E.T.S. de Ingeniería Industrial, Informática y de Telecomunicación

Procesamiento de imágenes satélite de tipo radar

Sentinel-1 y su utilidad para la clasificación automática de edificios y carreteras

Máster Universitario en Ingeniería Informática

Trabajo Fin de Máster

Autora: Sonia Elizondo Martínez
Director: Mikel Galar Idoate

Pamplona, 12 de marzo de 2021

upna
Universidad Pública de Navarra
Nafarroako Unibertsitate Publikoa

"That brain of mine is something more than merely mortal; as time will show."

Ada Byron

"Este cerebro mío es más que meramente mortal, como el tiempo demostrará."

Ada Byron

Agradecimientos

A Mikel Galar, director de este Trabajo Fin de Máster, por guiarme en su realización y dedicarme su tiempo, visión y consejos. Por ser uno de los artífices de mi gran interés por la Inteligencia Artificial.

A Christian Ayala, por sus conocimientos y apoyo diario para el desarrollo de este proyecto.

A mi madre y a mi padre, por respaldar mi vocación de ser ingeniera y por su inmenso cariño durante todas las etapas de mi vida.

A mi hermano, David Elizondo, por servir de inspiración para superarme a mí misma, por alentarme a perseguir aquello que me apasiona y por demostrarme, a través de su propia experiencia, que el esfuerzo tiene su recompensa.

A mi abuela Esther, mi *abu*, a tres meses de que cumpla 99 años, por enseñarme a valorar el haber accedido a este nivel de educación.

A mi tía Gema, por demostrarme siempre ese amor tan grande que guía su vida.

A Alejandra Fernández, mi hermana, aunque no de sangre, por ser mi mayor confidente y por creer sin fisuras que puedo lograr lo que me proponga, incluso cuando yo dudo de mí misma.

A Idoia Cerro, mi *compi*, por su gran amistad durante toda nuestra trayectoria universitaria, por llenarla de alegría, anécdotas y nuevos proyectos. Por estar tan segura de mis logros como lo estoy yo de los suyos.

Abstract

Satellite images can be used for training Deep Learning models which allow to classify the elements that compose these images. Designing proper processing methods for the images is key for both the learning and the segmentation tasks.

This work focuses on radar images due to their advantages. Specifically, it thoroughly studies European Spatial Agency's Sentinel-1 satellite and its application alongside Sentinel-2 (hyperspectral satellite) for the automatic classification of buildings and roads. Thus, the project's main goal is to evaluate different processing flows for these radar images and compare the obtained results for the subsequent classification.

As the processing of Sentinel-1 images depends on software tools based on graphic interfaces, a library in the programming language Python has been developed to gather their functionalities. This library then permits to dynamically design diverse processing flows via programming code.

Keywords*: Radar Imagery, Sentinel-1 Satellite, Processing Flows, Automatic Classification, Deep Learning.*

Resumen

Las imágenes satélite pueden utilizarse para el entrenamiento de modelos de Aprendizaje Profundo que permitan clasificar los elementos que las componen. Para dicho aprendizaje y clasificación, es necesario preparar las imágenes mediante un correcto método de procesamiento.

Este trabajo se centra en las imágenes de tipo radar debido a sus ventajas. En concreto, profundiza en el estudio del satélite Sentinel-1 de la Agencia Espacial Europea y su aplicación junto con Sentinel-2 (satélite hiperespectral) para la clasificación automática de edificios y carreteras. Con ello se pretende, en última instancia, evaluar diferentes flujos de procesamiento para dichas imágenes radar y comparar los resultados obtenidos en su posterior segmentación.

Dado que el procesamiento de imágenes Sentinel-1 depende de herramientas software con interfaz gráfico, se desarrolla a su vez una librería en lenguaje de programación Python para albergar sus funcionalidades. De este modo, se hace posible trabajar únicamente desde la programación y se agiliza el diseño de las diversas vertientes del procesamiento.

Palabras clave: Imágenes radar, Satélite Sentinel-1, Flujos de Procesamiento, Clasificación Automática, Aprendizaje Profundo.

Índice

1. Introducción **7**
1.1. Motivación . . . 7
1.2. Estructura de la memoria . . . 8

2. Preliminares **9**
2.1. Programa *Copernicus* . . . 9
2.2. Sentinel-1 . . . 10
2.2.1. Datos Ráster . . . 12
2.3. Herramientas . . . 14
2.3.1. QGIS . . . 14
2.3.2. SNAP . . . 15

3. Alternativas **21**
3.1. OST . . . 21
3.2. Snappy . . . 22
3.3. Decisión . . . 23

4. Desarrollo de un *wrapper* de *snappy* **25**
4.1. Fundamentos de *snappy* . . . 25
4.2. Programación de una librería de *Python* . . . 28
4.2.1. Grafo . . . 28
4.2.1.1. Ejemplo de uso: creación de un grafo . . . 31
4.2.1.2. Ejemplo de uso: modificación de un grafo . . . 32
4.2.2. Helper . . . 34
4.3. Mejoras para una próxima versión . . . 35

5. Fase de experimentación **37**
5.1. Objetivo . . . 37
5.2. Diseño . . . 37
5.2.1. Flujos de procesamiento . . . 37
5.2.2. *Deep Learning* . . . 43
5.3. Generación del *dataset* . . . 43

5.3.1. *GeoFlow* 46

5.3.1.1. Flujo de tareas inicial 46

5.3.1.2. Adaptación del flujo de tareas 47

5.4. Análisis de resultados 48

5.4.1. Detalle sobre las métricas 48

5.4.2. Experimentación base 49

5.4.3. Comparación de nuevos resultados 51

5.4.4. Estudio de peores resultados 54

5.4.4.1. Palma 54

5.4.4.2. Cáceres 55

5.4.4.3. Teruel 56

5.4.4.4. Pamplona 57

6. Conclusiones y líneas futuras **59**

Índice de figuras **60**

Índice de tablas **62**

1. Introducción

1.1. Motivación

En la actualidad, existen múltiples fuentes de datos satélite. El fácil acceso, aunque no siempre gratuito, a la gran cantidad de imágenes que se generan diariamente de cualquier parte de la Tierra ha llevado a investigar los posibles usos de dichos datos en pos de obtener información relevante sobre, entre otros, los cambios producidos por el cambio climático o los desplazamientos de terreno tras un terremoto.

Otra posible razón por la que tratar imágenes satélite es para obtener información relacionada con la forma en que evolucionan los núcleos urbanos. El estudio del crecimiento demográfico y la forma de estructurar las poblaciones pueden suponer una base firme sobre la que basar nuevos proyectos de construcción, por ejemplo.

Este análisis de la configuración de una ciudad contempla como primer paso la tarea de detectar los diferentes estratos que la componen. Esto es, los tipos de construcciones, las carreteras y demás vías de circulación, las zonas sin identificar, así como los ríos, los lagos e incluso las costas del mar. Esta tarea que, a simple vista puede parecer sencilla, se antoja impensable si quiere llevarse a cabo sobre una gran cantidad de datos. Surge entonces la necesidad de automatizar de algún modo esta clasificación.

La automatización de este tipo de procesos es hoy en día posible gracias al Aprendizaje Profundo (*Deep Learning*), rama específica del campo del Aprendizaje Máquina (*Machine Learning*). En el aprendizaje profundo se utilizan modelos basados en redes neuronales para aprender a optimizar el proceso que se le imponga a partir de una serie de datos preparados para tal fin. Es decir, para el presente caso, optimizar la clasificación de los diferentes estratos urbanos mediante imágenes satélite procesadas de un modo específico.

En concreto, para este proyecto, se utilizan imágenes satélite de tipo radar. Esta clase de datos satélite son adquiridos mediante la emisión de señales y la medición de la energía reflejada tras su interacción con la superficie terrestre. La naturaleza de estas señales implica que los datos incluyen información sobre la distribución del terreno y que pueden ser tomados de noche, entre otras características. Para el objetivo que se persigue, esta forma de adquisición supone una mejor opción que otro tipo de satélites, como pueden ser los dedicados a evaluar la superficie, el color y la temperatura marítima, o a monitorizar la disposición de la atmósfera. Específicamente, las imágenes de tipo radar son utilizadas a la par que las obtenidas por un satélite multiespectral, el cual se dedica también a observar la superficie terrestre. Es decir, el satélite de tipo multiespectral es utilizado en combinación con el de tipo radar para obtener una información más completa de la superficie de la Tierra que la que consiguen ambos por separado.

La utilización de estos dos tipos de satélites fue ya contemplada en un proyecto previo [1], del cual se parte para desarrollar el trabajo actual. En concreto, se utilizan imágenes tomadas por los satélites Sentinel-1 (radar) y Sentinel-2 (multiespectral) [2] de la Agencia Espacial Europea (ESA).

Precisamente, la segmentación semántica de estas imágenes satélite es el objetivo principal de este proyecto. En concreto, se pretende comparar las distintas formas de tratar las imágenes Sentinel-1 en relación al beneficio que implican en la posterior clasificación de edificios y carreteras.

Para ello, primero se estudian los posibles procesamientos a aplicar sobre las imágenes radar. Después, se analizan las herramientas que permiten tanto diseñar como ejecutar dichos tratamientos. No habiendo una aplicación que cumpla los requisitos impuestos, se desarrolla una librería que permita diseñar y realizar el procesamiento de los datos desde código Python. Por último, se realizan una serie de experimentos con el fin de evaluar la mejora que inducen ciertos flujos de procesamiento de las imágenes radar en la segmentación de edificios y carreteras.

1.2. Estructura de la memoria

Esta memoria abarca una detallada explicación sobre los pormenores del proyecto, empezando por el Capítulo 2, en el que se asientan las bases del trabajo realizado. En él, se explican los fundamentos de las imágenes utilizadas, como son su origen, la forma de adquirirlos y la naturaleza de los datos que albergan. Asimismo, se presentan las herramientas más utilizadas para la visualización y procesamiento de los datos empleados.

A continuación, se presenta una comparación entre posibles formas de dar uso a las herramientas comentadas por medio de la programación. Específicamente, en el Capítulo 3 se detallan las características de las librerías tenidas en cuenta, así como los puntos relevantes de dicha comparación.

Debido al hecho de que no existe actualmente una herramienta tan completa como para cumplir todos los requerimientos establecidos, se relata en el Capítulo 4 el desarrollo de una nueva librería. Esta opción persigue cumplir todas las necesidades del procesamiento de datos del proyecto.

Una vez presentada la nueva librería, se expone la fase de experimentación en el Capítulo 5. En concreto, se presenta la obtención del conjunto de datos a utilizar en los experimentos, el diseño de estos últimos y la configuración de la red neuronal convolucional utilizada para la clasificación ya comentada. De igual manera, se presentan y analizan los resultados de los experimentos realizados y las posibles razones por las que existen errores en la segmentación.

Por último, en el Capítulo 6, se presentan las conclusiones al trabajo acompañadas de potenciales futuras líneas de investigación en relación al mismo.

2. Preliminares

Se desglosan a continuación ciertos aspectos clave para la comprensión del grueso de este trabajo.

Primero de todo, se explica el origen de las imágenes satélite utilizadas durante el proyecto. En concreto, en la Sección 2.1, se detallan las misiones *Setinel*, pertenecientes al programa *Copernicus* de la Agencia Espacial Europea (o ESA, en sus siglas en inglés). Posteriormente, se profundiza en la misión Sentinel-1 en la Sección 2.2. Dentro de la misma, se destaca el uso de los datos ráster para almacenar la información de las imágenes tomadas por esta familia de satélites. Por último, se presentan, en la Sección 2.3, las herramientas que han sido empleadas para la visualización y procesamiento de los datos: QGIS y SNAP.

2.1. Programa *Copernicus*

Las imágenes satélite utilizadas para este trabajo han sido tomadas de la base de datos de libre acceso u *Open Access Hub* [3] del programa *Copernicus*. Aunque este nombre en honor al astrónomo Nicolás Copérnico no fue establecido hasta 2012, el programa existía desde octubre de 1998 bajo la denominación GMES (*Global Monitoring for Environment and Security*) [4].

El programa *Copernicus* es coordinado y gestionado por la Comisión Europea, y ejecutado por la Agencia Espacial Europea, de la mano de los Estados miembros, la Organización Europea para la Explotación de Satélites Meteorológicos (Eumetsat), el Centro Europeo de Previsiones Meteorológicos a Medio Plazo, las agencias de la Unión Europea y la empresa Mercator Océan. Su objetivo principal es el de proporcionar información precisa, actualizada y accesible para la mejora de la gestión medioambiental, así como para comprender y mitigar los efectos del cambio climático. [5]

El citado programa cuenta con la familia de misiones *Sentinel*, cuya representación puede encontrarse en la Figura 1. Cada misión está basada en una constelación de dos satélites cuyo fin último es la obtención de datos robustos sobre la observación de la Tierra, desde los océanos hasta la propia atmósfera. Actualmente, se compone de siete misiones [2]:

- **Sentinel-1**: se trata de una pareja de satélites, *Sentinel-1A* y *Sentinel-1B*, que fueron puestos en órbita en abril de 2014 y en abril de 2016, respectivamente. Su cometido es tomar imágenes radar de la tierra y los océanos desde su emplazamiento en la órbita polar, sin importar las condiciones meteorológicas o si es de día o de noche.
- **Sentinel-2**: se trata de una misión en la órbita polar que se encarga de tomar imágenes multiespectrales de alta resolución para monitorización de la superficie terrestre. El fin último es el de mantener un inventario de imágenes de, por ejemplo, la vegetación presente. Su satélite *Sentinel-2A* inició la misión en junio de 2015 y su satélite *Sentinel2-B* lo hizo en marzo de 2017.
- **Sentinel-3**: su objetivo es medir de manera precisa y fiable la topografía de la superficie marítima, además de la temperatura y del color tanto terrestre como marítimo. Esta misión se conforma como la base de la información de los sistemas de prevención oceánica y la monitorización medioambiental y climática. En febrero de 2016 y abril de 2018 llegaron a órbita sus satélites *Sentinel-3A* y *sentinel-3B*.
- **Sentinel 5-Precursor**: también conocido como *Sentinel-5P*, se desarrolló para cubrir la falta de datos entre los obtenidos por el satélite *Envisat* y el satélite *Sentinel-5*. Fue puesto en órbita en octubre de 2017 y proporciona datos puntuales sobre diversas trazas de gases y aerosoles que afectan negativamente a la calidad del aire y el clima.
- **Setinel-4**: es un satélite dedicado a la monitorización de la atmósfera que será utilizada para un satélite MTG-S (*Meteosat Third Generation-Sounder*) en órbita geoestacionaria.

- **Sentinel-5**: es un satélite de tipo *MetOp Second Generation* que monitoriza la atmósfera desde la órbita polar.
- **Sentinel-6**: presenta un altímetro radar para medir la altura global de la superficie marítima, principalmente para estudios climáticos y oceanográficos.

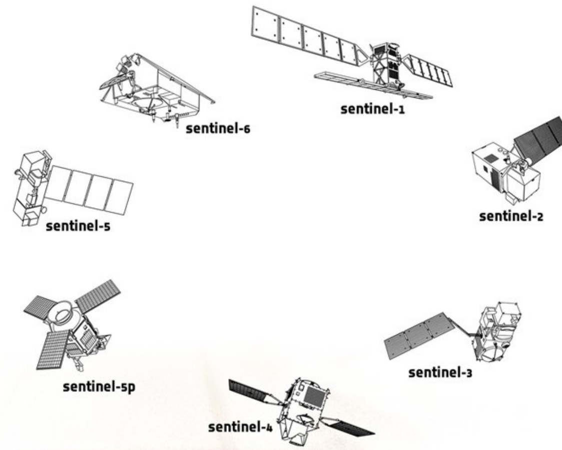


Figura 1: Representación de la familia de satélites Sentinel [2]

Para el desarrollo de este proyecto, se trabaja con imágenes de tipo radar obtenidas por la familia de satélites Sentinel-1: se procesan y utilizan para la segmentación de carreteras y edificios mediante modelos de inteligencia artificial.

2.2. Sentinel-1

Aunque ya se han explicado datos básicos sobre esta misión, cabe destacar ciertos aspectos más detallados a la par que significativos.

Sentinel-1 cubre por completo la extensión terrestre cada dos semanas; las zonas glaciares y las zonas de la costa europea y sus rutas náuticas de forma diaria; y la extensión oceánica de forma continuada mediante lo que se denomina *wave imagettes* (literalmente, miniaturas de las olas). Se espera que continúe proporcionando estos datos durante al menos 7 años (tras su puesta en órbita) [6].

Sentinel-1 cuenta con una instrumentación de una única banda C-SAR (*Synthetic Aperture Radar*). Se trata de una forma específica de recolección de datos activa: un sensor genera su propia energía y después anota la cantidad de energía que le llega reflejada tras interactuar esta con la Tierra. Este tipo de generación de señales implica una toma de datos que responde ante factores de la superficie como la distribución del terreno o la humedad presente. Además, posibilita la captura de imágenes sin afecciones por la presencia de nubes o la falta de iluminación por realizarse capturas durante la noche. [7]

La tecnología SAR permite la adquisición de datos mediante cuatro modos distintos de captura de imagen — *Stripmap* (SM), *Interferometric Wide Swath* (IW), *Extra Wide Swath* (EW) y *Wave Mode* (WV) — que se caracterizan por cierta resolución (por debajo de los 5 metros) y por cierta cobertura (hasta 400 kilómetros).

El modo IW, el cual permite combinar una amplia anchura de la región de toma de datos o *swath* (250 kilómetros) con una resolución geométrica más modesta (5x20 metros), se utiliza como único modo de captura en este proyecto. Esto se debe a que está globalmente extendido y, en concreto,

es el único utilizado para obtener datos en el territorio europeo; como queda plasmado en verde en la Figura 2.

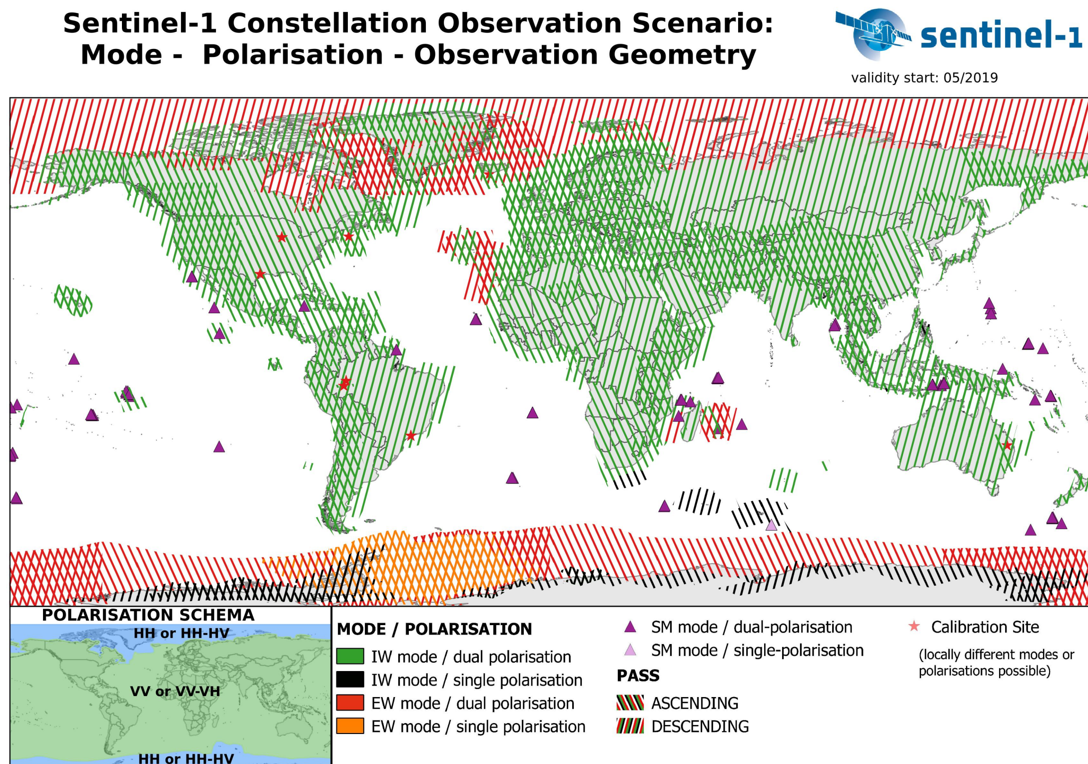


Figura 2: Distribución de la captura de datos y las polarizaciones de Sentinel-1 [8]

Las imágenes capturadas por el método IW se dividen en tres sub-regiones o *sub-swaths* debido a la utilización de la técnica de observación del terreno mediante escáneres SAR progresivos o TOPSAR (*Terrain Observation with Progressive Scans SAR*) [9]. Específicamente, se demoniman como IW1, IW2 e IW3. En la Figura 3 pueden contemplarse tanto los distintos modos de adquisición como los tres *sub-swaths* mencionados.

También gracias a la tecnología SAR pueden obtenerse imágenes según cierta polarización. Dicha polarización está presente en las señales radar que emite y recibe de vuelta cada satélite. Se denomina 'H' a la polarización horizontal y 'V' a la polarización vertical. Su combinación implica la obtención de señales de polarización simple (HH,VV) o de doble polarización (HH+HV, VV+VH). En ambos casos, la señal desde el satélite puede ser transmitida de forma horizontal o vertical. Sin embargo, su diferencia reside en la señal reflejada: en el caso de la simple, recibe una única cadena del mismo tipo que la que la origina; mientras que en el caso de la doble, recibe dos cadenas paralelas correspondientes a ambas polarizaciones [10]. Una posible representación de ello puede encontrarse en la Figura 34a.

Aunque diferentes polarizaciones pueden implicar la obtención de distintas propiedades físicas del terreno observado, durante el proyecto únicamente se utilizan datos tomados con polarización vertical, es decir, VV+VH, debido a que la polarización horizontal solo está disponible en las zonas polares, como podía verse anteriormente en la Figura 2.

Una vez conocidas en detalle las posibilidades y las ventajas que proporciona Sentinel-1 para la toma de datos, se ahonda en los tipos de datos que proporciona: los datos ráster.

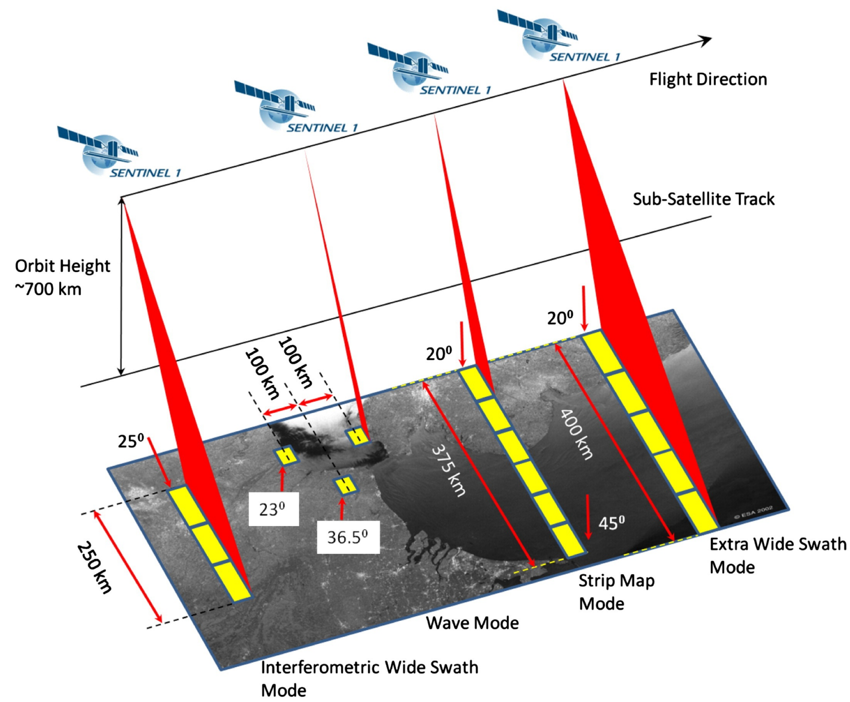


Figura 3: Modos de adquisición en Sentinel-1 [9]

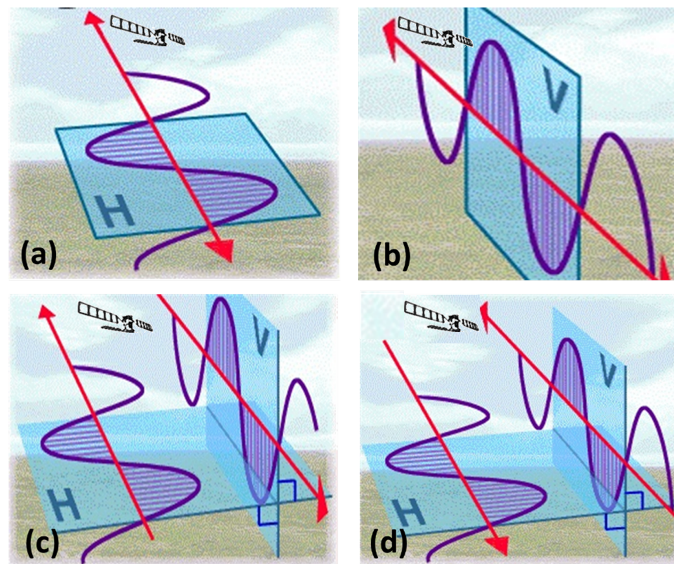


Figura 4: Representación de las distintas polarizaciones: (a)Polarización HH, (b)Polarización VV, (c)Polarización VH, (d)Polarización HV [11].

