UNIVERSIDAD DE LA RIOJA

TRABAJO FIN DE ESTUDIOS

Título

Modelado geométrico para el etiquetado automático de imágenes

Autor/es

MARTA ELVIRA REY

Director/es

ANA ROMERO IBÁÑEZ y JOSE DIVASÓN MALLAGARAY

Facultad

Escuela de Máster y Doctorado de la Universidad de La Rioja

Titulación

Máster universitario en Ciencia de Datos y Aprendizaje Automático

Departamento

MATEMÁTICAS Y COMPUTACIÓN

Curso académico

2020-21

publicaciones.unirioja.es
E-mail: publicaciones@unirioja.es

Trabajo de Fin de Máster

Modelado geométrico para el etiquetado automático de imágenes

Geometric modeling for automatic image labeling

Autor: ***Marta Elvira Rey***

Tutores: Ana Romero Ibáñez y Jose Divasón Mallagaray

MÁSTER:
Máster en Ciencia de Datos y Aprendizaje Automático

Escuela de Máster y Doctorado

UNIVERSIDAD DE LA RIOJA

AÑO ACADÉMICO: 2020/2021

Resumen

En este proyecto se ha estudiado una técnica de aumento de datos para el aprendizaje automático utilizando datos de modelos 3D ya creados.

Utilizando el programa Blender y a partir de modelos geométricos de distintos objetos, se han generado diferentes tipos de imágenes etiquetadas de forma automática modificando el objeto, la cámara, los puntos de luz y las condiciones del fondo. Se ha utilizado el lenguaje de programación Python para hacer las modificaciones aleatoriamente.

Se han validado los resultados construyendo modelos de aprendizaje supervisado con datasets de imágenes reales e imágenes generadas y comprobando su éxito para la clasificación y la detección de objetos sobre imágenes reales de diferentes datasets.

Abstract

In this project, a data augmentation technique for machine learning using data from already created 3D models has been studied.

Using the Blender program and starting from geometric models of different objects, different types of labeled images have been generated automatically by modifying the object, the camera, the light points and the background conditions. The Python programming language has been used to make the modifications randomly.

The results have been validated by building supervised learning models with datasets from real images and generated images and verifying their success for classifying and detecting objects on real images from different datasets.

Índice general

1. Introducción **1**
1.1. Antecedentes 1
1.2. Alcance 9
1.3. Tareas a realizar 11

2. Estudio de trabajos existentes **15**
2.1. Data generation for urban driving scenes 15
2.2. Data Augmentation Based on 3D Model Data 19
2.3. Using virtual people for real image analysis 20
2.4. Blender for Computer Vision Machine Learning 21
2.5. Data augmentation using 3d graphical engines 22

3. Generación de imágenes sintéticas **25**
3.1. Creación del script 25
3.2. Ejecución del script 34

4. Creación de modelos de redes neuronales **41**
4.1. Modelo de clasificación de imágenes 41
4.2. Modelo de detección de objetos 44

5. Validación de resultados **49**
5.1. Clasificación 49
5.2. Detección 53

Conclusiones y posibles mejoras **57**

Bibliografía **59**

Capítulo 1

Introducción

En este capítulo se van a explicar cuáles son los problemas que han motivado la realización de este proyecto y el objetivo que se intenta lograr.

1.1. Antecedentes

Este proyecto surge del problema que se tiene en el aprendizaje automático (*machine learning*), y sobre todo en el aprendizaje profundo (*deep learning*), cuando nos encontramos con datasets que tienen poca cantidad de datos. Los modelos de aprendizaje profundo, como las redes neuronales, requieren toneladas de datos etiquetados de calidad para evitar el sobreajuste en el entrenamiento. Aquí es donde entra en juego el aumento de datos. El aumento se realiza cuando los datos existentes son insuficientes, incompletos o simplemente si el problema requiere más variación en los datos [22].

El aumento de datos (*data augmentation*), es una técnica para crear un gran conjunto de imágenes de entrenamiento a partir de un pequeño conjunto de imágenes originales mediante diversas transformaciones, que incluyen volteo, rotación, recorte, cambio de color o luz, inyección de ruido y aplicación de filtros. El objetivo es enseñar al modelo las distintas configuraciones que puede adoptar un objeto [31]. Un ejemplo de esta técnica se ve en la Figura 1.1.

Para mejorar el rendimiento del reconocimiento de imágenes basado en el aprendizaje automático, en este proyecto estudiaremos un nuevo método de aumento de datos basado en modelos 3D. En vez de partir de imágenes 2D y realizar transformaciones para obtener imágenes diferentes del mismo objeto, lo que se propone es partir de un modelo geométrico 3D de un objeto al que se le aplicarán modificaciones de posición, rotación, escalado, luz, cámara, textura y fondo, y posteriormente obtener las imágenes bidimensionales. Además, mediante este proceso se podrá obtener un etiquetado de las imágenes de manera automática, cuyo proceso de forma manual es muy costoso y lento, sobre todo

Figura 1.1: *Data augmentation* a la imagen de un gato [22].

para problemas de detección y segmentación.

El uso de escenas virtuales para el aumento de datos y la generación de conjuntos de datos sintéticos se está empleando cada vez más en visión artificial. Por ejemplo, el conjunto de datos SYNTHIA [24] es un modelo enorme y completamente virtual de una ciudad, con personas reales e incluso mascotas, que se utiliza para producir datos para entrenar algoritmos de conducción autónoma. El mundo virtual permite colocar cámaras y recrear situaciones que son muy difíciles de hacer en la vida real. Por lo tanto, crear datos sintéticos es mucho más barato que crear un conjunto de datos reales. Sin embargo, crear escenas sintéticas es increíblemente difícil y requiere mucho tiempo, a menudo solo lo hacen expertos en 3D (modeladores y animadores). Aun así, una sola escena sintética, si está parametrizada adecuadamente, puede generar cientos o miles de muestras.

1.1.1. Modelado geométrico

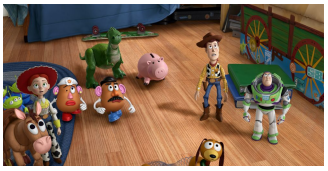
El modelado geométrico consiste en construir y representar objetos 3D en un ordenador [28]. Aunque en cada momento se muestren los objetos en el ordenador como una imagen 2D, los modelos geométricos nos permiten realizar operaciones como mover el objeto, rotarlo, modificar la escena, etc. Por tanto, nos permite ver el objeto desde distintas posiciones. Las imágenes 2D se representan con una matriz de números, que representan los píxeles, mientras que trabajar con modelos geométricos es más complejo ya que además de guardar la geometría del objeto se guarda mucha más información, como texturas, materiales, luces, fondos, etc. Por tanto, los procesos de renderizado, generación de imágenes 2D a partir de modelos 3D, pueden ser muy largos y costosos.

Hay muchas aplicaciones del modelado geométrico (Figura 1.2), entre otras, se utiliza en visualización de datos científicos, como medicina, en diseño de productos, piezas, edificios, interiores, etc., en simuladores, como los que usan los pilotos de aviones, videojuegos, películas de animación, mapas, etc.

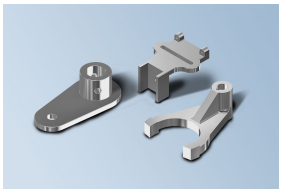
El modelado geométrico ha evolucionado mucho desde sus inicios, actual-

(a) Medicina [29]

(b) Diseño de interiores [12]



(c) Películas de animación [25]



(d) Piezas [18]

Figura 1.2: Aplicaciones del modelado 3D.

