas por él) para que prediga la segmentación. Es decir, se le van a enseñar imágenes nuevas a la red para que ésta se pronuncie indicando dónde cree que están los edificios que contienen las fotos.

En la implementación con código Python esto se consigue seleccionando, por ejemplo, el lote 1 del conjunto de test y llamando a la función predict como se ve en la figura 26:

```
# Se selecciona un lote, el 1 por ejemplo
[X_predict,y_predict]=testGenerator.__getitem__(1)

# Se hace una predicción de sus ground truths
# con el modelo secuencial entrenado y evaluado
prediction = theModel.predict(X_predict)
```

Figura 26: Bloque de predicción de un lote. Fuente: cuaderno de Jupyter

No se ha utilizado la aumentación de datos para controlar qué imágenes de salida se obtienen, y poder utilizar las mismas en el siguiente modelo para compararlas. Cuando se aplican flips o rotaciones, hay un random implicado que podría provocar que con una nueva llamada a *__getitem__* se obtuvieran imágenes diferentes de cada vez.

A continuación se explican las imágenes involucradas en la predicción que se pueden ver en el cuaderno.

La primera fila, que se muestra en la figura 27, está formada por las imágenes de referencia del lote inédito:

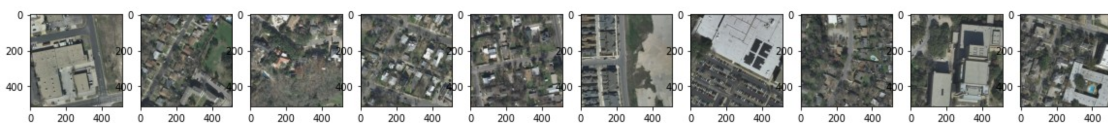

Figura 27: Imágenes de referencia del lote inédito. Fuente: cuaderno de Jupyter

En la figura 28 se enseña la segunda fila, que son los ground truths de las imágenes de referencia:

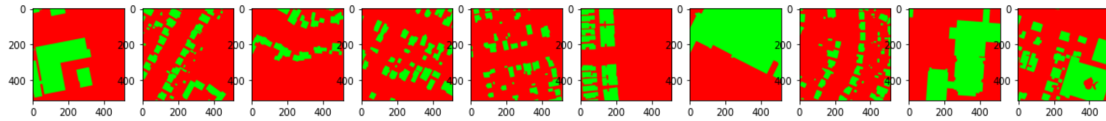

Figura 28: Ground truths del lote inédito. Fuente: cuaderno de Jupyter

La tercera fila, figura 29, son las predicciones por probabilidades (límites borrosos).

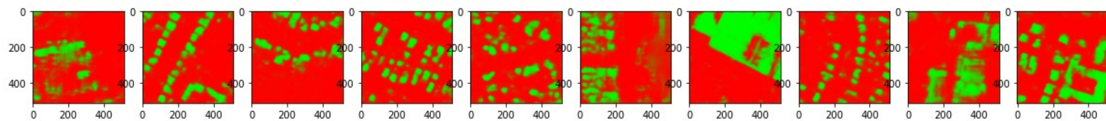

Figura 29: Predicciones por probabilidades del lote inédito. Fuente: cuaderno de Jupyter

La cuarta fila son también las predicciones pero redondeadas a una de las clases (utilizando la función *argmax*). Los límites son nítidos ya que pertenecen o a una clase o a otra (edificio o fondo) como se aprecia en la figura 30.

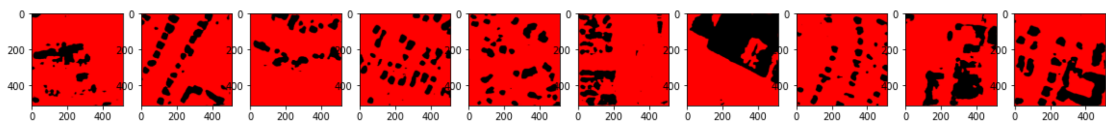

Figura 30: Predicciones redondeadas a las clases del lote inédito. Fuente: cuaderno de Jupyter

La quinta y última fila sirve para ayudar en el análisis cualitativo de la bondad de las predicciones. Para cada pixel se resta el valor en el ground truth del de la predicción redondeada y se coge el valor absoluto. Las imágenes reflejarán así, cuánto le falta a la predicción de cada edificio para llegar a ser el edificio exacto. Por lo tanto cuantos más píxeles negros haya, peor habrá resultado la predicción como se ve en la figura 31.