2.2.1. Datos Ráster

Los datos de tipo ráster se basan en una matriz de píxeles o celdas. Cada uno de estos píxeles representa una región geográfica y alberga un valor característico de dicha zona. Una posible representación sencilla de esta estructura se encuentra en la Figura 5. Cabe destacar que se trata de una imagen multibanda, es decir, que sus datos ráster se almacenan en varias bandas. Concretamente, al

número de bandas en una ráster se le llama resolución espectral. Dichas imágenes comparten analogía con, por ejemplo, las fotografías, ya que las bandas pueden combinarse entre sí para formar algo similar a una imagen RGB, de modo que se pueda visualizar mejor su contenido (Figura 6a); o, de tratarse de una imagen con una sola banda, podría adoptar la terminología de 'escala de grises' (Figura 6b) [12].

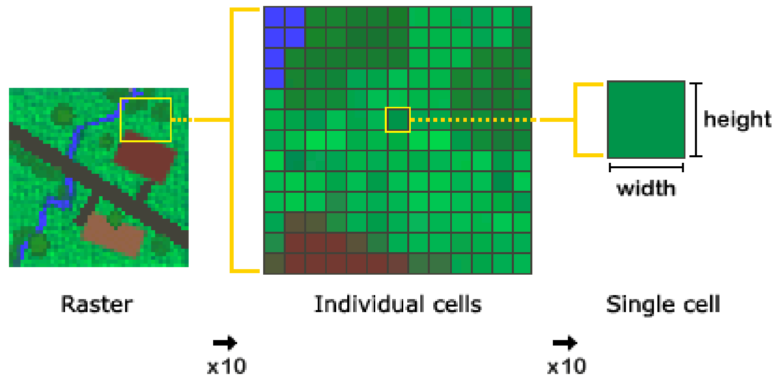
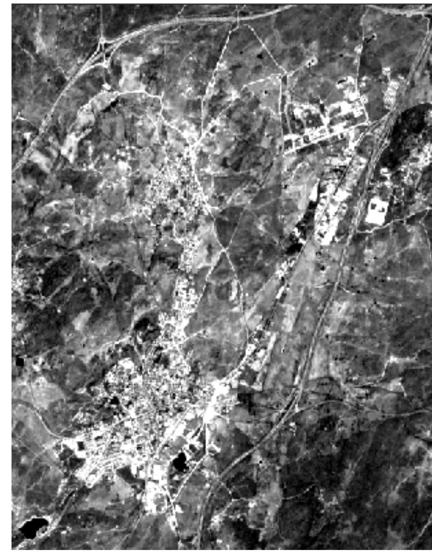


Figura 5: Ilustración de una matriz ráster [13].

(a) Superposición de varias bandas ráster (RGB)



(b) Una banda ráster (escala de grises)

Figura 6: Visualización de datos ráster.

El tamaño de los píxeles de una matriz (lo que en la Figura 5 se denomina como *cell* con unas dimensiones específicas), define la resolución espacial de los datos ráster. En concreto, se designa 'imagen de alta resolución' a aquella cuyo tamaño de píxel cubre un área reducida, ya que se aprecian numerosos detalles de la misma; mientras que se define 'imagen de baja resolución' a la que mantiene un gran área, puesto que la cantidad de detalle disminuye (Figura 7).

Las ya mencionadas bandas ráster deben estructurarse de cierto modo para que puedan visualizarse y procesarse. Para ello, se definen los llamados productos de datos. Dichos productos son proporcionados por los satélites antes comentados y se pueden diferenciar en tres niveles (de 0 a 2) según los tipos de datos que alberguen. En concreto, los productos de nivel 1 o *Level-1* son los utilizados para la experimentación presentada en posteriores secciones. Dicho nivel comprende tanto a los productos SLC (*Single Look Complex*) como a los GRD (*Ground Range Detected*) [15].

- **SLC**: se denomina a los productos que contienen datos SAR georeferenciados mediante el uso

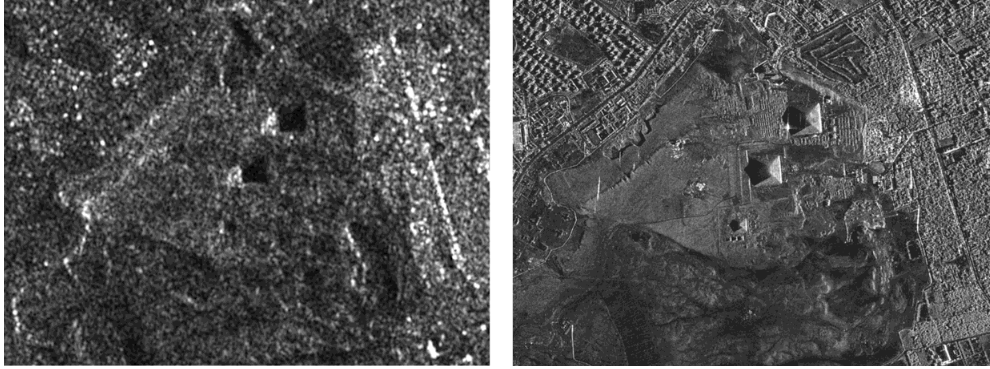


Figura 7: Comparación de la resolución espacial en imágenes SAR. (izq.) Imagen de baja resolución: píxel de 20x20 metros. (drcha.) Imagen de alta resolución: píxel de 1x1 metros [14].

de la información de situación del satélite, incluida la órbita, y proporcionados con geometría de rango 'zero-Doppler' (perpendicular al trayecto del satélite). Los datos son tomados tras una única observación por cada dimensión.

- **GRD**: hace referencia a los productos de datos SAR que han sido detectados y proyectados, tras varias observaciones, a un rango terrestre utilizando el modelo elipsoidal de la Tierra. El producto resultante posee una resolución espacial específica según sea de: *Full Resolution* (FR) o 'Resolución Completa', *High Resolution* (HR) o 'Alta Resolución' y *Medium Resolution* (MR) o 'Resolución Media'.

2.3. Herramientas

Con el fin de posibilitar la visualización y el procesamiento de los datos ráster en un ordenador, existen diversas herramientas SIG o 'Sistemas de Información Geográfica'. Entre ellas, se encuentran QGIS y SNAP, dos reconocidas aplicaciones con interfaz gráfica de usuario que han sido utilizadas de forma sostenida durante todo el proyecto.

2.3.1. QGIS

Entre los posibles programas para manejo de datos ráster, se trabaja con QGIS por dos razones principales: es gratuita y de código abierto (es decir, cuenta con una licencia *GNU General Public License*) y puede utilizarse en cualquier sistema operativo [16].

La herramienta se basa en ciertos servicios y librerías ya existentes, destacando entre ellas GDAL (*Geospatial Data Abstraction Library*), la cual permite trabajar con distintos formatos de datos geográficos. El formato comúnmente utilizado, también en este proyecto, es el formato GeoTIFF, un estándar de metadatos que permite que la información georreferenciada se almacene en una imagen de formato TIFF (*Tag Image File Format*).

QGIS proporciona una amplia diversidad de funcionalidades en relación con la creación, la edición, la gestión y la exportación de distintos tipos de datos para diferentes formatos, como puede apreciarse en el interfaz gráfico de la Figura 8. Además, permite revisar las propiedades de los productos que se estén tratando, ya sea su tamaño o su precisión numérica, entre otros (Figura 9).

Aunque el programa permite una gestión muy avanzada, se ha trabajado con él de forma básica. Fundamentalmente ha sido utilizado para visualizar tanto los productos originales obtenidos por Sentinel-1 como los datos resultantes de los procesamientos. También se ha usado para definir

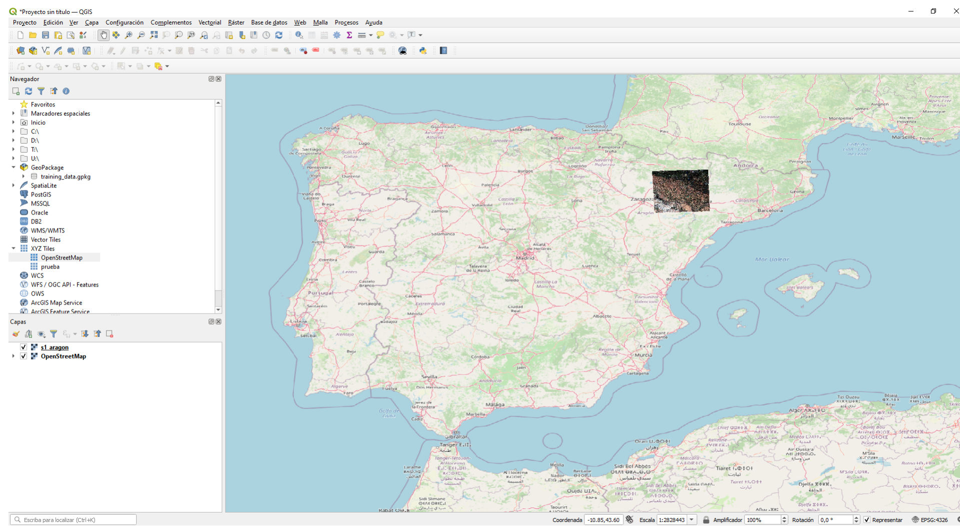


Figura 8: Interfaz gráfico de QGIS: visualización sobre mapa de un producto Sentinel-1 de una región de la provincia de Zaragoza.

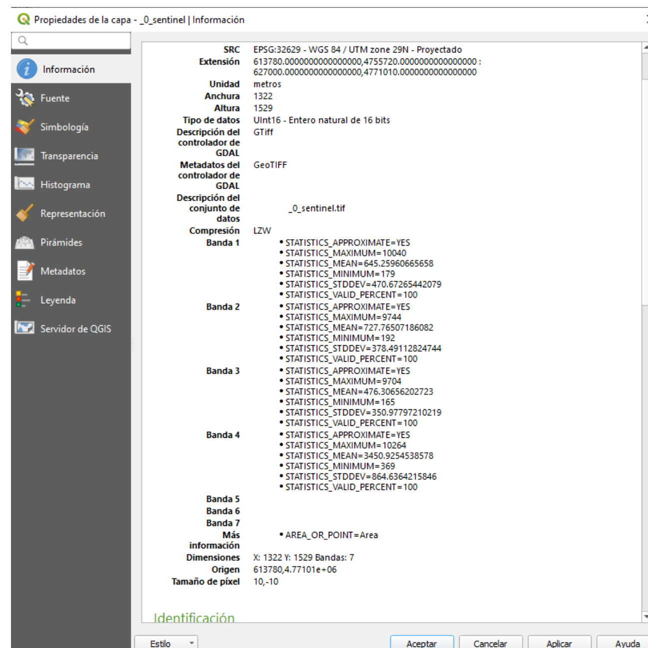


Figura 9: Cuadro de diálogo en QGIS con la información del producto en estudio

regiones de interés o *subsets* en las imágenes satélite en estudio, así como para comprobar que las regiones definidas por medio de código (en lugar de mediante el interfaz) se alineaban correctamente con los productos de interés, como puede verse en la Figura 10.

2.3.2. SNAP

Una de las herramientas más utilizadas para el procesamiento de datos ráster es SNAP (*Sentinel Application Platform*) [17], desarrollada por la ESA. Está basada en Java y utiliza librerías para tratar datos geospaciales como la ya nombrada GDAL. Esta aplicación alberga todas las funcionalida-

Figura 10: Ejemplo de comprobación de regiones para un producto concreto. En rosa, la representación de la superficie que ocupa la imagen satélite. En amarillo y en verde, dos posibles definiciones de *subsets* que cubran un cuarto de la superficie total.

des necesarias para trabajar con datos del programa Copernicus gracias a sus *toolboxes* (o paquetes de herramientas) para Sentinel 1, 2 y 3. Aunque pueda haber diferencias entre ellos, por las características concretas de los datos obtenidos por uno u otro tipo de satélite, mantiene una arquitectura común para una rápida navegación y visualización de las imágenes, y para el diseño de flujos de procesamiento mediante grafos: el denominado *Graph Processing Framework* (GPF). Además de por todas las posibilidades de trabajo que presenta, se utiliza ampliamente durante el proyecto por ser una herramienta de código abierto bajo la licencia *GNU GPL v3*, porque puede instalarse en cualquier sistema operativo [18] y por contar con una amplia comunidad de usuarios para mantener, actualizar y consultar cualquier cuestión sobre este *software* [19].

SNAP trabaja por medio de un interfaz gráfico de usuario (Figura 11). En él se pueden encontrar las diversas funcionalidades comentadas para visualización y tratamiento de los productos Sentinel. Durante el proyecto, se ha tendido a visualizar las imágenes resultantes de pruebas de procesamiento y a consultar ciertas de sus características, como pudieran ser sus dimensiones. También en relación a la forma de mostrar los datos, se han obtenido versiones RGB mediante la combinación de varias bandas ráster, de modo que destacasen ciertos detalles de interés. La división de la ventana en varias celdas para mostar y comparar diferentes imágenes al mismo tiempo ha servido de gran ayuda para apreciar cambios en los datos a simple vista.

Por medio de la búsqueda de operaciones en la barra superior de herramientas, se pueden ir procesando las bandas ráster paso a paso. Las imágenes ya tratadas pueden visualizarse en pantalla abriendo los productos que se vayan generando. Sin embargo, trabajar de esta forma intermitente podría ser tedioso, sobre todo si se están tratando una gran cantidad de imágenes.

En cuanto al tratamiento de datos, que supone la parte más interesante e importante, el número de posibles operaciones es extenso y sus aplicaciones diversas. Las más utilizadas durante el proyecto han sido *Read*, *Write*, *Apply Orbit File*, *Calibration*, *Band Maths*, *Band Merge*, *Terrain Correction*, *Thermal Noise Removal* y *Speckle Filter*. Su definición y su utilidad en el procesamiento serán comentadas en posteriores secciones, cuando se expliquen en detalle los flujos ejecutados.

Por la ya expresada necesidad de ejecutar flujos completos y no por partes, una de las posibilidades a la que más uso se ha dado durante el proyecto, ha sido al antes mencionado **GPF**. Esta herramienta interna de SNAP permite, también por medio del interfaz, trabajar con grafos de proce-

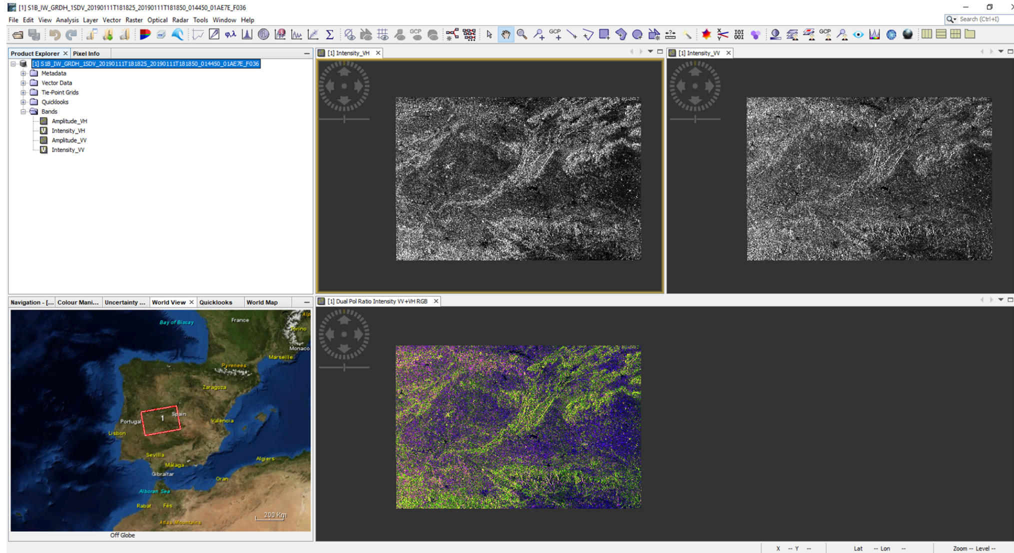
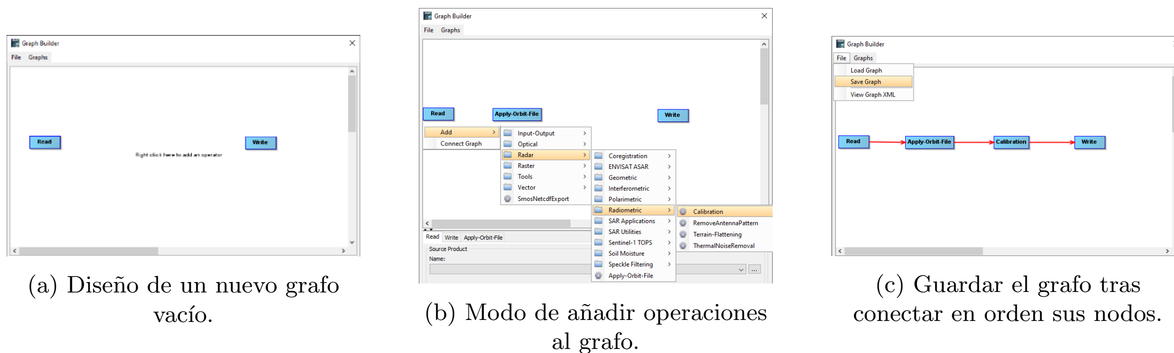


Figura 11: Interfaz gráfico de SNAP: visualización de un producto Sentinel-1 mediante dos de sus bandas ráster y la combinación de las mismas en formato RGB.

samiento, como se muestra en la Figura 12. Estos grafos deben ser dirigidos y acíclicos. Como pauta, siempre comienzan con una operación *Read*, es decir, de lectura, y finalizan con una operación *Write*, esto es, de escritura (Figura 12a). Ello supone que el grafo debe tomar los datos de un producto concreto que quiera tratarse y, tras una serie de operaciones, debe escribir el resultado en un nuevo producto con el formato especificado. Entre ambas operaciones pueden encontrarse múltiples otras, las cuales pueden añadirse de forma sencilla, como se muestra en la Figura 12b. La forma en que se conectan los nodos del grafo indican el orden de ejecución de sus operaciones. De hecho, una vez realizadas las conexiones, se considera finalizado y puede guardarse en el equipo en uso (Figura 12c).

La forma de almacenar los grafos de procesamiento es por medio de ficheros XML. De este modo, en el interfaz de GPF (o herramienta *Graph Builder*) se pueden cargar tanto grafos por defecto como grafos anteriormente diseñados, crear grafos vacíos para desarrollarlos desde cero, realizar cualquier cambio que se desee y guardarlos de nuevo en memoria. Esta variedad de formas de construir y modificar los grafos permite que se creen no solo grafos con pocas operaciones como el anteriormente sugerido, sino otros mucho más complejos. Esto es, es posible diseñar grafos que, por ejemplo, lean distintos productos mediante varias operaciones *Read*, realicen diferentes procesamientos, combinen en algún punto sus datos y escriban mediante *Write* el resultado final.



(a) Diseño de un nuevo grafo vacío.

(b) Modo de añadir operaciones al grafo.

(c) Guardar el grafo tras conectar en orden sus nodos.

Figura 12: Cuadro de diálogo para la creación de grafos de procesamiento.

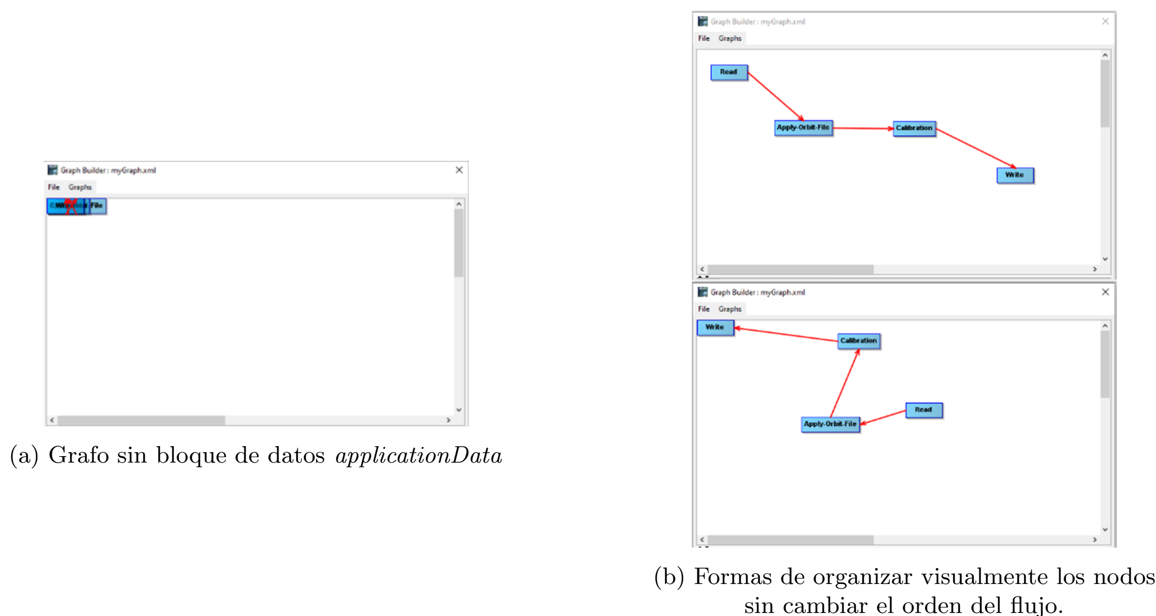
Puesto que los grafos se basan en **código XML**, bien podrían construirse desde un editor

de texto cualquiera (el interfaz permite ver el código pero no editarlo, como muestra la Figura 13), pero ello conllevaría un modo de trabajo poco eficaz y altamente costoso en tiempo, por el hecho de tener que escribir cada uno de los nodos con sus respectivas propiedades. La mencionada figura muestra el código XML que define el grafo anteriormente diseñado, de modo que puede apreciarse el formato que siguen los nodos que lo conforman. De igual manera, se puede visualizar con más detalle la forma de definir un nodo concreto, en este caso, para la operación *Apply-Orbit-File* (obtener los datos específicos de la órbita en la que se encuentra el satélite al tomar la información del producto en tratamiento). Cabe subrayar que, cuando se define un grafo mediante el interfaz de SNAP, éste añade ciertos datos relacionados con la forma de representar el grafo de forma visual: *applicationData*. Dichos datos incluyen la posición a ocupar en la ventana del interfaz por medio de coordenadas (x,y). Resultan irrelevantes para la ejecución del flujo de procesamiento, pero son convenientes si se trabaja mediante el interfaz de usuario, ya que permite ver de forma cómoda los nodos y sus conexiones. En caso de definir un grafo directamente en XML, sin incluir estos datos de posición, y de importarlo posteriormente en la herramienta *Graph Builder*, se mostrarán los nodos superpuestos en el origen de coordenadas (Figura 14a), siendo necesario desplazarlos uno por uno con el ratón para que pueda apreciarse el flujo completo (Figura 14b).

Figura 13: Opción de código en la herramienta *Graph Builder*

SNAP también puede ejecutarse por medio de la consola de comandos. Esto es posible gracias a su núcleo ejecutable **GPT** o *Graph Processing Tool*, a través del cual se pueden llevar a cabo procesamientos completos basados en un grafo previamente definido y guardado, o la ejecución de operaciones individuales. Asimismo, se puede consultar información básica sobre el comando con un simple `gpt -h`, el cual muestra una breve descripción de la herramienta y sus opciones a la hora de ejecutarlo, como la forma de indicar el nombre del archivo de salida (`-t <archivo>`) o el formato de dicho archivo (`-f <formato>`). También imprime una lista de todas las operaciones presentes en SNAP, que pueden conocerse más en detalle mediante `gpt <operacion> -h`.

Si se trabaja directamente sobre código XML, esta funcionalidad sirve de gran ayuda para conocer la estructura que debe seguir un nodo concreto así como las posibilidades y significados de sus parámetros, incluyendo el formato específico para indicarlos y los limitados valores que aceptan.



(a) Grafo sin bloque de datos *applicationData*

(b) Formas de organizar visualmente los nodos sin cambiar el orden del flujo.

Figura 14: Variantes de un mismo grafo en el *Graph Builder*

Por supuesto, a la hora de ejecutar operaciones individuales, esta consulta de información también es clave, ya que los parámetros concretos a utilizar deberán indicarse en un único comando por medio de opciones similares a, por ejemplo, `-Ssource` (esta opción establece los productos de entrada).

Una explicación general del comando puede resultar algo confusa, por lo que se muestra la forma de proceder en un caso concreto: ejecutar una única operación *Apply-Orbit-File* por línea de comandos. Puede realizarse de dos formas distintas: A) Mediante la ejecución de un grafo llamado 'grafo.xml' con tres nodos (*Read*, *Apply-Orbit-File* y *Write*); y B) Mediante la ejecución de *Apply-Orbit-File* como operación individual. En ambos casos deben especificarse ciertos parámetros y los valores que se desean emplear. En este ejemplo, se considera que dentro del grafo ya se han especificado los parámetros necesarios; mientras que, para la operación individual, deben indicarse explícitamente en el comando. Para ello, se establece que 'producto_entrada' y 'producto_salida' hacen referencia a los productos que lee y escribe el programa, respectivamente. Además, se indica solo un parámetro: '*continueOnFail*' con valor '*false*'; al no ser definidos expresamente, el resto de parámetros tomarán valores por defecto. El formato de salida se indica como '*GeoTIFF*', por ser de uso común. Siguiendo esas pautas, se definen los comandos tal que:

```
A)  gpt grafo.xml
B)  gpt Apply-Orbit-File -PcontinueOnFail='false' -t producto_salida
                          -f 'GeoTIFF' producto_entrada
```

Cabe destacar que a la hora de diseñar un grafo en XML pueden indicarse ciertos parámetros por medio de variables (con el formato ${nombre_parámetro}$) a los que dar un valor concreto desde línea de comandos dependiendo del caso en tratamiento. Por ejemplo, en la situación anterior, el valor del parámetro en el grafo podría haber quedado definido como ${continueOnFail}$ y su valor hubiese tenido que especificarse tal que: `gpt grafo.xml -PcontinueOnFail='false'`. Esta forma de definir los grafos es útil para poder reutilizarlos sin necesidad de editar el XML cada vez.

