UNIVERSITAT POLITÈCNICA DE VALÈNCIA

Escuela Técnica Superior de Ingeniería Geodésica, Cartográfica y Topográfica

Clasificación de objetos urbanos utilizando redes neuronales

Trabajo Fin de Máster

Máster Universitario en Ingeniería Geomática y Geoinformación

AUTOR/A: Oliveros Amorós, Antonio

Tutor/a: Coll Aliaga, Peregrina Eloína

CURSO ACADÉMICO: 2023/2024

ESCUELA TÉCNICA SUPERIOR
DE INGENIERÍA GEODÉSICA
CARTOGRÁFICA Y TOPOGRÁFICA

CLASIFICACIÓN DE OBJETOS URBANOS UTILIZANDO REDES NEURONALES

TRABAJO FINAL DE MÁSTER

AUTOR: ANTONIO OLIVEROS AMORÓS
TUTORA: ELOÍNA COLL ALIAGA
Curso académico: 2023-2024

Agradecimientos

Me gustaría agradecer, en primer lugar, a mi familia y amigos por acompañarme durante toda la travesía académica, también a la profesora Eloína Coll, por su guía.

Por último, a mis compañeros de GeoAI Tech Global por su colaboración en la elaboración de este proyecto.

Compromiso

“El presente documento ha sido realizado completamente por el firmante; no ha sido entregado como otro trabajo académico previo y todo el material tomado de otras fuentes ha sido convenientemente entrecomillado y citado su origen en el texto, así como referenciado en la bibliografía.”

Valencia, 08/07/2024 Antonio Oliveros Amorós

Resumen

El estudio tiene como objetivo principal utilizar redes neuronales para identificar y categorizar distintos elementos urbanos.

A través de una combinación de técnicas de procesamiento de imágenes y aprendizaje profundo, se diseñarán y entrenarán modelos de redes neuronales convolucionales (CNN) ya que estas son muy efectivas para tareas de visión artificial en la clasificación y segmentación de imágenes, entre otras aplicaciones.

Se explorarán diferentes arquitecturas de redes neuronales y técnicas de entrenamiento para optimizar el rendimiento de la clasificación. Además, se utilizarán conjuntos de datos de imágenes de alta resolución.

Los resultados obtenidos contribuirán al avance en la automatización de la interpretación de imágenes urbanas, con potenciales aplicaciones en campos como la planificación urbana y la cartografía digital.

Resum

L'estudi té com a objectiu principal utilitzar xarxes neuronals per identificar i categoritzar diferents elements urbans.

A través d'una combinació de tècniques de processament d'imatges i aprenentatge profund, es dissenyaran i entrenaran models de xarxes neuronals convolucionals (CNN) ja que aquestes són molt efectives per a tasques de visió artificial en la classificació i segmentació d'imatges, entre altres aplicacions.

S'exploraran diferents arquitectures de xarxes neuronals i tècniques d'entrenament per optimitzar el rendiment de la classificació. A més, s'utilitzaran conjunts de dades d'imatges d'alta resolució.

Els resultats obtinguts contribuiran a l'avanç en l'automatització de la interpretació d'imatges urbanes, amb potencials aplicacions en camps com la planificació urbana i la cartografia digital.

Abstract

The main objective of the study is to use neural networks to identify and categorize various urban elements.

Through a combination of image processing techniques and deep learning, convolutional neural network (CNN) models will be designed and trained as they are highly effective for computer vision tasks in image classification and segmentation, among other applications.

Different neural network architectures and training techniques will be explored to optimize classification performance. Additionally, high-resolution image datasets will be utilized.

The results obtained will contribute to advances in the automation of urban image interpretation, with potential applications in fields such as urban planning and digital cartography.

Índice de ilustraciones

Ilustración 1: Logo GeoAI 9
Ilustración 2: Objetivos Desarrollo Sostenible 11
Ilustración 3: Ortofotos Máxima Actualidad 13
Ilustración 4: Software SIG - QGIS 14
Ilustración 5: Imagen descargada 14
Ilustración 6: Imagen con máscara superpuesta 15
Ilustración 7: Imagen recortada 15
Ilustración 8: Código para recortar imágenes 16
Ilustración 9: Imagen enmascarada con mosaicos superpuestos 16
Ilustración 10: Código para eliminar imágenes sin información 17
Ilustración 11: Logo Label Studio 18
Ilustración 12: Comando label-studio start 18
Ilustración 13: Pestaña de parámetros 19
Ilustración 14: Código convertir TIFF a PNG 19
Ilustración 15: Apartado de importación de datos 20
Ilustración 16: Imágenes importadas correctamente 20
Ilustración 17: Pestaña selección de tipo de etiquetas 21
Ilustración 18: Menú creación etiquetas 21
Ilustración 19: Menú de Imágenes Etiquetadas 22
Ilustración 20: Imagen etiquetada 22
Ilustración 21: Menú Exportación 23
Ilustración 22: CNN LeNet 25
Ilustración 23: CNN AlexNet 25
Ilustración 24: Red VGGNet 26
Ilustración 25: Red ResNet 26
Ilustración 26: Red DenseNet 27
Ilustración 27: Red EfficientNet 27
Ilustración 28: Preparar el entorno Colab 30
Ilustración 29: Importar los datos 30
Ilustración 30: Crear el DataLoader 31
Ilustración 31: Crear el Modelo 31
Ilustración 32: Definir la Función de Pérdida 32
Ilustración 33: Entrenar el Modelo 33
Ilustración 34: Evaluación del Modelo 33
Ilustración 35: Front para utilizar la Red Neuronal 38
Ilustración 36: Predicción del modelo entrenado 38
Ilustración 37: Resultado obtenido sobre imagen 39
Ilustración 38: Piscinas no detectadas 39
Ilustración 39: Coches mal detectados 40
Ilustración 40: Elementos no detectados 40
Ilustración 41: Grupos profesionales 41
Ilustración 42: Tabla salarial 41
Ilustración 43: Plano Final Obtenido 43

Índice de tablas

Tabla 1: Formato YOLO ……23

Tabla 2: Valores obtenidos en los entrenamiento ……34

Tabla 3: Coste total ……42

No se encuentran entradas de índice.

Índice

1. Introducción 9
1.1. Antecedentes 9
1.2. Justificación 10
1.3. ODS 11
1.4. Localización 11
2. Objetivos 12
2.1. Objetivo general 12
2.2. Objetivos específicos 12
3. Datos 13
4. Metodología 14
4.1. Tratamiento de las imágenes 14
4.2. Etiquetado de imágenes 18
4.3. Selección de modelo de red neuronal 24
4.4. EfficientNetV2 28
4.5. Entrenamiento 30
4.6. Optimización de Hiperparámetros 34
4.7. Obtención de Resultados 38
5. Resultados 39
6. Presupuesto 41
7. Conclusiones 43
8. Bibliografía 44
9. Anexos 46
9.1. Anexo 1 46
9.2. Anexo 2 50
9.3. Anexo 3 52
10. Cartografía 54

1. Introducción

Este Trabajo de Final de Máster se lleva a cabo en colaboración con la empresa GeoAI Tech Global (Ilustración 1), especializada en análisis de datos geoespaciales, integración de modelos de aprendizaje computacional mediante procesos industriales y apoyo en la toma de decisiones con modelos de visualización de redes.

Ilustración 1: Logo GeoAI
Fuente: https://www.geoaitech.com

Las redes neuronales han demostrado ser herramientas muy útiles en el ámbito de la clasificación y detección de imágenes debido a su capacidad para aprender patrones complejos y jerarquizar las características de los datos (Javier Martínez Llamas, 2018). En este contexto, el objetivo de este proyecto es diseñar, entrenar y evaluar un modelo de red neuronal capaz de clasificar objetos urbanos dentro de una imagen. Para ello, se utilizarán imágenes de alta resolución, a partir de las cuales se obtendrán conjuntos de datos etiquetados que permitirán realizar el entrenamiento y testeo de la red.

Además, se explorarán diferentes tipos de redes y arquitecturas para obtener un modelo con el mejor rendimiento posible. Este enfoque no solo permitirá mejorar la precisión en la clasificación de objetos urbanos, sino que también contribuirá al avance en la gestión de entornos urbanos, facilitando la toma de decisiones y optimizando recursos en la planificación y mantenimiento de infraestructuras urbanas.

1.1. Antecedentes

El campo de la inteligencia artificial ha crecido de manera exponencial en los últimos tiempos (Castañeda et all., 2023), especialmente en el ámbito de las redes neuronales artificiales. Estas redes se basan en el funcionamiento del cerebro humano, tratando de imitar cómo funciona una neurona real teniendo como objetivo principal el reconocer patrones complejos como haría una persona. Dicho crecimiento es debido a distintos factores:

- **Incremento de la capacidad computacional de los ordenadores**: Esto ha permitido que los procesos como el entrenamiento de redes sean más rápidos, lo que ha generado que se diseñen redes neuronales más grandes y complejas.
- **Gran volumen de datos**: La recolección y el almacenamiento masivo de datos en formato digital ha proporcionado el material necesario para poder entrenar los modelos de aprendizaje de la manera más efectiva posible.
- **Avances en la creación de arquitecturas**: Debido a los dos razonamientos anteriores, se han estudiado y creado nuevas arquitecturas y algoritmos para optimizar y abordar de manera más eficiente el tratamiento de los modelos, como las redes neuronales convolucionales (CNN), las redes neuronales recurrentes (RNN) y las redes generativas adversarias (GAN).

Este auge ha generado un abanico de posibilidades en el proceso de automatización y análisis de distintas áreas como la gestión y análisis de entornos urbanos, lo cual permitirá mejorar la gestión de infraestructuras y facilitar el desarrollo de ciudades inteligentes.

El objetivo principal de este proyecto es desarrollar un modelo de clasificación que pueda identificar y categorizar distintos elementos, principalmente, en entornos urbanos.

1.2. Justificación

Los objetos urbanos son elementos existentes en las ciudades, y el tener controlada la cantidad, ubicación y distribución de ellos permite una gestión más eficiente de los recursos (Fernández, 2004). Esto es debido al rápido crecimiento de las ciudades, el aumento de la densidad y complejidad de los entornos urbanos provoca que la gestión de las infraestructuras deba ser más eficiente. Los métodos tradicionales, por lo general, requieren una mayor cantidad de tiempo, esfuerzo y mano de obra, ya que en muchos casos se requiere del traslado de personal a las distintas ubicaciones, lo que ralentiza los procesos. Mediante el uso de redes neuronales esta tarea puede realizarse de manera más automática, reduciendo los tiempos y optimizando los recursos.

La automatización permite ahorrar recursos en estos aspectos para poder destinarlos a otros proyectos para seguir mejorando la eficiencia y el desarrollo de las ciudades, además poder automatizar la detección de elementos ayuda a realizar un mantenimiento más eficiente de las infraestructuras, esto hace que se prolongue su vida útil y permita reducir costos de reparación.

Este es un proyecto innovador que promueve el desarrollo tecnológico de la gestión urbana, y sirve como base para investigaciones futuras y aplicaciones que sigan mejorando la vida de las personas. La clasificación de objetos urbanos es un paso hacia la modernización de las ciudades, ya no solo para cubrir una necesidad actual sino también para sentar las bases hacia un desarrollo urbano más inteligente y eficiente.

1.3. ODS

El proyecto se relaciona con 3 Objetivos de Desarrollo Sostenible (ODS) tal y como se puede observar en la ilustración 2.

- **ODS 8 – Trabajo Decente y Crecimiento Económico**: La relación con este objetivo se debe a la automatización de procesos, lo que provoca un ahorro que puede ser reinvertido en otros trabajos o desarrollar la economía en otros ámbitos.
- **ODS 9 – Industria, Innovación e Infraestructura**: El proyecto promueve la innovación tecnológica mediante el uso de redes neuronales, impulsando, además, unas infraestructuras más automatizadas e inteligentes.
- **ODS 11 – Ciudades y Comunidades Sostenibles**: Tiene una conexión directa con este objetivo ya que se basa en la detección de elementos que se encuentran en las ciudades.

Ilustración 2: Objetivos Desarrollo Sostenible
Fuente: Organización Mundial de la Salud

1.4. Localización

En el proyecto se han utilizado imágenes de distintas entidades de población, tanto ciudades grandes como municipios pequeños.

Las poblaciones utilizadas han sido:

- Llíria
- Benissanó
- Alcoi
- Alicante
- Novelda
- Albacete

Lo que se desea conseguir en el proyecto es una red que permita detectar objetos urbanos en cualquier población existente, da igual tamaño y ubicación.

2. Objetivos

2.1. Objetivo general

El objetivo general es obtener una red neuronal que permita detectar una serie de elementos urbanos, los cuales se han utilizado para entrenar la red, utilizando imágenes satelitales de alta resolución.

2.2. Objetivos específicos

El proyecto se puede dividir en objetivos específicos:

- **Objetivo Específico 1:**
  - Obtener las imágenes con las que se va a entrenar el modelo.
- **Objetivo Específico 2:**
  - Realizar el etiquetado de las imágenes indicando los elementos que se van a detectar.
- **Objetivo Específico 3:**
  - Estudiar los tipos de redes disponibles y seleccionar el tipo de red neuronal con mayor se adapte al objetivo general.
- **Objetivo Específico 4:**
  - Realizar los entrenamientos y pruebas pertinentes ajustando los parámetros de la red para obtener los mejores resultados posibles.
- **Objetivo Específico 5:**
  - Utilizar la red neuronal para obtener las imágenes con los elementos marcados automáticamente.