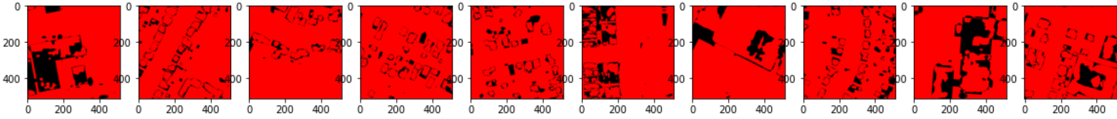

Figura 31: Diferencias entre los ground truths y las predicciones redondeadas del lote inédito. Fuente: cuaderno de Jupyter

Análisis cualitativo:

Para este lote en concreto las predicciones son más que aceptables. En general solo se muestran los bordes de los edificios donde la identificación de imágenes sí es más difusa. Quizás las peores predicciones sean la primera y la penúltima donde aparecen bastantes píxeles negros (que indican que existe diferencia entre el ground truth y la predicción redondeada).

Análisis cuantitativo:

Además de un análisis subjetivo de las predicciones, en cuanto a su naturaleza visual, se pueden extraer métricas que ayuden al enjuiciamiento objetivo de las predicciones.

Éstas son: Accuracy, Precision, Recall y F1 score.

Accuracy	0.883
Precision	0.885
Recall	0.969
F1 score	0.925

Tabla 2: Métricas del modelo secuencial seleccionado

Las métricas obtenidas son congruentes y están en el mismo orden que las del entrenamiento y la evaluación.

4. Estudio de modelos preentrenados

4.1 Breve descripción

Existen otros modelos de redes neuronales con eficacia demostrada segmentando imágenes, al haber sido entrenados previamente.

Se estudiaron distintos modelos preentrenados [13], a saber:

VGG-16: es un modelo secuencial que implementa un encoder.

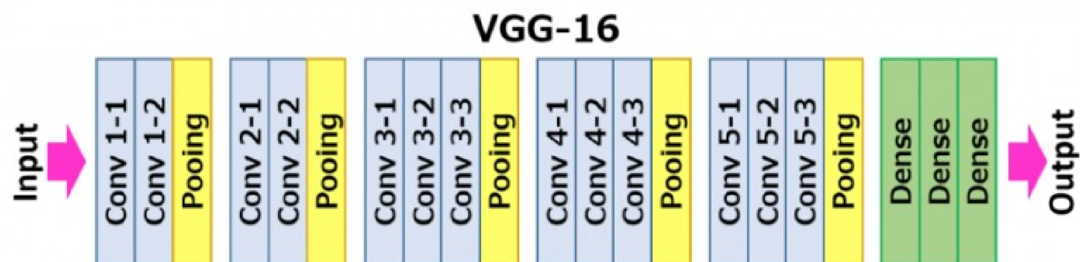

Figura 32: Arquitectura del modelo VGG-16. Fuente: [13]

Como se observa en esta imagen, cuenta con 21 capas y requiere un imagen de entrada con dimensiones (224, 224, 3). En el proyecto se trabaja con shape (512, 512, 3) así que habría que redimensionarla para poder usarlo. Como el objetivo es probar exactamente las mismas imágenes en distintos modelos de red sin modificaciones artificiales y compararlas, se descarta.

Inception versión 3: que no es más que una versión mejorada del Inception original de 22 capas. Es más rápido que VGG-16 pero también requiere una input shape fija de (150, 150, 3), con lo que es descartado.

EfficientNet versión B0: que usa un método de escalado nuevo (compound scaling). Como los 2 anteriores, se intenta probar con el dataset del INRIA pero requiere una input shape de (224, 224, 3) y se descarta también.

4.2 Principales configuraciones analizadas

El primer modelo preentrenado que pudo ser analizado fue **Resnet34**. Cuenta con 34 capas (de ahí su nombre) y 24 millones de parámetros de los cuales 3 (millones) son entrenables.

Al ser un modelo preentrenado no se pueden adicionar o quitar capas como en los secuenciales, de este modo las pruebas consistieron en entrenarlo con más o menos fotos y con o sin *freeze*: se congela la parte del encoder y el modelo solo se entrena con el decodificador.