Aunque la ejecución por línea de comandos es conveniente a la hora de prescindir del interfaz gráfico, es cierto que no permite dejar de utilizarlo por completo. Esto se debe a que, como ya se ha indicado antes, ejecutar un procesamiento complejo por medio de operaciones individuales no es la mejor forma de trabajar. No solo porque se deban tener en cuenta muchos parámetros y cada operación suponga definir un comando diferente, lo cual bien podría resumirse en un *script*, sino porque por cada

resultado a una operación se generaría un nuevo producto, aumentando así la escritura de datos intermedios no necesarios y, en definitiva, invirtiendo más tiempo del necesario en ejecutar el flujo completo.

Por otro lado, la ejecución mediante grafos parece bastante conveniente por línea de comandos, pero queda limitada precisamente por los grafos: deben estar ya definidos y guardados en versión XML. Es decir, se ha tenido que utilizar previamente el interfaz de SNAP para construir dichos grafos, ya que por medio de GPT no es posible ni concatenar operaciones ni mucho menos diseñar grafos completos.

Dado que el proyecto que se está desarrollando utiliza Python para la segmentación de las imágenes satélite ya tratadas, no resulta eficiente depender de un interfaz gráfico externo que produce archivos XML. En consecuencia, se considera necesario encontrar una herramienta que permita trabajar siempre desde código Python para la creación, modificación y ejecución de los grafos de una manera más eficiente y dinámica. Con este fin, se realiza una labor de investigación acerca de las aplicaciones disponibles en el momento que permitan trabajar de ese modo.

3. Alternativas

Con el objetivo de encontrar una herramienta que permita dejar de depender del interfaz gráfico de SNAP, se realizó una búsqueda exhaustiva de posibles aplicaciones. Se hallaron dos posibilidades bastante prometedoras para el uso de funcionalidades desde librerías de código *Python*: *Open SAR Toolkit* [20] y *SNAP-Python* [21].

3.1. OST

Open SAR Toolkit u OST engloba herramientas de procesamiento de imágenes SAR en una librería de código abierto. Su objetivo principal es reducir el nivel de dificultad que puede suponer el acceso y procesamiento de datos Sentinel-1 a usuarios con poco conocimiento de SAR y de Python.

Este *software* cuenta con un repositorio de código actualizado en GitHub [20] y otro repositorio con *notebooks* en forma de tutoriales para explicar cómo se utiliza [22]. Aunque en parte sirvieron de ayuda, se encuentran desfasados en muchos aspectos con la versión disponible de la librería, por lo que el aprendizaje tuvo que recaer en leer el código fuente. No solo ciertas funcionalidades cambiaban, sino que la definición de las funciones o los nombres de los parámetros variaban ampliamente.

Esta herramienta se basa en diversas librerías de tratamiento de datos geoespaciales como GDAL, en el paquete de funcionalidades para Sentinel-1 de SNAP (*Sentinel-1 Toolbox*) y en *Orfeo Toolbox* [23], un programa de código abierto para el procesamiento de imágenes SAR. Esta combinación permite usos concretos como los que se detallan a continuación.

En primer lugar, OST posibilita una buena gestión de productos Sentinel-1. Para ello, puede realizar consultas hasta en cuatro servidores de datos, según se le indique (siempre y cuando se tenga una cuenta válida): Alaska Satellite Facility [24], de la NASA; Copernicus SentinelHub, de la ESA; PEPS [25], del CNES [26]; y ONDA DIAS [27]. Estos dos últimos pertenecen a las agencias francesa e italiana, respectivamente, y forman parte a su vez del programa Copernicus.

Las consultas a cualquiera de estos servidores siguen el mismo formato. Una posibilidad es indicar el identificador de un producto, de modo que se conozcan todos los detalles sobre el mismo, como la fecha de adquisición o el tipo de producto. De igual manera, pueden consultarse todos los productos disponibles que cumplan ciertos criterios: es suficiente con especificar un diccionario de parámetros para realizar la búsqueda acotada. Los productos encontrados de esta forma pueden mostrarse por pantalla mediante un mapa centrado en la región de interés, como puede apreciarse en la Figura 15.

Siguiendo esta línea de muestra de datos, permite tanto la creación como la visualización de una imagen RGB resultado de la combinación de bandas, por medio de una simple llamada en cada caso. De estar trabajando desde un *notebook*, se muestra una pequeña ventana con la imagen obtenida como si de un gráfico se tratase (Figura 16). Aunque la calidad de la imagen que muestra esta función deja mucho que desear, permite al menos hacerse a la idea de si se trata de una zona urbana o de si alberga montañas o lagos, entre otros accidentes geográficos.

En cuanto al procesamiento de los datos, permite aplicar tareas incluidas en el *Sentinel-1 Toolbox* pero, en lugar de especificar las operaciones y sus parámetros de forma similar a los comandos, unifica todo el procesamiento en un diccionario de *Python*. Esto es, las claves del diccionario son o bien nombres de operaciones, o bien parámetros globales a todo el flujo. En el caso de los parámetros, si se quiere indicar que se utilicen, por ejemplo, las polarizaciones verticales, debe indicarse una entrada al diccionario tal que: `'polarisation': 'VV, VH'`. De tratarse de una operación, la clave es su nombre y su valor debe ser otro diccionario de parámetros. Es decir, si se quiere ejecutar una operación de filtrado *Speckle-Filter*, con tan solo dos parámetros con valor específico y el resto con valores por defecto, deberá indicarse como: `'speckle_filter': {'filter': 'Refined Lee','window_size': '5x5'}` [22].

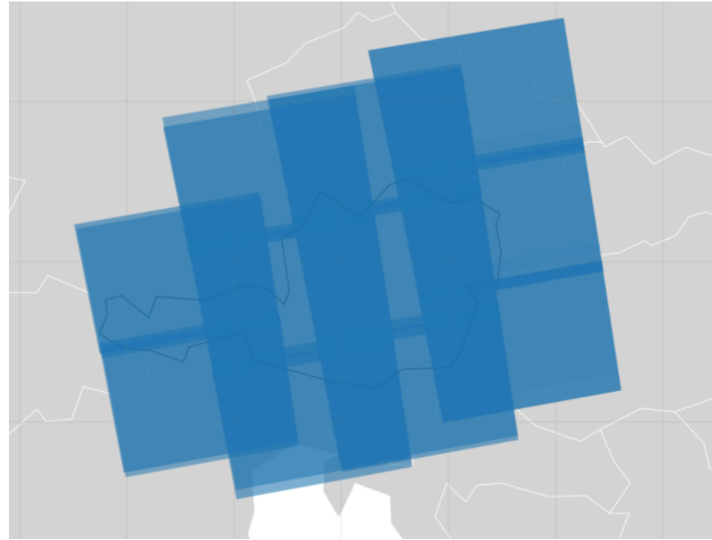


Figura 15: Muestra de productos sobre la superficie de Austria

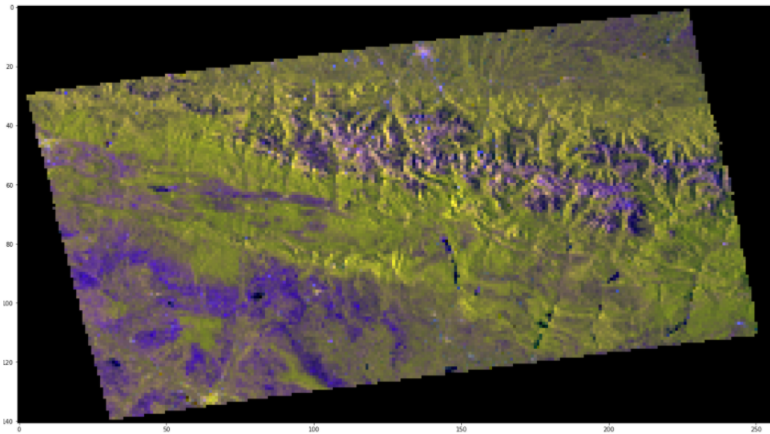


Figura 16: Visualización de una imagen RGB (calidad igual a la del *notebook*)

Cabe destacar que esta forma de trabajar permite cambiar los detalles del flujo de forma rápida y sencilla, pero éste no puede almacenarse posteriormente como flujo compatible con SNAP. Es decir, no puede guardarse como grafo XML ni como comandos GPT. Ello conlleva que, de necesitar realizar alguna comprobación en el interfaz o por línea de comandos, debería repetirse el diseño de los flujos. De hecho, la librería no es muy intuitiva en cuanto a pensar en estas dos vertientes, ya que las claves del diccionario son versiones adaptadas de los nombres reales, como ha podido verse con el ejemplo de *Speckle-Filter* con clave 'speckle_filter' (no todas son tan similares).

3.2. Snappy

SNAP-Python o *snappy* [21] surge por la necesidad de poder trabajar en SNAP desde Python. Como ya se ha comentado en la sección dedicada a SNAP, esta herramienta está basada en Java, por lo que, hasta el momento, solo podía accederse mediante una API de Java. Ahora es posible acceder a dicha API a través de Python gracias a *snappy*, siempre y cuando se tenga instalada la herramienta SNAP.

Este módulo permite dejar de depender del interfaz y trabajar desde código Python pero implica otra serie de requerimientos [28]. El aprendizaje del modo de uso de *snappy*, al menos superficialmente, puede apoyarse en ciertos tutoriales [29] y ejemplos [30] dentro de la propia librería. Durante el desarrollo de diversas pruebas se constata que aporta ciertas facilidades mediante puro código: lectura y escritura de productos, obtención y edición de sus propiedades básicas mediante funciones *get-set*, ejecución de operaciones individuales mediante el procesador de grafos o GPF, configuración de sus respectivos parámetros mediante HashMaps de Java (estructuras clave-valor) y visualización de imágenes y bandas por medio de librerías comunes de Python como *MatPlotLib* [31]. También pueden ejecutarse flujos de procesamiento completos. De hecho, existen dos formas distintas de proceder:

1. Encadenar diferentes operaciones individuales. Para ello, el argumento de salida de una operación debe ser el argumento de entrada de la siguiente. Por supuesto, el producto ha tenido que ser leído inicialmente desde un archivo guardado y el resultado de todo el proceso será almacenado en memoria solo cuando se indique una operación de escritura.

2. Importar mediante el submódulo *jpy* una serie de objetos Java que pertenecen al núcleo de SNAP. Esta forma de trabajar es todavía más cercana a la forma de proceder en el interfaz gráfico, ya que permite leer grafos almacenados como XML y ejecutarlos. Así mismo es posible diseñar nuevos flujos mediante operaciones y parámetros, y guardarlos en memoria como archivos XML. Aunque resulte una descripción generalizada, los referenciados objetos Java serán explicados en detalle en posteriores secciones, debido a la complejidad que pueden implicar.

3.3. Decisión

Ambas opciones estudiadas permiten trabajar desde código Python. En el caso de OST todas sus funcionalidades recaen sobre este lenguaje, pero en el caso de *snappy* también se deben tener conocimientos sobre Java. Por ejemplo, para saber utilizar un HashMap, puesto que siendo similar a un diccionario de Python, presenta otra forma de añadir y editar los valores.

En cuanto a la visualización, aunque se ha utilizado durante el proyecto la herramienta QGIS, las opciones en ambos casos son similares, ya que implican imprimir una imagen por pantalla de forma básica y únicamente en el caso de trabajar desde *notebooks*. Sí es cierto que OST presenta la novedad de visualizar la superficie que ocupan los productos en la región de búsqueda.

En relación a las dependencias que presentan estas librerías, *snappy* solo precisa de SNAP. Esto le otorga todo el potencial que posee el interfaz. Por el contrario, OST se basa en varias aplicaciones que trabajan conjuntamente para aportarle las funcionalidades ya explicadas. Y, cabe destacar, que no se basa en SNAP por completo, sino únicamente en el *toolbox* para Sentinel-1, por lo que todo lo que se desarrolle con esta librería no podrá extrapolarse al trabajo con productos de otras misiones.

En cuanto a la forma de ejecutar, ambas opciones presentan posibilidades respecto a las operaciones individuales pero no es así en el caso de los grafos. La forma de diseñar mediante diccionarios de OST puede ser confusa si albergan complejidad en los nodos y su orden, y los flujos generados no pueden guardarse como XML. Esta situación implica que no se puede revisar trabajo desde la interfaz o la línea de comandos, de ser necesario. En cambio, *snappy* sí aporta una forma de trabajar intuitiva respecto a los grafos, permitiendo tomar grafos ya diseñados, editarlos, crear nuevos y guardarlos.

Como punto también importante a la hora de decidir una herramienta, se sopesa la documentación relevante encontrada y, si bien es cierto que ambas opciones cuentan con repositorios y tutoriales, la falta de actualización en la parte de OST implicaría un aprendizaje mucho más lento y costoso que en el caso de *snappy*, aunque esta parezca una librería más compleja.

Las características clave comparadas anteriormente, bien pueden ser resumidas como muestra la Tabla 1.

Funcionalidad	***Open SAR Toolkit***	***Snap-Python***
Independiente del interfaz de SNAP	✓	✓
Uso exclusivo de código Python	✓	✗
Visualización de imágenes	✓	✓
Independiente de otras librerías (exceptuando SNAP)	✗	✓
Ejecución de operaciones individuales	✓	✓
Gestión de grafos de procesamiento	✗	✓
Válido para diferentes tipos de productos	✗	✓
Documentación relevante y actualizada	✗	✓

Tabla 1: Comparativa de funcionalidades: OST vs snappy

Tras sopesar todas ellas, puede considerarse que *snappy* queda más cercana a cumplir la pauta de prescindir del interfaz, ya que mantiene todas las funcionalidades del mismo pero desde código. Aún así, no supone una elección del todo válida, debido a que se trata de una API compleja. No solo depende de Java en los HashMap sino que también, para el uso de grafos, debe conocerse cada uno de los objetos que lo componen y cómo deben interactuar entre ellos. Dado que normalmente se trabaja en Python, sería mucho más adecuado abstraerse del núcleo de SNAP, de modo que ni se necesiten conocimientos de un segundo lenguaje de programación ni se deba invertir demasiado tiempo en aprender a utilizar la herramienta.

Por todo ello, se plantea una solución más acorde a las necesidades expresadas: programar un *wrapper* para *snappy*. Esto implica crear una librería en lenguaje Python que permita trabajar con grafos gracias a todo el potencial que presenta *snappy* solo que de forma más intuitiva e independiente de Java.

4. Desarrollo de un *wrapper* de *snappy*

Con el objetivo de hacer uso de todo el potencial de SNAP de una forma sencilla e intuitiva desde código Python, se desarrolló un *wrapper* de *snappy*. Para ello, se profundizó en el estudio del funcionamiento interno de SNAP en Java y la manera en que lo integra *snappy*. A su vez, se identificaron las funcionalidades necesarias que debía albergar el *wrapper* y el modo de adaptarlas al lenguaje Python.

Con el fin de presentar los pasos tomados para el buen desarrollo de dicho *wrapper*, se estructura la explicación del siguiente modo: en primer lugar, en la Sección 4.1 se detallan los pilares sobre los que se asienta *snappy*; en la Sección 4.2, se detalla cómo está conformada la nueva librería o *wrapper*; y, por último, en la Sección 4.3, se indican brevemente algunas pautas para seguir desarrollando y mejorando la librería actual.

4.1. Fundamentos de *snappy*

Como ya se ha indicado, el primer paso relevante fue estudiar a fondo *snappy*. A grandes rasgos, esta API se basa en el GPT y el GPF de SNAP y en ciertas clases Java. Su uso permite realizar las mismas operaciones que hay disponibles mediante línea de comandos o mediante el interfaz de SNAP.

Lo primero a destacar del uso de *snappy* es que permite trabajar de dos formas distintas:

1. Mediante una serie de clases a importar desde *snappy* que albergan la funcionalidad más importante, como son *ProductIO* para la gestión de los productos y *GPF* para ejecutar los procesamientos.
2. Mediante el uso de *jpy* y todas las clases que permite importar, las cuales son de más bajo nivel.

Ambas formas de utilizar *snappy* utilizan a su vez la clase *HashMap*, basada también en Java. Para dar una idea más detallada de qué finalidad tienen cada una de las clases mencionadas, se explican a continuación:

- **ProductIO** [32]: su nombre es la abrevación de *product input/output*, es decir, producto de entrada y de salida. Este tipo de clase tiene como fin albergar los datos de un producto. Dicho producto será considerado de entrada cuando sus datos provengan de un producto ya existente en memoria. En cambio, será considerado de salida cuando el producto ya procesado pase a guardarse en un archivo nuevo.

  Ambos pasos, tanto el de escritura como el de lectura, se basan en una simple llamada a funciones propias de la clase, denominadas *readProduct* y *writeProduct*, que toman como argumento las rutas de los productos, entre otros.

  Si se requiere realizar alguna consulta de información acerca del producto en tratamiento, como puede ser el tipo de formato o las características de las diversas bandas ráster que lo conforman, se pueden emplear funciones básicas de tipo *get-set*.
- **HashMap** [33]: se basa en la estructura de mismo nombre de Java [33], la cual es similar a un diccionario de Python. Sigue el mismo patrón de clave-valor, solo que utilizando funciones distintas al tratarse de lenguajes de programación diferentes.
- **GPF** [34]: se trata del núcleo de SNAP. Es imprescindible para el procesamiento de grafos. Su función más importante es *createProduct*.

  Esta función se utiliza, como su propio nombre indica, para crear productos. En concreto, se usa en caso de ejecutar operaciones de procesamiento individuales. Basta con indicar como parámetros de entrada al nombre de la operación, al *HashMap* con los parámetros concretos y a la

instancia *ProductIO* sobre la que actuar. Devuelve otra instancia de tipo *ProductIO* con los datos modificados.

- **jpy** [35]: constituye una vía bidireccional para insertar código Java en programas de Python y viceversa. Dentro de *snappy* permite tomar todos los tipos de clases necesarias para una buena gestión de grafos de procesamiento. Es decir, a través de él pueden importarse las clases que componen un grafo, como la propia *Graph* o la clase *Node* (representa cada nodo del grafo).

Tras un estudio preliminar sobre los usos de las anteriores clases, se percibió una diferencia destacable en cuanto al diseño de grafos. En el primer caso, el procesamiento solo puede realizarse paso a paso, es decir, solo puede definirse una operación individual en cada momento. Aunque sí es posible completar un flujo de procesamiento jugando con los productos de entrada y de salida de cada operación, no hay ningún modo de guardar este flujo en un archivo XML. En contraposición, en el caso de las clases propias de *jpy*, sí es posible crear flujos completos y guardarlos adecuadamente.

Dada esta situación y puesto que el objetivo principal del *wrapper* es gestionar los grafos de forma análoga al *GraphBuilder* del interfaz, se pone el foco en la segunda opción descrita y se buscan todas las clases a incluir en la librería.

Lo primero es entender el modo de importar las clases desde *jpy*. Sencillamente se utiliza una función denominada *'get_type'* a la que se le indica una ruta hasta la clase. Esta "ruta" está formada por el árbol de paquetes que lo almacenan. Es decir, en el caso de la clase *Node*, debe indicarse como '$jpy.get_type('org.esa.snap.core.gpf.graph.Node')$'.

Una vez resuelta la importación, se presentan las clases necesarias para la gestión de grafos a través de su definición y características principales:

- **GraphIO** [36]: se trata de un grafo de entrada/salida, es decir, almacena un grafo en relación a su lectura o a su escritura. Se denomina de entrada cuando se lee un grafo desde archivo XML, y de salida cuando se va a guardar en memoria. De hecho, estas dos posibilidades son las únicas acciones que puede llevar a cabo esta clase, mediante sus métodos *read* y *write*. El primero de ellos genera un *Graph* al deserializar un archivo XML y el segundo serializa un *Graph* a un fichero XML. Es importante subrayar, por tanto, que este tipo de grafo solo contiene la información relacionada a XML y su funcionalidad solo abarca la interpretación de su estructura característica.

  Es necesario concretar que esta clase no es capaz de leer o de escribir directamente desde un XML, esto es, no toma el nombre o ruta de un fichero como argumento de entrada. En cambio, trabaja con una instancia de *FileReader* o *FileWriter*, cuyas clases serán explicadas a continuación, y que sí permiten el uso de rutas.

- **Graph** [37]: tiene como objetivo albergar el grafo en sí. Esto es, para empezar a diseñar un grafo, el primer paso es instanciar un *Graph*. Este mantiene toda la información crucial para dicho diseño, desde la configuración de los nodos hasta el orden del flujo. Las funciones más relevantes que presenta son las básicas *add*, para incluir un nuevo nodo, y *get*, para consultar información sobre cualquier nodo.

- **GraphProcessor** [38]: se trata, literalmente, de un procesador de grafos. Se ocupa de preparar la ejecución desde el núcleo GPT de SNAP. Específicamente, para comenzar el procesamiento, solo hace falta llamar a la función *executeGraph* indicando la instancia *Graph* sobre la que se está trabajando.

- **Node** [39]: representa un nodo del grafo. Al generar un nodo nuevo, queda identificado mediante un nombre único y se establece a su vez la operación que representa. Es importante saber que así como el nombre puede definirse como se desee, la operación siempre debe indicarse como aparece en la documentación de SNAP. Por ejemplo, si se incluye un nodo de lectura, su nombre

podrá ser simplemente 'Read' o 'read_producto' o cualquier otro, pero su operación solo podrá ser 'Read', sin ninguna variante en su grafía.

El hecho de que el nombre sea único e identificativo conlleva que distintos nodos no pueden tener el mismo nombre. Siguiendo el ejemplo, si al primer nodo se le ha denominado 'Read', un segundo nodo de esa misma operación deberá llamarse 'otro_read' o cualquier nombre que se desee. Incluso podría seguirse la pauta por defecto del interfaz de SNAP y denominarlo 'Read (2)'. Aunque los nombres sean distintos, hay que recordar que la operación de ambos nodos debe ser estrictamente 'Read'.

Además de nombrarlos, los nodos pueden ser configurados. Pueden añadirse los parámetros correspondientes mediante la función *setConfiguration*, la cual toma como argumento de entrada una instancia de tipo *XppDomElement* (explicado a continuación).

- **NodeSource** [40]: representa "la fuente del nodo", esto es, el nombre del nodo cuya salida se toma como entrada del nodo en tratamiento.

  A la hora de configurar un nuevo nodo, puede indicarse que sus datos de entrada sean los datos resultantes de otro paso del procesamiento. Por ejemplo, si tras una operación de lectura quiere realizarse una de calibración, al configurar el nodo 'Calibration' se indicará que su *NodeSource* es 'Read'.

  Hay que tener en cuenta que un nodo de operación 'Read' siempre debe ser el primero de cualquier grafo, por lo que no admite *NodeSource*; mientras que, para el resto de operaciones, debe indicarse siempre al menos una fuente por nodo para que el flujo sea coherente.

  En caso de que algún nodo no esté conectado a otro de este modo, aparecerá una excepción indicando esta ausencia. Para evitar que se produzcan este tipo de errores cuando se esté diseñando con el *wrapper*, se realizarán conexiones por defecto conforme se añadan los nodos, como se explicará más adelante.

- **XppDomElement** [41]: representa la sección donde se indican los parámetros para cada nodo del grafo.

  Podría decirse que este tipo de clase pretende organizar los parámetros en forma de árbol. Partiendo del XppDom, cada vez que quiera indicarse un nuevo parámetro deberá utilizarse la función *createChild* (crear hijo) y para indicar el valor para ese parámetro bastará con *setValue*.

  Si un parámetro concreto, como pueda ser una banda, está definido a su vez por varios parámetros, como pueden ser su nombre y su expresión matemática, se deberán crear nuevos hijos (nombre y expresión) para el hijo ya existente (banda). De este modo la sección de parámetros contendrá un hijo y, dependientes del mismo, dos hijos de segunda generación.

- **PrintWriterProgressMonitor** [42]: pretende informar sobre el progreso real de la ejecución de un grafo que se esté llevando a cabo. Este tipo de "monitor de progreso" imprime por pantalla el porcentaje de avance del procesamiento en cada momento.

  Si bien es cierto que puede ser útil para cerciorarse de que la ejecución se desarrolla con normalidad, no se ha explorado su uso y quedaría pendiente para una nueva versión más refinada del *wrapper*.

- **FileReader** [43]: se trata de una clase originaria de Java cuyo único objetivo es el de leer un grafo desde un archivo XML. Indicando únicamente la ruta de dicho fichero XML, toma la configuración del grafo.

  Para aquellos grafos que tengan variables en su configuración, es posible leerlos exactamente así o leerlos a la vez que se sustituyen por valores específicos. Para ello, basta con indicar los valores a sustituir para los parámetros correspondientes a la hora de utilizar la función de lectura de esta clase.

  Esta acción de sustitución es posible debido a la notación del XML. Se recuerda que una variable se diferencia de un valor concreto mediante el uso de la nomenclatura ${variable}. Es decir,

un parámetro de tipo booleano puede contener un valor concreto, por ejemplo, *true*; o puede definirse mediante una variable, por ejemplo, *${parametro_booleano}*.