3. Datos

Los datos de partida del proyecto son imágenes satelitales de alta resolución que van a facilitar la labor de detección de los elementos. Para ello, se han utilizado las Ortofotos de Máxima Actualidad (Ilustración 3), que se pueden obtener del Centro Nacional de Información Geográfica (CNIG). Se ha decidido utilizar estas imágenes por varias razones: se encuentran corregidas geométricamente, son imágenes de máxima actualidad, por lo que pueden reflejar más precisamente la situación actual de los elementos que se van a detectar, además cuentan con una resolución de 0.25m, la cual es óptima para los objetivos de este proyecto permitiendo tener una mayor precisión a la hora de identificar y clasificar los objetos urbanos.

Ilustración 3: Ortofotos Máxima Actualidad
Fuente: Centro Nacional de Información Geográfica

Las imágenes fueron descargadas de manera manual seleccionando imágenes con la mayor cantidad de zonas pobladas disponibles. Se obtuvieron un total de 8 imágenes en las que se abarcan varias poblaciones dentro de cada imagen, cada una de estas imágenes pesa más de 2GB. El tamaño fue un factor fundamental a la hora de decidir la cantidad de imágenes descargadas.

4. Metodología

4.1. Tratamiento de las imágenes

Una vez se obtienen las imágenes se realiza un enmascaramiento de las zonas urbanas dentro de cada imagen para eliminar zonas inservibles para el proyecto. Las máscaras se generan manualmente dentro del software SIG, QGIS. (Ilustración 4)

Ilustración 4: Software SIG - QGIS
Fuente: qgis.org

Dentro del programa, visualizando las imágenes, se generan polígonos de las zonas urbanas y mediante la herramienta "Cortar ráster por capa de máscara" se obtuvieron unas imágenes resultantes en las que únicamente se visualizan las seleccionadas, dejando en negro las zonas que no se encuentran dentro del polígono, tal y como se puede visualizar en las ilustraciones 5,6 y 7.

Ilustración 5: Imagen descargada
Fuente: Elaboración propia

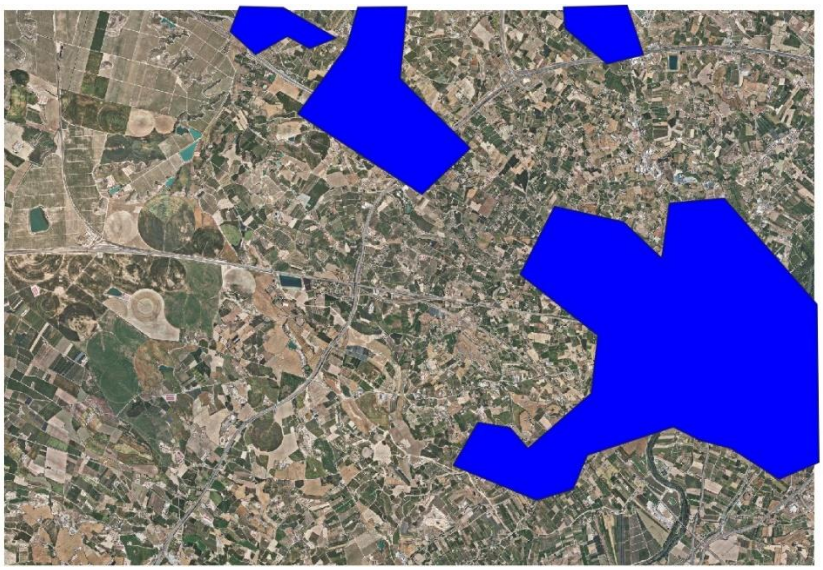


Ilustración 6: Imagen con máscara superpuesta
Fuente: Elaboración propia

Ilustración 7: Imagen recortada
Fuente: Elaboración propia

Posteriormente, partiendo de las imágenes resultantes en el paso anterior se generan unos códigos en Python para realizar el recorte de las imágenes en mosaicos de menor tamaño (Ilustración 8), consiguiendo trabajar con ellos con mayor rapidez y fluidez. Se decide obtener mosaicos de 1024x1024, lo que provocó que de las 8 imágenes que se partían se obtuvieran un total cercano a 18000 mosaicos de 1024x1024 (Cada imagen se divide en 38 filas por 57 columnas).

```python
def tiles(options):
        tile_size = 256
        # Calcular la cantidad de azulejos en cada dimensión
        num_tiles_x = math.ceil(width / tile_size)
        num_tiles_y = math.ceil(height / tile_size)

        # Recortar la imagen en azulejos de tile_size x tile_size
        for y in range(num_tiles_y):
            for x in range(num_tiles_x):
                left = x * tile_size
                upper = y * tile_size
                right = min((x + 1) * tile_size, width)
                lower = min((y + 1) * tile_size, height)

                # Leer la parte de la imagen correspondiente al azulejo
                tile_data = []
                for band_index in range(image.RasterCount):
                    band = image.GetRasterBand(band_index + 1)
                    band_data = band.ReadAsArray(left, upper, right - left, lower - upper)
                    tile_data.append(band_data)
```

Ilustración 8: Código para recortar imágenes
Fuente: Elaboración propia

El objetivo principal de este código es procesar imágenes en formato TIFF, dividiéndolas en mosaicos de menor tamaño y almacenándolo en un directorio de salida. Para ello se utiliza la librería *gdal* que está desarrollada para manipular datos geoespaciales. Primero se define una función para buscar el tipo de imagen indicada dentro de un directorio y todos sus subdirectorios. A continuación, se abren todas las imágenes encontradas en la ruta de entrada con la librería, se comprueba que la proyección sea la misma en todas, en caso afirmativo se guarda esta proyección en una variable, también se indica el tamaño de los mosaicos que se desean obtener. En el siguiente paso, mediante la librería *numpy*, se convierten los datos de cada banda de las imágenes a valores, se recorta el mosaico correspondiente y se calculan sus coordenadas correspondientes para, finalmente, almacenar todas las nuevas imágenes en un directorio previamente especificado.

Ilustración 9: Imagen enmascarada con mosaicos superpuestos
Fuente: Elaboración propia

De estos mosaicos hay muchos que no contienen información ya que se obtienen de las imágenes enmascaradas que cortan con zonas en negro, por lo que se realiza otro código de Python para obtener un depurado y eliminar todas las imágenes que no tuvieran información en ella (Ilustración 10), lo que redujo la cantidad de mosaicos a 7000 que contaban con información de zonas urbanas.

```python
def ValidarImagenes(options):
    log = logging.getLogger(__name__)
    log.info('Revisando imagenes validas')
    image_dir = r'G:\Mi unidad\TFM\Imagenes\33_tilesprueba'

    try:
        # Obtener las imágenes
        image_paths = open_images(image_dir)
        if not image_paths:
            log.error('No se encontraron imágenes válidas en el directorio especificado.')
            return

        # Eliminar imágenes negras
        for image_path in image_paths:
            if is_image_black(image_path):
                os.remove(image_path)
                log.info(f'Imagen eliminada: {image_path}')

    except Exception as oEx:
        log.error(oEx)
    log.info('...DONE!')
```

Ilustración 10: Código para eliminar imágenes sin información
Fuente: Elaboración propia

En este otro código se vuelve a utilizar la librería *gdal*, se define una función que abra una imagen, la convierta a matriz y compruebe si todos los valores son 0, si es así, la elimina. Esta función se introduce dentro de un bucle que recorre todo el directorio especificado.

4.2. Etiquetado de imágenes

Con las imágenes listas, el siguiente paso es el etiquetado. Este debe hacerse de una manera precisa y consistente para obtener el mejor rendimiento posible en el modelo. Las etiquetas son ejemplos necesarios para que la red neuronal aprenda a identificar y clasificar correctamente los objetos, por lo que cuanto mayor sea el número de etiquetas, mejor será para el posterior entrenamiento, permitiendo tener una mayor precisión y robustez.

Se utiliza la herramienta *Label Studio* (Ilustración 11). Es una herramienta de etiquetado de datos de código abierto que permite subir los archivos y etiquetarlos de distintas formas.

Ilustración 11: Logo Label Studio
Fuente: https://labelstud.io/

Para acceder a *Label Studio* una vez instalado se realiza desde los "Comandos del Sistema" utilizando, tal y como se observa en la ilustración 12, el comando *label-studio start*.

```
C:\Users\G531>label-studio start
=> Database and media directory: C:\Users\G531\AppData\Local\label-studio\label-studio
=> Static URL is set to: /static/
Current platform is win32, apply sqlite fix
Can't load sqlite3.dll from current directory
=> Database and media directory: C:\Users\G531\AppData\Local\label-studio\label-studio
=> Static URL is set to: /static/
Read environment variables from: C:\Users\G531\AppData\Local\label-studio\label-studio\.env
get 'SECRET_KEY' casted as '<class 'str'>' with default ''
Starting new HTTPS connection (1): pypi.org:443
https://pypi.org:443 "GET /pypi/label-studio/json HTTP/1.1" 200 32193
Performing system checks...

System check identified no issues (1 silenced).
July 06, 2024 - 21:37:56
Django version 3.2.25, using settings 'label_studio.core.settings.label_studio'
Starting development server at http://0.0.0.0:8080/
Quit the server with CTRL-BREAK.
[2024-07-06 21:37:59,217] [django.server::log_message::161] [INFO] "GET / HTTP/1.1" 302 0
[2024-07-06 21:37:59,217] [django.server::log_message::161] [INFO] "GET / HTTP/1.1" 302 0
[2024-07-06 21:37:59,317] [django.server::log_message::161] [INFO] "GET /projects/ HTTP/1.1" 200 31588
[2024-07-06 21:37:59,317] [django.server::log_message::161] [INFO] "GET /projects/ HTTP/1.1" 200 31588
[2024-07-06 21:37:59,348] [django.server::log_message::161] [INFO] "GET /static/css/uikit.e49a7a43adbd.css HTTP/1.1" 200
 3892
[2024-07-06 21:37:59,348] [django.server::log_message::161] [INFO] "GET /static/css/uikit.e49a7a43adbd.css HTTP/1.1" 200
 3892
[2024-07-06 21:37:59,349] [django.server::log_message::161] [INFO] "GET /static/css/main.05101f5b0b98.css HTTP/1.1" 200
```

Ilustración 12: Comando label-studio start
Fuente: Elaboración propia

Se crea un nuevo proyecto indicando el nombre del proyecto y una breve descripción (Ilustración 13). Se importan las imágenes que se desean etiquetar, en este caso, no se incorporaron todas las imágenes debido a limitaciones de hardware, también los formatos de importación son limitados por lo que se ha tenido que programar un código Python para convertir las imágenes en formato TIFF a formato PNG (Ilustración 14). Se selecciona este formato porque al realizar la conversión mantiene la calidad de la imagen igual que el original, lo cual es un factor fundamental.

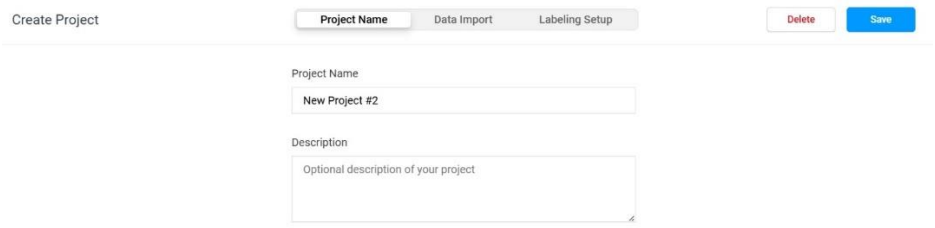


Ilustración 13: Pestaña de parámetros
Fuente: Elaboración propia

Para este programa se crea una función que busque todas las imágenes con terminación *.tif* en el directorio y sus correspondientes subdirectorios y se utiliza la librería *opencv*, que sirve para el procesamiento de imágenes, para leer las imágenes se crea el nuevo *path* de la imagen, esta vez con extensión *.png* y se almacena la imagen leída anteriormente en esa nueva ruta de salida.

```python
def tifapng(options):


    try:
        # Get image paths
        image_paths = open_images(image_dir)
        if not image_paths:
            log.error('No se encontraron imágenes válidas en el directorio especificado.')
            return

        if not os.path.exists(output_dir):
            os.makedirs(output_dir)

        # Loop through images and convert to PNG
        for image_path in image_paths:
            try:
                # Read TIFF image
                image = cv2.imread(image_path, -1)  # -1 for including alpha channel

                # Get image name and directory for output PNG
                filename = os.path.splitext(os.path.basename(image_path))[0]
                output_path = os.path.join(output_dir, filename + '.png')

                # Save as PNG
                cv2.imwrite(output_path, image)
                log.info(f'Imagen convertida: {image_path} -> {output_path}')
            except Exception as e:
                log.error(f'Error al convertir imagen {image_path}: {e}')
```