Las primeras pruebas se realizaron usando la congelación en el encoder y pocas fotos (las 500 usadas en los secuenciales). La precisión era notablemente peor que en los modelos anteriores (rondaba el 80%), y además el proceso de entrenamiento era mucho más pesado (unas 2 horas por epoch). Sin congelación se obtuvieron resultados parecidos.
Se intentó entonces aumentar el tamaño del dataset, a unas 2.000 imágenes, para que la red tuviera una base de aprendizaje mejor, pero se obtuvieron también resultados parecidos y ejecuciones (lógicamente) aún más lentas.
Las trazas de los entrenamientos pueden verse en el cuaderno.
No habiendo conseguido batir al secuencial con ninguna de las configuraciones, se descarta el modelo.

El segundo modelo que pudo ser también entrenado, fue **Resnet50**. Cuenta con más parámetros aún que Resnet34 ya que tiene más capas, 50: unos 32,5 millones, de las que 9 millones son entrenables.

Desde los primeros entrenamientos se comprobó su buen rendimiento. A las pocas epochs ya alcanzaba una precisión del 95%, superior al del modelo secuencial seleccionado. Fueron varios los entrenamientos realizados (con ejecuciones de horas de duración) que se llevaron a cabo conformados por unas 20 epochs cada uno, pero el entorno de ejecución de Colab siempre abortaba antes de terminar. Se decide entonces determinar el overfitting con los entrenamientos disponibles y entrenar el modelo justo hasta esa epoch, con el objetivo de conseguir una ejecución completa.
El overfitting se detectó en el epoch número 14.
Se volvió a entrenar el modelo planificando esas 14 epochs y la ejecución terminó exitosamente.

4.3 Persistencia del entrenamiento y gráficas de precisión y pérdida

Entrenar el modelo es muy costoso en tiempo y en recursos computacionales. Los entrenamientos se prolongan durante horas, aunque se utilice la unidad de procesamiento gráfico (GPU) que provee Colab o se realicen solo hasta detectar el overfitting.
Es fundamental, pues, guardar los modelos para poder usarlos a posteriori para realizar predicciones, sin necesidad de volver a entrenarlos de cada vez.
Se usa la misma función (*save_trained_model()*) que en los modelos secuenciales como se aprecia en la figura 33.

```
save_trained_model('resnet50',modelUnet,trainHistory)
```

Figura 33: Llamada a la función save_trained_model. Fuente: cuaderno de Jupyter

Las gráficas de precisión y pérdida serán diferentes de las del modelo secuencial, ya que como se ha explicado, solo se entrena el modelo hasta que deja de aprender.

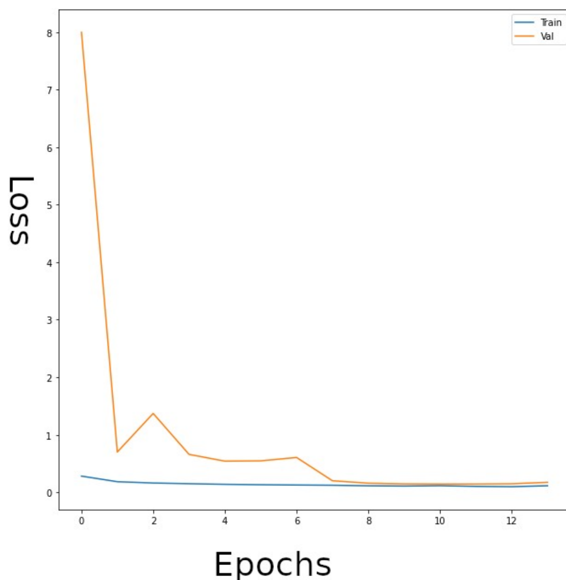

Figura 35: Gráfica de pérdida. Fuente: cuaderno de Jupyter

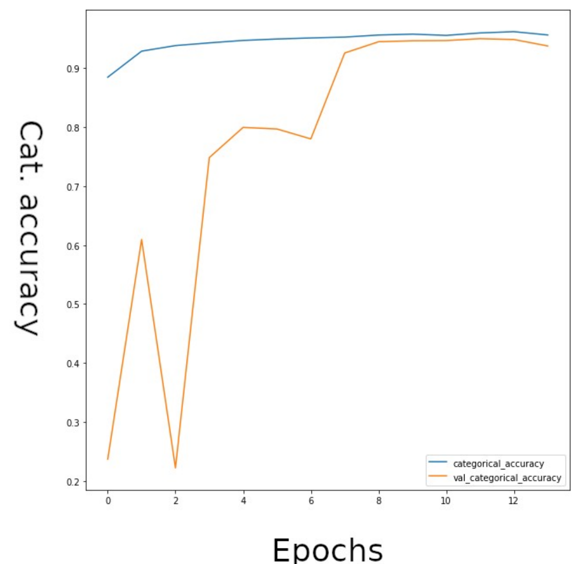

Figura 34: Gráfica de precisión. Fuente: cuaderno de Jupyter

En la figura 34, donde se muestra la gráfica de pérdida, la de entrenamiento no continúa bajando. En la figura 35, en la que se ve la gráfica de precisión, la de entrenamiento no continúa subiendo.