A la hora de sustituir las variables por valores específicos, la función de lectura busca internamente las marcas indicativas de la presencia de las mencionadas variables (${}). Si encuentra alguna y el nombre dentro de las llaves ({}) coincide con el nombre especificado en la llamada, entonces será sustituido por el valor que se haya indicado. Esto es, si se indica como argumento 'parametro_booleano'='false', al encontrar la variable antes descrita la sustituirá por 'false'. Si no hubiera variable con dicho nombre, no se produciría ningún cambio en el grafo. Cabe destacar que si en varios puntos del XML se encuentran variables con el mismo nombre, se sustituirían todas ellas por el valor que se haya indicado.

- **FileWriter** [44]: es una clase con base en Java que se utiliza para escribir en un archivo XML un grafo diseñado por código. Basta con indicar el nombre que quiera darse al archivo (o la ruta completa) para guardar la configuración actual del grafo de forma instantánea.

4.2. Programación de una librería de *Python*

En base a las clases recién explicadas se construyó el *wrapper*. Su forma de englobar toda la funcionalidad vista en los apartados anteriores permite al usuario abstraerse de la exigencia que conlleva aprender cómo estructurar los grafos en *snappy*. En cambio, podrá diseñarlos de forma intuitiva y sencilla.

Primero de todo, es necesario refrescar la funcionalidad que alberga la librería:

1. **Lectura de grafos**: leer archivos XML, sin importar que los parámetros estén definidos por valores o por variables. En este segundo caso, puede decidirse entre leer las variables o sustituirlas en la propia lectura por valores concretos.
2. **Creación de grafos**: generar un nuevo grafo completamente vacío sobre el que empezar a diseñar.
3. **Edición de grafos**: sin importar si se ha tomado un grafo desde XML o de si se ha creado uno nuevo, es posible añadir todas las operaciones (nodos) que se desee e indicar los parámetros necesarios (con valores o como variables). También puede redefinirse en cualquier momento el flujo de procesamiento, es decir, el orden en el que se ejecutan las operaciones.
4. **Ejecución de grafos**: ejecutar el grafo que se esté tratando en el momento. En este caso, sí que es necesario que todos los parámetros tengan asignados valores concretos (no sean variables), de modo que no se eviten errores en el procesamiento.
5. **Escritura de grafos**: siempre que se crea conveniente, puede guardarse como XML el grafo que se esté diseñando. De este modo, pueden guardarse diversas variantes de un grafo en archivos distintos. Para ello, es suficiente con indicar distintos nombres por cada uno de ellos, de manera que no se sobrescriban en memoria.

Con el fin de abarcar toda la funcionalidad descrita y de hacer el uso de la librería lo más sencillo posible, se desarrollaron dos clases.

4.2.1. Grafo

Alberga toda la funcionalidad principal comentada en el punto anterior. Para ello, se compone de diversas funciones que, en su gran mayoría, toman argumentos similares, como son el nombre de un nodo sobre el que actuar y un diccionario con los parámetros a utilizar.

Con el objetivo de asemejarse todo lo posible a la descripción de las acciones, las funciones siguen la siguiente política de nombres: deben comenzar por un verbo (en minúsculas) que indique la acción que quiere realizarse, al cual debe seguirle un sujeto (empezando en mayúscula) que represente la parte del *Grafo* sobre la que se quiere actuar. De este modo, en el momento de escribir código, es muy sencillo referirse a una función concreta por ser muy parecida a lo que se necesita hacer. Por ejemplo, para añadir un nodo hay que referirse a la función *addNode* y para reordenar el flujo de procesamiento, basta con llamar a *connectGraph* (nombre similar al utilizado por el interfaz de SNAP).

Siguiendo la referencia a las funciones, a continuación se detallan todas las que se incluyen en la clase *Grafo*. En concreto, se indica por cada una de ellas su nombre y una breve descripción de su cometido. Los argumentos de entrada que admiten en general son el nombre de un nodo sobre el que actuar y, si corresponde, un diccionario de parámetros, conformado por sus respectivos nombres y sus valores a asignar. Es destacable que no producen argumentos de salida, sino que modifican la instancia de *Grafo* que se esté tratando en el momento.

- **createGraph**: crea un nuevo *Grafo*, inicializando sus atributos.

  Entre estos atributos se encuentra el número de cada tipo de operación que haya en el *Grafo* y la lista de nombres únicos de los nodos presentes en el grafo.

  Es claro que si se trata de un grafo vacío, el número por cada operación se inicializará a cero y la lista de nombres empezará estando vacía.

  Si, por el contrario, el *Grafo* es generado tras la lectura de un archivo XML, sus atributos serán inicializados conforme a la información que en él se encuentre. Lo cual significa que el número de operaciones y los nombres de los nodos no serán todos cero ni vacío, sino que contendrán los datos pertinentes.

- **addNode**: añade un nuevo nodo al *Grafo*, incluyendo la configuración de parámetros que se le haya dado.

  Actualmente, hay un número limitado de operaciones que son reconocidas por el *wrapper* como posibles nodos a incluir en los grafos. Aunque solo se indica una breve descripción del objetivo de cada una de ellas, serán comentadas más en detalle cuando se planteen los flujos de procesamiento empleados en la experimentación.

  - **Read**: lee los datos radar de un producto Sentinel-1, entre otros.
  - **Apply-Orbit-File**: actualiza los metadatos del producto según la posición y la velocidad propias del satélite en el momento de capturar los datos.
  - **Calibration**: aplica a los píxeles una calibración radiométrica en base a cierta información adicional contenida en el propio producto.
  - **Band Maths**: modifica o crea bandas ráster mediante la aplicación de expresiones matemáticas sobre una o varias de las bandas originales del producto.
  - **Band Merge**: copia la información ráster de varios productos a un producto único.
  - **Subset**: crea una región espacial a partir de un producto de datos.
  - **Terrain-Correction**: corrige las posibles distorsiones que puedan aparecer en los datos debido a la topografía del terreno.
  - **ThermalNoiseRemoval**: reduce los efectos del ruido entre diferentes zonas de adquisición o *sub-swaths* dentro de un mismo producto.
  - **Speckle-Filter**: elimina el ruido granular consecuencia de la interferencia entre ondas reflejadas.
  - **Write**: almacena un nuevo producto formado por datos ya procesados.

- **addParameters**: añade nuevos parámetros al nodo en cuestión. Es importante destacar que el parámetro sobre el que actuar no debe haber sido previamente definido para ese nodo. En caso de no ser así, se indicará un mensaje por pantalla con el motivo del error y se instará al usuario a utilizar la función *replaceValues*.

- **replaceValues**: cambia los valores de los parámetros de un nodo. Es importante destacar que el parámetro debe estar previamente definido, ya sea mediante un valor o mediante una variable, para poder realizar este cambio. En caso de que no exista este parámetro, se mostrará un mensaje comentando el error e instando al usuario a utilizar la función *addParameters*.
- **addSources**: añade nuevas fuentes de datos a un nodo, es decir, configura los *NodeSource* para un nodo concreto. Una operación puede depender de varias operaciones anteriores, como ocurre en el caso de Band Merge, por ejemplo. Es importante recalcar que si antes de utilizar esta función ya existían otras fuentes para el nodo objetivo, las nuevas fuentes indicadas serán añadidas a estas, no reemplazadas.
- **removeSources**: elimina las fuentes de un nodo, es decir, el nodo concreto deja de tener *NodeSource* asignados. Suele ser utilizada como función auxiliar para changeSources.
- **changeSources**: cambia las fuentes de datos de un nodo, es decir, edita los *NodeSource* para un nodo concreto. Esta función sustituye las fuentes actuales por las nuevas fuentes indicadas. De hecho, hace uso de las dos funciones anteriores para completar su objetivo.
- **connectGraph**: redefine el flujo de procesamiento (el orden de ejecución de las operaciones). No es necesario indicar todos los nodos presentes, basta con indicar aquellos que se quieran modificar. Su uso es sencillo y abarca los cambios de fuentes contemplados en las anteriores funciones. Aunque estas siguen siendo necesarias para realizar bien los cambios, se recomienda que el usuario utilice cómodamente connectGraph.
- **readGraph**: lee un grafo previamente almacenado en XML. Puede leer su contenido tal cual esté definido o puede sustituir por valores las variables que encuentre. Esta lectura propicia la generación de un *Grafo* cuyos atributos serán actualizados conforme al grafo reflejado en el XML.
- **writeGraph**: escribe un *Grafo* a XML. Se puede indicar un nombre de archivo si se quiere almacenar con nombre personalizado.
- **execGraph**: ejecuta el *Grafo*.
- **showParameters**: presenta por pantalla todos los parámetros que admite la operación asignada a un nodo concreto, resaltando aquellos que son obligatorios (si los hubiera).
- **listNodes**: muestra por pantalla qué nodos componen el *Grafo* en el momento de la consulta.
- **showNode**: imprime por pantalla qué parámetros, incluyendo sus valores, posee un nodo concreto en el momento de la consulta.
- **showNodes**: imprime por pantalla al igual que showNode pero para todos los nodos del *Grafo* a la vez.

Una vez conocidas las funciones, cabe destacar algunos aspectos relevantes que el *wrapper* aporta a la hora de diseñar los grafos.

Para empezar, sobresale la función "connectGraph". Esta permite un gran dinamismo a la hora de diseñar los grafos, superando en ello al interfaz de SNAP. En el interfaz, al indicar que se conecte el grafo, el orden de ejecución de los nodos es exactamente igual al orden en el que se han ido creando. En el *wrapper*, esta conexión, con orden por defecto, se hace de forma automática conforme se añaden nuevos nodos, sin necesidad de utilizar ninguna función. Si, por cualquier razón, se requiere cambiar el orden establecido, en el *wrapper* bastaría con utilizar *connectGraph*, indicando un diccionario con el orden a establecer. En cambio, en el interfaz, debería eliminarse cada unión no válida y volver a realizar las conexiones pertinentes una a una. En cualquiera de los dos casos, se puede cambiar únicamente el orden de ciertos nodos y no necesariamente rediseñar el flujo completo. Es claro, por tanto, que el *wrapper* permite el diseño ágil de una amplia combinación de flujos distintos para una misma configuración de nodos.

También las funciones *addParameters* y *replaceValues* pueden resultar claves, ya que, aunque a la hora de añadir nuevos nodos es común indicar sus parámetros en el momento de crearlos, dichas funciones permiten que sean añadidos o modificados posteriormente. Esta posibilidad continúa aportando dinamismo al diseño ya que, de este modo, los grafos pueden modificarse en cualquier momento y guardarse o ejecutarse por cada una de sus variantes.

Por último, es necesario recalcar las funciones de muestra de información por pantalla. Estas funciones constituyen un intento de integrar y de mejorar las consultas de información que permite el GPT por línea de comandos. Para conseguirlo, permiten consultar cualquier detalle sobre la estructura del grafo en tratamiento, para tener una mejor percepción del mismo. Asimismo permiten conocer los parámetros válidos, de modo que el usuario no dude a la hora de escribir correctamente sus nombres.

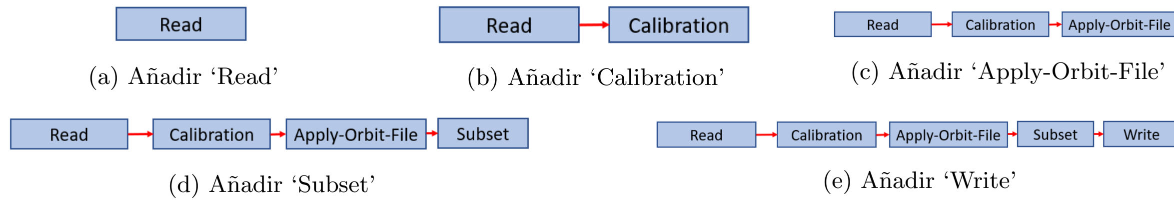
4.2.1.1 Ejemplo de uso: creación de un grafo

Para una mejor compresión de las funciones vistas y de las ventajas que pueden aportar en comparación con el interfaz de SNAP, se detallan a continuación los pasos a seguir para crear un grafo completo. Por un lado, se comenta el código y su propósito específico, y por otro, se muestra la evolución del grafo de manera visual. Además, se compara en paralelo la forma en la que deberían realizarse las mismas acciones pero en el *GraphBuilder* del interfaz.

Como grafo de ejemplo se va a construir un flujo de procesamiento básico. Va a contener cinco nodos (cinco operaciones distintas): Read, Calibration, Apply-Orbit-File, Subset y Write. Inicialmente, el orden de los nodos va a ser establecido exactamente igual a como han sido listados. En cuanto a parámetros, solo se va a especificar uno en una de las operaciones: en Calibration debe indicarse que se genere la banda *gamma0*. En este caso, no se trata de obtener un grafo perfectamente diseñado en cuanto a parámetros obligatorios, sino de mostrar una posible configuración mediante diccionario.

El código necesario para generar ese grafo exacto es el siguiente (se indica de forma visual cómo iría evolucionando el diseño al mismo tiempo):

1. Crear un *Grafo* vacío.
   ```
   grafo = Grafo()
   grafo.createGraph()
   ```
2. Añadir un nodo de operación 'Read' (Figura 17a).
   ```
   grafo.addNode("read")
   ```
3. Definir un diccionario de parámetros para el siguiente nodo 'Calibration'.
   ```
   parametros = {"outputGammaBand":"true"}
   ```
4. Añadir un nodo de operación 'Calibration' indicando sus parámetros (Figura **??**).
   ```
   grafo.addNode("calibration",parametros)
   ```
5. Añadir un nodo de operación 'Apply-Orbit-File' (Figura 17c).
   ```
   grafo.addNode("applyorbitfile")
   ```
6. Añadir un nodo de operación 'Subset' (Figura 17d).
   ```
   grafo.addNode("subset")
   ```
7. Añadir un nodo de operación 'Write' (Figura 17e).
   ```
   grafo.addNode("write")
   ```



(a) Añadir 'Read' (b) Añadir 'Calibration' (c) Añadir 'Apply-Orbit-File'

(d) Añadir 'Subset' (e) Añadir 'Write'

Figura 17: Evolución del diseño de un *Grafo* mediante el *wrapper* de *snappy*

Nótese que la funcionalidad del *wrapper* antes explicada, de ir conectando por defecto los nodos en el orden en el que se añaden, implica que tras los pasos anteriores, el *Grafo* ya tiene un flujo correctamente definido (Figura 17e).

Los pasos a dar en el interfaz de SNAP para conseguir el mismo resultado que mediante código se indican a continuación, incluyendo las imágenes del progreso en el diseño:

- Crear un grafo vacío.

  En el *GraphBuilder* del interfaz, al crear un nuevo grafo siempre se inicializan las operaciones 'Read' y 'Write' por defecto (Figura 18a).

- Añadir un nodo de operación 'Calibration' (Figura 18c).
- Configurar los parámetros del nodo 'Calibration'.

  Este paso puede realizarse en cualquier momento del diseño, como puede verse en la Figura 18d, en la cual se configura el parámetro después de añadir todos los nodos.

- Añadir un nodo de operación 'Apply-Orbit-File' (Figura 18e).
- Añadir un nodo de operación 'Subset' (Figura 18f).
- Definir el orden de ejecución de los nodos (Figura 18g).

Finalmente, el grafo queda definido en su totalidad como muestra la Figura 18h.

Cabe destacar que para la inclusión de cada uno de los nodos debe realizarse una búsqueda por las categorías de tipos de procesamiento del menú desplegable del *GraphBuilder*. Esto conlleva una gran pérdida de tiempo si el usuario no recuerda específicamente en qué categoría se encuentra la operación que busca. En el caso del nodo 'Calibration' se debe navegar por varios menús hasta llegar a dicha operación, como se muestra en la Figura 18b. En contraposición a este hecho, no es necesario conocer la forma de agrupar las operaciones para utilizar el *wrapper*.

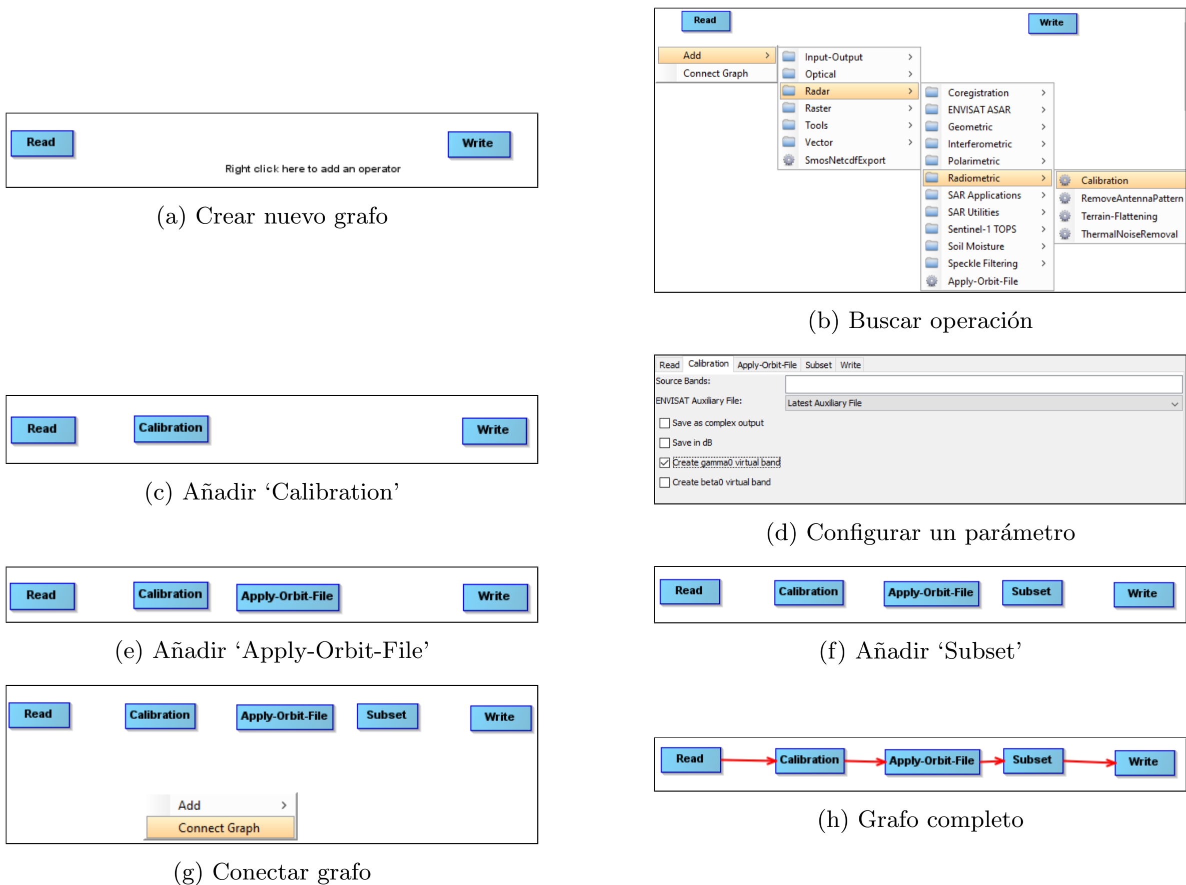
4.2.1.2 Ejemplo de uso: modificación de un grafo

Siguiendo el ejemplo, se percibe que la forma en la que están conectados los nodos no es la más óptima ni tampoco la más común. Por lo tanto, se quiere cambiar el flujo por otro mejor: Read, Subset, Apply-Orbit-File, Calibration y Write. Para poder realizar este cambio de orden, mediante código basta con indicar:

1. Definir un diccionario que indique el nuevo orden para los nodos.

   Como método para no liar los nodos, debe pensarse siempre que las claves del diccionario son los nodos a configurar y que los valores del diccionario son los meros nombres a imponer como datos de entrada para dichos nodos.

   ```
   flujo = {'Subset':'Read','Apply-Orbit-File':'Subset',
             'Calibration':'Apply-Orbit-File','Write':'Calibration'}
   ```



(a) Crear nuevo grafo

(b) Buscar operación

(c) Añadir 'Calibration'

(d) Configurar un parámetro

(e) Añadir 'Apply-Orbit-File'

(f) Añadir 'Subset'

(g) Conectar grafo

(h) Grafo completo

Figura 18: Evolución del diseño de un grafo mediante el interfaz de SNAP

2. Indicar esta nueva configuración a la función pertinente.

   `grafo.connectGraph(flujo)`

   El cambio es instantáneo. Tanto es así que se muestra la comparación del flujo anterior y el actual en la Figura 19 para poder apreciar las diferencias.

(a) Configuración previa

(b) Grafo final

Figura 19: Rediseño del flujo de un *Grafo* mediante el *wrapper* de *snappy*

En el interfaz de SNAP, el cambio de flujo comprenderá más pasos como se puede confirmar a continuación:

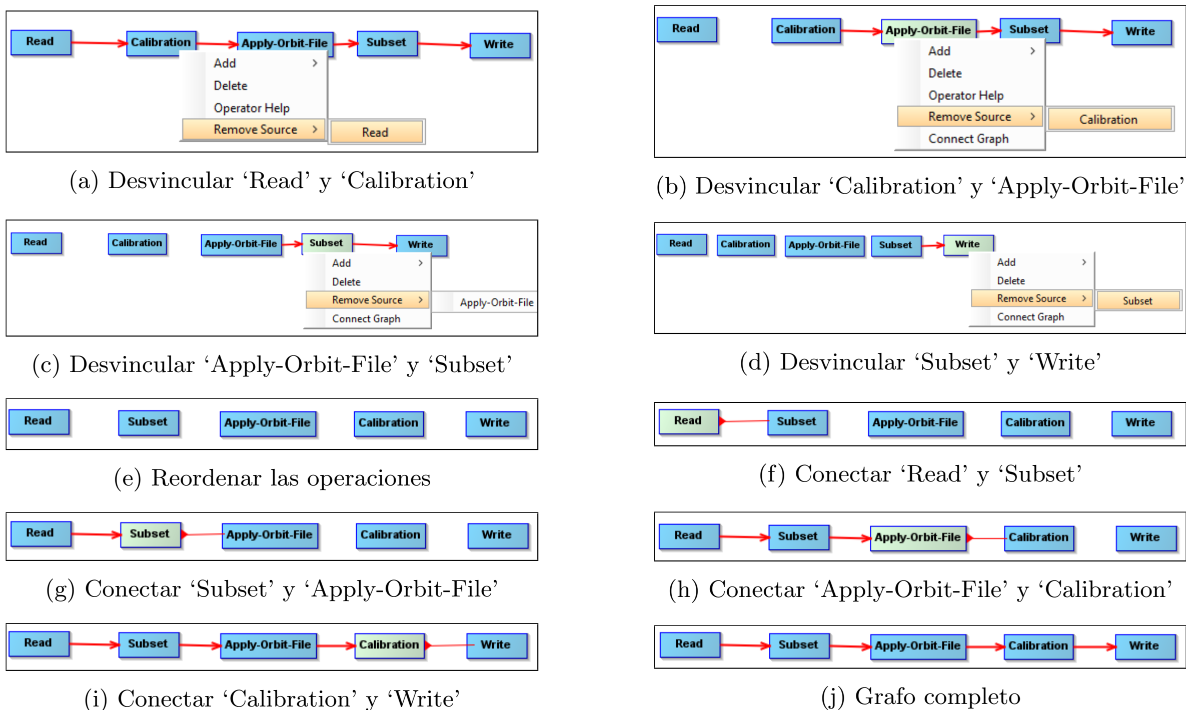
1. Eliminar el arco entre el nodo 'Read' y el nodo 'Calibration' (Figura 20a).
2. Eliminar el arco entre el nodo 'Calibration' y el nodo 'Apply-Orbit-File' (Figura 20b).
3. Eliminar el arco entre el nodo 'Apply-Orbit-File' y el nodo 'Subset' (Figura 20c).
4. Eliminar el arco entre el nodo 'Subset' y el nodo 'Write' (Figura 20d).
5. Reordenar los nodos (*click* y arrastrar, como en cualquier interfaz gráfico).

De esta manera, la nueva ordenación de nodos se puede visualizar de forma más cómoda, como se puede ver en la Figura 20e. Pero no solo eso: este paso, que puede parecer opcional, permite que, con tal solo indicar 'Connect Graph', el grafo quede dispuesto perfectamente, como puede apreciarse en la Figura 20j.

Es decir, esta acción evita tener que realizar los pasos que se indican a continuación. Estos se mantienen en la explicación por suponer una alternativa en caso de no querer llevar a cabo la reordenación visual, aunque sea más recomendable.

6. Conectar el nodo 'Read' y el nodo 'Subset' (Figura 20f).
7. Conectar el nodo 'Subset' y el nodo 'Apply-Orbit-File' (Figura 20g).
8. Conectar el nodo 'Apply-Orbit-File' y el nodo 'Calibration' (Figura 20h).
9. Conectar el nodo 'Calibration' y el nodo 'Write' (Figura 20i).

Finalmente, en la Figura 20j se muestra el grafo completo con el nuevo orden de nodos, que si bien es idéntico al obtenido mediante código, ha supuesto una mayor inversión de tiempo.