Ilustración 14: Código convertir TIFF a PNG
Fuente: Elaboración propia

Por último, se selecciona el tipo de etiquetado que se desea realizar (Ilustración 17). Este puede ser mediante:

- Polígonos
- Máscaras
- Bounding boxes
- Puntos clave

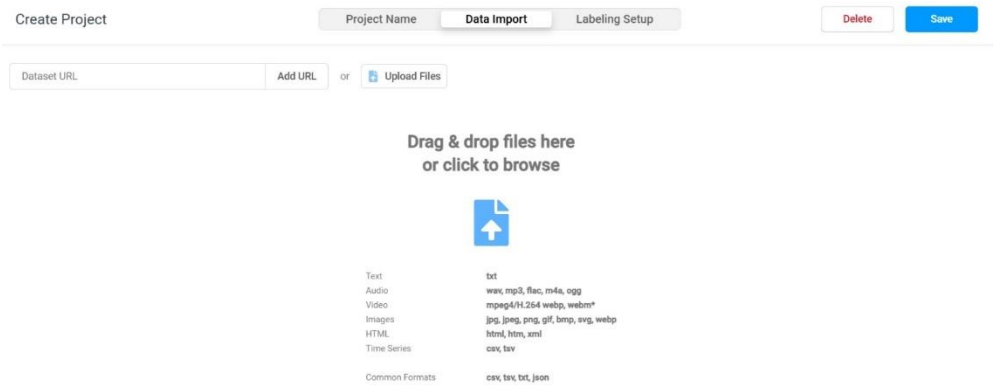


Ilustración 15: Apartado de importación de datos
Fuente: Elaboración propia

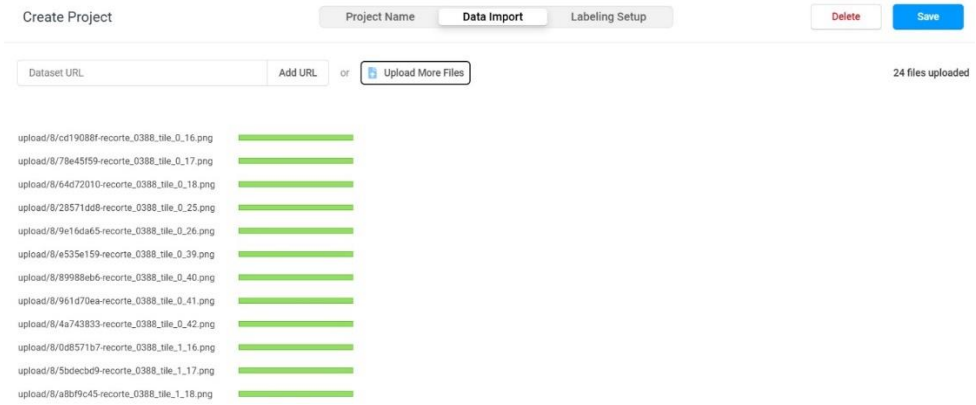


Ilustración 16: Imágenes importadas correctamente
Fuente: Elaboración propia

Para este proyecto se realizó mediante polígonos ya que era la forma óptima para etiquetar todos los elementos que se querían marcar en la imagen. Una vez seleccionado el método de etiquetación hay que indicar el número y nombre de tipos de etiquetas. Se decidió detectar 5 elementos en esta red como se muestra en la ilustración 18:

- Vehículos.
- Rotondas.
- Pasos de cebra.
- Piscinas.
- Contenedores.

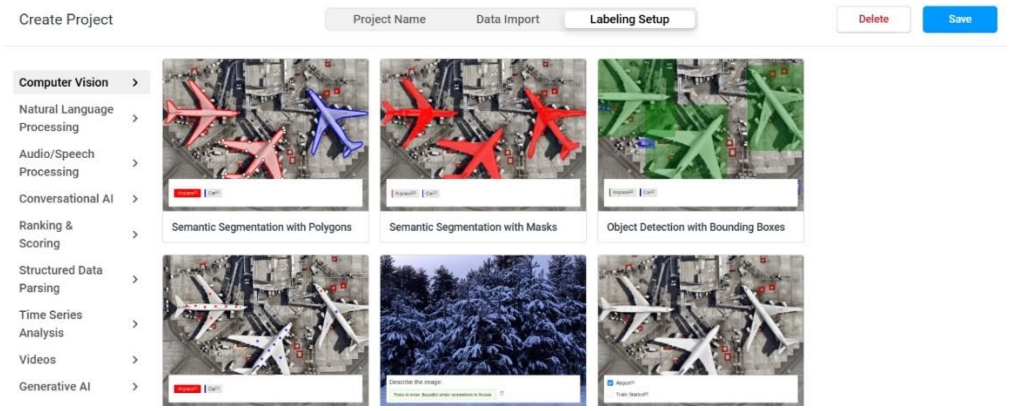


Ilustración 17: Pestaña selección de tipo de etiquetas
Fuente: Elaboración propia

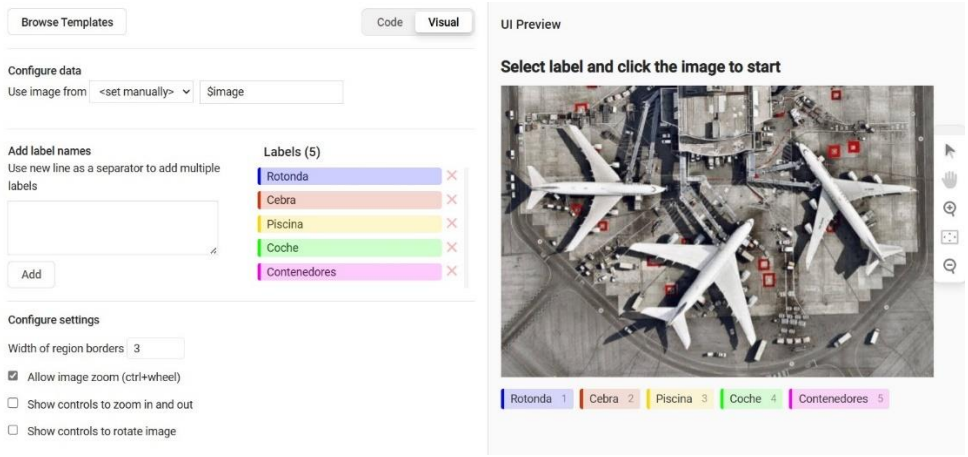


Ilustración 18: Menú creación etiquetas
Fuente: Elaboración propia

El etiquetado es un proceso muy tedioso y que requiere mucho tiempo. Debe a ser un paso fundamental de la red y todos los elementos visibles que se desean detectar en la red tienen que ser correctamente etiquetados en las imágenes de entrenamiento para poder reducir el sesgo lo más posible. Si en una imagen no se etiquetan correctamente surgen problemas como:

- **Baja precisión y exactitud**: El modelo no podrá aprender a identificar correctamente esos objetos y no tendrá suficiente información para predicciones correctas.
- **Confusión entre clases**: Si los objetos no tienen una etiquetación correctamente delimitada puede llegar a haber confusión entre las clases haciendo que la red detecte elementos incorrectamente.

Se obtuvieron más de 10000 etiquetas de las distintas clases y en las siguientes ilustraciones se muestra un detalle del proceso.

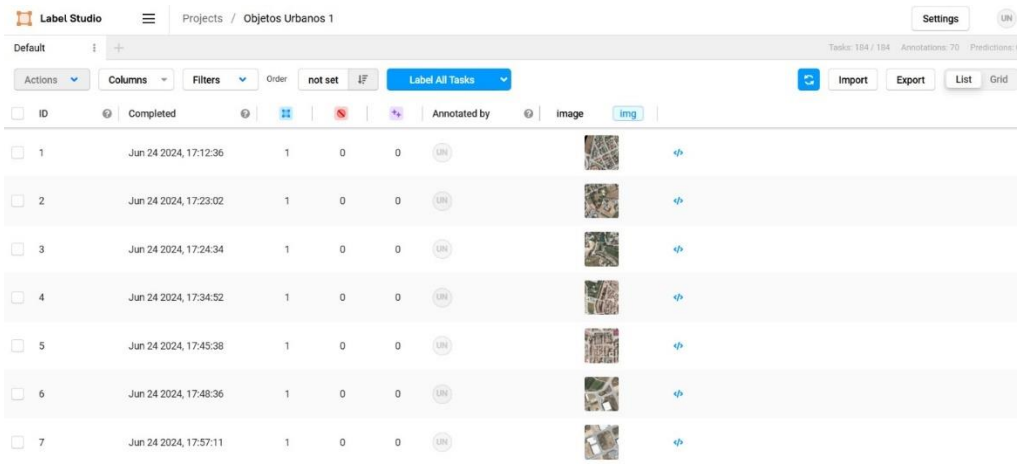


Ilustración 19: Menú de Imágenes Etiquetadas
Fuente: Elaboración propia

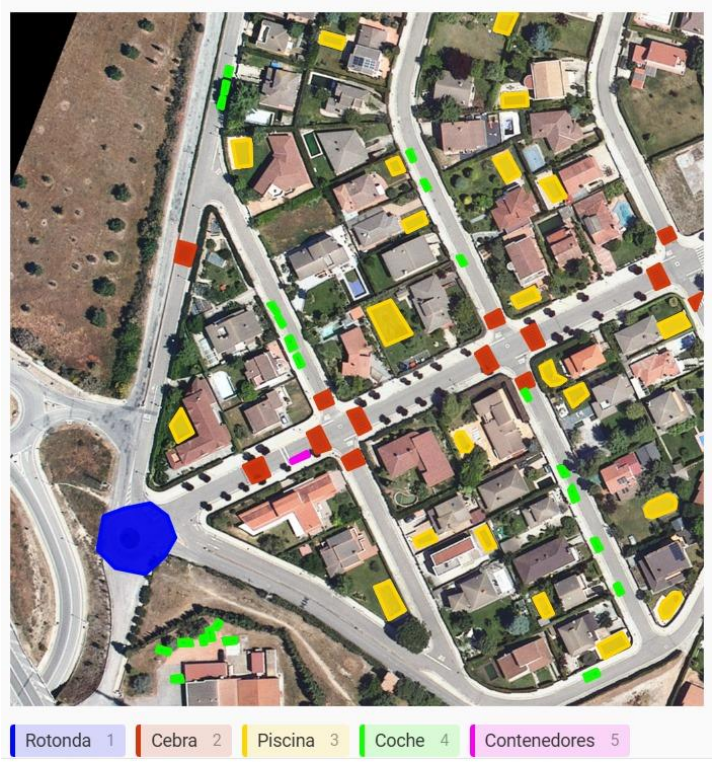


Ilustración 20: Imagen etiquetada
Fuente: Elaboración propia

El formato de exportación puede variar dependiendo del tipo de uso que se le va a hacer a las etiquetas, los formatos de exportación permitidos por el programa son (Ilustración 21):

- Texto
- JSON
- CSV
- TSV
- COCO
- YOLO

La Ilustración 21 muestra los formatos y la explicación de cada uno de ellos.

Para la red que se va a utilizar en este trabajo final de master se elige **YOLO**, como el formato de exportación óptimo.

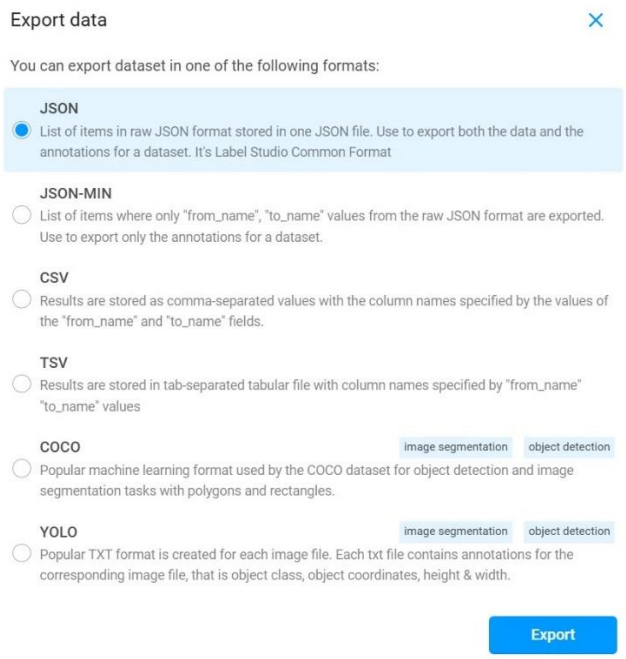


Ilustración 21: Menú Exportación
Fuente: Elaboración propia

YOLO (*You Only Look Once*) es un formato de exportación de etiquetas muy utilizado debido a su simplicidad, eficiencia y fácil implementación a la hora de representar las anotaciones para entrenar los modelos de detección de objetos.