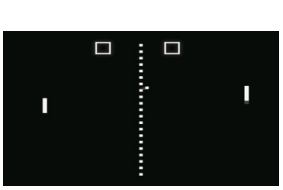
mente se ha conseguido obtener imágenes muy realistas [29]. En películas de animación el gran cambio en el modelado 3D se da con la película Toy Story (Figura 1.2c). Fue la primera película hecha entera con modelos 3D y utilizaron nuevas técnicas que supusieron un gran avance en el modelado geométrico.

La evolución en los gráficos de los videojuegos, en cuanto a realismo, ha sido muy grande (Figura 1.3). Empezaron siendo gráficos muy básicos con figuras 2D en blanco y negro, posteriormente intentaron dar la sensación de profundidad jugando con los fondos y el tamaño, más tarde añadieron complejidad a las escenas utilizando luces y sombras y finalmente con los modelos 3D se consiguió un gran realismo.

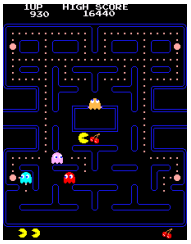
Existen varias formas de generar modelos geométricos: a mano, a partir de objetos reales y matemáticamente o algorítmicamente. Además, existen diferentes tipos de modelos geométricos o de representación de un objeto, todos ellos basados en matemáticas [28, 29]. Los más usados son:

Modelos de puntos: (Figura 1.4)

- **Nube de puntos:** son muestras de puntos en 3D no estructurados, normalmente son adquiridos mediante técnicas de visión por ordenador. Su codificación en el ordenador suele ser en un fichero de texto en el que aparecen las coordenadas de cada punto.
- **Mapa de profundidad:** es un conjunto de puntos en 3D con valor dependiendo de la profundidad, normalmente son adquiridos a través de un escáner 3D.



(a) 1972 Pong



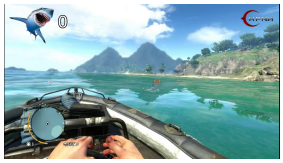
(b) 1980 Pac-Man

(c) 1981 Turbo

(d) 1986 OutRun

(e) 1994 Donkey Kong Country

(f) 2004 Half-Life 2

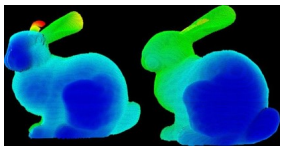


(g) 2012 Far Cry 3

(h) 2018 God of War

Figura 1.3: Evolución gráfica en los videojuegos [13].

(a) Nube de puntos [1]

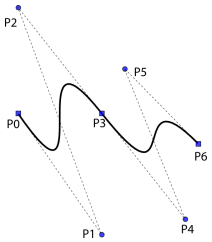


(b) Mapa de profundidad [29]

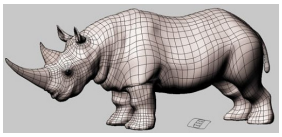
Figura 1.4: Modelos de puntos.

(a) Geometría analítica (Banda de Möbius) [4]

(b) Interpolación de Hermite

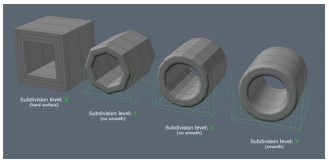


(c) Aproximación (curva de Bézier) [9]

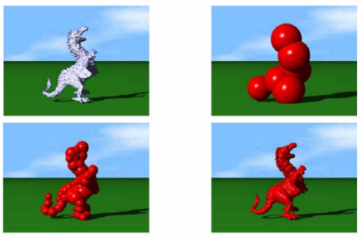


(d) NURBS [19]

(e) Mallas poligonales [7]



(f) Superficies de subdivisión [29]



(g) Metaball [29]

Figura 1.5: Modelos de curvas y superficies.

Modelos de curvas y superficies: Representan la información geométrica en términos de elementos matemáticos. (Figura 1.5)

- **Geometría analítica:** se utilizan las ecuaciones matemáticas de las curvas o superficies. En su codificación aparecen las ecuaciones con las que se han creado las curvas o superficies.
- **Interpolación:** se construye una curva o superficie que pase por una serie de puntos dados. Se utilizan técnicas matemáticas como la interpolación lineal, los polinomios de Lagrange, Hermite y Splines.
- **Aproximación:** se construye una curva o superficie que pase cerca de una serie de puntos dados, sin llegar a pasar por dichos puntos. El método más utilizado son las curvas y superficies de Bézier.
- **NURBS (Non Uniform Rational BSplines):** son una familia de cur-

vas y superficies paramétricas que generalizan la interpolación y aproximación. Una gran ventaja es que utilizando estas curvas se pueden construir muchas de las curvas habituales y muchas curvas suaves que son muy útiles para el modelado geométrico.

Su origen se debe a los ingenieros franceses Pierre Bézier y Paul de Casteljau, quienes crearon las curvas y superficies al mismo tiempo y de forma independiente en dos empresas de automóviles, Renault y Citroën, respectivamente, y las utilizaron para representar el diseño de carrocerías de automóviles.

- **Mallas poligonales:** son una colección de vértices, aristas y polígonos conectados de forma que cada arista es compartida como máximo por dos polígonos. Lo más común es utilizar triángulos, pero se puede usar cualquier tipo de polígono. Además, cuanto más pequeños y más cantidad de polígonos haya se consigue una mayor precisión y por tanto un mayor realismo de los objetos. Los modelos poligonales son muy utilizados debido a su velocidad de procesamiento y a la exactitud de definición que permite.

  Respecto a la codificación, hay distintos tipos de ficheros para guardar las mallas poligonales, pero básicamente se guarda una lista de los vértices, con coordenadas en el espacio, que forman parte de los polígonos y una lista de los polígonos indicando el número de lados y la lista de los índices de los vértices correspondientes a cada polígono.

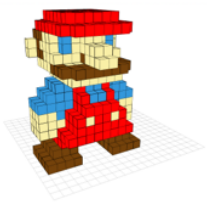
- **Superficies de subdivisión:** consiste en aplicar una secuencia limitada de refinamientos a un objeto básico para conseguir un mayor número de vértices y polígonos y por tanto, una superficie más suave. La posición de los nuevos vértices se calcula mediante unos modelos de refinamiento matemático, interpolación o aproximación.

- **Metaballs o superficies implícitas:** consiste en construir un objeto uniendo bolas. A mayor número de bolas y menor tamaño de estas se consigue mayor nivel de detalle.

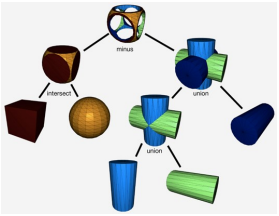
Modelos de sólidos: Estos modelos se crean mediante operaciones con objetos primitivos y a diferencia de los modelos de superficies permiten distinguir entre interior, exterior y superficie de un objeto. Estos modelos permiten calcular diferentes propiedades de los objetos como volumen, masa, etc. (Figura 1.6)

- **Modelos de descomposición:** consiste en describir un objeto como un conjunto de células elementales cuya yuxtaposición llena todo el espacio ocupado por el objeto. Entre ellos se encuentra el modelo de descomposición mediante enumeración exhaustiva, en el que las células son pequeños cubos del mismo tamaño y orientación.

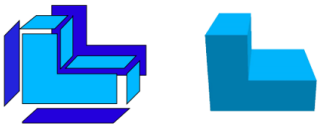
- **Modelos constructivos:** se parte de elementos básicos (cubo, esfera, cilindro...), se les aplica determinadas transformaciones (traslación, rotación, escalado, extrusión, biselado...) y posteriormente se combinan los objetos aplicándoles operaciones booleanas (unión, intersección y diferencia) para obtener el objeto deseado. Las transformaciones de extrusión y biselado son similares. En la extrusión se parte de una curva plana o superficie y se arrastra de forma perpendicular al plano en el que se encuentra



(a) Modelos de descomposición



(b) Modelos constructivos



(c) Modelos de fronteras

Figura 1.6: Modelos de sólidos [29].

para obtener una superficie u objeto. En el biselado se parte de una curva y se arrastra siguiendo una circunferencia u otra curva obteniéndose una especie de tubería. La representación de un modelo constructivo se hace por medio de un árbol.