(a) Desvincular 'Read' y 'Calibration'

(b) Desvincular 'Calibration' y 'Apply-Orbit-File'

(c) Desvincular 'Apply-Orbit-File' y 'Subset'

(d) Desvincular 'Subset' y 'Write'

(e) Reordenar las operaciones

(f) Conectar 'Read' y 'Subset'

(g) Conectar 'Subset' y 'Apply-Orbit-File'

(h) Conectar 'Apply-Orbit-File' y 'Calibration'

(i) Conectar 'Calibration' y 'Write'

(j) Grafo completo

Figura 20: Rediseño del flujo de un grafo mediante el interfaz de SNAP

4.2.2. Helper

Sirve como apoyo a la funcionalidad de la clase anterior para que no se produzcan incongruencias en el diseño de los grafos. Para ello, cuenta con diversas funciones que devuelven datos específicos de la información de SNAP, destacando las listas de parámetros para cada posible operación. Su uso está pensado exclusivamente para proporcionar información a *Grafo*, el cual o bien dará formato a los datos para mostrarlos al usuario, o bien realizará comprobaciones internas.

Estas comprobaciones sirven para verificar en todo momento que el usuario no indica por código nombres mal deletreados u operaciones o parámetros que no existen en el contexto del *wrapper*.

De este manera, nunca habrá problemas en la posterior ejecución del *Grafo*. Por ejemplo, si se quiere añadir un nodo de operación 'Reed', aparecerá un error indicando que esa operación no es válida y no será añadida al *Grafo*.

Otra funcionalidad importante que contiene este *script* es la de transformar un diccionario de Python en un HashMap de Java. Esta necesidad surge de que ciertos elementos propios de SNAP, como pudiera ser el *FileReader*, solo permiten como argumento de entrada un objeto de tipo HashMap. Puesto que la motivación para el desarrollo de este *wrapper* es, precisamente, trabajar únicamente en lenguaje Python, esta función auxiliar interna evita que el usuario tenga que tener conocimientos sobre Java.

En definitiva, las funciones que componen esta segunda clase son:

- **helpAll**: proporciona un listado con todos los parámetros que pueden indicarse para cierta operación.
- **helpMandatory**: proporciona un listado solo con aquellos parámetros que obligatoriamente deben tener asignado un valor concreto para una operación dada. Los parámetros obligatorios son aquellos que de no asignarse específicamente un valor, quedarían sin asignar. En cambio, los parámetros prescindibles son aquellos que ya tienen valores predefinidos asignados y solo son cambiados de ser indicado explícitamente.
- **showParameters**: proporciona toda la información sobre los parámetros que contiene en el momento el nodo que se esté consultando, incluyendo sus valores.
- **init_num_ops**: funcionalidad auxiliar que permite inicializar el contador de cada tipo de operación a cero cuando se crea un nuevo *Grafo*. Ello indica que el *Grafo* está completamente vacío.
- **list_num_ops**: funcionalidad auxiliar que contabiliza las operaciones de cada tipo presentes en el *Grafo* en cada momento. De este modo, si ya existe un nodo para cierta operación, póngase 'Read', el número de operaciones será de uno; y si se añade uno nuevo será actualizado a dos, y así sucesivamente.

  La necesidad de contabilizar los nodos de esta forma recae en la obligatoriedad, antes comentada, de que los nodos mantengan nombres identificativos únicos. Al añadir nuevos nodos al grafo, no es imprescindible indicar para ellos nombres personalizados por el usuario, sino que pueden establecerse internamente por defecto. Esta funcionalidad requiere conocer en todo momento el número de nodos por operación para evitar incongruencias en el grafo. Por ejemplo, si ya existe un 'Read', un nuevo nodo de lectura tomará el nombre 'Read2' (si el usuario no especifica otro), puesto que dicho *2* será el número que corresponda según la *list_num_ops*.
- **dictToHash**: realiza la transformación de diccionario de Python a HashMap de Java comentada anteriormente para aquellos casos en los que sea imprescindible utilizar esta última estructura como entrada de algunas funciones utilizadas en la librería.

4.3. Mejoras para una próxima versión

Aunque el *wrapper* ya puede ser utilizado con normalidad para gestionar los grafos como se ha explicado, no hay que olvidar que se trata de una primera versión. Por ello, es requisito indispensable tener en cuenta aquellos aspectos que pueden mejorarse en una segunda iteración del desarrollo.

Entre dichas futuras mejoras están la inclusión de nuevos posibles nodos. Aunque las operaciones que hasta ahora están presentes en el *wrapper* son las utilizadas en el actual proyecto, otras muchas pueden pasar a ser necesarias conforme se avance en el tratamiento de imágenes radar. Debido a ello, será conveniente ampliar las operaciones que son reconocidas por la librería. Esto supone, entre otras tareas de implementación, añadir la información correspondiente a los parámetros de las nuevas operaciones.

Como mejora complementaria a las funciones de añadir y modificar nodos, está el añadir funciones que permitan la eliminación de parámetros o la eliminación completa de uno o varios nodos.

Como posible ganancia de tiempo, puede modificarse la función *createGraph* para asemejarse al comportamiento del interfaz de SNAP a la hora de crear un nuevo grafo. Como se ha visto antes, el *wrapper* genera un grafo vacío, mientras que el *GraphBuilder* incluye por defecto un nodo de operación 'Read' y otro de 'Write'. Siendo una norma obligatoria el hecho de que todo grafo comience y termine de este modo (o con varias de dichas operaciones), puede suponer un importante ahorro de tiempo incorporar esta funcionalidad por defecto. Bastaría con que, internamente, la función *createGraph* hiciese uso de la función *addNode* indicando como argumento "read" y "write" respectivamente y que, posteriormente, se gestionase automáticamente el *NodeSource* de este último nodo conforme se añadiesen nuevos nodos al grafo.

Por último, en relación a la aportación de información al usuario y la comprobación de los grafos para evitar excepciones en la ejecución, puede ser muy beneficioso incluir la información y comprobación de los valores dados a los parámetros. Esto es, hacer posible la consulta de qué valores específicos puede tomar un parámetro, ya sea una opción en formato *string* o un rango de valores en formato número. Asimismo, se evitarían malas configuraciones de los parámetros en aquellos casos en los que, entre otras posibilidades, se indicase un número donde solo es posible indicar una opción o se escribiese incorrectamente dicha opción.

5. Fase de experimentación

El desarrollo del *wrapper* de *snappy* trajo consigo la posibilidad de diseñar numerosos grafos de procesamiento de forma ágil y dinámica. Este hecho sentó las bases para la realización de una serie de experimentos que comparasen la efectividad en la segmentación de imágenes Sentinel dependiendo del procesamiento aplicado.

El objetivo específico de la experimentación puede encontrarse en la Sección 5.1. Después, se detalla la forma en la que se diseñaron los casos para la experimentación, en la Sección 5.2, concretando los aspectos más relevantes de la configuración de los grafos y de las redes neuronales empleadas para la segmentación. Posteriormente, en la Sección 5.3 se relata la forma en la que se obtuvieron los datos necesarios para el conjunto de los experimentos, haciendo especial mención a la librería *GeoFlow* [45]. Por último, se presentan los resultados obtenidos y se analizan los aspectos más relevantes de los mismos, en la Sección 5.4.

5.1. Objetivo

El fin último de este proyecto es mejorar la segmentación semántica de edificios y carreteras basada en imágenes Sentinel-1 y Sentinel-2, tratando de optimizar el procesamiento de las imágenes Sentinel-1 en concreto. Segmentación semántica quiere decir, de forma simple, clasificar cada píxel de una imagen en una clase específica. Para lograr una clasificación adecuada, es necesario obtener datos de calidad sobre los que una red neuronal pueda aprender y extraer información relevante. Dichos datos se obtienen mediante el tratamiento de las mencionadas imágenes, pero para ello no existe un único método de procesamiento.

Precisamente en esta falta de un flujo de procesamiento único se asienta la experimentación: pretende comparar diferentes diseños para el tratamiento de imágenes Sentinel-1 mediante la evaluación de la posterior segmentación de edificios y carreteras. Es decir, un flujo concreto se considera más adecuado que otro si el primero propicia una mejor segmentación que este último.

5.2. Diseño

El objetivo antes planteado requiere planificar tanto los tratamientos a aplicar como la configuración de la red neuronal a utilizar en la segmentación.

Específicamente, en la Sección 5.2.1 se detalla el flujo de procesamiento utilizado hasta el momento para tratar las imágenes Sentinel-1 y los nuevos flujos diseñados para tratar de mejorar su aportación a la clasificación. Asimismo, se presenta en la Sección 5.2.2 el modelo de red neuronal utilizado para el aprendizaje y posterior segmentación de las mencionadas imágenes.

5.2.1. Flujos de procesamiento

La forma de tratar las imágenes para obtener datos de calidad es clave para la realización de los experimentos. Hasta ahora, el procesamiento utilizado para Sentinel-1 conllevaba la utilización de una serie de operaciones de SNAP como muestra la Figura 21. Esta configuración concreta se denomina como 'rebotes' y este nombre será utilizado en adelante para referirse a este grafo inicial.

Para entender mejor este tratamiento inicial del que van a partir la serie de variantes a comparar, es necesario concretar cada una de las operaciones, incluyendo los parámetros utilizados.

- **Read**: operación de lectura de datos desde un producto Sentinel-1. Precisa del nombre del

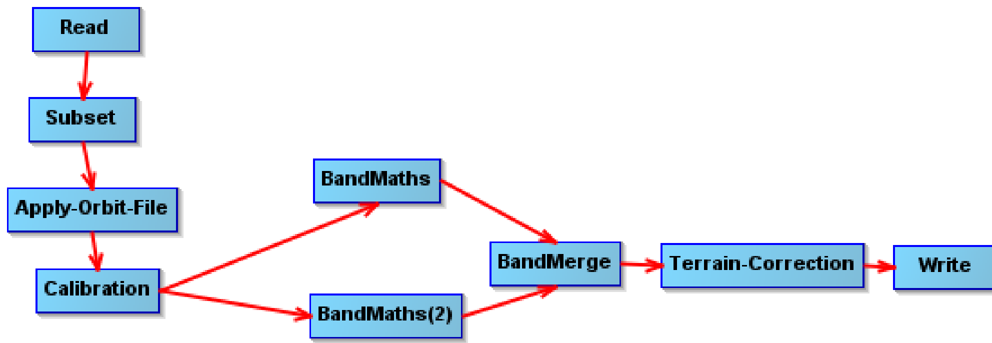


Figura 21: Grafo de procesamiento inicial, 'Rebotes'

producto a tomar como entrada de datos. Este parámetro, denominado *file*, se deja como variable, de modo que en cada momento de la experimentación se detalle el producto a utilizar en cada caso.

- **Subset**: toma únicamente los datos contenidos en cierta región del producto que haya sido leído. La mencionada región es llamada *geoRegion* y se considera como variable, ya que para cada producto distinto se requerirá de una región diferente.

  Otro parámetro incluido es *copyMetadata*, booleano establecido a *true*, que indica si los metadatos del producto original deben ser tomados también por la región seleccionada. Esta es buena práctica para no perder información relevante, puesto que algunas operaciones requieren de ellos para realizarse correctamente. De no indicar este parámetro de esta forma, no se podría realizar la operación 'Calibration', por ejemplo, después de generar la región.

  Utilizar 'Subset' inmediatamente después de la lectura es la mejor disposición para el grafo si la región tomada es la única zona de interés para el procesamiento, ya que de esta manera no se invierte tiempo en procesar datos que van a desecharse posteriormente.

- **Apply-Orbit-File**: actualiza los vectores de estado contenidos en los metadatos del producto según la posición y la velocidad propias del satélite en el momento de capturar los datos. Como ya se ha dicho, los metadatos pueden ser críticos para algunas operaciones, luego su actualización es necesaria para que no se malinterpreten los datos.

  Es relevante comentar que estos datos conocidos como "órbita"se encuentran disponibles en un plazo de días o incluso semanas. Hasta que no se cumple dicho plazo, no puede conocerse esta información específica del satélite para el producto concreto. Esto puede suponer un problema a la hora de aplicar esta operación en productos de muy reciente adquisición. En el caso de este proyecto, no supuso un problema por no trabajar con datos tan cercanos en el tiempo.

- **Calibration**: aplica a los píxeles una calibración radiométrica en base a cierta información adicional. La fuente de la que tomar dicha información se indica en el parámetro *auxFile*. Su valor por defecto es *Latest Auxiliary File*, pero, en este caso, se especifica el valor *Product Auxiliary File* para que se tenga en cuenta la información contenida en el propio producto.

- **Band Maths**: modifica o crea bandas ráster mediante la aplicación de expresiones matemáticas sobre una o varias de las bandas originales del producto.

  Todos los flujos diseñados cuentan con dos operaciones de este tipo: una que aplica una expresión matemática sobre la banda VH y otra que aplica la misma expresión solo que sobre la banda VV. Para que las bandas resultantes no sean confundidas, se indican como *name* "VH_DB" y "VV_DB", respectivamente. La expresión matemática utilizada es "$log_{10}(banda) \times 10$", siendo *banda* VH o VV según corresponda.

Los datos a obtener son de tipo *float32* y, en caso de que se produzcan datos no válidos para alguno de los píxeles (valores negativos, por ejemplo), estos serán considerados con el valor 0, el cual es indicado para el parámetro *noDataValue*.

- **Band Merge**: copia la información ráster de varios productos a un producto único.

  Esta operación es utilizada para poner en común las dos bandas obtenidas tras la aplicación de 'Band Maths'. De este modo, en lugar de trabajar independientemente con cada una de las bandas por separado, se pone en común un producto que alberga las dos bandas.

- **Terrain-Correction**: corrige las posibles distorsiones que puedan aparecer en los datos debido a la topografía del terreno.

  Para esta operación se establecen hasta cinco parámetros con valores personalizados, siendo tres de ellos variables a concretar en cada caso particular. El primero es *pixelSpacinginMeter*, cuyo valor implica la correspondencia de metros contenidos por cada píxel y puede depender de la resolución de cada producto.

  Además, se encuentra *mapProjection* que representa el sistema de coordenadas de referencia del producto, el cual alberga en sí mismo varios parámetros y se indican dos como variable: la zona UTM en la que se encuentra el producto y el meridiano sobre el que se halla.

  En cuanto a los parámetros con valores específicos se indica *nodataValueatSea* como booleano *false* y *demName* a *SRTM 1sec HGT*. El primero implica que no se descarten las zonas de mar a la hora de realizar la corrección. El segundo, es algo más complejo.

  El parámetro *demName* hace referencia al nombre de los DEM (Data Elevation Models) o modelos de datos de elevación. Esto quiere decir que el modelo que se indique será tomado como referencia para conocer información sobre las elevaciones del terreno. El valor especificado, *SRTM 1sec HGT*, implica utilizar el modelo derivado de la *Shuttle Radar Topography Mission* de la NASA [46], que contiene datos obtenidos en intervalos de latitud y longitud de un arcosegundo o $1''$. Por su parte, *HGT* es la extensión utilizada para los archivos que contienen esta información y sus datos son números enteros con y sin signo de dos bytes [47].

- **Write**: almacena un nuevo producto formado por datos ya procesados. El parámetro *file* en el que se indica el nombre del nuevo producto a generar se deja como variable para poder establecer uno distinto para cada ejecución.

Aunque no existen gran cantidad de ejemplos en la literatura de flujos de procesamiento concretos (y menos mediante operaciones específicas de SNAP) sí se encontró un caso en el que se añaden nuevas operaciones no utilizadas en el grafo anterior. En concreto, se especifican las operaciones 'Speckle-Filter' y 'ThermalNoiseRemoval' [48], no conocidas previamente y cuyo objetivo es el de eliminar diferentes tipos de ruido en las imágenes. Por ello, se planteó la posibilidad de añadir estas dos nuevas operaciones para comprobar si la teórica mejor calidad de los datos tras un procesamiento más extenso es confirmada en los experimentos. Los detalles sobre las operaciones se presentan a continuación.

- **ThermalNoiseRemoval**: reduce los efectos del ruido entre diferentes zonas de adquisición o *sub-swaths* dentro de un mismo producto.

- **Speckle-Filter**: elimina el ruido granular consecuencia de la interferencia entre ondas reflejadas.

  Esta operación permite utilizar diferentes filtros, de los cuales se tienen en cuenta *GammaMap* y *RefinedLee* como valores para el parámetro *filter*. También se aplican diferentes tamaños para dichos filtros: *windowSize* toma los valores 5x5, 7x7 o 9x9 según corresponda. Por último, el parámetro *estimateENL* queda a *true*, el cual indica que se estime automáticamente el *Equivalent Number of Looks*, que podría traducirse como número de puntos de vista equivalentes. Este parámetro es dependiente del modo de adquisición del producto y de la resolución del mismo [49]. Precisamente por este hecho, y porque se van a manejar un gran número de productos diferentes, es una mejor opción que este número se calcule automáticamente.

A través del estudio de estas nuevas operaciones, surgen una serie de posibles combinaciones en cuanto a su inclusión y el uso de sus parámetros. Esto es, a los nodos presentes en el grafo de 'Rebotes' se suma la operación 'ThermalNoiseRemoval' en una primera versión (Figura 22), la operación 'Speckle-Filter' en otra (Figura 23), y las dos operaciones en una tercera (Figura 24).

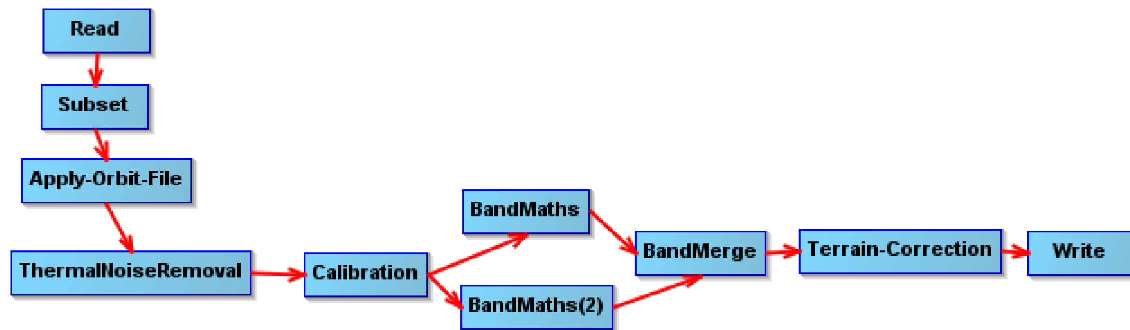


Figura 22: Nuevo grafo de procesamiento con 'ThermalNoiseRemoval'

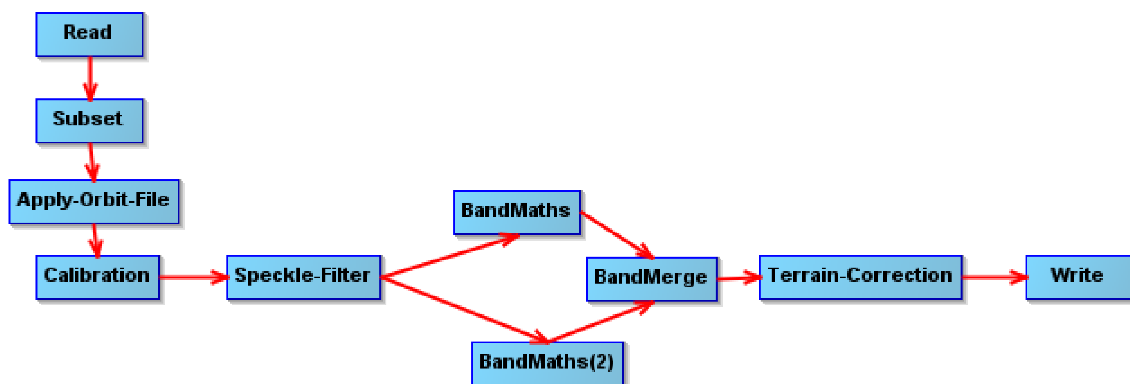


Figura 23: Nuevo grafo de procesamiento con 'Speckle-Filter'

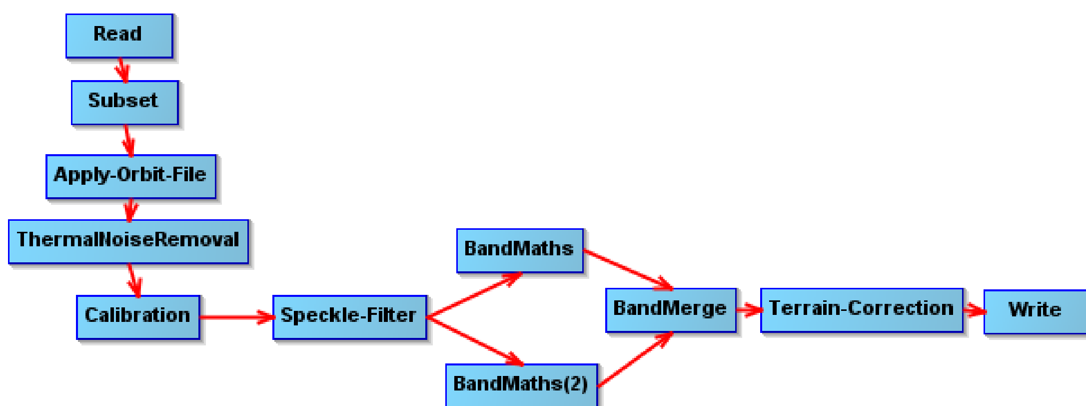


Figura 24: Nuevo grafo de procesamiento con 'ThermalNoiseRemoval' y 'Speckle-Filter'

Aunque viendo los diseños de los grafos pueda parecer que únicamente se tienen tres nuevas opciones de ejecución, la realidad es que existe un número más amplio de posibles procesamientos debido a la variación de los parámetros que puede realizarse. Como aclaración, se enumeran todas las posibles configuraciones (para resumir, se parte del grafo inicial 'Rebotes' en todos los casos):

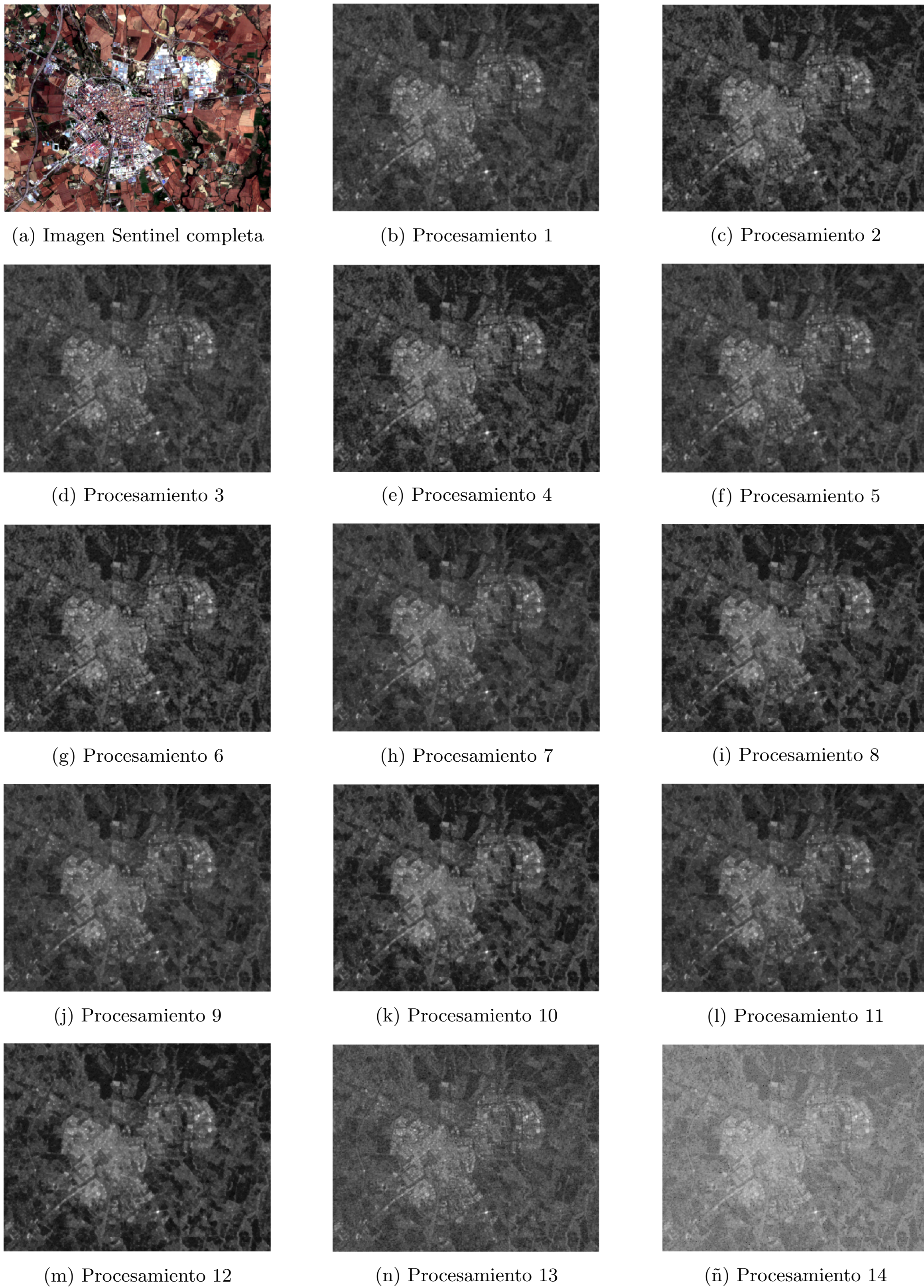
1. Rebotes
2. Rebotes + ThermalNoiseRemoval
3. Rebotes + Speckle-Filter (GammaMap, 5x5)
4. Rebotes + Speckle-Filter (GammaMap, 5x5) + ThermalNoiseRemoval
5. Rebotes + Speckle-Filter (GammaMap, 7x7)
6. Rebotes + Speckle-Filter (GammaMap, 7x7) + ThermalNoiseRemoval
7. Rebotes + Speckle-Filter (GammaMap, 9x9)
8. Rebotes + Speckle-Filter (GammaMap, 9x9) + ThermalNoiseRemoval
9. Rebotes + Speckle-Filter (RefinedLee, 5x5)
10. Rebotes + Speckle-Filter (RefinedLee, 5x5) + ThermalNoiseRemoval
11. Rebotes + Speckle-Filter (RefinedLee, 7x7)
12. Rebotes + Speckle-Filter (RefinedLee, 7x7) + ThermalNoiseRemoval
13. Rebotes + Speckle-Filter (RefinedLee, 9x9)
14. Rebotes + Speckle-Filter (RefinedLee, 9x9) + ThermalNoiseRemoval

Como ejemplo de las diferencias en los procesamientos, en la Figura 25 se presentan los resultados obtenidos para un mismo producto localizado en la ciudad de Huesca. Cada figura muestra la banda ráster VV tras la ejecución de los grafos de procesamiento de SNAP antes mencionados.