Este formato utiliza archivos de texto (*.txt*) para almacenar las etiquetas de las imágenes. Cada archivo de texto corresponde a una imagen y contiene una línea por cada etiqueta en la imagen. La estructura es la siguiente:

$$< class\_id >< x\_center >< y\_center >< width >< height >$$

Tabla 1: Formato YOLO
Fuente: Elaboración propia

Donde:

- *Class_id*: Identificador de la clase del objeto
- *X_center*: Coordenada X del centro del objeto normalizada
- *Y_center*: Coordenada Y del centro del objeto normalizada
- *Width*: Ancho del objeto normalizado
- *Height*: Altura del objeto normalizado

4.3. Selección de modelo de red neuronal

Una clasificación de objetos no es posible realizarse con utilizar redes neuronales normales, por lo que hay que utilizar redes neuronales convolucionales (CNN) (Artola, 2019).

Las redes neuronales convolucionales son una parte del *Machine Learning* y parte principal del *Deep Learning*. Estas redes están formadas por capas de entrada, capas ocultas y capas de salida. Cada unión entre neuronas tiene un peso y un umbral distintos asociados a esa conexión (Jiménez de las Heras, 2018).

Las ventajas de las CNN son:

- **Entrenamiento más rápido**: Las CNN entrenan cada parte de una red para que haga una tarea específica, esto hace que se reduzca el número de capas ocultas haciendo más rápido el procesamiento.
- **Invarianza a la traslación de patrones**: Las CNN pueden reconocer patrones, aunque no se encuentren siempre en la misma posición, las redes neuronales tradicionales solo reconocen patrones si se encuentran todo el rato en la misma posición.
- **Eficiencia en detección de patrones**: Al no tener el problema del posicionamiento, las CNN tiene mayor facilidad para identificar patrones visuales, esta es la principal razón por la que se usan para clasificar imágenes, por su capacidad para detectar patrones da igual la posición, rotación o tamaño.
- **Capacidad de procesar muchos datos**: Su mayor velocidad de procesamiento permite que las cantidades de datos con las que se pueden trabajar sean mayores.

Existen distintas maneras de trabajar con redes neuronales convolucionales, se pueden crear desde cero, lo cual no es recomendable debido a su complejidad, ya que crear una CNN sin ninguna base requiere una gran cantidad de tiempo y recursos debido a que hay que diseñar una arquitectura, implementar los algoritmos para realizar el entrenamiento y definir los parámetros. Además, habría que realizar el entrenamiento para que entienda todos los procesos pertinentes.

El otro método es utilizar redes neuronales convolucionales pre-entrenadas o evaluadas. Utilizar este tipo de redes permite ahorrar muchos recursos, ya que las arquitecturas ya han sido diseñadas y testadas, por lo que son redes que funcionan.

También, al ser modelos pre-entrenados sirven como punto de partida para que el proceso de entrenamiento se realice mediante un aprendizaje por transferencia, lo cual reduce mucho los tiempos necesarios para obtener resultados.

Por último, las CNN evaluadas se actualizan periódicamente para estar optimizadas lo máximo posible.

Existen distintos tipos de CNN evaluadas (Randellini, 2023):

LeNet

Es una de las primeras CNN que se diseñaron, en 1998 y se puede ver en la Ilustración 22. Se utilizaba para clasificar dígitos escritos a manos (MNIST).

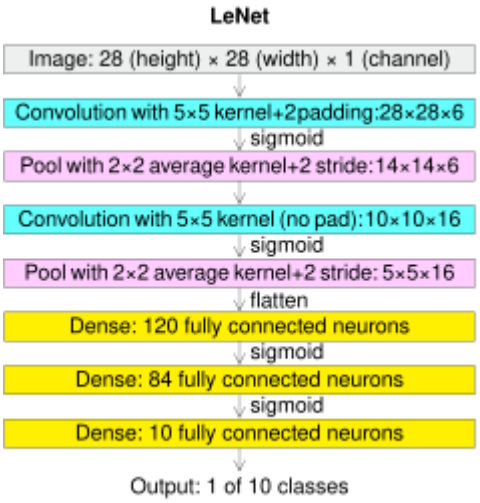


Ilustración 22: CNN LeNet
Fuente: https://en.wikipedia.org/wiki/LeNet

AlexNet

Fue diseñada en 2012 y marcó el inicio de la visualización por ordenador de las CNN (Ilustración 23).

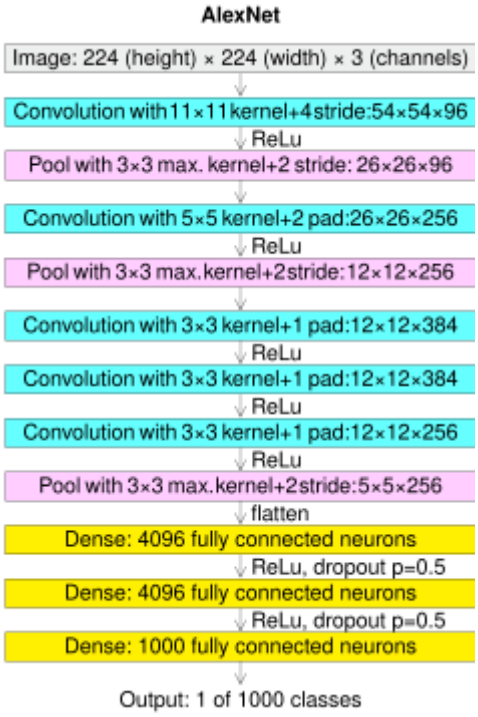


Ilustración 23: CNN AlexNet
Fuente: https://en.wikipedia.org/wiki/AlexNet

VGGNet

Es una CNN que destaca por su uso en clasificación de imágenes a gran escala debió a su aprendizaje de representaciones jerárquicas tal y como vemos en la Ilustración 24.

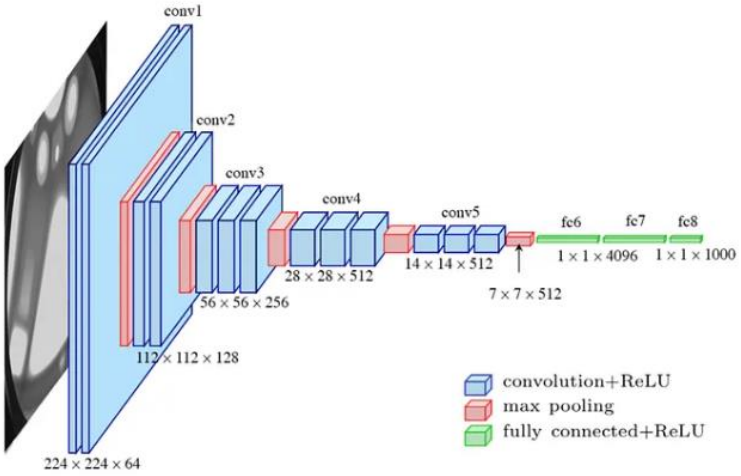


Ilustración 24: Red VGGNet
Fuente: https://medium.com/@siddheshb008/vgg-net-architecture-explained-71179310050f

ResNet

La ilustración 25 muestra ResNet, desarrollada por Microsoft, que añadió el concepto de conexiones residuales para no poder tener redes muy profundas sin que se degraden.

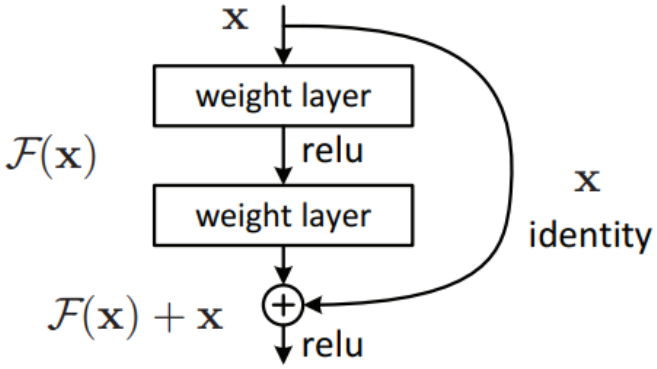


Ilustración 25: Red ResNet
Fuente: https://www.geeksforgeeks.org/residual-networks-resnet-deep-learning/

DenseNet

En esta CNN, cada capa recibe información de todas las capas anteriores. La lustración 26 muestra este tipo de red.

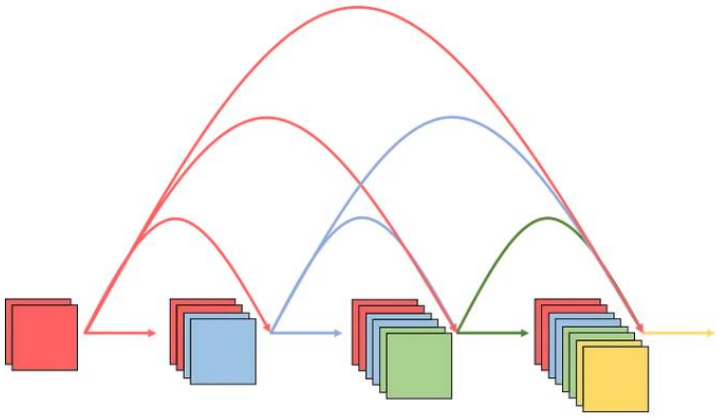


Ilustración 26: Red DenseNet
Fuente: https://medium.com/@alejandro.itoaramendia/densenet-a-complete-guide-84fedef21dcc

EfficientNet

La ilustración 27 muestra EfficienNet. Esta red ajusta de maneta equilibrada la profundidad, el ancho y la resolución de la red.

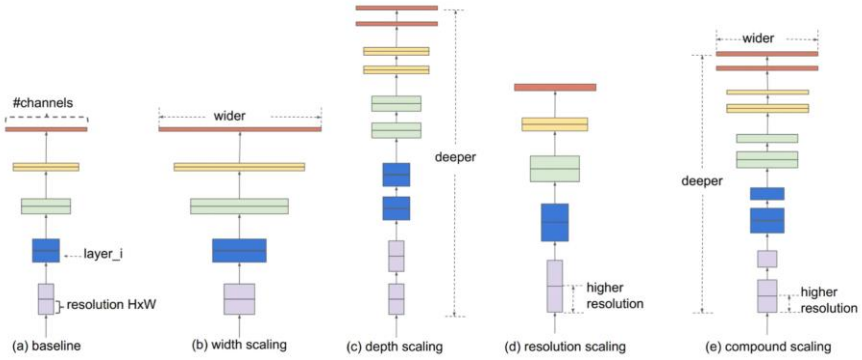


Ilustración 27: Red EfficientNet
Fuente: https://arjun-sarkar786.medium.com/understanding-efficientnet-the-most-powerful-cnn-architecture-eaeb40386fad

Después de revisar diversas fuentes, incluida la investigación de Randellini (2023), se ha decidido utilizar la EfficientNet como la red neuronal convolucional (CNN) para este proyecto debido a que esta red se destaca por su eficiencia y precisión en tareas de clasificación de imágenes, la principal ventaja de esta red es el equilibrio entre el rendimiento y el consumo de recursos (Potrimba, 2023). Esto se logra mediante una escala equilibrada y uniforme de las dimensiones de profundidad, ancho y resolución de la red, lo que permite obtener mejores resultados en comparación con otras arquitecturas de CNN.

4.4. EfficientNetV2

Es una red neuronal convolucional basada en un concepto llamado "escalado compuesto". Este concepto aborda la relación entre el tamaño del modelo, la precisión y la eficiencia computacional (Mingxing, Quoc, 2021). La idea detrás de este escalado compuesto es escalar tres dimensiones esenciales de una red neuronal: ancho, profundidad y resolución.

- **Ancho**: el escalado de ancho se refiere al número de canales en cada capa de la red neuronal. Al aumentar el ancho, el modelo puede capturar patrones y características más complejas, lo que resulta en una mejor precisión.
- **Profundidad**: el escalado de profundidad se refiere al número total de capas en la red. Los modelos más profundos pueden capturar representaciones de datos más complejas, pero también requieren más recursos computacionales. Por otro lado, los modelos más superficiales son computacionalmente eficientes, pero conllevan una pérdida en la precisión.
- **Resolución**: el escalado de resolución implica ajustar el tamaño de la imagen de entrada. Las imágenes de mayor resolución proporcionan información más detallada, lo que mejora el rendimiento. Sin embargo, también requieren más memoria y potencia computacional. Por otro lado, las imágenes de menor resolución consumen menos recursos, pero pueden llevar a una pérdida de detalles finos.

Una de las fortalezas de EfficientNet es su capacidad para equilibrar estas tres dimensiones de manera sistemática. Se parte de un modelo base para realizar una búsqueda exhaustiva y así encontrar la combinación óptima de ancho, profundidad y resolución. A diferencia de las prácticas convencionales que escalan estos factores de manera arbitraria, el método de escalado de EfficientNet ajusta uniformemente el ancho de la red, la profundidad y la resolución utilizando un conjunto fijo de coeficientes de escala. Este ajuste se guía por un coeficiente compuesto, conocido como "phi" (ϕ), que ajusta uniformemente las dimensiones del modelo. Este valor ϕ actúa como un parámetro definido por el usuario que determina la complejidad general del modelo y los recursos requeridos.

Su funcionamiento comienza con un modelo base, normalmente es una red neuronal de tamaño medio que funciona bien en una tarea específica pero que no está optimizada para eficiencia computacional. Se introduce dicho coeficiente compuesto, ϕ, que determina cuánto escalar uniformemente las tres dimensiones de la red.