4.4 Evaluación de la alternativa seleccionada

Se usa nuevamente la función evaluate, cuya llamada en código Python se ve en la figura 36, y se comprueba que la salida devuelta es coherente con lo obtenido en el entrenamiento, con una precisión que ronda el 95%. Por lo tanto, se da por bueno el entrenamiento hasta el overfitting.

```
results = modelUnet.evaluate(testGenerator, verbose=1)

40/40 [==============================] - 911s 23s/step - loss: 0.1694 - categorical_accuracy: 0.9409
```

Figura 36: Llamada a la función evaluate. Fuente: cuaderno de Jupyter

4.5 Predicción: representación gráfica y medidas

Se usará el mismo lote que en el modelo anterior (el 1) para poder posteriormente comparar cualitativa y cuantitativamente ambos modelos.

Se usa la función predict y se representan las filas con las imágenes de forma análoga al punto 3.5.

La primera y segunda fila muestran nuevamente las imágenes de referencia del lote 1 inédito y sus ground truths, como se aprecia en las figuras 27 y 28 respectivamente.

La tercera fila, figura 37, son las predicciones por probabilidades (límites borrosos):

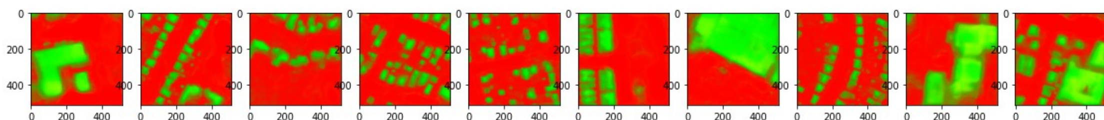

Figura 37: Predicciones por probabilidades del lote inédito. Fuente: cuaderno de Jupyter

La cuarta fila, mostrada en la figura 38 son las predicciones redondeadas a una de las clases gracias a la función argmax.

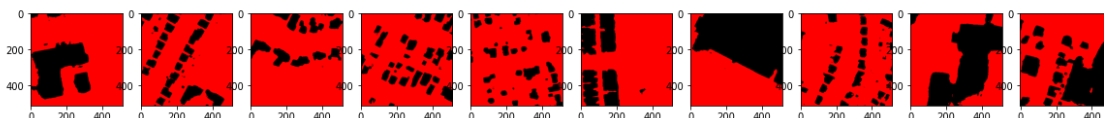

Figura 38: Predicciones redondeadas a las clases del lote inédito. Fuente: cuaderno de Jupyter

La quinta y última fila refleja las diferencias entre la predicción redondeada y el ground truth. Se muestra gráficamente en la figura 39 en qué grado acierta el modelo.

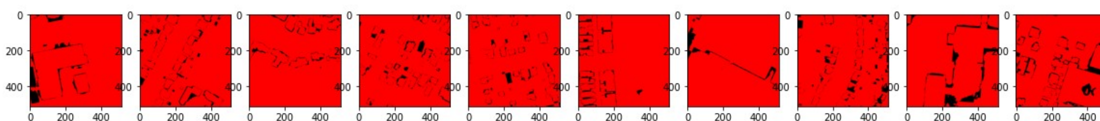

Figura 39: Diferencias entre los ground truths y las predicciones redondeadas del lote inédito. Fuente: cuaderno de Jupyter

Análisis cualitativo:

Se observa que las predicciones muestran gran fiabilidad. La única que destaca en sentido negativo sobre las demás, es la penúltima y quizás la primera. Se observan trazos negros más gruesos en los bordes de los edificios que indican que hay más diferencia entre la predicción y las clases reales.

Análisis cuantitativo:

Se calcularán de nuevo las métricas, como se hizo en el modelo secuencial seleccionado. Sus resultados están la tabla 3.