La combinación de posibilidades acaba resultando en catorce flujos diferentes. Es un número elevado, pero a la hora de evaluar los resultados de los experimentos puede inferirse que si, por ejemplo, Speckle-Filter (GammaMap, 5x5) implica un mal rendimiento de la segmentación, entonces su combinación con ThermalNoiseRemoval no va a ser buena tampoco. Por este modo de proceder, pueden reducirse las posibilidades contempladas y definirse los siguientes procesamientos (la numeración es relevante para posteriores referencias a los experimentos):

- **Procesamiento 1**: Rebotes
- **Procesamiento 2**: Rebotes + Speckle-Filter (GammaMap, 5x5)
- **Procesamiento 3**: Rebotes + Speckle-Filter (GammaMap, 7x7)
- **Procesamiento 4**: Rebotes + Speckle-Filter (GammaMap, 9x9)
- **Procesamiento 5**: Rebotes + Speckle-Filter (RefinedLee, 5x5)
- **Procesamiento 6**: Rebotes + Speckle-Filter (RefinedLee, 7x7)
- **Procesamiento 7**: Rebotes + Speckle-Filter (RefinedLee, 9x9)
- **Procesamiento 8**: Rebotes + ThermalNoiseRemoval
- **Procesamiento 9**: Rebotes + Speckle-Filter (Mejor Configuración) + ThermalNoiseRemoval

Como puede verse, el último experimento agrupa todas las posibilidades antes definidas para el uso de las dos nuevas operaciones al mismo tiempo. Así, en cuanto se conozca el beneficio de Speckle-Filter, si es lo que hubiera, y la configuración precisa que lo propicia, puede evaluarse la ganancia o pérdida de precisión si es acompañado por ThermalNoiseRemoval.



(a) Imagen Sentinel completa
(b) Procesamiento 1
(c) Procesamiento 2
(d) Procesamiento 3
(e) Procesamiento 4
(f) Procesamiento 5
(g) Procesamiento 6
(h) Procesamiento 7
(i) Procesamiento 8
(j) Procesamiento 9
(k) Procesamiento 10
(l) Procesamiento 11
(m) Procesamiento 12
(n) Procesamiento 13
(ñ) Procesamiento 14

Figura 25: Aplicación de nuevos procesamientos en Huesca

5.2.2. *Deep Learning*

Una vez el tratamiento de datos está definido, cabe preguntarse el modo en el que se van a emplear los datos procesados. Es decir, debe configurarse un modelo de inteligencia artificial para aprender de los datos tratados y ser capaz de, a través del aprendizaje realizado, segmentar semánticamente cualquier nueva imagen.

El modelo utilizado es una red neuronal. Su configuración parte de la ya presentada en un trabajo previo de similares características [50]. En este trabajo se busca tanto aprender características de calidad como segmentar adecuadamente en base a lo aprendido. Por ello, se trabaja con una red completamente convolucional basada en U-Net [51, 52]. La U-Net, cuya estructura se muestra en la Figura 26, sigue una topología *encoder-decoder* que debe su origen a la segmentación de imágenes médicas. Actualmente, este tipo de redes son utilizadas en otros ámbitos, como el que se trata en este proyecto. En este caso, el mencionado *encoder* queda modificado para albergar la arquitectura de una ResNet-34 [53, 54]. De esta forma, se combina la capacidad de U-Net para problemas de segmentación con la capacidad de ResNet-34 para extraer características óptimas con un bajo coste computacional [50].

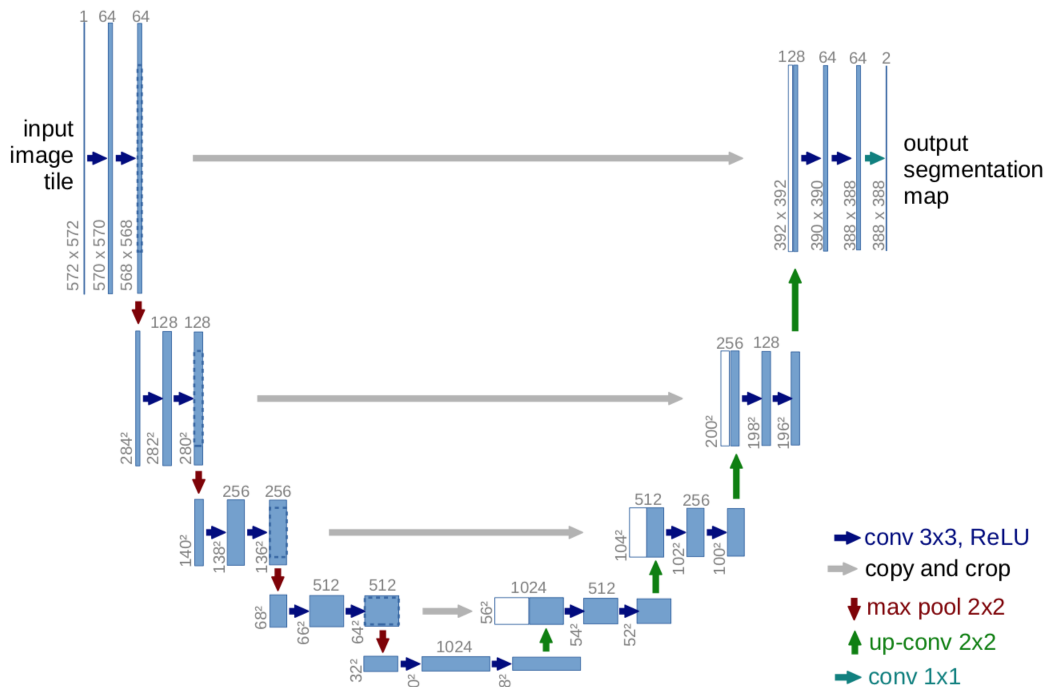


Figura 26: Arquitectura de una red neuronal U-Net [51]

La arquitectura final de la red neuronal consta de un total de 24.468.710 parámetros, de los cuales 24.451.352 pueden ser entrenados. Pueden encontrarse más detalles sobre la red neuronal en [55].

5.3. Generación del *dataset*

Además de hacer un buen uso de los recursos referentes al procesamiento y clasificación de imágenes, es muy relevante centrar la atención en el modo de adquirir y preparar el conjunto de datos a emplear en el aprendizaje y posterior segmentación.

Por esto mismo, se construye un amplio *dataset* conformado por ciudades españolas, en su mayoría capitales de provincia. Con ello se busca aportar diversidad, de modo que los datos constituyan una muestra representativa adecuada de los diferentes terrenos existentes. Las ciudades seleccionadas pueden visualizarse distribuidas por el mapa de la península en la Figura 27 a través de las miniaturas de sus imágenes.

Figura 27: Distribución geográfica del conjunto de datos

En más detalle, se presentan en orden alfabético las ciudades en la Tabla 2 identificadas por sus nombres y acompañadas por las dimensiones en píxeles (altura por anchura) de las imágenes que abarcan cada una de ellas y del conjunto en el que han sido empleadas: siendo *train* el conjunto de entrenamiento (sobre fondo blanco) y *test* el conjunto de prueba (sobre fondo gris).

Puede apreciarse que Barcelona y Madrid aparecen en dos ocasiones denominadas como 'N.', referente a norte, y 'S.', referente a sur. En definitiva, se toman 43 ciudades diferentes, de las cuales 14 pertenecen al conjunto de *test*. Esto implica que el *dataset* queda balanceado con un 67.44 % de las ciudades para el entrenamiento y un 32.56 % para la posterior comprobación del aprendizaje.

Cabe destacar que para cada ciudad no se obtuvieron datos equivalentes a un único producto Sentinel, sino que se obtuvieron un total de cuatro: una imagen por cada trimestre del tiempo comprendido entre el 1 de junio de 2018 y el 1 de junio de 2019. Específicamente, los intervalos de tiempo tenidos en cuenta están comprendidos entre junio y septiembre de 2018, entre septiembre y diciembre de 2018, entre diciembre y marzo de 2019 y entre marzo y junio de 2019, ambos extremos delimitados por el primer día del mes. Ampliar la cantidad de datos puede aportar mayor diversidad al *dataset* y, con ello, una mejor base de aprendizaje para que la red sepa generalizar ante imágenes nuevas. De hecho, esta forma de utilizar la temporalidad puede ser más beneficiosa que otras transformaciones típicas de la técnica *data augmentation* (literalmente, aumento de datos), puesto que se trata de datos reales y no de datos creados artificialmente. Puede contemplarse en la Figura 28 la diferencia entre las imágenes capturadas para una misma ciudad en cada uno de los cuatro trimestres mencionados.

El *dataset* conformado por estas imágenes sirve de entrada para la red neuronal, pero no las utiliza de forma directa. En realidad, las imágenes son troceadas mediante parches, lo cual es una forma habitual de proceder en los estudios de segmentación semántica. Para este proyecto, se utilizan

Ciudad	Dimesiones	Conjunto
Albacete	1280 x 1152	Train
Alicante	1216 x 1472	Train
Baracaldo	1088 x 896	Train
Barcelona N.	1152 x 1728	Test
Barcelona S.	896 x 1088	Test
Bilbao	576 x 832	Train
Burgos	512 x 704	Train
Cáceres	1024 x 896	Test
Cartagena	768 x 1216	Train
Castellón	1024 x 1024	Train
Córdoba	1088 x 1792	Train
Denia	640 x 768	Train
Ferrol	384 x 704	Test
Gijón	704 x 832	Test
Gerona	1536 x 1216	Train
Granada	1664 x 1600	Test
Huesca	448 x 576	Train
La Coruña	704 x 576	Train
León	1216 x 768	Train
Lérida	576 x 768	Test
Logroño	768 x 960	Train
Lugo	768 x 576	Test
Madrid N.	1920 x 2688	Train
Madrid S.	1280 x 2624	Train
Majadahonda	1472 x 1344	Test
Málaga	1024 x 1472	Train
Mérida	512 x 640	Train
Murcia	1792 x 1600	Train
Ourense	960 x 704	Train
Oviedo	960 x 896	Train
Palma	1024 x 1344	Test
Pamplona	1600 x 1536	Test
Pontevedra	384 x 512	Train
Rivas-vacía	1088 x 1088	Train
Salamanca	832 x 960	Train
San Sebastián	512 x 768	Test
Santander	1152 x 1216	Train
Sevilla	2176 x 2368	Train
Teruel	640 x 768	Test
Valencia	2304 x 1728	Test
Valladolid	1408 x 1408	Test
Vigo	704 x 1024	Train
Vitoria	576 x 896	Train
Zamora	512 x 576	Train
Zaragoza	2304 x 2752	Train

Tabla 2: Detalle del *dataset* de ciudades

parches de 128 x 128 píxeles, ya que es un tamaño suficiente y adecuado para que se pueda reconocer aquello presente en cada imagen. También es apropiado mencionar que las imágenes de entrada son redimensionadas al cuádruple de su resolución original (y, por tanto, las imágenes de salida también tienen dichas dimensiones), puesto que esta práctica implica un mejor rendimiento en la clasificación,

(a) Intervalo Junio 2018 - Septiembre 2018

(b) Intervalo Septiembre 2018 - Diciembre 2018

(c) Intervalo Diciembre 2018 - Marzo 2019

(d) Intervalo Marzo 2019 - Junio 2019

Figura 28: Imágenes temporales de San Sebastián

como se indica más adelante al detallar el modelo utilizado en los experimentos.

5.3.1. *GeoFlow*

GeoFlow [45] es una librería de Python desarrollada durante la realización de un proyecto previo al que se está tratando. Esta librería se formó como respuesta a la necesidad de automatizar las tareas de pre-procesado de los productos Sentinel para generar conjuntos de imágenes de áreas urbanas geolocalizadas y anotadas a nivel de píxel. Hasta ese momento no existía una forma de realizar dichas tareas de manera automática, ya que a pesar de existir algunas herramientas gráficas para el pre-procesado, como QGIS, suponían un tiempo excesivo si se tiene en cuenta el número de ciudades sobre las que actuar.

GeoFlow se basa en la librería EO-Learn [56]. Esta librería presenta un paradigma de programación de flujo de procesamiento: una serie de tareas que se ejecutan de forma ordenada y comparten entre sí sus parámetros y los datos que se van procesando. A continuación, se describe el orden y el objetivo de cada una de dichas tareas.

5.3.1.1 Flujo de tareas inicial

El flujo que realiza *GeoFlow* sobre cada una de las ciudades podría resumirse de la siguiente forma [45]:

1. Se realiza una petición de productos Sentinel-2 al repositorio SentinelHub en función, entre otros

parámetros, de un área de interés para delimitar el terreno, de un intervalo de tiempo para limitar las fechas de adquisición y de un porcentaje que indique la presencia de nubes permitida (cuanto menor sea, menos píxeles se verán afectados por este tipo de obstrucción).

2. Se genera una imagen en formato *GeoTIFF* por cada producto obtenido en el paso anterior. Específicamente, para las bandas RGB e Infrarrojo cercano.
3. Se crea una imagen en formato GeoTIFF como combinación de las imágenes obtenidas en el paso previo.
4. Se recorta la imagen resultado de la tarea anterior utilizando la región de interés.
5. Se realiza una petición similar a la contemplada en el punto 1, pero esta vez para productos Sentinel-1.
6. Se realiza el procesamiento de los productos descargados en la pauta previa. Esto es, se aplica el grafo de procesamiento de SNAP 'Rebotes' sobre dichos productos.
7. Se toma el resultado del paso anterior y se recorta en base a la región de interés.
8. Se extraen las geometrías de las clases definidas para la segmentación (edificios y carreteras) y se agregan como objeto georeferenciado según se indique una clase u otra.
9. Se *rasteriza* el objeto georeferenciado en el paso previo para que este resultado haga las veces de máscara en el aprendizaje automático posterior.
10. Se elabora el *dataset* a partir de los ráster generados y se guarda el resultado en disco. Esto es, combina los resultados multiespectrales de Sentinel-2 con los datos radar de Sentinel-1, y computa la banda adicional NVDI. Esto es, cada entrada al flujo conlleva un resultado provisto de 7 bandas.

5.3.1.2 Adaptación del flujo de tareas

Cabe destacar que esta librería tuvo que ser adaptada para abarcar la nueva forma de experimentación, es decir, el tratamiento de los datos mediante distintos flujos de procesamiento. Con este fin, se modificaron algunas de las tareas comentadas hasta ahora. Las más afectadas y merecedoras de mayor detalle son la sexta tarea, relacionada con el procesamiento de Sentinel-1, y la última, relacionada con la obtención de productos definitivos.

En el caso del paso para el procesamiento de Sentinel-1, hasta el momento se ejecutaba únicamente un grafo de procesamiento de SNAP, el llamado 'Rebotes'. Para la presente experimentación, debía hacerse posible obtener diferentes datos procesados partiendo de los mismos productos de entrada. Por ello, se modificó la tarea de modo que no solo se tomase el XML de 'Rebotes' sino todos aquellos grafos que fuera necesario ejecutar. Era importante también que los resultados guardados para ser tomados por la siguiente tarea estuvieran bien nombrados y no se mezclase un tipo de procesamiento con otro.

Siguiendo esta línea de obtener en una única ejecución del flujo de *GeoFlow* los distintos resultados para cada procesamiento, se cambió expresamente la última tarea encargada de guardar a disco. La necesidad de ser alterada recae en que, si bien el producto final está compuesto por 7 capas, dos de ellas provienen directamente del procesamiento de Sentinel-1. Esto conlleva que si, por ejemplo, se aplican 3 procesamientos distintos, no se quiere obtener un único producto con 11 bandas ráster, sino que se necesita guardar 3 resultados distintos con 7 bandas cada uno. Para tal fin, como ocurría en la tarea antes comentada, lo más relevante era no mezclar distintos procesamientos, sino diferenciar en cada momento unos de otros.

La ejecución de este nuevo flujo de *GeoFlow* permite, por tanto, obtener un producto tratado por cada tipo de procesamiento que se indique como entrada. Esto conlleva que ejecutando dicho flujo una vez por cada ciudad y por cada uno de los cuatro trimestres antes descritos, se complete el *dataset*.

5.4. Análisis de resultados

Una vez definidos los pormenores de la obtención del conjunto de datos implicado en el aprendizaje y la segmentación de edificios y carreteras, se plasma a continuación la forma en la que se han tratado los resultados obtenidos.

Primero, en la Sección 5.4.1, se detalla la forma de obtener las métricas y el modo en que van a ser presentadas. Después, se comenta el modo en el que se obtuvo el modelo base sobre el que comparar los nuevos tipos de procesamiento en la Sección 5.4.2. A continuación, en la Sección 5.4.3, se realiza la mencionada comparación con los nuevos resultados. Por último, se analizan en detalle aquellas ciudades que han presentado peores métricas, aportando ciertas sugerencias para mejorar dichos casos.

5.4.1. Detalle sobre las métricas

Antes de empezar a analizar los resultados, es relevante explicar en detalle qué métricas han sido empleadas para conocer el rendimiento del modelo, la forma de calcularlas y la manera de plasmarlas en forma de tabla.

Primero de todo, hay que entender que la segmentación de edificios y carreteras es un problema de clasificación multiclase. Está claro que, en este caso, un píxel concreto puede ser clasificado como edificio o como carretera o como ninguna de las dos etiquetas. Esta clasificación bien puede ser obtenida mediante un único modelo que clasifique las tres clases, o bien puede descomponerse en dos subproblemas de clasificación binaria: un modelo que detecta carreteras, otro modelo que detecta edificios y la posterior combinación de sus salidas para obtener la clasificación completa. Aunque ambas opciones tienen ventajas y desventajas, para este proyecto se considera la segunda opción siguiendo el trabajo desarrollado en [57].

Una vez definida la forma de segmentar las clases, es necesario explicar la forma de evaluar los modelos. Dicha evaluación se realiza sobre cada una de las clases, es decir, de forma individual, y también para las dos clases a la vez, es decir, de forma agregada. Para agregar las clases se sigue la pauta de siempre superponer los edificios a las carreteras.

La métrica utiliza es la IoU (*Intersection over Union*), la cual cuantifica el porcentaje de solapamiento entre la máscara deseada (*true values*, valores reales) y la obtenida en la segmentación (*predicted values*, valores predichos). Su cálculo consiste sencillamente en dividir el número total de píxeles que tienen en común la máscara real y la predicha, entre el número total de píxeles que se obtienen al unir ambas máscaras:

$$IoU = \frac{y \cap \hat{y}}{y \cup \hat{y}}$$

Específicamente, en las tablas que muestran los resultados, en todas las celdas se indica la mIoU, media de los IoUs por clase, tratándose simplemente de la IoU si se trata de métricas individuales y de la mIoU si se trata de la versión agregada. Dichas métricas se muestran redondeadas a tres cifras significativas.

La Figura 3 representa la forma de resumir las métricas resultantes. La columna 'Modo' hace referencia al tipo de métrica que se está presentando, es decir, si esta es 'Individual' significa que se ha obtenido teniendo en cuenta una única clase, y si es ' Agregado' se trata de métricas para ambas clases de forma unificada. En el caso de la clase, puede indicarse respecto a una de ellas, véase 'Edificio' o 'Carretera', o puede contemplarse el promedio de los resultados obtenidos por separado bajo el mismo nombre 'Promedio'. A la hora de plantear las pruebas o experimentos, constará como indicador un número 'i', el cual denota el número de referencia del experimento en concordancia con el tipo de procesamiento que se haya realizado para Sentinel-1. Se recuerda que la relación entre la numeración y el procesamiento correspondiente ha sido presentada anteriormente en la Sección 5.2.1.

En cuanto a los colores presentes en las tablas completas, se define un color crema para denotar los valores finales a tener en cuenta para cada modelo (el valor promedio del agregado para cada experimento) y un color verde para remarcar la métrica más prometedora en comparación con las que se estén evaluando. Para facilitar el seguimiento de las métricas también se indican en formato *negrita* los mejores valores por cada clase.

Modo	Clase	i
Individual	Edificio	
	Carretera	
	Promedio	
Agregado	Edificio	
	Carretera	
	Promedio	

Tabla 3: Ejemplo de la disposición de una tabla de resultados

5.4.2. Experimentación base

Como ya ha sido comentado anteriormente, los experimentos diseñados se basan en los ya realizados en un proyecto previo. De hecho, el modelo con mejor rendimiento de esa experimentación previa ha sido tomado como modelo base para la actual valoración de los resultados. Este modelo base implica el entrenamiento y segmentación con la red neuronal ResNet-34 + U-Net antes mencionada, pero además contempla otra serie de disposiciones para los datos.

Para comprender en detalle todo lo que implica el llamado 'modelo base', se detallan seguidamente las pautas que fueron teniéndose en cuenta en la experimentación previa. Cabe destacar que los siguientes puntos solo corresponden a configuraciones que mejoraban el modelo diseñado hasta el momento (experimentación completa en [58]):

1. Se trata del primer modelo implementado durante la experimentación previa. Se basa en la red neuronal compuesta por ResNet-34 + U-Net utilizando la función de coste *Binary Crossentropy* [59] (habitual en los problemas de clasificación binaria).

2. Como ya se ha comentado, las imágenes son tomadas por parches. Hasta el momento, cada imagen quedaba limitada a una única posible malla o *grid*. Sin embargo, se demostró que permitir mayor diversidad en este aspecto mejora el rendimiento. Esto es así debido a que la red no memoriza ejemplos, de forma que evita el sobreaprendizaje y, por extensión, mejora su capacidad de generalizar en el caso de las imágenes de *test*.

3. El uso de los parches también dio pie a preguntarse sobre la verdadera utilidad que podían tener algunos de ellos para el aprendizaje. Cada imagen queda troceada, pero no se había comprobado si algunos de esos parches no aportaban información relevante sobre alguna de las clases. Por ejemplo, un parche que no tenga ningún edificio no aporta información para esa clase. Del mismo modo, un parche con alta presencia de nubes entorpeciendo la visualización del terreno tampoco aporta ningún beneficio (Figura 29).

   Para evitar que estos casos puedan impedir optimizar el aprendizaje, se incorporan las máscaras de validación. Se trata de máscaras, similares a las de clasificación, en las cuales cada píxel representa una área de 64 x 64 píxeles de la imagen Sentinel. Para denotar la validez de cada parche en concreto se establece la leyenda en escala de grises de la Tabla 4. Además, en la Figura 30 pueden verse dos ejemplos reales de máscaras de validación para dos ciudades distintas.

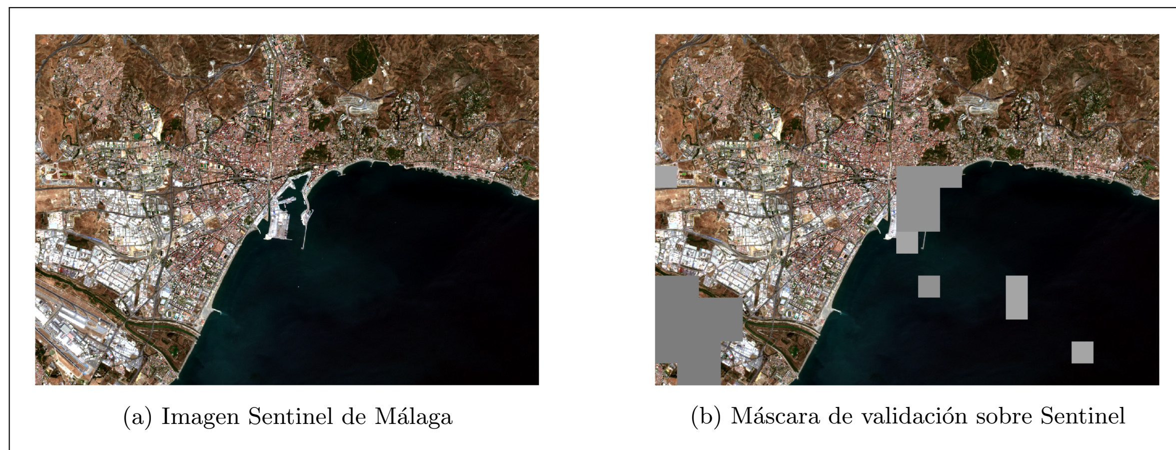
   Cabe destacar que la generación de estas nuevas máscaras es manual y requiere de un estudio riguroso de los datos presentes en cada imagen. Tratándose de un proceso manual existe el riesgo

(a) Presencia de nubes (b) Terrenos agrícolas (c) Zona de mar (barcos)

Figura 29: Ejemplos de parches sin información relevante para el aprendizaje

ID	Descripción	Código de color	Color
0	Válido	#ffffff	
1	Nube	#e1e1e1	
2	Sin rebotes	#cdcdcd	
3	Sin puente	#b9b9b9	
4	Flash	#a5a5a5	
5	Puertos	#919191	
6	Aeropuerto	#7d7d7d	
7	Sin edificio	#696969	
8	Cementerio	#555555	
9	Sombra de nube	#373737	

Tabla 4: Leyenda para las máscaras de validación



(a) Imagen Sentinel de Málaga (b) Máscara de validación sobre Sentinel

Figura 30: Ejemplo de máscara de validación

de que se produzcan fallos humanos a la hora de definir las máscaras, como el hecho de clasificar como válido un parche que no lo sea o viceversa. Como añadido, y debido a la actual presencia de píxeles válidos y no válidos sin un orden preestablecido, fue necesario redefinir la manera de obtener los parches en los que se divide la imagen [60].