Las dimensiones se escalan de manera coordinada. El escalado del ancho implica elevar ϕ a un exponente α, el escalado de la profundidad a un exponente β, y el escalado de la resolución multiplicando la resolución original (r) por ϕ elevado a un exponente γ.

El siguiente paso es determinar los exponentes óptimos, α, β y γ, momento vital. Estas constantes generalmente se encuentran mediante búsquedas empíricas en cuadrículas o a través de un proceso de optimización para lograr el mejor equilibrio entre precisión del modelo y eficiencia computacional.

Una vez que se establecen los exponentes de cada escala, se aplican al modelo base, lo que resulta en una variante de EfficientNet con un valor específico de ϕ. Hay multitud de modelos de EfficientNet, cada uno correspondiente a diferentes valores de ϕ. Valores

de ϕ más pequeños proporcionan modelos ligeros adecuados para recursos limitados, mientras que valores más grandes ofrecen más potencia a costa de una mayor intensidad computacional.

El método de escalado compuesto de EfficientNet permite explorar una amplia gama de arquitecturas que equilibran eficazmente precisión y uso de recursos. Este enfoque ha revolucionado el aprendizaje profundo al lograr un rendimiento de vanguardia en tareas de visión por computadora en diversos entornos de hardware.

Su arquitectura utiliza capas Mobile Inverted Bottleneck (MBConv), que combinan convoluciones separables en profundidad y bloques residuales invertidos. Además, la arquitectura del modelo utiliza la optimización Squeeze-and-Excitation (SE) para mejorar aún más el rendimiento del modelo.

La capa MBConv es un bloque de construcción fundamental de la arquitectura de EfficientNet. Comienza con una convolución en profundidad, seguida de una convolución de puntos (convolución 1x1) que expande el número de canales, y finalmente otra convolución 1x1 que reduce los canales de vuelta al número original. Este diseño de botella permite que el modelo aprenda de manera eficiente mientras mantiene un alto grado de capacidad representativa.

Además de las capas MBConv, EfficientNet incorpora el bloque SE, que ayuda al modelo a aprender a centrarse en características esenciales y suprimir las menos relevantes. Dicho bloque, utiliza agrupación promedio global para reducir las dimensiones espaciales del mapa de características a un solo canal, seguido de dos capas completamente conectadas. Estas capas permiten al modelo aprender dependencias de características por canal y crear pesos de atención que se multiplican con el mapa de características original, enfatizando la información importante.

La principal innovación de EfficientNetV2 radica en su enfoque en la mejora de la eficiencia de entrenamiento y la optimización de parámetros. Para lograr esto, se han implementado tres mejoras clave: la modificación del espacio de búsqueda, una función objetivo actualizada y un enfoque de entrenamiento progresivo (Potrimba, 2023).

EfficientNetV2 introduce varias mejoras significativas con respecto a EfficientNet original (Toolify.ia, 2024):

- **Optimización de la arquitectura**: Optimiza aún más la arquitectura de red mediante el uso de nuevas estructuras y técnicas. Ayudando a aumentar la eficiencia y el rendimiento del modelo.
- **Escalado compuesto refinado**: El proceso de escalado compuesto se ha refinado para obtener modelos aún más eficientes y rápidos.
- **Reducción en el número de parámetros**: Se enfoca en reducir el número de parámetros necesarios para lograr un rendimiento dado, lo que resulta en modelos más pequeños y rápidos. Esto se logra mediante la optimización de la estructura de las capas y el uso eficiente de técnicas de regularización.
- **Mejora en la velocidad de entrenamiento**: Gracias a las optimizaciones mencionadas y a la reducción en la complejidad del modelo, EfficientNetV2 ofrece tiempos de entrenamiento más rápidos en comparación con su predecesor.

EfficientNetV2 representa una evolución significativa de EfficientNet, mejorando la eficiencia, el rendimiento y la velocidad de entrenamiento mediante técnicas avanzadas de optimización de redes neuronales y escalado compuesto refinado.

4.5. Entrenamiento

Una vez seleccionada la red se diseña el programa por el cual se va a entrenar la red con la información que se desea. Se ha utilizado Google Colab para esta función por su capacidad de trabajar en la nube. La ilustración 28 muestra la preparación del entorno.

El código de entrenamiento se puede dividir en distintos apartados:

- **Preparar el entorno**

  Se ha instalado el framework de aprendizaje profundo llamado P*ytorch*, es una librería de código abierto desarrollada por Facebook. Es una de las principales herramientas, junto con *Tensorflow*, utilizadas para la programación en el ámbito de la inteligencia artificial y aprendizaje automático.

```
!pip install torch torchvision
!pip install efficientnet_pytorch
```

Ilustración 28: Preparar el entorno Colab
Fuente: Elaboración propia

- **Importar los datos**

  La ilustración 29 muestra parte del código para importar los datos que se habían obtenido en el formato YOLO al entorno de Colab. Para esto, se utiliza la librería *gdown*, que sirve para descargar un archivo de un enlace y almacenarlo en una ruta especificada. También se utiliza la librería *zipfile* para descomprimir el archivo descargado.

```
from google.colab import drive
drive.mount('/content/drive')

import gdown
import zipfile

url = 'https://drive.google.com/uc?id=1eqknpyuK1krWlLBaHERpcOOZeYwQ9AUk'
#https://drive.google.com/file/d/1eqknpyuK1krWlLBaHERpcOOZeYwQ9AUk/view?usp=drive_link
output = '/content/YOLO.zip'

gdown.download(url, output, quiet=False)

with zipfile.ZipFile('/content/YOLO.zip', 'r') as zip_ref:
    zip_ref.extractall('/content/YOLO')

# ver los archivos extraídos
!ls /content/YOLO
print("\nImages Directory:")
!ls /content/YOLO/images
print("\n--------------------\nLabels Directory:")
!ls /content/YOLO/labels
```

Ilustración 29: Importar los datos
Fuente: Elaboración propia

- **Crear el *DataLoader***

En la ilustración 30 primero se especifican las clases que tiene que detectar, las cuales son 5 (Cebra, Coche, Contenedores, Piscina, Rotonda). A continuación, se crean los *datasets*, tanto de entrenamiento como de validación, se definen los parámetros del valor al que se quiere redimensionar la imagen, el tamaño de la cuadricula de la red YOLO y el tamaño del *batch*.

```
class YoloDataset(Dataset):
    def __init__(self, images_dir, labels_dir, S, B, C, transform=None):
        self.images_dir = images_dir
        self.labels_dir = labels_dir
        self.transform = transform
        self.S = S
        self.B = B
        self.C = C
        self.image_files = [f for f in os.listdir(images_dir) if f.endswith('.png')]

        print(f"Found {len(self.image_files)} image files.")
        if len(self.image_files) > 0:
            print("First few image files:", self.image_files[:5])

    def __len__(self):
        return len(self.image_files)

    def __getitem__(self, idx):
        img_path = os.path.join(self.images_dir, self.image_files[idx])
        label_path = os.path.join(self.labels_dir, self.image_files[idx].replace('.png', '.txt'))

        # Cargar imagen
        image = Image.open(img_path).convert('RGB')
        image = image.resize((128, 128), Image.LANCZOS)  # Redimensionar a 128x128 píxeles

        # Cargar label
        target = np.zeros((self.S, self.S, self.B * 5 + self.C))
        with open(label_path, 'r') as file:
            label_data = file.readlines()

            for line in label_data:
                values = list(map(float, line.strip().split()))
                class_id = int(values[0])
                num_coords = (len(values) - 1) // 2
                for i in range(num_coords):
                    x = values[1 + 2*i]
                    y = values[2 + 2*i]
```

Ilustración 30: Crear el DataLoader
Fuente: Elaboración propia

- **Definir el Modelo**

La ilustración 31 muestra cómo se definen los pesos pre-entrenados que se va a utilizar. Para la EfficientNet existen 8 tipos (desde b0 hasta b7) los cuales van realizando un escalado compuesto para aumentar el ancho, la resolución y la profundidad. Teniendo en cuenta la precisión y las limitaciones computacionales y consultándolo con un equipo de expertos se llega a la conclusión de utilizar el modelo b4.

```
from efficientnet_pytorch import EfficientNet
import torch.nn as nn

class EfficientNetV2Model(nn.Module):
    def __init__(self, num_classes, S, B):
        super(EfficientNetV2Model, self).__init__()
        self.effnet = EfficientNet.from_pretrained('efficientnet-b4', num_classes=num_classes)
        self.S = S  #cuadrícula
        self.B = B  #bounding boxes por celdita
        self.num_classes = num_classes
        self.classifier = nn.Linear(self.effnet._fc.in_features, S * S * (B * 5 + num_classes))
        self.effnet._fc = nn.Identity()

    def forward(self, x):
        x = self.effnet(x)
        x = self.classifier(x)
        x = x.view(-1, self.S, self.S, self.B * 5 + self.num_classes)  # Asegurarse de que la salida tenga la forma correcta
        return x

# Crear una instancia del modelo
model = EfficientNetV2Model(num_classes=num_classes, S=S, B=B)

# configurar el dispositivo (GPU o CPU)
device = torch.device("cuda" if torch.cuda.is_available() else "cpu")
model = model.to(device)
```

Ilustración 31: Crear el Modelo
Fuente: Elaboración Propia

- **Definir la Función de Pérdida**

La función de perdida es una medida que evalúa cuan bien o mal se desempeña el modelo de aprendizaje en comparación con las etiquetas verdaderas (ilustración 32). Calcula la discrepancia entre las predicciones del modelo y los valores reales de los datos.

Al usar datos en formato YOLO la función de pérdida que se utiliza también es la YOLO, la cual está diseñada para penalizar las predicciones incorrectas en los *bounding boxes*.

```
class YoloLoss(nn.Module):
    def __init__(self, S, B, C, lambda_coord=5, lambda_noobj=0.5):
        super(YoloLoss, self).__init__()
        self.S = S  # tamaño de la grid
        self.B = B  #  bounding boxes por celdita
        self.C = C  # clases
        self.lambda_coord = lambda_coord
        self.lambda_noobj = lambda_noobj

    def compute_iou(self, box1, box2):
        #Calculo de la Interseccion sobre la Union
#Convertir las coordenadas del centro (x, y) y las dimensiones (w, h) de las bounding boxes en coordenadas de las esquinas (x1, y1, x2, y2).
        box1 = box1.unsqueeze(0)
        box2 = box2.unsqueeze(0)

        box1_x1 = box1[..., 0] - box1[..., 2] / 2
        box1_y1 = box1[..., 1] - box1[..., 3] / 2
        box1_x2 = box1[..., 0] + box1[..., 2] / 2
        box1_y2 = box1[..., 1] + box1[..., 3] / 2
        box2_x1 = box2[..., 0] - box2[..., 2] / 2
        box2_y1 = box2[..., 1] - box2[..., 3] / 2
        box2_x2 = box2[..., 0] + box2[..., 2] / 2
        box2_y2 = box2[..., 1] + box2[..., 3] / 2
          #coordenadas del rectangulo de interseccion
        inter_x1 = torch.max(box1_x1, box2_x1)
        inter_y1 = torch.max(box1_y1, box2_y1)
        inter_x2 = torch.min(box1_x2, box2_x2)
        inter_y2 = torch.min(box1_y2, box2_y2)
          #area del rectangulo de interseccion
        inter_area = (inter_x2 - inter_x1).clamp(0) * (inter_y2 - inter_y1).clamp(0)
          #area de cada bounding box
        box1_area = (box1_x2 - box1_x1) * (box1_y2 - box1_y1)
        box2_area = (box2_x2 - box2_x1) * (box2_y2 - box2_y1)
          #area de la union
        union_area = box1_area + box2_area - inter_area
```

Ilustración 32: Definir la Función de Pérdida
Fuente: Elaboración propia

- **Entrenar el modelo**

La ilustración 33 muestra el código para entrenar el modelo.

Se definen la tasa de aprendizaje, el número de clases y las épocas de entrenamiento, así como el modelo y la función de pérdida que se han definido y el optimizador que se va a usar, en este caso Adam, debido a su adaptabilidad y eficiencia.