Accuracy	0.950
Precision	0.970
Recall	0.962
F1 score	0.966

Tabla 3: Métricas del modelo preentrenado seleccionado

Las métricas obtenidas son congruentes y están en el mismo orden que las del entrenamiento y la evaluación, con precisiones que rondan el 95%.

5. Comparativa entre modelos

Antes de proceder a describir las diferencias, es preciso aclarar que el estudio se realiza sobre los dos modelos concretos seleccionados de cada tipo y que no es extrapolable a la generalidad de los modelos analizados, como se comentó en el *abstract*.

Análisis cualitativo:

Se aprecia a simple vista que las predicciones del modelo preentrenado son mucho más fieles a la realidad que las del secuencial.

Se mostrarán dos ejemplos bastante descriptivos analizando distintas imágenes de la **quinta fila**, que como se explicó permiten visualizar las diferencias entre la predicción redondeada y el ground truth.

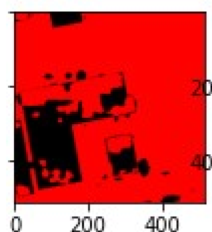

Figura 40: Primera foto del lote 1 del modelo secuencial. Fuente: cuaderno de Jupyter

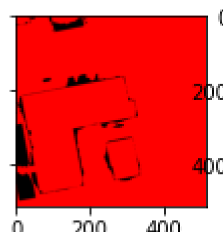

Figura 41: Primera foto del lote 1 del modelo preentrenado. Fuente: cuaderno de Jupyter

En el primer ejemplo, se aprecia que en la figura 40 (modelo secuencial) se observan una cantidad considerable de píxeles negros dentro del edificio con forma de r minúscula, mientras que en la figura 41 (modelo preentrenado) las diferencias se concentran en una delgada línea que indica el límite de la construcción. Los píxeles negros indicaban la diferencia en valor absoluto entre la clase predicha y la clase en el ground truth de referencia y están en mucha menor proporción en el preentrenado que resulta claramente ganador.

El segundo ejemplo se ilustra con las figuras 42 y 43 que muestran la séptima imagen del lote 1 predicha por cada uno de los modelos.

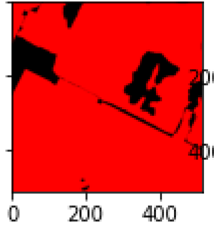

Figura 42: Séptima foto del lote 1 del modelo secuencial. Fuente: cuaderno de Jupyter

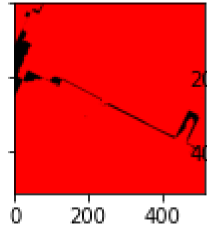

Figura 43: Séptima foto del lote 1 del modelo preentrenado. Fuente: cuaderno de Jupyter

Se pueden obtener las mismas conclusiones que en el primer ejemplo, las diferencias son claramente menores en el modelo preentrenado, donde prácticamente solo se acierta a adivinar la silueta del edificio.

Análisis cuantitativo:

Los valores de precisión en el entrenamiento y en la valuación muestran un claro ganador en el análisis cualitativo: el modelo preentrenado. Mientras el modelo secuencial arroja una precisión del 91%, se obtiene alrededor de un 95% en el preentrenado.

En las métricas obtenidas como resultado de predecir el lote nº 1, el resultado se refrenda con valores parecidos. Solo cabe destacar que son algo menores de lo esperado en el secuencial que no llega al 90%. De todas formas en otras ejecuciones realizadas sí se llegó a esa cifra, e incluso a la esperada del 91%, ya que los resultados dependen de la ejecución concreta.