4. La función de coste queda sustituida por una combinación de *Focal Loss* [61] con *Dice Loss* [62]. Esta forma de medir la evolución del aprendizaje obtiene mejores resultados que la inicial *Binary Crossentropy*, puesto que proporciona resultados de mayor calidad.

5. Se valora la posibilidad de aumentar la resolución de las imágenes, ya sea a la entrada o a la

salida del modelo para comprobar si aportan mejores resultados. Se constata que se obtiene mejor rendimiento al cuadruplicar la resolución de las imágenes de entrada mediante la técnica de *nearest-neighbour* o vecino más cercano.

6. Por último, se señala la posibilidad de que, debido a que el modelo debe entrenar millones de parámetros, la cantidad de imágenes en el *dataset* no es suficiente para conseguir un buen aprendizaje. La forma más útil de ampliar los datos y mejorar el rendimiento, para este caso en concreto, es la de aplicar transformaciones afines a las imágenes. Lo cual implica combinar rotaciones de 90º con volteos horizontales y verticales de las imágenes, evitando las repeticiones por posiciones equivalentes.

Cada una de esas disposiciones del modelo, fueron evaluadas y comparadas como se resume en la Tabla 5. Se indica *P* como referencia a *Previo* y la numeración es referida a cada uno de los puntos recién explicados.

Resultados Experimentos Previos							
Modo	**Clase**	**P1**	**P2**	**P3**	**P4**	**P5**	**P6**
Individual	Edificio	0,473	0,552	0,491	0,556	0,568	**0,699**
	Carretera	0,498	0,498	0,533	0,524	0,589	**0,689**
	Promedio	0,485	0,524	0,511	0,540	0,578	**0,694**
Agregado	Edificio	0,473	0,552	0,491	0,556	0,568	**0,699**
	Carretera	0,476	0,467	0,513	0,561	0,537	**0,636**
	Promedio	0,475	0,510	0,502	0,559	0,553	**0,668**

Tabla 5: Resumen de las métricas para los experimentos previos

El experimento denotado por ‘P6’ es el que mejor rendimiento obtiene. De hecho, esta última configuración se trata del mencionado ‘modelo base’ con el que comparar los resultados de los nuevos experimentos.

En definitiva, el modelo final de estos experimentos previos queda definido como el compendio de los puntos anteriores: una red neuronal U-Net combinada con ResNet-34, utilizando la función de coste *Focal Loss* + *Dice Loss*, que toma imágenes Sentinel al cuádruple de su resolución original y que han sido repetidas para diferentes disposiciones geométricas, junto con máscaras de validación (además de las de clasificación). Este modelo pasa a ser el ‘modelo base’ para los experimentos de este proyecto.

5.4.3. Comparación de nuevos resultados

Los experimentos basados en diferentes procesamientos se ejecutaron utilizando el modelo base y el *dataset* de ciudades ya comentado. Seguidamente, se muestra en la Tabla 6 un resumen de las métricas resultantes para cada nuevo experimento. Cada denominación *E* corresponde a la abreviación de *Experimento* (‘E1’ significa ‘Experimento 1’). Nótese que el primer experimento corresponde al tratamiento mediante ‘Rebotes’, luego sus resultados coinciden con los presentados para el último experimento del proyecto previo, esto es, ‘P6’ en la figura anterior es ‘E1’ en la actual.

Como puede verse en las métricas resultantes, ningún experimento nuevo ha logrado superar el rendimiento del modelo utilizado hasta el momento. Esto quiere decir que ninguno de los procesamientos nuevos ha mejorado las métricas conseguidas por el procesamiento ‘Rebotes’. El experimento más cercano al ‘E1’, en razón de buenas métricas, es el cuarto experimento: 0,668 y 0,661 de IoU en edificios y carreteras, respectivamente (diferencia del 1,059 %). De hecho, si se comparan en detalle se pueden percibir mejor sus diferencias: individualmente, para la clase ‘Edificio’ difieren en un 0,431 % (0,699 respecto de 0,696) y para ‘Carretera’ un 2,990 % (0,689 respecto de 0,669).

Resultados Experimentos Procesamiento Sentinel-1										
Modo	**Clase**	**E1**	**E2**	**E3**	**E4**	**E5**	**E6**	**E7**	**E8**	**E9**
Individual	Edificio	**0,699**	0,693	0,684	0,696	0,694	0,691	0,687	0,682	0,678
	Carretera	**0,689**	0,634	0,644	0,669	0,663	0,653	0,653	0,652	0,684
	Promedio	**0,694**	0,663	0,664	0,682	0,679	0,672	0,670	0,667	0,681
Agregado	Edificio	**0,699**	0,693	0,684	0,696	0,694	0,691	0,687	0,682	0,678
	Carretera	**0,636**	0,582	0,594	0,625	0,614	0,610	0,615	0,617	0,619
	Promedio	**0,668**	0,638	0,639	0,661	0,654	0,651	0,651	0,650	0,649

Tabla 6: Resultados de los experimentos sobre el procesamiento de Sentinel-1

Si se analizan los resultados por clases individuales, el tratamiento correspondiente al experimento 9 es el menos aconsejable a la hora de clasificar edificios (0,678, un 3,097 % peor que el mejor resultado individual para esa clase), mientras que para segmentar carreteras debería evitarse el procesamiento relacionado con el experimento 2 (0,634, un 8,675 % peor que el mejor resultado individual para dicha clase). De hecho, este mismo experimento 2, consistente en Rebotes + Speckle-Filter (Gamma Map, 5x5), es el que peor resultado obtiene en la métrica final correspondiente al promedio de las clases agregadas (0,638, un 4,702 % por debajo del mejor promedio agregado).

Por tanto, parece que la ejecución del procesamiento de ‘Rebotes’ sobre imágenes Sentinel-1 es la mejor opción para la segmentación de edificios y carreteras.

Aunque, como se ha dicho, no existe un procesamiento que se comporte mejor entre los nuevos diseños, sí es cierto que el más próximo al primer experimento es el ‘E4’. Como muestra de la buena predisposición de este mismo experimento, se puede encontrar en la Figura 31 la predicción realizada para la ciudad de Gijón. Dicha ciudad es la que mejores resultados ha obtenido en dicho caso. Es importante saber que en color amarillo se representa las zonas en las que existe tanto predicción por parte del modelo como etiquetado en la máscara correspondiente (aciertos); en color salmón se muestra únicamente el etiquetado real de las clases (zonas no etiquetadas por el modelo pero que deberían estarlo) y en color verde se representan las zonas de falsos positivos, es decir, edificios o carreteras que han sido etiquetados como tal pero que no existen en la máscara real. Para interpretar bien dichos falsos positivos es importante comprobar que las máscaras utilizadas no se encuentran desactualizadas.

(a) Imagen Sentinel (b) Predicción de edificios (c) Predicción de carreteras

Figura 31: Ejemplo de predicción del experimento 4 sobre una región de Gijón

Dado que, aunque no es una mejor opción que el experimento 1, se postula como el siguiente mejor procesamiento, se diseña una nueva versión sobre el experimento 4. Esto es, se quiere realizar una nueva ejecución del mismo pero esta vez con más recursos: aumentando el número de GPUs en

uso. Cabe la posibilidad de que al ser capaz la red neuronal de tomar más ejemplos por cada iteración (tamaño de *batch*), se produzca una mejor generalización, ya que esto aumenta la diversidad de las imágenes en análisis.

Los datos obtenidos para esta nueva ejecución se muestran en la Tabla 7 en comparación con los datos resultantes del experimento 4. Asimismo, se muestran los resultados tanto para el experimento 1 antes realizado como para el experimento 1 en calidad de múltiples GPUs [63].

GPU vs. MultiGPU					
Modo	**Clase**	**E1**	**E1+MultiGPU**	**E4**	**E4+MultiGPU**
Individual	Edificio	0,699	**0,717**	0,696	0,706
	Carretera	0,689	**0,718**	0,669	0,686
	Promedio	0,694	**0,717**	0,682	0,696
Agregado	Edificio	0,699	**0,717**	0,696	0,706
	Carretera	0,636	**0,644**	0,625	0,632
	Promedio	0,668	**0,681**	0,661	0,669

Tabla 7: Comparación de resultados respecto a número de GPUs

Se puede apreciar que el resultado promedio de las clases agregadas es mejor en los casos con varias GPUs. Si se compara el 'E1' en sus dos vertientes, el caso MultiGPU supera al anterior en un 1,946 % (0,681 respecto a 0,668), y para sus clases de forma individual un 2,575 % en el caso de 'Edificio' y un 4,209 % en el de 'Carretera'. Igualmente, el 'E4' es superado por su versión para múltiples GPUs en un 1,210 % (0,669 frente a 0,661), al igual que para las clases individualmente: una diferencia del 1,437 % en la clase 'Edificio' y del 2,541 % en la clase 'Carretera'.

Si bien los experimentos 'E1' y 'E4' ya se diferenciaban claramente, la inclusión de las GPUs supone ampliar todavía más la distancia entre ambas opciones. No solo se hace visible al comparar las clases de manera individual (un 1,558 % en favor de E1+MultiGPU para la clase 'Edificio' y un 4,665 % en favor de la misma para la de 'Carretera'), sino también y de forma más acentuada al comparar su métrica promedio agregada: un 0,681 de E1+MultiGPU frente al 0,669 de E4+MultiGPU (un 1,794 % de diferencia).

Los resultados presentados parecen indicar que la ejecución por medio del uso de varias GPU (tres unidades, concretamente) mejora el rendimiento de la segmentación. Cabe destacar que existe normalmente una gran semejanza entre la clasificación de edificios y la de carreteras de forma individualizada, la cual empeora al superponer los edificios a las carreteras (al agregar las clases). Está claro que la clase 'Edificio' no pierde precisión por la manera de superponer las clases, pero se pierde mucho acierto en el caso de la clase 'Carretera', lo cual impide que los modelos consigan un promedio agregado más alto. El caso E1+MultiGPU acentúa más esta situación, ya que la clase 'Carretera' desciende hasta en un 10,181 % al pasar de métrica individual a agregada.

En resumen, según los experimentos realizados los nuevos procesamientos con operaciones SNAP para las imágenes Sentinel-1 no parecen resultar una mejor opción que el ya utilizado 'Rebotes'. También tienden a indicar que el empleo de múltiples unidades de procesamiento gráfico favorece la obtención de mejores métricas en la segmentación semántica de edificios y carreteras.

Se hacer notar que, aunque los nuevos grafos de procesamiento, en un principio, mejoran la calidad de los datos, esta mejora no se ve reflejada a la hora de segmentar las imágenes. Para intentar comprender la posible razón de este hecho, primero hay que analizar cómo actúan tanto las nuevas operaciones de SNAP como la red neuronal convolucional utilizada.

Dichas redes neuronales aplican una serie de filtros convolucionales cuyos parámetros o pesos se van aprendiendo con el objetivo de maximizar el porcentaje de acierto a la hora de segmentar las imágenes. Por otro lado, las operaciones de SNAP eliminan el ruido presente en los datos en bruto (*raw data*) mediante una serie de convoluciones caracterizadas, entre otros, por su tamaño de ventana

(las ya mencionadas dimensiones 5x5, 7x7 y 9x9). Los pesos para este tipo de filtros están prefijadas dentro de la herramienta, de modo que eliminan solo el ruido específico que quiera tratarse. Por tanto, la diferencia recae en que, en un caso, se optimizan dichos pesos para la casuística concreta y, en el otro, se utilizan los mismos en todo momento. Este hecho, unido a la falta de mejora del rendimiento al ejecutar las nuevas operaciones, puede llevar a pensar que la red neuronal convolucional es capaz de conseguir una mejor disposición de esos filtros convolucionales sobre *raw data* que sobre datos ya tratados mediante filtros estáticos no optimizados, debido a que en los segundos puede existir una pérdida de información que la red puede aprovechar.

La formulación de esta hipótesis y su posible confirmación mediante experimentación más minuciosa se puede postular como continuación al trabajo ya iniciado en este proyecto. Esta posibilidad, al igual que otras muchas que pueden derivarse del tratamiento de Sentinel-1 y de la segmentación semántica de imágenes radar, se comentan posteriormente.

5.4.4. Estudio de peores resultados

Dados los resultados detallados previamente, es lógico afirmar que sigue habiendo un amplio margen de mejora en la clasificación. De modo que, como forma de conocer aquellos casos en los que la red erra de manera continuada, se analizan en profundidad y de forma individualizada aquellas ciudades con peores métricas.

Para empezar, se presentan tres ciudades cuyos resultados de mIoU para la clase 'Edificio' están por debajo de 0,5 para todos los trimestres considerados y todos los procesamientos vistos.

Después, y siendo el último caso relevante, se encuentra una única ciudad con malas predicciones para el trimestre comprendido entre diciembre de 2018 y marzo de 2019, y para todos los procesamientos. En esta ocasión, llama la atención la clase 'Carretera'.

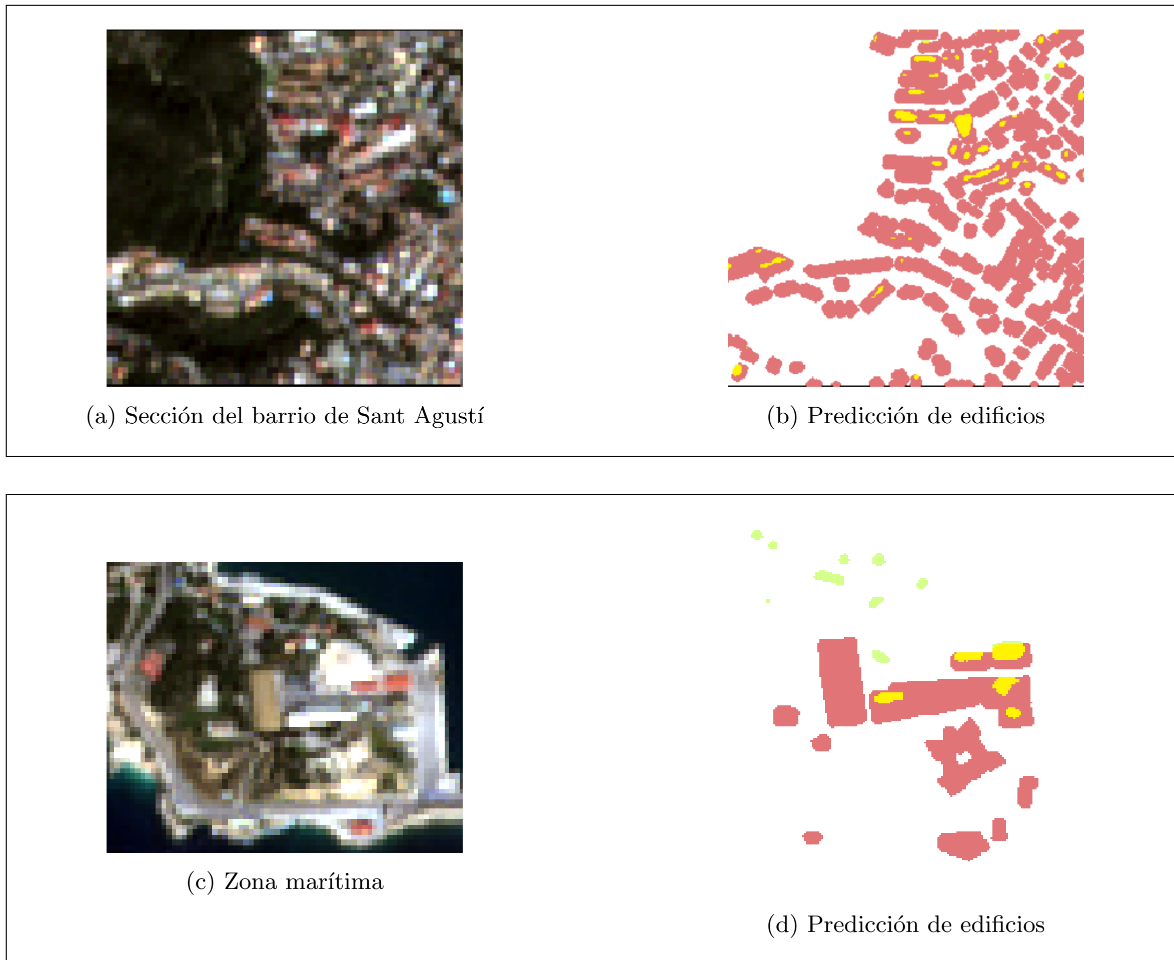
Siendo hechos tan recurrentes, no solo se intentan analizar las razones de su mala predicción, sino que se muestran ejemplos de su evaluación en imágenes y se plantean posibles pautas para solucionar su situación.

En relación a las predicciones que se muestran visualmente, el color salmón representa las etiquetas reales (o falsos negativos) y el amarillo las predichas correctamente (o verdaderos positivos). Además, puede aparecer el color verde para denotar una zona de falsos positivos, es decir, o bien edificios o carreteras que han sido etiquetados como tal pero que no existen en la máscara real, o bien predicciones que ocurren dentro de zonas invalidadas. Por último, si se muestran fracciones con fondo negro quiere decir que no son válidas, mientras que si mantienen el fondo blanco, se trata de parches válidos.

5.4.4.1 Palma

En el caso de la ciudad de Palma, en aquellas zonas de la imagen Sentinel denotadas como válidas (máscara de validación de edificios) se visualizan gran cantidad de edificios. Sin embargo, al analizar las predicciones se halla muy poco acierto en ellas. En la Figura 32 puede verse la diferencia entre los edificios presentes y la predicción acertada en dos zonas diferentes de la ciudad.

Esta mala predicción generalizada no parece atender a una razón específica, puesto que ni existen incongruencias en las máscaras de clasificación ni de validación, ni se trata de zonas de terreno inusuales.



(a) Sección del barrio de Sant Agustí (b) Predicción de edificios

(c) Zona marítima (d) Predicción de edificios

Figura 32: Predicción de edificios en zonas de Palma

5.4.4.2 Cáceres

En esta ciudad existen zonas válidas con poca presencia de edificios y en general son bien predichos. Sin embargo, existen dos regiones muy delimitadas que pueden estar afectando negativamente al rendimiento global. Estos casos se detallan más en profundidad para analizar su razón de ser.

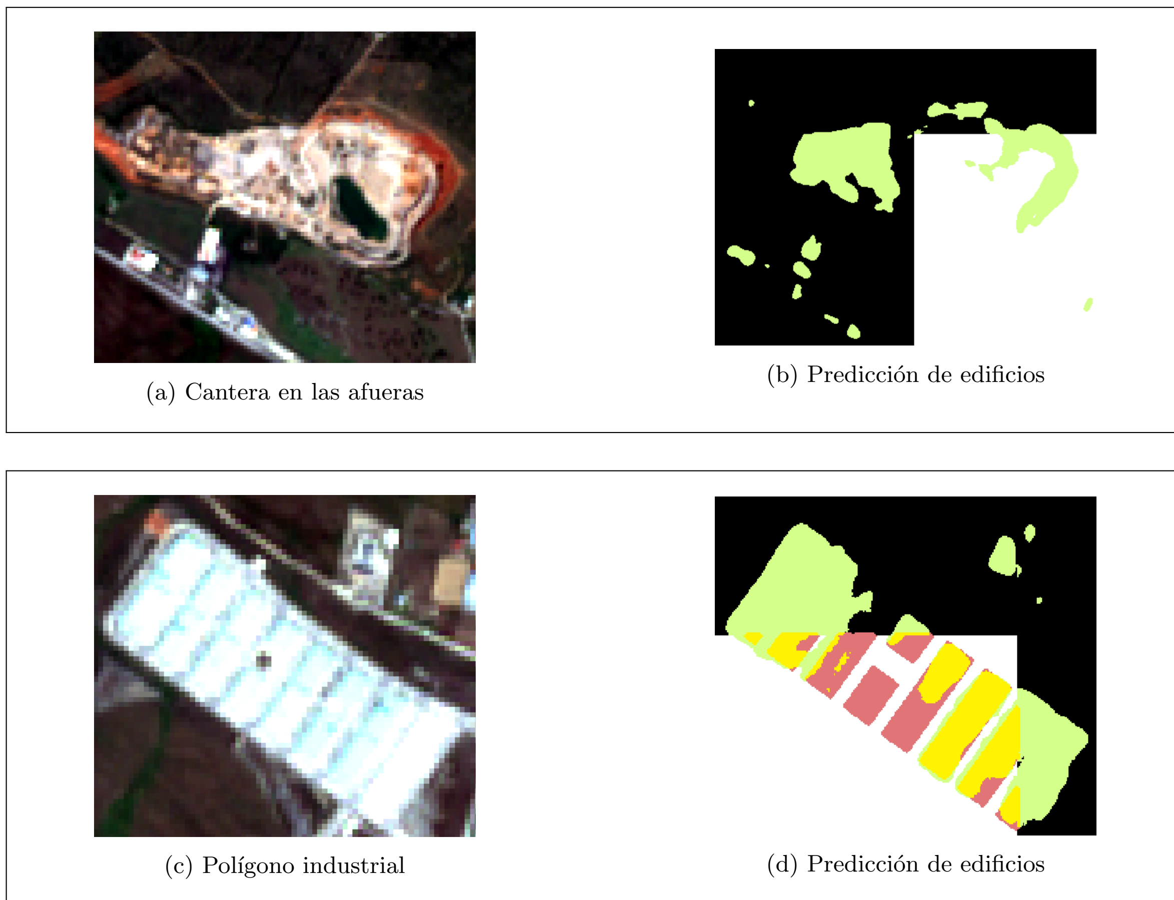
Por un lado, existe una zona válida llena de falsos positivos. En dicha región se halla una cantera (Figura 33a), la cual alberga ciertas edificaciones provisionales para llevar a cabo su explotación. La red está detectando estos edificios y etiquetándolos como tal, pero en la máscara de clasificación no están contemplados. Aunque parezca entonces que la máscara es errónea por no albergar estos edificios, se trata de construcciones provisionales que normalmente se descartan para el aprendizaje. De hecho, en la máscara de validación ya aparece parte de esa cantera como invalidada pero no toda ella (Figura 33b). Esto denota que debería actualizarse la máscara, incluyendo esos parches de la cantera como no válidos. De esa manera, tampoco las métricas se verían tan penalizadas.

Este caso de la cantera también abre la posibilidad de añadir una nueva etiqueta a los parches en escala de grises ya explicados, como son los que invalidan zonas con puertos o con nubes. De este modo, no solo quedaría como no válida sino que estaría siendo indicada de forma concreta como 'Cantera' en el tono de gris escogido para ello. Bien es cierto que no se ha dado en ninguna otra imagen del *dataset* la presencia de una cantera dentro del área de interés a la que es recortada la

imagen Sentinel-1, pero eso no quiere decir que para otras ciudades no pueda ocurrir. Por lo tanto, es una opción a tener en cuenta.

Por otro lado, existe una zona mal predicha en la que se encuentra un polígono industrial (Figura 33c). Sobre el mismo se dan ciertas predicciones correctas pero sobre todo destacan los falsos negativos, es decir, zonas en las que existen edificios pero que no han sido etiquetados por el modelo (Figura 33d).

Es particular la forma en la que se comporta el modelo respecto a estas predicciones, ya que los edificios siguen un patrón muy distinguible y, en cambio, solo reconoce parte de ellos. Se formula la hipótesis de que haya caminos internos entre los almacenes, que ni siquiera puedan considerarse carreteras, que creen confusión en la red respecto a los límites de cada una de las naves.