Este proceso dependiendo de los parámetros que se han establecido anteriormente tiene distintas duraciones, la media en este proyecto ha sido de 1 hora por entrenamiento.

```python
# Parámetros del modelo
num_classes = 5
num_epochs = 50
learning_rate = 0.001

# Definir optimizador y parámetros de entrenamiento
model = EfficientNetV2Model(num_classes=num_classes, S=S, B=B)
criterion = YoloLoss(S=S, B=B, C=num_classes)
optimizer = optim.Adam(model.parameters(), lr=learning_rate)

# Función de entrenamiento
def train_model(model, train_loader, val_loader, criterion, optimizer, num_epochs):
    model.to(device)
    model.train()


    for epoch in range(num_epochs):
        epoch_loss = 0.0
        model.train()
        for images, targets in train_loader:
            images = images.to(device)
            targets = targets.to(device)

            optimizer.zero_grad()
            outputs = model(images)

            # elementos que se esperan
            num_elements = outputs.numel()

            # Adaptar la salida del modelo a la forma esperada por YoloLoss
            outputs = outputs.view(-1, criterion.S, criterion.S, criterion.B * 5 + criterion.C)
            assert outputs.numel() == num_elements, "Numero de elementos despues de reshaping no es el mismo que el output original"

            # Verificar las formas de las salidas y las etiquetas
            print(f"Output shape: {outputs.shape}")
            print(f"Target shape: {targets.shape}")
```

Ilustración 33: Entrenar el Modelo
Fuente: Elaboración propia

- **Evaluación del Modelo**

En la ilustración 33 se muestra la función de evaluación del modelo, donde se obtiene los valores de pérdida de la evaluación, de precisión y de exactitud.

```python
# Función de evaluación
def evaluate_model(model, dataloader, criterion):
    model.eval()
    eval_loss = 0.0
    all_preds = []
    all_targets = []

    with torch.no_grad():
        for images, targets in dataloader:
            images = images.to(device)
            targets = targets.to(device)

            outputs = model(images)
            loss = criterion(outputs, targets)

            eval_loss += loss.item()

            # Recopilar predicciones y targets para calcular métricas
            all_preds.append(outputs.cpu())
            all_targets.append(targets.cpu())

    print(f'Evaluation Loss: {eval_loss/len(dataloader)}')

    # Convertir listas a tensores
    all_preds = torch.cat(all_preds)
    all_targets = torch.cat(all_targets)

    # Convertir las predicciones y targets continuas a etiquetas binarias (0 o 1)
    all_preds = (all_preds > 0.5).long()
    all_targets = (all_targets > 0.5).long()

    #métricas
    precision, recall, f1, _ = precision_recall_fscore_support(
        all_targets.view(-1),
        all_preds.view(-1) > 0.5,
        average='macro'
    )

    print(f'Precision: {precision:.4f}, Recall: {recall:.4f}, F1 Score: {f1:.4f}')

evaluate_model(model, val_loader, criterion)
```

Ilustración 34: Evaluación del Modelo
Fuente: Elaboración propia

4.6. Optimización de Hiperparámetros

Una vez el código está en funcionamiento hay que realizar distintas pruebas para obtener los resultados con los mejores valores posibles. La tabla 2 muestra los 11 entrenamientos que se han realizado y que presentan estos resultados:

Entrenamiento	Loss	Precision	Recall	F1 Score
1	1434.1876831054688	0.6937	0.6134	0.6376
2	2.5520519018173218	0.6937	0.6134	0.6376
3	3.8397676944732666	0.4319	0.499	0.4634
4	2.3793258666992188	0.6817	0.5905	0.6135
5	2.71382999420166	0.6931	0.6603	0.6741
6	2.3581748008728027	0.7093	0.6511	0.6731
7	2.294529438018799	0.6657	0.5932	0.6144
8	2.373502731323242	0.7072	0.6018	0.6282
9	2.2696008682250977	0.7206	0.6236	0.6523
10	2.1888604164123535	0.7524	0.6163	0.6488
11	2.2089715003967285	0.7429	0.5664	0.5878

Tabla 2: Valores obtenidos en los entrenamientos
Fuente: Elaboración propia

En cada uno de ellos, se ha ido variando algunos parámetros para ir mejorando el modelo. Los parámetros que pueden variar son los siguientes:

- ***Batch_size***: Define el número de muestras de entrenamiento que se utilizarán para actualizar los parámetros del modelo en cada iteración. Cuanto más pequeño menos memoria se requiere para procesar, aunque cuanto mayor sea más estable es el modelo, suele variar entre los 8, 16, 32, 64, 128.
- ***Image_resize***: Redimensiona los datos de entrada para ajustar el tamaño a una dimensión específica para que todas las imágenes de entrada sean compatibles con la arquitectura. Las imágenes más pequeñas reducen el tiempo de procesamiento, pero hace que se pierdan detalles.
- ***Lambda_coord***: Es un factor de ponderación que controla la importancia relativa del error de coordenadas de los *bounding boxes*. Sirve para marcar lo que se considera error o no calculando la diferencia de las coordenadas predichas y reales de los delimitadores.

- ***Lambda_noobj***: Es un factor de ponderación que afecta a la predicción de los delimitadores, si contienen objetos o no. En el entrenamiento se predicen muchas *bounding boxes*, muchas de las cuales no contienen objetos, con este hiperparámetro se reduce la importancia de las cajas sin objetos dándole mayor importancia a las que tienen objetos.
- ***Learning_rate***: Es un escalar que determina la variación de los pesos de una red en cada paso del entrenamiento. Si la tasa de aprendizaje es muy alta puede saltarse la mejor solución, si es muy baja avanza muy lentamente.
- ***Num_epochs***: Indica el número de veces que la red va a ver y aprender todo el conjunto de datos. Si el número de épocas es muy alto puede haber un sobreajuste (*overfitting*), lo cual no permite detectar bien los datos nuevos.
- ***Reduction***: Indica cómo se agregan los valores de pérdida de cada época a la pérdida total, hay 3 maneras: *Sum* (suma de pérdidas individuales), *Mean* (promedio de pérdidas individuales), *None* (devuelve una pérdida para cada muestra). La diferencia entre *Sum* y *Mean* es que *Mean* es independiente del tamaño del lote mientras que *Sum* es dependiente.

Las variaciones de los parámetros en los distintos entrenamientos realizados han sido las siguientes:

- 1º Entrenamiento:
    - Batch_size = 8
    - Image_resize = 128x128
    - Lambda_coord = 5
    - Lambda_noobj = 0.5
    - Learning_rate = 0.001
    - Num_epochs = 50
    - Reduction = “sum”

- 2º Entrenamiento:
    - Batch_size = 8
    - Image_resize = 128x128
    - Lambda_coord = 5
    - Lambda_noobj = 0.5
    - Learning_rate = 0.001
    - Num_epochs = 50
    - Reduction = “mean”

- 3º Entrenamiento:
    - Batch_size = 8
    - Image_resize = 128x128
    - Lambda_coord = 5
    - Lambda_noobj = 0.5
    - Learning_rate = 0.00001
    - Num_epochs = 50
    - Reduction = “mean”

- 4º Entrenamiento:
    - Batch_size = 16
    - Image_resize = 128x128
    - Lambda_coord = 5
    - Lambda_noobj = 0.5
    - Learning_rate = 0.001
    - Num_epochs = 50
    - Reduction = “mean”

- 5º Entrenamiento:
    - Batch_size = 32
    - Image_resize = 128x128
    - Lambda_coord = 5
    - Lambda_noobj = 0.5
    - Learning_rate = 0.001
    - Num_epochs = 50
    - Reduction = “mean”

- 6º Entrenamiento:
    - Batch_size = 24
    - Image_resize = 256x256
    - Lambda_coord = 5
    - Lambda_noobj = 0.5
    - Learning_rate = 0.001
    - Num_epochs = 50
    - Reduction = “mean”

- 7º Entrenamiento:
    - Batch_size = 16
    - Image_resize = 256x256
    - Lambda_coord = 5
    - Lambda_noobj = 0.5
    - Learning_rate = 0.001
    - Num_epochs = 50
    - Reduction = “mean”

- 8º Entrenamiento:
    - Batch_size = 24
    - Image_resize = 256x256
    - Lambda_coord = 5
    - Lambda_noobj = 0.5
    - Learning_rate = 0.001
    - Num_epochs = 75
    - Reduction = “mean”

- 9º Entrenamiento:
    - Batch_size = 24
    - Image_resize = 256x256
    - Lambda_coord = 5
    - Lambda_noobj = 0.5
    - Learning_rate = 0.001
    - Num_epochs = 100
    - Reduction = “mean”

- 10º Entrenamiento:
    - Batch_size = 20
    - Image_resize = 288x288
    - Lambda_coord = 5
    - Lambda_noobj = 0.5
    - Learning_rate = 0.001
    - Num_epochs = 150
    - Reduction = “mean”

- 11º Entrenamiento:
    - Batch_size = 20
    - Image_resize = 288x288
    - Lambda_coord = 5
    - Lambda_noobj = 0.5
    - Learning_rate = 0.0001
    - Num_epochs = 150
    - Reduction = “mean”

El entrenamiento que mejores resultados ha tenido ha sido el 10. Este es el modelo de aprendizaje que se utilizará para realizar la detección.

4.7. Obtención de Resultados

A continuación, la ilustración 35 muestra el código Python para utilizar la red que se acaba de entrenar en imágenes sin etiquetar para detectar los elementos urbanos.

```python
# Definir la estructura del modelo
class EfficientNetV2Model(nn.Module):
    def __init__(self, num_classes, S, B):
        super(EfficientNetV2Model, self).__init__()
        self.effnet = EfficientNet.from_pretrained('efficientnet-b4')
        self.S = S
        self.B = B
        self.num_classes = num_classes
        self.classifier = nn.Linear(self.effnet._fc.in_features, S * S * (B * 5 + num_classes))
        self.effnet._fc = nn.Identity()

    def forward(self, x):
        x = self.effnet(x)
        x = self.classifier(x)
        x = x.view(-1, self.S, self.S, self.B * 5 + self.num_classes)
        return x

num_classes = 5
S = 7
B = 2

# Cargar el modelo
model = EfficientNetV2Model(num_classes=num_classes, S=S, B=B)
model.load_state_dict(torch.load(model_path))
model.eval()  # Modelo en modo de evaluación
```

Ilustración 35: Front para utilizar la Red Neuronal
Fuente: Elaboración propia

Para realizar este código se define el modelo que se va a utilizar y se cargan los parámetros que se han exportado del modelo entrenado a un archivo *.pth*. Se realiza un re-escalado de la imagen de entrada para que tenga el mismo tamaño que las imágenes con las que se ha entrenado el modelo (ilustración 36), se generan los polígonos de predicción y se les aplica la clase que el modelo detecta que son los distintos elementos.

Por último, se exporta los polígonos obtenidos a formato *.shp* para poder visualizarlos sobre la imagen en un software SIG.

```python
cell_size = 1.0 / S
polygons = []
for i in range(S):
    for j in range(S):
        for b in range(B):
            confidence = output[0, i, j, b * 5 + 4]
            if confidence > threshold:
                x, y, w, h = output[0, i, j, b * 5:b * 5 + 4]
                x = (j + x) * cell_size
                y = (i + y) * cell_size
                w = w * cell_size
                h = h * cell_size

                class_probs = output[0, i, j, B * 5:]
                class_id = torch.argmax(class_probs)
                class_name = classes[class_id]

                # Convertir coordenadas de la normalización al tamaño de la imagen
                img_height, img_width = image.shape[:2]
                x1, y1 = int((x - w/2) * img_width), int((y - h/2) * img_height)
                x2, y2 = int((x + w/2) * img_width), int((y + h/2) * img_height)

                # Crear un polígono
                polygon = Polygon([(x1, y1), (x2, y1), (x2, y2), (x1, y2)])
                polygons.append({'geometry': polygon, 'class': class_name})

                rect = patches.Rectangle((x1, y1), x2-x1, y2-y1, linewidth=2, edgecolor='r', facecolor='none')
                ax.add_patch(rect)
                plt.text(x1, y1, class_name, bbox=dict(facecolor='white', alpha=0.5))
```

Ilustración 36: Predicción del modelo entrenado
Fuente: Elaboración propia

5. Resultados

EL resultado obtenido consiste en una capa vectorial de polígonos formada por elementos pertenecientes a las clases correspondientes al modelo. Al introducirlo en QGIS sobre la ortofoto correspondiente se visualiza el resultado que podemos observar en la ilustración 37.

Ilustración 37: Resultado obtenido sobre imagen
Fuente: Elaboración propia

Se observa que el resultado es bastante bueno, pero hay que ser precavidos porque al evaluar el entrenamiento hemos obtenido una precisión de 0.7524 y un *recall* de 0.6163, por lo tanto, es posible que haya elementos erróneos.

Al acercarse a los detalles se observan algunos como:

- En la ilustración 38 existen piscinas no detectadas.

Ilustración 38: Piscinas no detectadas
Fuente: Elaboración propia

- La ilustración 39 detecta Contenedores cuando en verdad son coches.

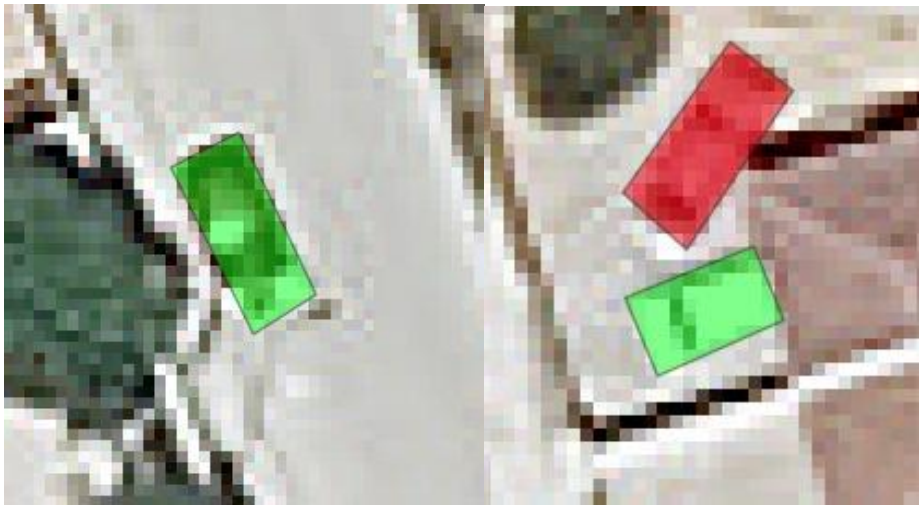


Ilustración 39: Coches mal detectados
Fuente: Elaboración propia

- La ilustración 40 muestra elementos no detectados

Ilustración 40: Elementos no detectados
Fuente: Elaboración propia

A pesar de estos errores, la red detecta la mayoría de los elementos de manera satisfactoria por lo que se podría concluir que la red funciona correctamente dentro de un pequeño margen de error.