- **Modelos de fronteras o B-rep (Boundary representation):** Se definen los objetos en función de la superficie que los encierra (su frontera), describiendo sus vértices, aristas y caras. La representación de fronteras más simple es una lista de caras poligonales.

Otra forma de crear modelos geométricos es a partir de objetos reales y se hace mediante la reconstrucción 3D. La reconstrucción 3D es el proceso mediante el cual un objeto real es construido en el ordenador a partir de alguna técnica de modelado geométrico manteniendo sus características físicas (dimensiones, volumen y forma). Hay dos tipos de técnicas:

- **Mediante un escáner 3D:** el escáner va girando y mediante una cámara va obteniendo la forma y la textura del objeto y a partir de ahí obtiene una nube de puntos y su textura. A partir de la nube de puntos se forma una malla poligonal mediante alguna técnica de triangulación, como la triangulación de Delaunay. Esta técnica consiste en formar triángulos a partir de una serie de puntos siguiendo unas condiciones.
- **Fotogrametría:** partiendo de una serie de imágenes bidimensionales de un objeto desde diferentes vistas se construye el modelo 3D del objeto.

1.1.2. Renderizado

Una vez que se tiene creado el modelo geométrico se le pueden añadir texturas y modificar los demás objetos de la escena como la cámara, la luz,

el fondo, etc, y finalmente, renderizarlo. El proceso de renderizado consiste en generar una imagen bidimensional a partir de un modelo geométrico. La propiedad más importante, que hace que una imagen obtenida a partir de un modelo 3D sea realista, es la luz y su interacción con los objetos (reflejos y sombras). Existen varios tipos de motores de renderizado que tratan de forma diferente la luz. Ordenados del más antiguo al más nuevo y del más rápido al más lento en tiempo de ejecución tenemos:

- ***Rasterization*:** utiliza trucos geométricos para detectar qué se va a ver en la pantalla. Es una técnica muy rápida que permite hacer renderizado en tiempo real pero que no es realista. No simula la luz, pero mediante cálculos geométricos se hacen trucos para conseguir sombras y reflejos y así, que las imágenes parezcan más realistas.
- ***Ray casting*:** consiste en trazar un rayo por pixel desde la cámara y encontrar el objeto más cercano que interrumpe su trayectoria. Usando las propiedades del material y el efecto de las luces en la escena, este algoritmo puede estimar el color del objeto pixel a pixel. Esta técnica es rápida y también se utiliza para renderizado en tiempo real, pero no es posible obtener sombras ni reflejos realistas, aunque pueden simularse.
- ***Ray tracing*:** consiste en lanzar un rayo por pixel y cuando intercepta un objeto se le hace rebotar hasta las fuentes de luz para poder calcular mejor la intensidad de la luz, el color y las sombras. Dependiendo del material del objeto la luz rebotará de una manera distinta. Esta técnica produce sombras, refracciones y reflexiones, aunque las sombras son todas fuertes, es decir, no hay posibilidad de crear sombras débiles o más difuminadas. Además, solo muestra lo que se ve a través de la cámara, ya que solo dibuja lo que se ve con luz directa a través de la cámara. Este algoritmo no es adecuado para renderizado en tiempo real.
- ***Path tracing*:** se lanzan muchos rayos por pixel y cuando un rayo contacta con un objeto rebota varias veces hasta que encuentra una fuente de luz. La dirección en la que rebota el rayo depende del tipo de material y superficie. Con esta técnica se consiguen sombras fuertes y débiles y permite mostrar reflexión y refracción, incluso de los objetos que no están en el campo de visión de la cámara. La cantidad de rayos lanzados por pixel puede variar y cuanto menor es el número de rayos mayor es el ruido que hay en la imagen. Esta técnica es mucho más lenta ya que cada rayo lanzado puede convertirse en varios al chocar con un objeto y así sucesivamente hasta que llegan a una fuente de luz.

1.1.3. Blender

Hay muchos programas de modelado, tanto de pago como gratuitos. En este proyecto se va a utilizar Blender, con versión 2.92.0, y lo que se hará es modificar propiedades de modelos geométricos que ya están creados, en concreto un donut y una manzana, mediante un script de Python.

Blender, lanzado en 1998, es uno de los mejores programas enfocado princi-

palmente al modelado, control de iluminación, renderizado, animación y creación de gráficos en 3D [36]. Blender permite crear entornos totalmente virtuales y filmarlos con cámaras virtuales que producen imágenes de aspecto real utilizando algoritmos de renderizado de trazado de rayos. Este programa es totalmente gratuito y de código abierto, y está más que a la altura de otras alternativas similares dentro del ámbito profesional. Incluso algunos estudios de cine (como Marvel) lo han utilizado para animar y procesar sus películas y efectos.

La principal característica de este programa es que es un software totalmente gratuito, de código abierto y multiplataforma, aunque no siempre fue así. Aunque no es un programa especialmente sencillo de utilizar, viene de serie con una gran variedad de figuras geométricas primitivas, incluyendo curvas, mallas, vacíos y metaballs. También cuenta con simulaciones dinámicas para cuerpos blandos y con una gran variedad de herramientas de animación, como cinemática inversa, deformaciones y partículas estáticas y dinámicas. También está pensado para desarrollo de juegos por lo que encontramos herramientas de detección de colisiones y sistemas de recreaciones dinámicas y lógicas.

Este software es compatible con el lenguaje de programación Python. Con él se puede automatizar o controlar varias tareas de edición. En Blender se encuentran todos los tipos de modelos geométricos que se han explicado anteriormente y tiene dos motores de renderizado, *Eevee* que es de tipo *Rasterization* y *Cycles* de tipo *Path tracing*.

1.2. Alcance

El objetivo de este proyecto es generar imágenes sintéticas para hacer *data augmentation* y comprobar si los modelos de aprendizaje supervisado mejoran. Para la validación se usarán modelos de aprendizaje profundo. La gran diferencia entre los modelos de aprendizaje automático (*machine learning*) y aprendizaje profundo (*deep learning*) se encuentra en la extracción de características y por tanto en los datos con los que entrenan. En el aprendizaje automático las características que describen las imágenes se extraen manualmente y los modelos se entrenan con ellas, mientras que en el aprendizaje profundo el proceso de extracción de características lo hace el modelo automáticamente, se entrena con las imágenes directamente (Figura 1.7).

En este proyecto se utilizarán las redes neuronales para comprobar si al aumentar datasets de imágenes reales con imágenes sintéticas los resultados para clasificación y detección de objetos reales mejoran (Figura 1.8). Una red neuronal es una serie de algoritmos que intentan reconocer las relaciones subyacentes en un conjunto de datos a través de un proceso que imita la forma en que funciona el cerebro humano [32]. Se suelen utilizar para problemas de clasificación y regresión pero realmente tienen un gran potencial para resolver multitud de problemas. Son muy buenas para detectar patrones. Las redes neuronales requieren mucha capacidad de procesamiento y memoria y estuvieron

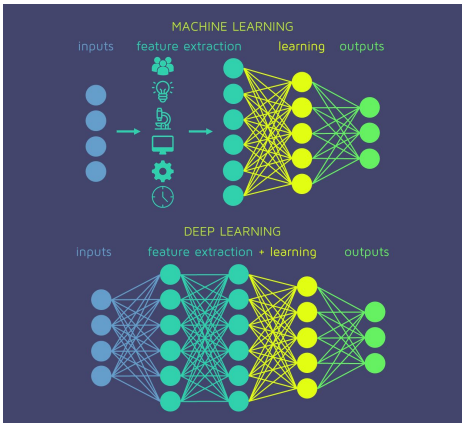


Figura 1.7: Diferencia entre *machine learning* y *deep learning* [17].