(a) Cantera en las afueras

(b) Predicción de edificios

(c) Polígono industrial

(d) Predicción de edificios

Figura 33: Predicción de edificios en zonas de Cáceres

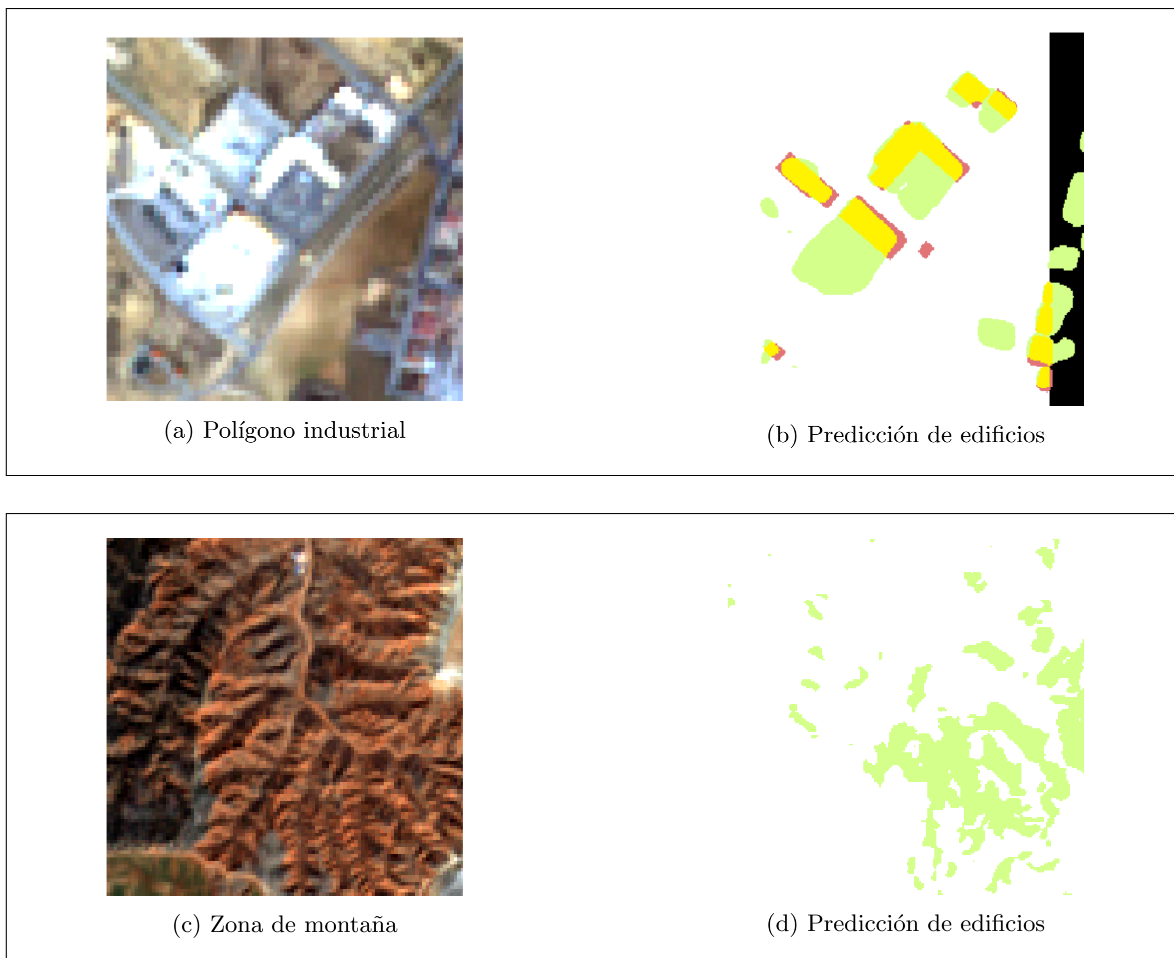
5.4.4.3 Teruel

Siguiendo con las incidencias en los términos donde se encuentran polígonos industriales, Teruel presenta el caso inverso al anterior (Figura 34a). Se encuentran muchos falsos positivos, es decir, predicciones de edificios que no existen (Figura 34b). Esta vez parece algo más claro que el polígono cuenta con patios interiores que son detectados erróneamente como edificios. Estos casos tan particulares parecen difíciles de resolver, ya que para poder discernir entre patio interior y edificio en lugares como el que se está analizando, quizá tuviese que realizarse todo un entrenamiento centrado en esa diferencia. Es decir, puede requerirse mayor presencia de parches con esta disposición en los datos de

entrada al modelo.

Además del polígono industrial, se analiza una zona de montaña (Figura 34d). Se da el caso de predicciones de edificios donde no existe ningún tipo de urbanización (Figura 34c). Se trata de una zona de monte en la que no hay ninguna edificación visible. Este problema afecta considerablemente al desempeño del modelo, por lo que debería encontrarse una solución factible para que no se repita.

Una posible causa de la situación de las montañas es que a la hora de entrenar el modelo se balancean las clases de entrada de forma que si se está prediciendo para la clase 'Edifico', como es el caso actual, solo se toman parches que contienen como mínimo un 10 % de píxeles con edificios. Para la clase 'Carretera' se aplica lo mismo pero con los píxeles correspondientes. Para casos en los que no existan edificios, como el actual, en lugar de haber aprendido de zonas que siempre tengan un mínimo de edificios, quizá podría ser beneficioso aprender también de zonas sin edificio alguno pero que alberguen un mínimo de, por ejemplo, el 5 % de carreteras. De esta forma, aquellas zonas sin ningún tipo de edificación podrían ser mejor interpretadas, como la de la montaña.



(a) Polígono industrial

(b) Predicción de edificios

(c) Zona de montaña

(d) Predicción de edificios

Figura 34: Predicción de edificios en zonas de Teruel

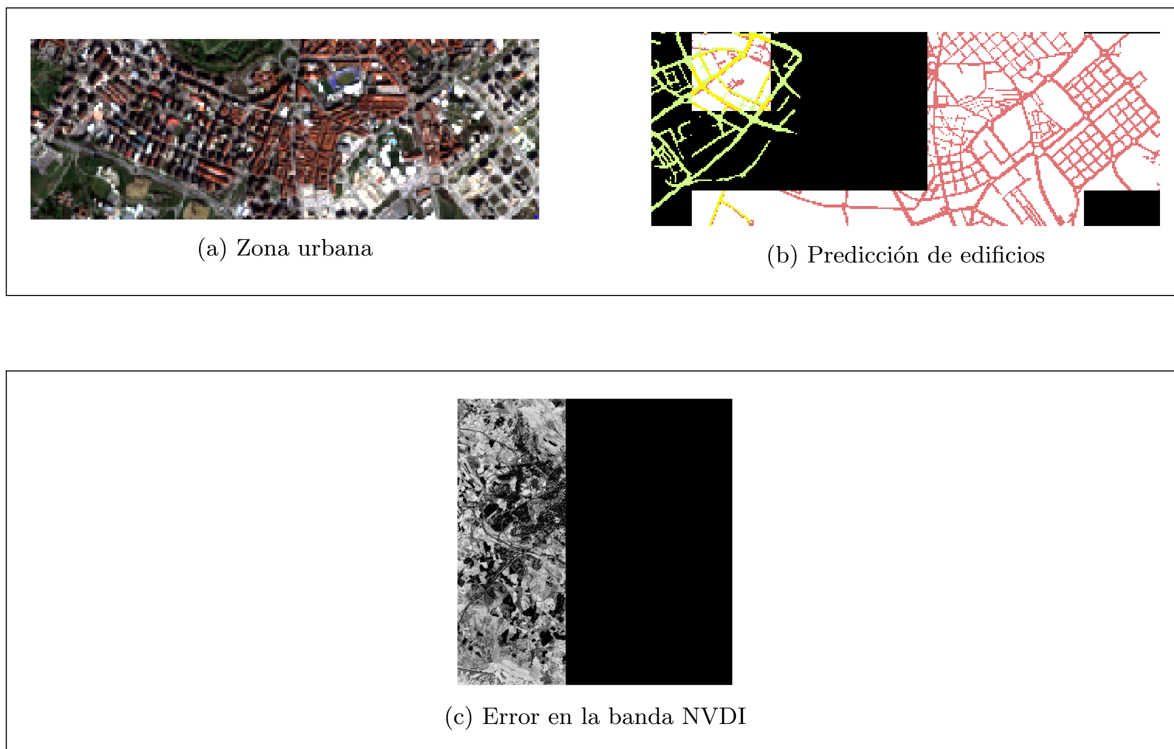
5.4.4.4 Pamplona

Esta ciudad destaca por sus malas métricas para la clase 'Carretera'. En un principio, al visualizar las predicciones se contemplan buenas prácticas pero parece existir una especie de corte a

partir de la mitad derecha de la imagen (Figura 35b). Ese corte se percibe por el hecho de que no hay ni una sola predicción en la parte derecha de la imagen, solo falsos negativos. Se busca la causa de esta falta de datos en esa zona y se encuentra mediante la inspección del producto capa por capa: la banda 5, referente a la banda NVDI, no se ha generado adecuadamente (Figura 35c).

Este hecho implica que, si bien en las zonas correctamente generadas sí que se estaba realizando una predicción bastante acertada, la imposibilidad del modelo de predecir en la zona no existente provoca que los resultados decaigan de la forma que lo han hecho.

La solución a esta situación es la generación correcta del producto, revisando que la NVDI se crea completa. Este fallo puede darse, aunque no resulta frecuente, al generar el *dataset* mediante *GeoFlow*. Justamente para evitar estas situaciones, todos los productos son revisados al detalle, tanto para identificar bandas mal creadas como para generar las propias máscaras de validación. Sin embargo, al tratarse de una comprobación manual, existe mayor predisposición para que se produzcan fallos humanos. En el caso de Pamplona, el fallo recae en no haber identificado la banda mal generada antes de utilizar el producto para la predicción.



(a) Zona urbana

(b) Predicción de edificios

(c) Error en la banda NVDI

Figura 35: Predicción de edificios en zona de Pamplona

6. Conclusiones y líneas futuras

En este trabajo se ha estudiado la aplicación de imágenes satélite de tipo radar para la clasificación automática de edificios y carreteras. Específicamente, se han evaluado diversos tratamientos sobre imágenes Sentinel-1 en combinación con los datos aportados por el sátelite Sentinel-2. Dicha evaluación ha consistido en la ejecución de una serie de experimentos. En base a los resultados obtenidos en ellos, se han destacado ciertos flujos de procesamiento frente a otros según su implicación en la mejora de la segmentación de las imágenes. Para hacer posible el diseño de la experimentación, se han estudiado los posibles flujos de procesamiento a ejecutar sobre Sentinel-1, se han sopesado varias herramientas para el procesamiento de las imágenes y se ha desarrollado una librería en código *Python* con el fin de albergar toda la funcionalidad necesaria para dicho tratamiento.

El comentado análisis de los resultados de la experimentación sugiere varias conclusiones importantes y posibles líneas de trabajo a desarrollar en relación con este proyecto.

Para empezar, las métricas obtenidas por cada uno de los posibles procesamientos indican que el flujo definido como 'Rebotes' sigue siendo una mejor opción para la clasificación de edificios y carreteras que los nuevos flujos diseñados. Aún así, la investigación de nuevas operaciones de SNAP para mejorar la segmentación no debería abandonarse en base a esta única experimentación. En cambio, se abren varias posibilidades al respecto. Una de ellas es profundizar en la forma en que se ejecutan las operaciones 'Speckle-Filter' y 'ThermalNoiseRemoval', tanto en relación a sus múltiples parámetros como a la confirmación de si la red neuronal actúa mejor sin utilizar dichas operaciones debido a su carácter convolucional. Otra alternativa de trabajo es analizar nuevas operaciones de procesamiento de imágenes Sentinel-1 que puedan resultar más adecuadas para el caso concreto de segmentación semántica, ya que SNAP presenta una amplia variedad de operaciones todavía por estudiar.

Según las métricas obtenidas, también se considera que en futuras fases de experimentación puede ser una buena práctica utilizar varias unidades de procesamiento gráfico, es decir, ejecutar los experimentos como en la versión MultiGPU. Además, cabe recordar que ciudades concretas de las empleadas en el *dataset* han presentado inconvenientes a la hora de realizarse sobre ellas una buena clasificación. Por esta razón, si se utilizan los mismos datos en futuros experimentos, sería adecuado asegurar la buena disposición de dichas imágenes, de modo que los resultados no se vean afectados negativamente por errores ya conocidos.

En cuanto a la librería de *Python* desarrollada sobre *snappy*, se ha podido comprobar durante la realización de este proyecto que agiliza considerablemente la manera de definir los grafos de procesamiento. A pesar de constituir un *wrapper* válido y que alberga las funcionalidades más importantes del diseño de grafos mediante la herramienta SNAP, puede admitir ciertas modificaciones. En relación a este hecho, ya se presentaron en la Sección 4.3 las posibles mejoras para una segunda versión de la librería.

Además de dichas actualizaciones internas, puede ser muy favorable la integración completa de esta librería con la de *GeoFlow*. De este modo, el *wrapper* podría utilizarse directamente en las ejecuciones del flujo de tareas de *GeoFlow* sin necesidad siquiera de que existan grafos predefinidos en XML. Esto es, que en la tarea referente al tratamiento de imágenes Sentinel-1, y en base a cierta configuración en forma de argumentos de entrada, se creen flujos de procesamiento al vuelo. Es más, podría ser de interés, si 'Rebotes' continúa suponiendo el punto común entre los futuros procesamientos, que su diseño puede generarse como base por defecto para agilizar todavía más la generación de nuevas variedades de flujos.

En definitiva, el procesamiento de imágenes Sentinel-1 para su aplicación en la segmentación semántica de edificios y carreteras en combinación con Sentinel-2 implica una amplia investigación, la cual ha sido apenas iniciada con la realización de este proyecto.

Índice de figuras

1. Representación de la familia de satélites Sentinel [2] 10
2. Distribución de la captura de datos y las polarizaciones de Sentinel-1 [8] 11
3. Modos de adquisición en Sentinel-1 [9] 12
4. Representación de las distintas polarizaciones: (a)Polarización HH, (b)Polarización VV, (c)Polarización VH, (d)Polarización HV [11]. 12
5. Ilustración de una matriz ráster [13]. 13
6. Visualización de datos ráster. 13
7. Comparación de la resolución espacial en imágenes SAR. (izq.) Imagen de baja resolución: píxel de 20x20 metros. (drcha.) Imagen de alta resolución: píxel de 1x1 metros [14]. 14
8. Interfaz gráfico de QGIS: visualización sobre mapa de un producto Sentinel-1 de una región de la provincia de Zaragoza. 15
9. Cuadro de diálogo en QGIS con la información del producto en estudio 15
10. Ejemplo de comprobación de regiones para un producto concreto. En rosa, la representación de la superficie que ocupa la imagen satélite. En amarillo y en verde, dos posibles definiciones de *subsets* que cubran un cuarto de la superficie total. 16
11. Interfaz gráfico de SNAP: visualización de un producto Sentinel-1 mediante dos de sus bandas ráster y la combinación de las mismas en formato RGB. 17
12. Cuadro de diálogo para la creación de grafos de procesamiento. 17
13. Opción de código en la herramienta *Graph Builder* 18
14. Variantes de un mismo grafo en el *Graph Builder* 19
15. Muestra de productos sobre la superficie de Austria 22
16. Visualización de una imagen RGB (calidad igual a la del *notebook*) 22
17. Evolución del diseño de un *Grafo* mediante el *wrapper* de *snappy* 32
18. Evolución del diseño de un grafo mediante el interfaz de SNAP 33
19. Rediseño del flujo de un *Grafo* mediante el *wrapper* de *snappy* 33
20. Rediseño del flujo de un grafo mediante el interfaz de SNAP 34
21. Grafo de procesamiento inicial, ‘Rebotes’ 38
22. Nuevo grafo de procesamiento con ‘ThermalNoiseRemoval’ 40
23. Nuevo grafo de procesamiento con ‘Speckle-Filter’ 40
24. Nuevo grafo de procesamiento con ‘ThermalNoiseRemoval’ y ‘Speckle-Filter’ 40
25. Aplicación de nuevos procesamientos en Huesca 42
26. Arquitectura de una red neuronal U-Net [51] 43
27. Distribución geográfica del conjunto de datos 44

28. Imágenes temporales de San Sebastián 46
29. Ejemplos de parches sin información relevante para el aprendizaje 50
30. Ejemplo de máscara de validación 50
31. Ejemplo de predicción del experimento 4 sobre una región de Gijón 52
32. Predicción de edificios en zonas de Palma 55
33. Predicción de edificios en zonas de Cáceres 56
34. Predicción de edificios en zonas de Teruel 57
35. Predicción de edificios en zona de Pamplona 58

Índice de tablas

1. Comparativa de funcionalidades: OST vs snappy 24
2. Detalle del *dataset* de ciudades 45
3. Ejemplo de la disposición de una tabla de resultados 49
4. Leyenda para las máscaras de validación 50
5. Resumen de las métricas para los experimentos previos 51
6. Resultados de los experimentos sobre el procesamiento de Sentinel-1 52
7. Comparación de resultados respecto a número de GPUs 53

Referencias

[1] Christian Ayala Lauroba. *A Deep Learning Approach to Land Use Classification in High Resolution Satellite Imagery*. 2020.

[2] European Space Agency. *Copernicus Overview*. https://www.esa.int/Applications/Observing_the_Earth/Copernicus/Overview4.

[3] European Space Agency. *Copernicus Open Access Hub*. https://scihub.copernicus.eu/. 2014.

[4] European Space Agency. *Copernicus (European Commission's Earth Observation Programme / formerly GMES*. https://earth.esa.int/web/eoportal/satellite-missions/c-missions/copernicus. 2018.

[5] European Space Agency. *Sobre Copernicus*. https://www.copernicus.eu/es/sobre-copernicus.

[6] European Space Agency. *Mission Objectives*. https://sentinel.esa.int/web/sentinel/missions/sentinel-1/mission-objectives.

[7] NASA: EarthData. *What is Synthetic Aperture Radar?* https://earthdata.nasa.gov/learn/backgrounders/what-is-sar. 2020.

[8] European Space Agency. *Sentinel-1 Observation Scenario*. https://sentinel.esa.int/web/sentinel/missions/sentinel-1/observation-scenario.

[9] European Space Agency. *Instrument Payload*. https://sentinel.esa.int/web/sentinel/missions/sentinel-1/instrument-payload.

[10] NASA. *NASA ARSET: Basics of Synthetic Aperture Radar (SAR), Session 1/4*. https://www.youtube.com/watch?v=Xemo2ZpduHA&ab_channel=NASAVideo. 2018.

[11] European Space Agency. *Dual Polarisation*. http://envisat.esa.int/handbooks/asar/CNTR1-1-5.html.

[12] QGIS. *Datos Raster*. https://docs.qgis.org/2.14/es/docs/gentle_gis_introduction/raster_data.html.

[13] ArcGIS. *Acerca de los datos ráster en Spatial Analyst*. https://desktop.arcgis.com/es/arcmap/latest/extensions/spatial-analyst/about-raster-data-in-spatial-analyst.htm.

[14] Alberto Moreira. «Digital beamforming: A paradigm shift for spaceborne SAR». En: *2013 14th International Radar Symposium (IRS)*. Vol. 1. IEEE. 2013, págs. 23-26.

[15] European Space Agency. *Data Products*. https://sentinel.esa.int/web/sentinel/missions/sentinel-1/data-products.

[16] QGIS. *Foreword*. https://docs.qgis.org/3.16/en/docs/user_manual/preamble/foreword.html.

[17] European Space Agency. *SNAP*. https://step.esa.int/main/toolboxes/snap/.

[18] European Space Agency. *SNAP FAQ: Frequently Asked Questions About the SNAP Platform*. http://step.esa.int/main/toolboxes/snap/snap-faq/.

[19] European Space Agency. *Step Forum*. https://forum.step.esa.int/.

[20] ESA-PhiLab. *Open SAR Toolkit (OST)*. https://github.com/ESA-PhiLab/OpenSarToolkit. 2018.

[21] SNAP Wiki. *How to use the SNAP API from Python*. https://senbox.atlassian.net/wiki/spaces/SNAP/pages/19300362/How+to+use+the+SNAP+API+from+Python.

[22] ESA-PhiLab. *OST Notebooks*. https://github.com/ESA-PhiLab/OST_Notebooks. 2019.

[23] Orfeo ToolBox. *Open Source processing of remote sensing images*. https://www.orfeo-toolbox.org/.

[24] NASA. *Alaska Satellite Facility*. https://asf.alaska.edu/.

[25] CNES. *PEPS: French Access to the Sentinel products*. https://peps.cnes.fr/rocket/#/home.

[26] CNES. *Centre National d'Ètudes Spatiales*. https://cnes.fr/fr/.

[27] Serco Italia S.p.A. *Copernicus Data and Information Access Service (DIAS)*. https://www.onda-dias.eu/cms/.

[28] SNAP Wiki. *Configure Python to use SNAP-Python (snappy) interface*. https://senbox.atlassian.net/wiki/spaces/SNAP/pages/50855941/Configure+Python+to+use+the+SNAP-Python+snappy+interface.

[29] *Earth Observation Project - SNAP-Python Repository*.

[30] SNAP Wiki. *Snap Engine*. https://github.com/senbox-org/snap-engine/tree/master/snap-python/src/main/resources/snappy/examples.

[31] matplotlib. *Matplotlib: Visualization with Python*. https://matplotlib.org/.

[32] European Space Agency. *Class ProductIO*. http://step.esa.int/docs/v7.0/apidoc/engine/org/esa/snap/core/dataio/ProductIO.html.

[33] Oracle. *Class HashMap*. https://docs.oracle.com/javase/8/docs/api/java/util/HashMap.html.

[34] European Space Agency. *Class GPF*. http://step.esa.int/docs/v7.0/apidoc/engine/org/esa/snap/core/gpf/GPF.html.

[35] Read the Docs. *jpy - Introduction*. https://jpy.readthedocs.io/en/latest/intro.html.

[36] European Space Agency. *Class GraphIO*. http://step.esa.int/docs/v6.0/apidoc/engine/org/esa/snap/core/gpf/graph/GraphIO.html.

[37] European Space Agency. *Class Graph*. http://step.esa.int/docs/v6.0/apidoc/engine/org/esa/snap/core/gpf/graph/Graph.html.

[38] European Space Agency. *Class GraphProcessor*. http://step.esa.int/docs/v6.0/apidoc/engine/org/esa/snap/core/gpf/graph/GraphProcessor.html.

[39] European Space Agency. *Class Node*. http://step.esa.int/docs/v6.0/apidoc/engine/org/esa/snap/core/gpf/graph/Node.html.

[40] European Space Agency. *Class NodeSource*. http://step.esa.int/docs/v6.0/apidoc/engine/org/esa/snap/core/gpf/graph/NodeSource.html.

[41] European Space Agency. *Class XppDomElement*. http://step.esa.int/docs/v6.0/apidoc/engine/com/bc/ceres/binding/dom/XppDomElement.html.

[42] European Space Agency. *Class PrintWriterPM*. http://step.esa.int/docs/v6.0/apidoc/engine/com/bc/ceres/core/PrintWriterProgressMonitor.html.

[43] Oracle. *Class FileReader*. https://docs.oracle.com/javase/7/docs/api/java/io/FileReader.html.

[44] Oracle. *Class FileWriter*. https://docs.oracle.com/javase/7/docs/api/java/io/FileWriter.html.

[45] Christian Ayala Lauroba. *A Deep Learning Approach to Land Use Classification in High Resolution Satellite Imagery*. 2020, págs. 84-100.

[46] NASA Jet Propulsion Laboratory. *U.S. Releases Enhanced Shuttle Land Elevation Data*. https://www2.jpl.nasa.gov/srtm/.

[47] Safe Software. *SRTM HGT (Shuttle Radar Topography Mission Height) Reader*. https://docs.safe.com/fme/html/FME_Desktop_Documentation/FME_ReadersWriters/srtmhgt/srtmhgt.htm.

[48] Federico Filipponi. «Sentinel-1 GRD preprocessing workflow». En: *Multidisciplinary Digital Publishing Institute Proceedings*. Vol. 18. 1. 2019, pág. 11.

[49] European Space Agency. *Level-1 Ground Range Detected*. https://sentinel.esa.int/web/sentinel/user-guides/sentinel-1-sar/resolutions/level-1-ground-range-detected.

[50] Christian Ayala Lauroba. *A Deep Learning Approach to Land Use Classification in High Resolution Satellite Imagery*. 2020, págs. 65-66; 103-104.

[51] ArcGIS API for Python. *How U-net Works?* https://developers.arcgis.com/python/guide/how-unet-works/.

[52] Towards Data Science. *UNet — Line by Line Explanation*. https://towardsdatascience.com/unet-line-by-line-explanation-9b191c76baf5.

[53] Kaiming He, Xiangyu Zhang, Shaoqing Ren y Jian Sun. «Deep residual learning for image recognition». En: *Proceedings of the IEEE conference on computer vision and pattern recognition*. 2016, págs. 770-778.

[54] Kaggle. *ResNet-34*. https://www.kaggle.com/pytorch/resnet34.

[55] Christian Ayala Lauroba. *A Deep Learning Approach to Land Use Classification in High Resolution Satellite Imagery*. 2020, págs. 103-104.

[56] Sentinel-Hub. *EO-Learn*. https://github.com/sentinel-hub/eo-learn.

[57] Christian Ayala Lauroba. *A Deep Learning Approach to Land Use Classification in High Resolution Satellite Imagery*. 2020, págs. 105-106.

[58] Christian Ayala Lauroba. *A Deep Learning Approach to Land Use Classification in High Resolution Satellite Imagery*. 2020, págs. 109-131.

[59] Towards Data Science. *Understanding binary cross-entropy / log loss: a visual explanation*. https://towardsdatascience.com/understanding-binary-cross-entropy-log-loss-a-visual-explanation-a3ac6025181a.

[60] Christian Ayala Lauroba. *A Deep Learning Approach to Land Use Classification in High Resolution Satellite Imagery*. 2020, págs. 112-118.

[61] Towards Data Science. *Neural Networks Intuitions: 3. Focal Loss for Dense Object Detection — Paper Explanation*. https://towardsdatascience.com/neural-networks-intuitions-3-focal-loss-for-dense-object-detection-paper-explanation-61bc0205114e.

[62] AI Salon Medium. *Understanding Dice Loss for Crisp Boundary Detection*. https://medium.com/ai-salon/understanding-dice-loss-for-crisp-boundary-detection-bb30c2e5f62b.

[63] Christian Ayala Lauroba. *A Deep Learning Approach to Land Use Classification in High Resolution Satellite Imagery*. 2020, pág. 136.