6. Conclusiones

Una vez concluido el trabajo se pueden extraer varias lecciones. Las redes neuronales son capaces hoy en día de segmentar imagenes con calidad y rapidez. Sorprende la enorme destreza con la que definen casi a la perfección la localización de los edificios en imágenes inéditas y aunque es cierto que se pierden bastantes horas entrenando a los modelos, hay que destacar que es un tarea que solo hay que realizar en una ocasión. Una vez que el entrenamiento queda guardado en disco, se pueden realizar segmentaciones de imágenes a posteriori sin tener que volver a repetirlo. Por otro lado, este trabajo se ha particularizado en la segmentación de edificios, y se ha apuntado la detección de irregularidades urbanísticas como una de sus posibles aplicaciones, pero es evidente que esta tecnología puede ser utilizada para segmentar cualquier ítem y abarcar ilimitados campos, medicina por ejemplo, con sus correspondientes soluciones.
Durante el estudio de los modelos preentrenados, se ha podido comprobar lo enormente fácil que resulta usarlos. Los secuenciales necesitan de bastantes conocimientos previos para poder definirlos, y requieren de un proceso iterativo de «afinado». Por el contrario en los preentrenados, al estar ya predefinidos, todo resulta mucho más sencillo: basta con escoger el modelo concreto, los pesos, el nº de clases de salida y las funciones de pérdida y métricas como se observa en la figura 44.

```
modelUnet = Unet(backbone_name='resnet50', encoder_weights='imagenet', encoder_freeze=True, input_shape=(512,512,3), classes=2)
modelUnet.compile('Adam', loss='categorical_crossentropy', metrics=['categorical_accuracy'])
```

Figura 44: Instanciación y compilación de un modelo preentrenado. Fuente: cuaderno de Jupyter

Aunque como se demostró en el trabajo, se pueden hallar configuraciones de arquitecturas conocidas con buen rendimiento, es preferible usar modelos preentrenados. Tienen un funcionamiento predecible de antemano, e incluso como se ha visto, se pueden conseguir mejores resultados. Respecto a la consecución de objetivos, se constata que se alcanzó tanto el objetivo general perseguido (segmentar imágenes aéreas con una calidad más que aceptable), como los objetivos parciales planteados necesarios para alcanzar esa meta: depuración del dataset y estudio de diferentes modelos de redes neuronales. La planificacion inicial se siguió durante todo el proyecto, pero no se pudieron alcanzar algunos hitos en las fechas estimadas. Tras la primera parte del desarrollo y debido a la inexperiencia del autor, no se había siquiera entrenado con éxito el modelo secuencial. El *gap* se solventó gracias que se había previsto más esfuerzo en esa primera parte descargando la segunda por si había que encajar retrasos previos como así aconteció. Algunos desarrollos posibles que afloraron durante

la ejecución del proyecto, se apuntarán como líneas de trabajo futuras, sin embargo la previsión inicial se cumplió.
La metodología de desarrollo de software clásica (waterfall) también resultó adecuada. El proyecto tenía un horizonte temporal prefijado, y unos objetivos y alcance definidos perfectamente en la PACs previas al desarrollo. Con un solo recurso no había necesidad de puestas común con el resto del equipo, y la naturaleza iterativa de sus fases encajaba perfectamente en los estudios de los distintos modelos de redes neuronales, muchas veces basados en la «prueba y error». Además la necesidad de aportar una documentación amplia de cada tarea junto a esta memoria final encajan más en un enfoque clásico.
Los desarrollos ágiles están más pensados para el trabajo en equipo o la flexibilidad que aporta para responder a nuevos requisitos en ciclos rápidos (sprints) y donde la puesta en producción de nuevas releases prima sobre la creación de documentación como se requiere en un TFM.

7. Líneas de trabajo futuras

Si se desea continuar con este proyecto existen varias líneas de trabajo interesantes de dos tipos claramente diferenciados.

Parte científico-técnica o de investigación:

Una primera idea sería lograr un segmentación más precisa de las imágenes de partida, las del dataset inicial del INRIA. En el proyecto actual, solo se consigue una única asociación aproximada de cada píxel, en los cortes disjuntos, a una clase objetivo (edificio o fondo). Se propone deslizar una ventana del tamaño elegido en el proyecto (512*512 píxeles), con un avance en píxeles a determinar (de uno en uno, de dos en dos, etcétera), hasta «barrer» toda la imagen padre. Se aplica una red neuronal a esa ventana y se obtienen «n» segmentaciones de cada píxel. Esas segmentaciones pueden combinarse con alguna función matemática (la media, por ejemplo) con lo que se podría obtener una segmentación de la predicción mucho más aproximada a la del ground truth original, ya que cada uno de los píxeles no se segmenta una sino varias veces, incrementando de esta forma la precisión.

La segunda línea que se puede seguir, parte de la idea planteada en la introducción de crear un sistema que detecte irregularidades urbanísticas. No basta con conseguir una buena segmentación de edificios en una imagen aérea de un instante concreto del tiempo. Se propone aplicar una red neuronal a imágenes de series de tiempo, que identifique nuevas construcciones irregulares con el paso de los meses y que distinga además, por ejemplo, fenómenos naturales como una erupción como la ocurrida en la isla de La Palma en 2021 o fuertes lluvias que generen zonas anegadas en un terreno y sean susceptibles de confundirse con una piscina en forma de playa.