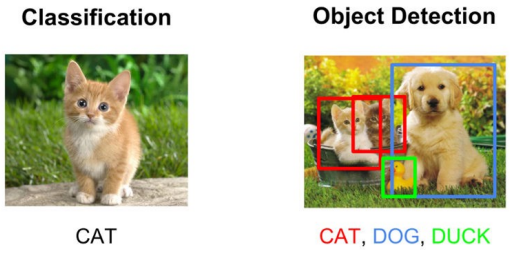


Figura 1.8: Diferencia entre clasificación y detección de objetos [10].

muy limitadas por la tecnología del pasado hasta estos últimos años en los que resurgieron con mucha fuerza dando lugar al aprendizaje profundo [20].

Las redes neuronales se componen de capas de nodos, que contienen una capa de entrada, una o más capas ocultas y una capa de salida (Figura 1.9). Cada nodo, o neurona artificial, se conecta a otro y tiene un peso y un umbral asociados. Si la salida de cualquier nodo individual está por encima del valor de umbral especificado, ese nodo se activa y envía datos a la siguiente capa de la red. De lo contrario, no se transmiten datos a la siguiente capa de la red [30].

En concreto, en este proyecto se utilizará la red neuronal ya entrenada *ResNet-18* para clasificación, a la que solo entrenaremos la última capa con las imágenes que queramos para evitar entrenar toda la red desde cero, lo cual sería muy costoso en tiempo. *ResNet-18* es una red neuronal convolucional que está entrenada en más de un millón de imágenes de la base de datos ImageNet, así, tiene un buen desempeño en la clasificación de imágenes en color. La red tiene 18 capas de profundidad (Figura 1.10) y puede clasificar imágenes en 1000 categorías de objetos. Como resultado, la red ha aprendido representaciones de características ricas para una amplia gama de imágenes. Las imágenes de entra-

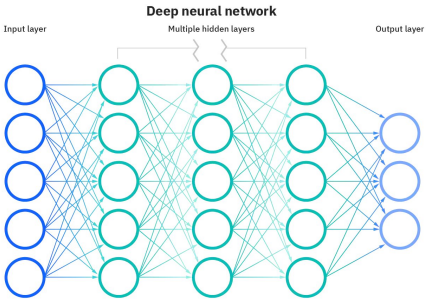


Figura 1.9: Estructura de una red neuronal [30].

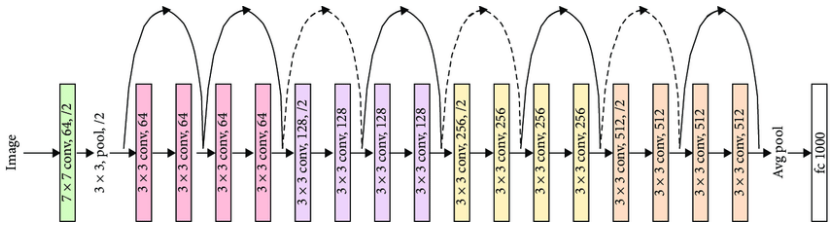


Figura 1.10: Arquitectura de la red neuronal *ResNet-18* [27].

da tienen que tener el mismo tamaño.

Para detección de objetos utilizaremos la red neuronal *Faster R-CNN*. Esta red, ya entrenada, es similar a *R-CNN* y *Fast R-CNN*, sin embargo es mucho más rápida y es capaz de detectar varios objetos en una misma imagen. En la Figura 1.11 se puede ver una versión simplificada de su arquitectura.

1.3. Tareas a realizar

En esta sección se describe el procedimiento que se va a seguir para la realización del proyecto.

1. **Estudio de trabajos existentes:** en primer lugar buscaremos proyectos en los que se utilice el modelado geométrico para el aumento de datasets y poder así coger ideas para realizar este proyecto.

2. **Estudio de Blender:** aprenderemos a manejar el programa Blender mirando su documentación. Nos centraremos en los comandos de Python para así poder automatizar la generación de imágenes sintéticas [3].

   Para usar Blender con Python lo más sencillo es activar la opción *Python tooltips* dentro de la configuración. Todas las opciones de Blender tienen asociadas un comando de Python y así dejando el cursor del ratón encima de un botón del programa aparece su comando asociado. Por tanto es necesario conocer dónde se encuentra cada opción dentro de la interfaz gráfica de Blender.

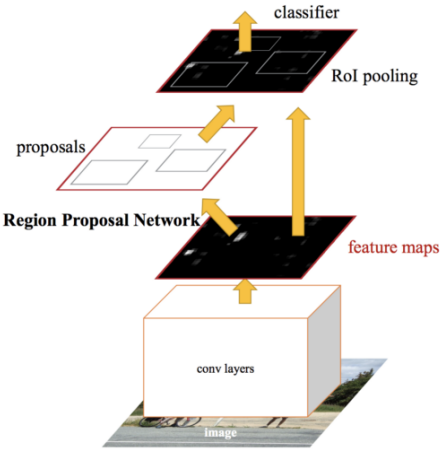


Figura 1.11: Arquitectura de la red neuronal *Faster R-CNN* [21].

El primer problema que nos encontramos es que muchos tutoriales sobre el uso del programa están realizados con versiones anteriores del mismo. Esto es un problema ya que a partir de la versión 2.81 la interfaz gráfica de Blender cambió mucho.

3. **Generación de imágenes sintéticas:** crearemos un script de Python en Blender que haga modificaciones aleatorias a los modelos 3D y genere las imágenes bidimensionales resultantes. Utilizaremos dos modelos gratuitos, un donut [11] y una manzana [2], y les aplicaremos modificaciones de posición, rotación, luces, texturas, etc. Para el problema de detección, haremos que el script defina automáticamente las posiciones de los objetos de la imagen.

4. **Descarga de datasets:** descargaremos imágenes de Google de varios objetos a través de un Notebook de Jupyter para hacer clasificación. Cada imagen se guardará en la carpeta de su clase correspondiente y así tendremos las imágenes etiquetadas. Los objetos que utilizaremos son: donut, manzana, peras, cupcakes, ciruelas y roscón de reyes. Para los modelos de detección de objetos utilizaremos un dataset de COCO [6], del cual nos quedaremos con las clases: plátano, naranja, manzana, donut y pastel.

5. **Creación de modelos de redes neuronales:** realizaremos varias pruebas para poder comprobar la mejoría de los modelos de clasificación y detección al aumentar los datos de entrenamiento con imágenes sintéticas. En primer lugar entrenaremos el modelo solo con las imágenes descargadas de Google y COCO, para cada modelo respectivamente. Después, entrenaremos de nuevo un modelo con las imágenes anteriores y añadiremos algunas sintéticas. Por último, añadiremos mayor cantidad de imágenes sintéticas.

6. **Validación de resultados:** validaremos los modelos anteriores con las imágenes reales y compararemos los resultados de cada modelo.