6. Presupuesto

Para el presupuesto de este proyecto hay que tener en cuenta tanto los costes de software, de la adquisición de los datos y el coste del personal técnico especializado.

Por la parte de software se han utilizado QGIS y *Label Studio*, los dos son gratuitos, por lo que no supone ningún gasto.

Los datos se han obtenido de forma gratuita del CNIG por lo que tampoco existe coste alguno en este apartado.

Por lo que respecta al coste del personal, este aparece reflejado en la resolución del 27 de febrero de 2023 del convenio colectivo nacional de empresas de ingeniería; oficinas de estudios técnicos; inspección, supervisión y control técnico y de calidad. En esta resolución, tal y como se muestra en la ilustración 41, aparece el grupo profesional correspondiente (Grupo profesional 1 / nivel Salarial 1 - INGENIERO; ARQUITECTO; DOCTOR; LICENCIADO; TITULADO 2.º Y 3.º CICLO UNIVERSITARIO; GRADUADO UNIVERSITARIO CON MÁSTER UNIVERSITARIO OFICIAL HABILITANTE O MÁSTER UNIVERSITARIO OFICIAL).

Grupo profesional	Nivel salarial	Personal Técnico	Personal Administrativo
		Puestos de trabajo (relación no exhaustiva)	Puestos de trabajo (relación no exhaustiva)
I	1	INGENIERO; ARQUITECTO; DOCTOR; LICENCIADO; TITULADO 2.º Y 3.º CICLO UNIVERSITARIO; GRADUADO UNIVERSITARIO CON MÁSTER UNIVERSITARIO OFICIAL HABILITANTE O MÁSTER UNIVERSITARIO OFICIAL (MÍN. 60 ECTS), CUANDO APORTE ESPECIALIZACIÓN Y COMPETENCIAS PROFESIONALES NECESARIAS PARA EL DESEMPEÑO DEL PUESTO DE TRABAJO. ANALISTA.	LICENCIADO; TITULADO 2.º Y 3.º CICLO UNIVERSITARIO; GRADUADO UNIVERSITARIO CON MÁSTER UNIVERSITARIO OFICIAL HABILITANTE O MÁSTER UNIVERSITARIO OFICIAL (MÍN. 60 ECTS), CUANDO APORTE ESPECIALIZACIÓN Y COMPETENCIAS PROFESIONALES NECESARIAS PARA EL DESEMPEÑO DEL PUESTO DE TRABAJO.
	2	GRADUADO UNIVERSITARIO; INGENIERO TÉCNICO; ARQUITECTO TÉCNICO; APAREJADOR; DIPLOMADO UNIVERSITARIO; TITULADO 1er. CICLO UNIVERSITARIO.	GRADUADO UNIVERSITARIO; DIPLOMADO UNIVERSITARIO; TITULADO 1er. CICLO UNIVERSITARIO
II	3	TÉCNICO DE CÁLCULO O DISEÑO; PROGRAMADOR INFORMÁTICO.	JEFE 1.ª ADMINISTRATIVO
	4	DELINEANTE-PROYECTISTA.	JEFE 2.ª ADMINISTRATIVO
III	5	DELINEANTE; TÉCNICO 1.ª; TÉCNICO MODELADOR BIM; TÉCNICO INFORMÁTICO.	OFICIAL 1.ª ADMINISTRATIVO; TRADUCTOR E INTÉRPRETE NO JURADO DE UNO O MÁS IDIOMAS EXTRANJEROS
	6	TÉCNICO 2.ª	OFICIAL 2.ª ADMINISTRATIVO

Ilustración 41: Grupos profesionales
Fuente: BOE

Nivel salarial	Tabla salarial art. 33		Plus convenio según art. 38 convenio	Total anual
	Mes x 14	Anual		
1	1.765,51	24.717,14	2.396,68	27.113,82
2	1.331,06	18.634,84	2.396,68	21.031,52
3	1.283,52	17.969,28	2.396,68	20.365,96
4	1.176,74	16.474,36	2.396,68	18.871,04
5	1.051,43	14.720,02	2.396,68	17.116,70
6	905,87	12.682,18	2.396,68	15.078,86
7	875,48	12.256,72	2.396,68	14.653,40
8	864,28	12.099,92	2.396,68	14.496,60
9	857,97	12.011,58	2.396,68	14.408,26

Ilustración 42: Tabla salarial
Fuente: BOE

El salario del convenio aparece en la ilustración 42 y es 1.765,51€ al mes, siendo un salario anual de 24.717,14€, añadiendo el plus del convenio deja un total anual de 27.113,82. Añadiéndole un 40% del salario bruto anual (10.845,53€), el coste anual del ingeniero es de 37.959,35€.

Esto hace un coste diario de 158,17€, lo que se convierte en 19,77€ la hora. La estimación de horas del proyecto es de 400 horas, repartidas en tres meses y medio, incluyendo las fases de obtención de datos, procesamiento, análisis de metodologías, aplicación y redacción del informe.

$$Coste\ Total = 400\ horas\ x\ 19{,}77€ = 7.908€$$

Tabla 3: Coste total
Fuente: Elaboración propia

7. Conclusiones

Tanto el objetivo general como los objetivos específicos propuestos en este proyecto se han conseguido satisfactoriamente.

Se ha realizado una clasificación de una selección de objetos urbanos mediante el uso de una red neuronal. Para ello, se han obtenido las imágenes y se ha realizado un tratamiento de estas mediante la implementación de distintos programas Python para poder realizarles su posterior etiquetado en la plataforma *Label Studio* el cual se ha exportado con el formato YOLO. Estas imágenes etiquetadas se han utilizado como entrenamiento y evaluación de una red neuronal convolucional del tipo EfficientNet obteniendo una precisión y *recall* aceptables para los objetivos. Para finalizar, se ha implementado otro programa Python para poder utilizar ese modelo obtenido en el entrenamiento en imágenes sin etiquetar lo cual ha proporcionado una salida en formato vectorial de las predicciones que ha realizado, tal y como podemos verlo en la ilustración 43.

Este trabajo ha servido para diseñar una red que puede facilitar mucho el trabajo de distintos usuarios como empresas o instituciones y así facilitar la gestión de los recursos tanto económicos como humanos que disponen.

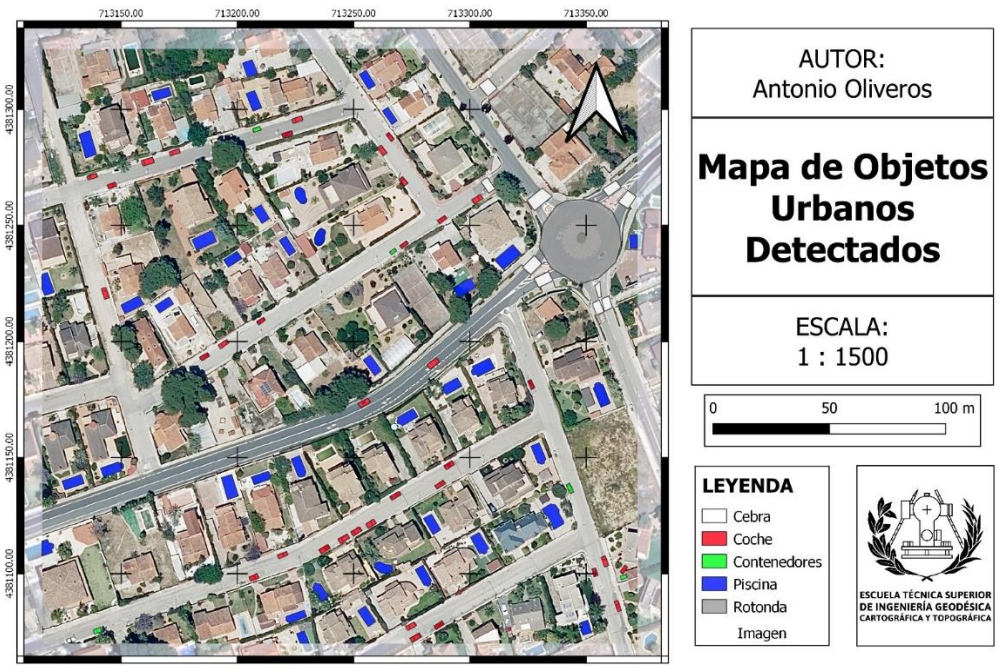


Ilustración 43: Plano Final Obtenido
Fuente: Elaboración Propia

8. Bibliografía

Álvaro Artola Moreno (2019). Clasificación de imágenes usando redes neuronales convolucionales en Python. Depósito de Investigación Universidad de Sevilla. (https://hdl.handle.net/11441/89506)

Aniket Thomas (2023). EfficientNet Implementation from scratch in Pytorch: A step-by-step guide. Medium. (https://medium.com/@aniketthomas27/efficientnet-implementation-from-scratch-in-pytorch-a-step-by-step-guide-a7bb96f2bdaa)

Antony García González (2023). PyTorch y MNIST: Cómo entrenar una red neuronal para clasificación de imágenes. Panamahitek. (https://panamahitek.com/pytorch-y-mnist-como-entrenar-una-red-neuronal-para-clasificacion-de-imagenes/?utm_content=cmp-true)

Bases de Datos para Deep y Machine Learning. (https://virtual.cuautitlan.unam.mx/intar/?page_id=1060)

Centro Nacional de Información Geográfica. Centro de Descargas. (https://centrodedescargas.cnig.es/CentroDescargas/index.jsp)

Documentación oficial de Hugging Face para EfficientNet en la biblioteca Transformers. (https://huggingface.co/docs/transformers/v4.41.3/model_doc/efficientnet)

Enrico Randellini (2023). Image classification: ResNet vs EfficientNet vs EfficientNet_v2 vs Compact Convolutional Transformers. Medium. (https://medium.com/@enrico.randellini/image-classification-resnet-vs-efficientnet-vs-efficientnet-v2-vs-compact-convolutional-c205838bbf49)

Jacob Solawetz (2020). How to train EfficientNet – Custom Image Classification. Roboflow. (https://blog.roboflow.com/how-to-train-efficientnet/)

Javier Martínez Llamas (2018). Reconocimiento de imágenes mediante redes neuronales convolucionales. Archivo Digital UPM. (https://oa.upm.es/53050)

Jonathan Víctor Aguilar-Alvarado, Milton Alfredo Campoverde-Molina (2020). Clasificación de frutas basadas en redes neuronales convolucionales. Dialnet. (https://dialnet.unirioja.es/servlet/articulo?codigo=7436055)

Manlio Massiris, Claudio Delrieux, Juan Álvaro Fernández (2018). Detección de equipos de protección personal mediante red neuronal convolucional YOLO. Repositorio Universidad de Coruña. (http://hdl.handle.net/10662/8846)

Marta Fernández Rebollos (2004). Mobiliario urbano: un elemento diferenciador en las ciudades. Bricojardinería & Paisajismo, 8. (https://www.academia.edu/download/34699810/10_17.pdf)

Mingxing Tan, Quoc V. Le (2019). EfficientNet: Rethinking Model Scaling for Convolutional Neural Networks. Arxiv Cornell University. (https://arxiv.org/abs/1905.11946)

Mingxing Tan, Quoc V. Le (2021). EfficientNetV2: Smaller Models and Faster Training. Arxiv Cornell University. (https://arxiv.org/pdf/2104.00298)

Petru Potrimba (2023). What is EfficientNet? The Ultimate Guide. Roboflow. (https://blog.roboflow.com/what-is-efficientnet/)

Repositorio de Pytorch con la implementación de una EfficientNet. (https://github.com/Levigty/EfficientNet-Pytorch/tree/master)

Repositorio oficial de Tensorflow, implementación oficial de EfficientNet. (https://github.com/tensorflow/tpu/tree/master/models/official/efficientnet)

Sección de la documentación oficial de Keras que describe la implementación de EfficientNet. (https://keras.io/api/applications/efficientnet/)

Silvia Jiménez de las Heras (2018). Clasificación de imágenes médicas mediante redes convolucionales. Repositorio UAM. (https://repositorio.uam.es/handle/10486/688150)

Siria Sadeddin (2020). Clasificación de imágenes (Rayos-X) de Tórax con EfficientNet. SIRIASADEDDINAI. (https://siriasadeddin.wixsite.com/siriaai/post/clasificaci%C3%B3n-de-im%C3%A1genes-rayos-x-de-t%C3%B3rax-con-efficientnet)