Figura 45: Ejemplo de piscina en forma de playa. Fuente: Piscinas Natursand

Parte desarrollo:

En cuanto a posibles trabajos futuros de desarrollo, podría plantearse implementar y planificar procesos batch que se lancen periódicamente para cargar imágenes nuevas en el sistema.
El objetivo último sería disponer de una aplicación de escritorio o web que, con la ayuda de esos nuevos datos cargados y teniendo en cuenta los anteriores, permita la identificación y visualización de las irregularidades urbanísticas sobre un mapa. Como funcionalidades adicionales se puede plantear la creación de notificaciones y alertas que se disparen al encontrar alguna ilegalidad.

8. Glosario

- **4K**: tamaño de imagen que tiene sobre 4000 píxeles de resolución horizontal.
- **Accuracy**: (TP + TN) / (TP + TN + FP + FN)
- **Agile**: metodología de desarrollo de proyectos rápida y flexible.
- **Argmax**: función que encuentra los índices de los valores más altos de un array.
- **CAPTCHA** (Completely Automated Public Turing test to tell Computers and Humans Apart): prueba que determina si el usuario de un sistema informático es humano.
- **Clockwise**: sentido de las agujas del reloj.
- **CNN** (Convolutional Neural Network): es un tipo de red neuronal artificial capaz de ver e identificar objetos.
- **Colab**: producto de Google que permite la edición y ejecución de código Python en un navegador de forma interactiva.
- **Concello**: ayuntamiento de la comunidad autónoma de Galicia.
- **Convolución**: filtrado de una imagen usando una máscara.
- **Counter clockwise**: sentido contrario al de las agujas del reloj.
- **Cuaderno de Jupyter**: archivo con extensión .ipynb que contiene celdas de código Python, texto o metadatos raw (imágenes, por ejemplo).
- **Data augmentation**: generación artificial de nuevos datos mediante modificaciones de los datos originales.
- **Dataset**: conjunto de datos, en concreto de imágenes en este contexto.
- **Epoch**: cada uno de los ciclos de un entrenamiento.
- **F1 score**: 2*TP / (2*TP+FP+FN)
- **Flip**: inversión de una imagen.
- **FN** (False Negative): el modelo predice que la muestra es negativa y falla.
- **FP** (False Positive): el modelo predice que la muestra es positiva y falla.
- **Gantt**: cronograma de un proyecto.
- **Gap**: problema de complejidad variable detectado en una implantación.
- **Gdrive**: servicio de almacenamiento de archivos de Google.
- **Google**: filial de la compañia estadounidense Alphabet que presta servicios y ofrece productos por internet.
- **Google maps**: servidor de aplicaciones de mapas de Google.
- **GPU** (Graphics Processing Unit): unidad de procesamiento gráfico.
- **Ground truth**: imagen que asigna a cada uno de sus píxeles, una clase en su imagen de referencia.
- **Input**: entrada.
- **INRIA** (Institut National de Recherche en Informatique et en Automatique): Instituto Nacional de Investigación en Informática y Automática de Francia.