Para la realización de este proyecto no hemos utilizado ninguna metodología de desarrollo de software ni planificación específica ya que es un proyecto de investigación y experimentación, lo cual hace que sea muy difícil estimar el tiempo de cada tarea. Se partía de cero en temas como Blender, modelado geométrico o renderizado y es imposible prever las dificultades que nos íbamos a ir encontrando. También, debido a la diferente carga que había en las asignaturas que se estaban cursando y el inicio de las prácticas de empresa era difícil saber qué número de horas se iba a poder dedicar al día para el proyecto. Sin embargo, en la última parte de la realización del proyecto, al acabar el período de prácticas, se le ha podido dedicar más horas. Además, al no haber cursado las asignaturas del Grado de Informática en las que se explica la planificación de proyectos no se tenía experiencia sobre ello. Por todo lo dicho, no planificamos un calendario detallado donde se asignen horas y entregables, ya que no sería realista.

Capítulo 2

Estudio de trabajos existentes

Existen varios proyectos en los que tratan el aumento de datos con modelos geométricos. En esta sección se explicarán brevemente algunos de ellos.

2.1. Augmented reality meets computer vision: Efficient data generation for urban driving scenes [26]

En este artículo se proponen mejorar el resultado de los modelos de aprendizaje profundo para segmentación de instancias y detección de objetos utilizando datos aumentados, en particular se pretende detectar objetos de automóviles en escenarios de conducción al aire libre. Los datos aumentados son una combinación entre datos reales y datos sintéticos. En los datos sintéticos toda la escena se crea a partir de modelos geométricos, en cambio los datos aumentados utilizan fondos reales a los que se les añade algún objeto creado con modelos 3D como se ve en la Figura 2.1. Esto permite mantener el realismo total del fondo mientras se generan cantidades arbitrarias de configuraciones de objetos en primer plano. De esta forma, se mejoran considerablemente la precisión de las redes neuronales profundas de última generación entrenadas con datos reales. En particular, se demuestra que un modelo entrenado usando un conjunto de datos aumentado se generaliza mejor que los modelos entrenados puramente con datos sintéticos, así como los modelos que usan un número menor de imágenes reales anotadas manualmente.

En el artículo explican el proceso y las pruebas que realizaron para ver como se obtenían los mejores resultados. Trabajaron con Blender usando su renderizado *Cycles*. Sin embargo, las representaciones obtenidas carecen de los artefactos típicos del proceso de formación de la imagen, como el desenfoque de movimiento, el desenfoque de la lente, las aberraciones cromáticas, etc. Para

(a) Escena real

(b) Escena sintética

(c) Escena real aumentada con coches sintéticos

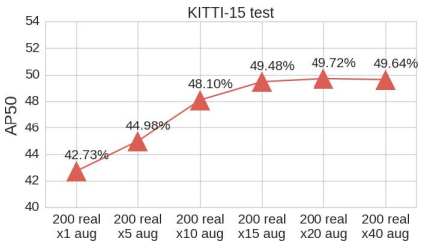
Figura 2.1: Diferentes tipos de escenas [26].

igualar mejor las estadísticas de la imagen del fondo, se realizó un posprocesado a las imágenes para simular esos efectos, lo que dio como resultado representaciones que son visualmente más similares al fondo. Se aplicaron cambios de color para simular aberraciones cromáticas en la lente de la cámara, así como desenfoque de profundidad para que coincida con la profundidad de campo de la cámara. Finalmente, utilizaron varias curvas de color y transformaciones Gamma para que coincidan mejor con las estadísticas de color y contraste de los datos reales.

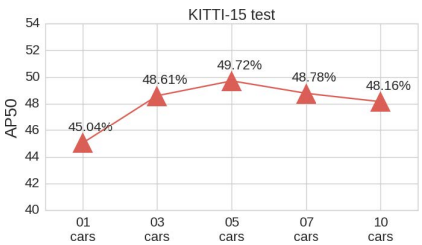
Para lograr un aumento realista de alta calidad, es esencial colocar correctamente los objetos virtuales en la escena en ubicaciones aceptables, haciendo coincidir la distribución de poses y oclusiones en los datos reales. Estudiaron cuatro estrategias de muestreo de ubicación diferentes:

- **Anotaciones manuales de las ubicaciones de los automóviles:** se marcan las posibles trayectorias de los automóviles, se toman muestras de las ubicaciones de estas anotaciones y se establece la rotación a lo largo del eje vertical del automóvil para que se alinee con la trayectoria.
- **Segmentación automática de carreteras:** se usa un algoritmo que segmenta la imagen en áreas viales y no viales con alta precisión, se proyectan hacia atrás esos píxeles de la carretera y se calcula su ubicación en el plano del suelo para obtener posibles ubicaciones de automóviles y se utiliza una rotación aleatoria alrededor del eje vertical del vehículo.
- **Estimación de planos de carreteras:** Dado que se conocen los parámetros intrínsecos de la cámara de captura y su pose exacta, es posible estimar el plano del suelo en la escena.
- **Ubicaciones aleatorias:** se muestrean aleatoriamente ubicaciones y rotaciones de una distribución arbitraria.

Empíricamente, obtienen que las anotaciones de ubicación de automóviles manuales funcionan ligeramente mejor que la segmentación automática de carreteras y están a la par con la estimación de planos de carreteras.



(a) Variación del número de aumentos por imagen real



(b) Variación del número máximo de coches sintéticos renderizados en cada imagen aumentada

Figura 2.2: Rendimiento de la segmentación de instancias utilizando datos aumentados [26].

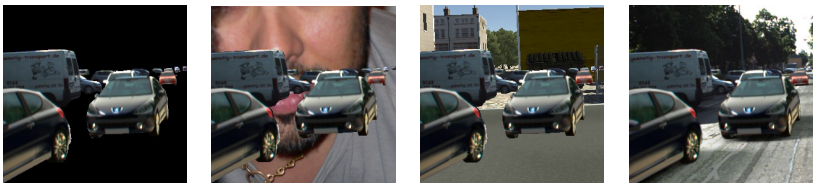
También estudiaron cómo la calidad y la cantidad de datos aumentados afectan al rendimiento de un modelo de segmentación de instancias de última generación. En particular, exploraron cómo el número de aumentos por imagen real y el número de coches sintéticos añadidos afecta a la calidad de los modelos aprendidos.

- **Número de imágenes aumentadas creadas a partir de cada imagen real:** en la Figura 2.2a se muestra cómo el crecimiento del número de aumentos por imagen real mejora el rendimiento del modelo entrenado a través de la diversidad agregada de la clase objetivo, pero al superar los 20 aumentos se satura.
- **Número de coches sintéticos renderizados en cada imagen aumentada:** en la Figura 2.2b se ve que al principio agregar más coches sintéticos mejora el rendimiento al introducir más instancias al conjunto de entrenamiento. Esto proporciona poses de automóvil más novedosas y oclusiones realistas sobre automóviles reales, lo que conduce a modelos más generalizables. Sin embargo, aumentar el número de automóviles más allá de 5 por imagen da como resultado una disminución notable en el rendimiento.

Por tanto, se puede lograr el mejor rendimiento utilizando una combinación equilibrada de datos reales y sintéticos.

Analizaron el efecto del realismo del fondo comparando modelos entrenados con los mismos objetos en primer plano, que consisten en una mezcla de coches reales y sintéticos, mientras cambiaban el fondo, como se ve en la Figura 2.3 usando las siguientes cuatro variaciones:

- **Fondo negro:** tener el mismo fondo negro en todas las imágenes de entrenamiento conduce a un ajuste excesivo al fondo y, en consecuencia, a un rendimiento deficiente en los datos de prueba reales.
- **Imágenes aleatorias:** el uso de imágenes aleatorias mejora el rendimien-



(a) Fondo negro (b) Imagen aleatoria (c) Imagen virtual (d) Imagen real

Figura 2.3: Imágenes aumentadas sobre diferentes fondos [26].

to al evitar el ajuste excesivo del fondo, pero no proporciona ninguna pista semántica significativa para el modelo.