Toolify.ia (2024). EfficientNetV2 - Modelos más pequeños y entrenamiento más rápido | Explicación del artículo. Toolify.ia. (https://www.toolify.ai/es/ai-news-es/efficientnetv2-modelos-ms-pequeos-y-entrenamiento-ms-rpido-explicacin-del-artculo-484946)

Vikram Sandu (2022). EfficientNet from Scratch. Kaggle. (https://www.kaggle.com/code/vikramsandu/efficientnet-from-scratch/notebook)

Willy Alex Castañeda Sánchez, Benjamín Roldan Polo Escobar, Fernando Vega Huincho (2023). Redes neuronales artificiales: una medición de aprendizajes de pronósticos como demanda potencial. Universidad, Ciencia y Tecnología. (https://ve.scielo.org/scielo.php?script=sci_arttext&pid=S1316-48212023000100051)

9. Anexos

9.1. Anexo 1

Código de archivo Python para recortar imágenes

```
import os
import shutil
from pathlib import Path
import math
import logging
import logging.handlers
import requests
from optparse import OptionParser, OptionGroup
from osgeo import gdal, ogr, osr
import glob
import cv2
import numpy as np
import json
import matplotlib



gdal.UseExceptions()


matplotlib.use('TkAgg')


logging.getLogger("matplotlib").setLevel(logging.WARNING)
logging.getLogger("shutil").setLevel(logging.WARNING)


def open_images(image_dir):
    log = logging.getLogger(__name__)
    image_paths = glob.glob(os.path.join(image_dir, '**', '*.tif'), recursive=True)
    images = []
```

```
    for image_path in image_paths:
        log.info(f"Image path: {image_path}")
        images.append(image_path)  # Agregar la ruta de la imagen, no el objeto Dataset
    return images


def tiles(options):
    log = logging.getLogger(__name__)
    log.info('Ejecutando tiles')


    try:
        image_dir = r'**********' # directorio donde se encuentran las carpetas donde estan las fotos (no pueden estar dentro de un zip)
        images = open_images(image_dir)
        if len(images) < 1:
            log.error('No se encontraron suficientes imágenes.')
            return -1


        output_dir = r'**********'
        if not os.path.exists(output_dir):
            os.makedirs(output_dir)


        # Iterar sobre cada imagen y recortar en azulejos
        for image_path in images:
            image_name = os.path.basename(image_path)
            try:
                image = gdal.Open(image_path)
                if image is None:
                    log.warning(f"No se pudo abrir la imagen: {image_name}. No se encontró información dentro del archivo.")
                    continue  # Saltar esta imagen y continuar con la siguiente
            except Exception as e:
                log.warning(f"No se pudo abrir la imagen: {image_name}. Error: {e}")
```

```
            continue  # Saltar esta imagen y continuar con la siguiente

        width = image.RasterXSize
        height = image.RasterYSize

        # Obtener la información de la proyección y la transformación geotransform de la imagen de origen
        projection = image.GetProjection()
        geotransform = image.GetGeoTransform()

        tile_size = 1024
        # Calcular la cantidad de azulejos en cada dimensión
        num_tiles_x = math.ceil(width / tile_size)
        num_tiles_y = math.ceil(height / tile_size)

        # Recortar la imagen en azulejos de tile_size x tile_size
        for y in range(num_tiles_y):
            for x in range(num_tiles_x):
                left = x * tile_size
                upper = y * tile_size
                right = min((x + 1) * tile_size, width)
                lower = min((y + 1) * tile_size, height)

                # Leer la parte de la imagen correspondiente al azulejo
                tile_data = []
                for band_index in range(image.RasterCount):
                    band = image.GetRasterBand(band_index + 1)
                    band_data = band.ReadAsArray(left, upper, right - left, lower - upper)
                    tile_data.append(band_data)

                # Convertir la lista de datos de bandas en un arreglo numpy 3D
```

```
            tile_data = np.array(tile_data)

            # Verificar si el resultado tiene el formato esperado
            if tile_data.ndim != 3 or tile_data.shape[0] != image.RasterCount:
                log.warning(f"El azulejo {image_name} no tiene el formato esperado. Saltando este azulejo.")
                continue  # Saltar este azulejo y continuar con el siguiente

            # Calcular las coordenadas geográficas del azulejo utilizando la transformación geotransform
            tile_geo_x = geotransform[0] + left * geotransform[1] + upper * geotransform[2]
            tile_geo_y = geotransform[3] + left * geotransform[4] + upper * geotransform[5]

            # Guardar el azulejo en formato GeoTIFF
            tile_name = f"{os.path.splitext(image_name)[0]}_tile_{y}_{x}.tif"  # Se quita la extensión .jp2 del nombre de la imagen de origen
            tile_path = os.path.join(output_dir, tile_name)

            # Crear una nueva imagen GeoTIFF para el azulejo
            driver = gdal.GetDriverByName('GTiff')
            dataset = driver.Create(tile_path, tile_data.shape[2], tile_data.shape[1], image.RasterCount, gdal.GDT_Float32)
            for band_index in range(image.RasterCount):
                dataset.GetRasterBand(band_index + 1).WriteArray(tile_data[band_index])
            dataset.SetProjection(projection)  # Aplicar la misma proyección que el archivo de origen
            dataset.SetGeoTransform((tile_geo_x, geotransform[1], geotransform[2], tile_geo_y, geotransform[4], geotransform[5]))  # Aplicar la transformación geotransform al azulejo
            dataset.FlushCache()
            log.info(f"Guardado tile: {tile_path}")
    except Exception as oEx:
        log.error(oEx)
    log.info('...DONE!')
```

9.2. Anexo 2

Código de archivo Python para eliminar imágenes sin información

```
import os
import shutil
from pathlib import Path
import math
import logging
import logging.handlers
import requests
from optparse import OptionParser, OptionGroup
from osgeo import gdal, ogr, osr
import glob
import cv2
import numpy as np
import json
import matplotlib

gdal.UseExceptions()

matplotlib.use('TkAgg')

logging.getLogger("matplotlib").setLevel(logging.WARNING)
logging.getLogger("shutil").setLevel(logging.WARNING)

def open_images(image_dir):
    log = logging.getLogger(__name__)
    image_paths = glob.glob(os.path.join(image_dir, '**', '*tile*.tif'), recursive=True)
    return image_paths

def is_image_black(image_path):
    """
```

```
    Verifica si una imagen es completamente negra.
    """
    with gdal.Open(image_path) as dataset:
        if dataset is None:
            return False
        array = dataset.ReadAsArray()
        return np.all(array == 0)

def ValidarImagenes(options):
    log = logging.getLogger(__name__)
    log.info('Revisando imagenes validas')
    image_dir = r'G:\Mi unidad\TFM\Imagenes\33_tilesprueba'

    try:
        # Obtener las imágenes
        image_paths = open_images(image_dir)
        if not image_paths:
            log.error('No se encontraron imágenes válidas en el directorio especificado.')
            return

        # Eliminar imágenes negras
        for image_path in image_paths:
            if is_image_black(image_path):
                os.remove(image_path)
                log.info(f'Imagen eliminada: {image_path}')

    except Exception as oEx:
        log.error(oEx)
    log.info('...DONE!')
```

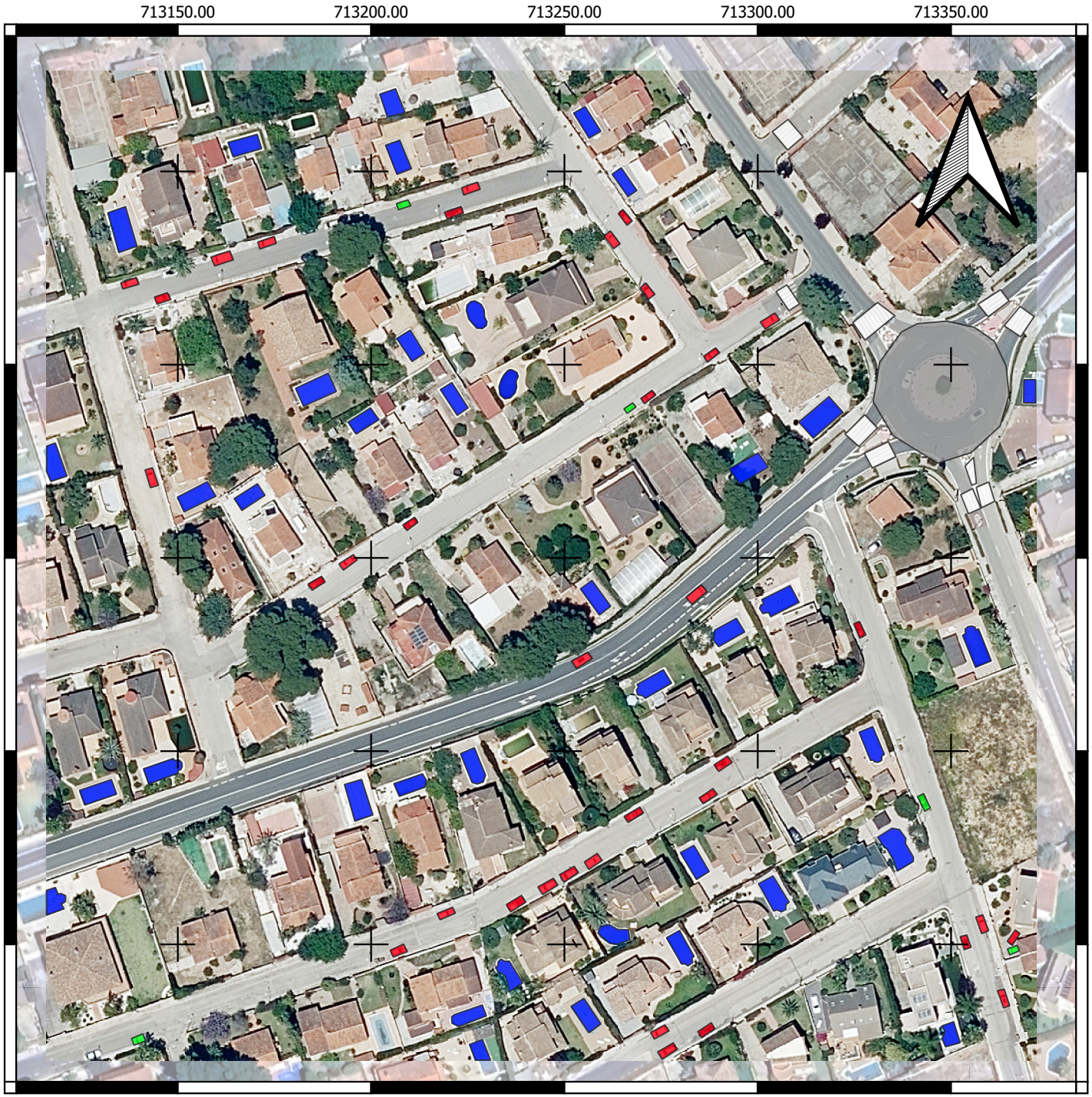
9.3. Anexo 3

Código de archivo Python para convertir archivos TIFF a PNG

```
import os
import shutil
from pathlib import Path
import math
import logging
import logging.handlers
import requests
from optparse import OptionParser, OptionGroup
from osgeo import gdal, ogr, osr
import glob
import cv2
import numpy as np
import json
import matplotlib

gdal.UseExceptions()

matplotlib.use('TkAgg')

logging.getLogger("matplotlib").setLevel(logging.WARNING)
logging.getLogger("shutil").setLevel(logging.WARNING)

def open_images(image_dir):
    log = logging.getLogger(__name__)
    image_paths = glob.glob(os.path.join(image_dir, '**', '*.tif'), recursive=True)
    return image_paths

def tifapng(options):
    log = logging.getLogger(__name__)
```

```
log.info('Transformando (TIF a PNG)')
image_dir = r'G:\Mi unidad\TFM\Imagenes\3_Tiles'
output_dir = r'G:\Mi unidad\TFM\Imagenes\4_PNGs'



try:
    # Get image paths
    image_paths = open_images(image_dir)
    if not image_paths:
        log.error('No se encontraron imágenes válidas en el directorio especificado.')
        return

    if not os.path.exists(output_dir):
        os.makedirs(output_dir)

    # Loop through images and convert to PNG
    for image_path in image_paths:
        try:
            # Read TIFF image
            image = cv2.imread(image_path, -1)  # -1 for including alpha channel
            # Get image name and directory for output PNG
            filename = os.path.splitext(os.path.basename(image_path))[0]
            output_path = os.path.join(output_dir, filename + '.png')
            # Save as PNG
            cv2.imwrite(output_path, image)
            log.info(f'Imagen convertida: {image_path} -> {output_path}')
        except Exception as e:
            log.error(f'Error al convertir imagen {image_path}: {e}')
except Exception as oEx:
    log.error(oEx)
log.info('...DONE!')
```

10. Cartografía



AUTOR:
Antonio Oliveros

Mapa de Objetos Urbanos Detectados

ESCALA:
1 : 1500

0 50 100 m

LEYENDA

- Cebra
- Coche
- Contenedores
- Piscina
- Rotonda

Imagen

ESCUELA TÉCNICA SUPERIOR
DE INGENIERÍA GEODÉSICA
CARTOGRÁFICA Y TOPOGRÁFICA