- **Keras**: biblioteca de código abierto escrita en Python para trabajar con redes neuronales.
- **Matriz de confusión**: matriz de dimensión 2*2, cuya primera fila es TP FP y cuya segunda fila es FN TN.
- **Max pooling**: operación de agrupación que calcula el valor máximo de una submatriz.
- **Output**: salida.
- **Overfitting**: fenómeno que se presenta cuando un modelo memoriza los datos del entrenamiento en vez de aprender.
- **PAC** (Prova d'Avaluació Contínua): prueba de evaluación continua.
- **PGOM**: Plan General de Ordenación Municipal.
- **Píxel**: cada uno de los puntos de color que componen una imagen.
- **PNG** (Portable Network Graphics): formato de imagen con compresión sin pérdida.
- **Precision**: TP / (TP + FP)
- **Principio de Pareto**: ley que establece que el 20% del esfuerzo dedicado a realizar una tarea produce el 80% de los resultados.
- **Python**: lenguaje de programación de propósito general. Puede usarse de forma imperativa, orientada a objetos o funcional.
- **Recall**: TP / (TP + FN)
- **ReLU**: función de activación utilizada en modelos de aprendizaje profundo. Para un input negativo, devuelve 0. Para un input positivo, devuelve ese mismo valor.
- **Resnet** (Residual neural network): estructura de red ganadora de varios campeonatos de segmentación de imágenes.
- **RGB** (Red Green Blue): modo de color basado en la suma de los colores primarios rojo, verde y azul.
- **Segmentación**: es un proceso que categoriza cada píxel de una imagen en una determinada clase, en este contexto.
- **Sequence**: clase de Keras para secuencias de datos.
- **Softmax**: función de activación que traduce los valores de las clases a probabilidades (valores entre 0 y 1).
- **Sprint**: ciclo de ejecución corto en un proyecto.
- **Test set**: conjunto de prueba.
- **TFM**: Trabajo Fin de Máster.
- **TIFF** (Tagged Image File Format): formato de imagen de mapa de bits.
- **TN** (True Negative): el modelo predice que la muestra es negativa y acierta.
- **TP** (True Positive): el modelo predice que la muestra es positiva y acierta.
- **Training set**: conjunto de entrenamiento.
- **Upsampling**: función que aumenta la resolución de la capa anterior.
- **Validation set**: conjunto de validación.
- **Waterfall**: cascada.
- **Xunta**: gobierno regional de la comunidad autónoma de Galicia.

9. Bibliografía

[1]: https://airdroneview.com/2014/07/04/historia-de-la-fotografia-aerea/. Fecha de visita: 22/12/2021.
[2]: https://www.google.es/maps. Fecha de visita: 24/12/2021.
[3]: https://www.bing.com/maps/. Fecha de visita: 21/12/2021.
[4]: https://wego.here.com/. Fecha de visita: 22/12/2021.
[5]: https://apps.sentinel-hub.com/eo-browser. Fecha de visita: 22/12/2021.
[6]: https://arxiv.org/pdf/2001.05566.pdf. Fecha de visita: 24/12/2021.
[7]: https://github.com/chrieke/awesome-satellite-imagery-datasets. Fecha de visita: 20/12/2021
[8]: https://project.inria.fr/aerialimagelabeling/. Fecha de visita: 20/12/2021
[9]: https://note.nkmk.me/en/python-opencv-numpy-rotate-flip/. Fecha de visita: 21/12/2021
[10]: stanford.edu/~shervine/blog/keras-how-to-generate-data-on-the-fly. Fecha de visita: 24/12/2021
[11]: researchgate.net/post/70_training_and_30_testing_split. Fecha de visita: 24/12/2021
[12]: keras.io/guides/sequential_model. Fecha de visita: 24/12/2021
[13]: top-4-pre-trained-models-for-image-classification. Fecha de visita: 24/12/2021

10. Anexos

El cuaderno de Jupyter que contiene el código fuente, puede ser descargado en este link (solo desde una cuenta @*uoc.edu*):

https://colab.research.google.com/drive/1fFoGQStClopvl6WYV5FRF-fugEtMmkOT?usp=sharing

Manual de Instalación del cuaderno de Jupyter:

*** Requisitos previos: contar con una cuenta de Google nombre_de_usuario@gmail.com ***

1. Acceder a Gdrive y crear en el directorio raiz una carpeta llamada DATA.
2. Dentro de DATA y todas al mismo nivel, crear las siguientes carpetas: INPUT, INPUT_GT, OUTPUT, OUTPUT_GT y MODELS_BACKUP.
3. Descargar el dataset de la web del INRIA: https://project.inria.fr/aerialimagelabeling/download/
4. Descomprimir la carpeta descargada con la herramienta 7z.
5. Navegar hasta la ruta ..\train\images, seleccionar los .tif deseados y copiarlos en la carpeta de Gdrive INPUT.
6. Navegar hasta la ruta ..\train\gt, seleccionar los mismos .tif deseados del punto 5 y copiarlos en la carpeta de Gdrive INPUT_GT.
7. Teclear en un navegador https://colab.research.google.com/notebooks/intro.ipynb
8. Menú Archivo>Subir cuaderno> Subir
9. Pulsar el botón Seleccionar archivo y buscar el cuaderno PAC3.ipynb.
10. Una vez subido el cuaderno, pulsar el botón play de la primera celda y dar permisos de acceso a Gdrive.
11. Menú Entorno de ejecución>Cambiar tipo de entorno de ejecución. Seleccionar en Acelerador por hardware: «GPU» para acelerar las ejecuciones de las celdas.