- **Imágenes virtuales:** las imágenes virtuales proporcionan un mejor contexto para los coches en primer plano mejorando la segmentación. Sin embargo, su rendimiento es insuficiente debido a la diferencia de apariencia entre el primer plano y el fondo en comparación con el uso de fondos reales.
- **Imágenes reales:** se obtienen los mejores resultados.

Los resultados muestran claramente el papel importante de las imágenes de fondo y su impacto incluso cuando se utiliza la misma instancia en primer plano.

Otro aspecto importante que puede sesgar la distribución del conjunto de datos aumentado es la ubicación de los coches sintéticos. Experimentaron con cuatro variantes:

- **Colocar aleatoriamente los coches en la escena 3D con rotación 3D aleatoria:** tiene un rendimiento notablemente peor que colocarlos en el plano del suelo. Esto no es sorprendente, ya que los automóviles pueden colocarse en lugares físicamente inverosímiles, que no aparecen en nuestros datos de validación.
- **Colocar aleatoriamente los coches en el plano del suelo con una rotación aleatoria alrededor del eje ascendente:** los resultados obtenidos mediante este proceso son buenos.
- **Usar segmentación semántica para encontrar píxeles de la carretera y proyectarlos en el plano del suelo 3D mientras se configura la rotación alrededor del eje ascendente de forma aleatoria:** tiende a colocar más coches sintéticos en las áreas despejadas de la carretera más cerca de la cámara, lo que cubre la mayoría de los coches reales más pequeños en el fondo, y esto conduce a resultados ligeramente peores.
- **Colocar manualmente los coches:** los resultados obtenidos son similares a los que se obtenían en el segundo caso. Esto indica que las anotaciones manuales no son necesarias para colocar los coches aumentados siempre que se conozcan el plano de tierra y los parámetros de la cámara.

Figura 2.4: Rotaciones al objeto kanji 3D [31].

A través de un extenso conjunto de experimentos, llegaron a obtener el conjunto correcto de parámetros para producir datos aumentados que pueden mejorar al máximo el rendimiento de los modelos de segmentación de instancias. Además, comprobaron que el entrenamiento con datos puramente sintéticos conduce a modelos sesgados que tienen un rendimiento inferior al de datos reales. De manera similar, el entrenamiento con un conjunto de datos de tamaño limitado de imágenes reales restringe el rendimiento de generalización del modelo. Por el contrario, la composición de imágenes reales y automóviles sintéticos en un solo cuadro puede ayudar al modelo a aprender características compartidas entre las dos distribuciones de datos sin ajustar demasiado a las sintéticas. El programa que hicieron para realizar este proyecto es específico para coches y no está disponible públicamente.

2.2. Data Augmentation Based on 3D Model Data for Machine Learning [31]

En este artículo se propone crear modelos 3D para generar automáticamente un conjunto de datos de entrenamiento que cubren un rango continuo de ángulos de visión y diversos fondos. El objeto que se utiliza es el kanji 3D, que es una forma tridimensional generada a partir de un símbolo japonés bidimensional y lo que se quiere conseguir es clasificar 5 tipos de objetos kanji 3D. Para clasificar se utilizan dos redes neuronales convolucionales (CNN) en una relación padre-hijo donde el objeto se considera como el "padre"(clase principal) y los ángulos de visión se consideran como "hijo"(clase secundaria).

En el reconocimiento de imágenes, la dificultad se encuentra en las diferentes proyecciones que produce un objeto según la dirección de la vista, dos proyecciones del mismo objeto podrían ser bastante diferentes pero deben reconocerse como el único objeto. Además, las similitudes en los fondos de las imágenes también son problemáticas. Por tanto, partiendo de un modelo geométrico del kanji 3D hacen rotaciones al objeto sobre los ejes x e y para producir varias imágenes (Figura 2.4). Para cada una de las imágenes generadas en el paso anterior, el fondo negro se reemplaza por diferentes imágenes y cada una de las imágenes generadas se etiqueta con el nombre del objeto y la cantidad de rotación que han recibido. Los mismos objetos que se crearon mediante modelado 3D fueron fabricados por una impresora 3D y se usaron para tomar fotografías mientras los sostenían con las manos y los giraban (Figura 2.5).

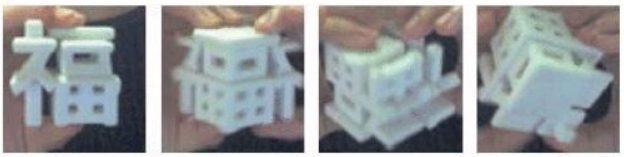


Figura 2.5: Imágenes reales del objeto kanji 3D [31].

Utilizaron la red neuronal *DenseNet121* entrenada con ImageNet y una capa completamente conectada. Se usó ReLU para la activación de la capa completamente conectada y Softmax como activación de la capa de salida. La diferencia entre la red secundaria y la red principal está en el número de nodos por capa de salida. En total, se realizaron tres experimentos utilizando las redes de padres e hijos.

- **Categorización de clases secundarias de imágenes sintéticas utilizando la red secundaria:** Los resultados muestran que la red secundaria se entrenó con éxito para categorizar las formas kanji 3D.
- **Categorización de la clase principal de imágenes sintéticas con ambas redes:** Los resultados muestran que la precisión de la red secundaria es mejor que la red principal correspondiente en más de la mitad de los casos.
- **Categorización de la clase principal de imágenes fotografiadas en el mundo real con ambas redes:** Los resultados muestran que la precisión fue tan baja como esperaban, probablemente debido a la diferencia en la calidad de imagen de las imágenes generadas y fotografiadas que estaba más allá del rango permitido. Mientras que las imágenes generadas tienen una alta resolución y una iluminación homogénea, las imágenes fotografiadas son de menor calidad.

2.3. Receipt data augmentation: Using virtual people for real image analysis [22]

En este artículo se describe el proceso que utiliza la empresa Way2VAT para analizar imágenes de recibos para sus clientes, utilizando una gran cantidad de datos obtenidos a partir de las cámaras de teléfonos móviles. El proceso que siguen es encontrar el recibo en la imagen, segmentarlo del fondo, desenvolverlo de nuevo en un rectángulo plano y luego hacer reconocimiento óptico de caracteres (OCR) y análisis de texto. Cada uno de estos pasos tiene un algoritmo que se basa en montones de datos para entrenamiento, validación y pruebas.

Aplicar *data augmentation* tradicional a las imágenes de recibos no da buenos resultados, debido a que las rotaciones hacen que los bordes del documento del recibo escapen de los límites de la imagen y los giros en espejo y las deformaciones no rígidas destruyen el texto. Por otro lado, las imágenes escaneadas desde el móvil tienen mucha variación. Cada persona tiene su propia

Figura 2.6: Imágenes sintéticas de un recibo de compra [22].

forma de sostener el papel frente a la cámara, además de la cantidad infinita de condiciones de iluminación que puede haber. También existe una increíble variedad de cámaras para teléfonos inteligentes que toman fotografías, para lo cual necesitan crear una solución de escaneo unificada.

Esto les llevó a emplear la generación de datos sintéticos, esto significa crear más imágenes de recibos falsos, utilizando las que ya tienen y utilizando actores humanos en 3D (Figura 2.6). Preparan a los actores virtuales para tomar una foto con un teléfono virtual falso para simular una situación real. También agregan deformaciones por arrugas y rizos a la malla del papel de recibo. Así crean miles de muestras de imágenes de recibos de aspecto real capturadas desde un teléfono móvil para entrenar sus algoritmos. Utilizan scripts de Python en el programa Blender ya que de esta forma es posible programar el proceso, que consume mucho tiempo. Por lo general, realizan los trabajos de renderizado durante el fin de semana o durante el invierno. El resultado final del proyecto no se especifica en el artículo y los scripts que diseñaron no están disponibles públicamente y son específicos para este problema.

2.4. Blender for Computer Vision Machine Learning [35]

Este artículo describe un proyecto que se hizo para la empresa Greppy de robots autónomos para la limpieza de baños. Necesitaban una forma de entrenar su software de visión por computador para reconocer de manera rápida y precisa los asientos de los inodoros a través del aprendizaje automático. El principal desafío era cómo encontrar suficientes imágenes para alimentar el algoritmo de aprendizaje automático y después se necesitaba etiquetarlas con el contorno del asiento del inodoro.

En lugar de recopilar fotos reales y etiquetarlas manualmente, se decidió que sería más rápido generar todas las imágenes mediante modelos 3D y luego enviar esas imágenes al ordenador. Este enfoque tiene algunas ventajas sobre el etiquetado de fotografías de origen colectivo: se pueden generar máscaras de píxeles perfectos y datos de profundidad automáticamente y cubre una amplia gama de diseño, distribución, iluminación y colores de baños.

Por tanto, crearon un baño 3D en Blender y generaron imágenes de más

Figura 2.7: Imágenes de inodoros generadas automáticamente [35].

de 40 modelos y diseños de inodoros reales, incluso incluyeron lavabos y bañeras para asegurarse de que el algoritmo de aprendizaje automático no sobreajustara los datos y pensase que todo lo que es brillante y puede contener agua es una taza de inodoro.

Crearon un script para obtener todas las posibles combinaciones de elementos de la escena. Además, posicionaron aleatoriamente la cámara para que se cubriesen todos los ángulos posibles en los que el robot puede percibir al sujeto. En total obtuvieron más de 86.000 imágenes como las que se ven en la Figura 2.5. Las imágenes no las obtuvieron con la mejor calidad, disminuyeron la fidelidad del color, es decir, cómo de fiel es la representación del color en la pantalla. Esto fue a propósito ya que la cámara para la que se entrena no tiene el rango dinámico ni la respuesta a la luz de una cámara de película de alta gama y así se consigue imitar la curva de respuesta de la cámara robótica real.

El resultado final resultó bueno, la empresa pudo integrar con éxito las imágenes en su canal de aprendizaje automático y se obtuvieron excelentes resultados en la producción de una red neuronal que identifica con precisión y rapidez los asientos de los inodoros de todas las formas y tamaños.

2.5. A semi-supervised data augmentation approach using 3d graphical engines [34]

En este artículo crean un conjunto de datos sintéticos con diferentes poses del cuerpo humano a partir de la reconstrucción 3D. Partiendo de escaneos 3D de solo 7 individuos crean modelos 3D, a los que posteriormente irán moviendo sus articulaciones para crear imágenes bidimensionales de una figura humana con la pose de una persona físicamente válida y etiquetada con precisión.

El objeto 3D es esencialmente un esqueleto digital unido a una malla 3D que consta de articulaciones y huesos. Las articulaciones y los huesos se pueden

Figura 2.8: Imágenes sintetizadas con diferentes poses [34].

mover modificando el descriptor de estado de pose, lo que conduce a la animación de los modelos 3D. Para manipular el modelo escaneado en 3D, emplean herramientas de animación de Blender, así, manipulan las partes del cuerpo y dan la pose que se desea.

Se asigna una geometría de las extremidades al modelo escaneado predefiniendo 14 partes móviles. Entre otras, cabeza, torso, brazo superior izquierdo, brazo inferior izquierdo, palma izquierda, muslo izquierdo, etc. El descriptor de estado de pose es un vector que indica el estado de la articulación mediante un ángulo de rotación.

Además del estado del sujeto, también introducen los parámetros del entorno para generar imágenes realistas. El entorno incluye todos los elementos de la escena y también la iluminación y los parámetros de la cámara. Para el fondo, utilizan imágenes aleatorias dentro del contexto deseado. Para que las imágenes sintéticas fuesen más similares a las reales aplicaron un filtro gaussiano. En la Figura 2.8 se ve un conjunto de imágenes sintéticas generadas.

Para evaluar la calidad de los conjuntos de datos sintetizados, emplearon un algoritmo de estimación de pose humana 2D basado en redes neuronales profundas de última generación y lo entrenaron desde cero con el conjunto de datos sintetizados y compararon su rendimiento de estimación de pose con modelos entrenados en un conjunto de datos de imágenes de pose humana real y un conjunto de datos de pose humana sintética. El modelo de aprendizaje profundo de estimación de pose humana utilizando el conjunto de datos sintéticos pudo lograr una precisión de estimación de pose del 91.2 %.

Capítulo 3

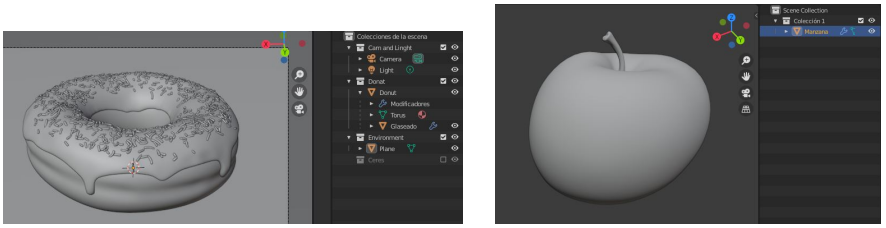
Generación de imágenes sintéticas

En este capítulo se explicará con detalle todo el proceso realizado para la generación de imágenes sintéticas de donuts y manzanas con Blender. Se explicará el script final aunque se empezó haciendo un script para clasificación básico para el donut, luego se hizo otro para la manzana utilizando el ya creado, más tarde se unieron los dos para tener solo un script compatible con los dos modelos y finalmente se fueron añadiendo más detalles para realizar más modificaciones a los objetos, como añadir más de un objeto a la escena y obtener las coordenadas 2D del cuadro delimitador de los objetos, llamado *bounding box*. El código empleado para la generación de imágenes en Blender se encuentra en el Anexo A.

Como ya hemos dicho los modelos que vamos a modificar son un donut (Figura 3.1a) y una manzana (Figura 3.1b). Son dos modelos gratuitos que hemos descargado de las páginas [11] y [2], respectivamente. Como vemos, el modelo de la manzana consta de un solo objeto, por tanto, en primer lugar mediante las opciones en la interfaz gráfica de Blender vamos a separar el rabo del resto de la manzana (Figura 3.2). Para ello desde el modo edición seleccionamos todos los vértices que pertenecen al rabo (Figura 3.2a) y los separamos del objeto manzana, creando así el objeto rabo (Figura 3.2d).

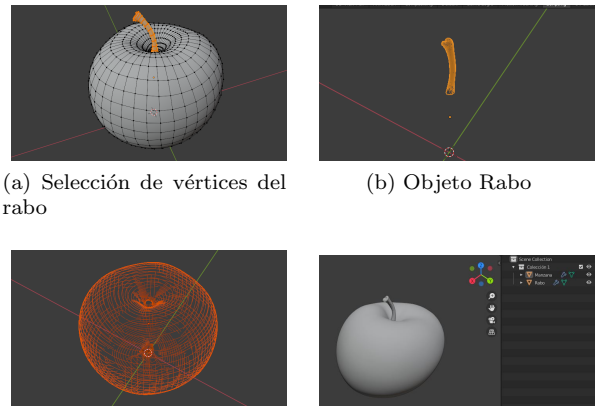
3.1. Creación del script

Una vez que tenemos los dos modelos listos para modificarlos comenzamos a explicar el script. Consta de una serie de funciones que para cada modelo se llamarán con unos parámetros específicos. La llamada de las funciones se hará mediante un bucle para así obtener una imagen diferente en cada iteración. Las librerías que vamos a utilizar en el script son las siguientes:



(a) Donut (b) Manzana

Figura 3.1: Modelos originales.



(a) Selección de vértices del rabo (b) Objeto Rabo

(c) Objeto Manzana (d) Resultado final

Figura 3.2: Proceso de división del modelo de la manzana.

```
import bpy
import random
from pathlib import Path
import colorsys
from math import radians
from time import time
import datetime
import os
import bpy_extras
import xml.etree.ElementTree as ET
import numpy as np
```

La librería `bpy` es la que utiliza Python para realizar operaciones de Blender, `random` lo utilizaremos para obtener valores aleatorios, `Path` sirve para obtener las rutas de los directorios, con `colorsys` modificaremos colores, `radians` es necesaria para las rotaciones, con `os` obtendremos el directorio en el que nos encontramos, `bpy_extras` y `numpy` son necesarias para obtener las coordenadas del *bounding box* del objeto, con `xml.etree.ElementTree` crearemos los archivos xml que contienen las anotaciones para detección y `time` y `datetime` los utilizaremos para mostrar el tiempo que tarda en ejecutarse todo el proceso.

3.1.1. Modificaciones de los objetos

En primer lugar, definimos una función que haga que un objeto sea padre de los demás indicados. En el modelo del donut ya viene inicialmente el donut como padre del glaseado, sin embargo en el modelo de la manzana no aparece de esta forma ya que hemos sido nosotros los que hemos separado en dos objetos independientes. De esta forma va a facilitar algunas de las funciones siguientes.

```
def objeto_padre(padre, hijos)
```

La función tiene como parámetros el nombre del objeto que queremos que sea el padre y una lista con los nombres de los objetos que van a ser hijos. Se comienza deseleccionando todos los objetos para seleccionar únicamente los objetos hijos. Indicamos que el objeto padre sea el objeto activo y finalmente realizamos el parentesco. En Blender, el último elemento seleccionado se denomina objeto activo, por tanto, hay exactamente un objeto activo en cualquier momento. Muchas acciones en Blender usan el objeto activo como referencia, por ejemplo, la rotación y el escalado ocurren alrededor del origen del objeto activo. Todos los demás objetos seleccionados simplemente se seleccionan, pero no son activos.

Construimos una función que modifique aleatoriamente la posición de un objeto, su rotación y su tamaño. Como parámetros tiene una lista con los objetos a los que queremos hacer la misma transformación y los límites, máximo y mínimo, para cada variable, por ejemplo, el valor mínimo y máximo entre los que puede desplazarse el objeto sobre cada eje (x, y, z). Dichos valores de los

parámetros los estimamos experimentalmente haciendo movimientos del objeto desde la interfaz gráfica de Blender para que el objeto no se salga del todo del campo de visión de la cámara. En el caso de los valores correspondientes a la rotación se indican en grados.

```
def objeto_transformacion(names, mov_x_min, mov_x_max, mov_y_min,
↪  mov_y_max, mov_z_min, mov_z_max, rot_x_min, rot_x_max, rot_y_min,
↪  rot_y_max, rot_z_min, rot_z_max, scal_min, scal_max)
```

En primer lugar deseleccionamos todo y seleccionamos los objetos que se especifican en la lista `names` pasada como parámetro para aplicarles transformaciones. Si tenemos los objetos en una relación padre-hijo solo es necesario pasar el objeto padre a la función, si se pasa el hijo también, a este se le aplicarán dos veces las transformaciones. Para mover los objetos usamos la función `bpy.ops.transform.translate` para que mueva los objetos seleccionados. A esta función se le especifica unos valores de x, y, z, pero no corresponden con las coordenadas en las que queremos posicionar los objetos, sino que desplaza los objetos de su posición en el número de unidades indicado a lo largo de cada eje. Los valores x, y, z los obtenemos mediante la función `random.uniform` que devuelve un número aleatorio, mediante una distribución uniforme, dentro del rango de valores que se especifica en los parámetros en la llamada de la función.

Procedemos de forma similar para rotar y escalar los objetos seleccionados. Para cada eje se obtiene un valor aleatorio dentro de su rango correspondiente especificado en los parámetros y aplicamos la función `bpy.ops.transform.rotate` en la que transformamos en radianes el valor obtenido aleatoriamente. Para el escalado obtenemos aleatoriamente un único valor para los tres ejes dentro de los límites indicados y aplicamos la función `bpy.ops.transform.resize`, que aumenta o disminuye el tamaño actual de los objetos proporcionalmente a la cantidad indicada.

Creamos una función que cree un plano dado el tamaño, la posición y la rotación. El nombre del objeto será `Plane`. El hecho de tener un plano sirve para modelar el lugar donde se coloca el objeto, en nuestro caso utilizaremos texturas para simular una mesa y una encimera.

```
def agregar_plano(size, pos_x, pos_y, pos_z, rot_x, rot_y, rot_z)
```

La siguiente función que definimos nos permite cambiar la textura de cada objeto mediante una imagen. Como parámetros tiene el nombre del objeto al que le queremos añadir la textura y un booleano, que por defecto es falso, que nos permite cambiar la forma de proyectar la textura en el objeto. Las imágenes de textura para cada objeto se encuentran en carpetas con el nombre del objeto y dentro de la carpeta Texturas.

(a) Textura para la base del donut

(b) Textura para el glaseado del donut

(c) Textura para la manzana

(d) Textura para el fondo

Figura 3.3: Ejemplos de texturas.

(a) Proyección por defecto

(b) Proyección desde la vista

Figura 3.4: Diferentes resultados para a textura de la manzana.

```
def textura(name_obj, proyect_vista=False)
```

Comenzamos leyendo las imágenes de la carpeta de texturas correspondiente al objeto indicado en los parámetros. Después, seleccionamos aleatoriamente una de las imágenes, creamos un material al que le asignamos la imagen como textura y por último, asignamos ese material al objeto. Las texturas son una imagen en color que visualmente simula una superficie real de un objeto. Por ejemplo, una textura que utilizamos para la base del donut se ve en la Figura 3.3a, para el glaseado utilizamos imágenes como la Figura 3.3b, para la manzana empleamos imágenes como la Figura 3.3c y para el fondo usamos imágenes del estilo a la Figura 3.3d que simulan una mesa de madera u otros materiales.

Para algún objeto damos la posibilidad de que la textura se proyecte desde la vista, ya que si no puede que se noten los bordes de las imágenes en el objeto, como se ve en la Figura 3.4a. Para ello activamos el objeto, entramos en el modo edición y utilizamos la función `bpy.ops.uv.project_from_view` [15]. El resultado no es demasiado realista (Figura 3.4b) pero sí mejor que el anterior.

3.1.2. Modificaciones al entorno

La siguiente función agrega una cámara a la escena en la posición y con la rotación indicadas.

```
def agregar_camara(pos_x, pos_y, pos_z, rot_x, rot_y, rot_z)
```

Una vez que haya una cámara en la escena creamos una función para modificarla. Utilizaremos dos tipos de cámara, ortogonal y perspectiva, que serán elegidos aleatoriamente. Como parámetros hay que indicar el nombre del objeto cámara, el tamaño horizontal del área del sensor de la imagen, el valor de escala de la cámara ortogonal y el valor de lente de la cámara perspectiva. Por tanto, esta función elige aleatoriamente el tipo de cámara y en caso de que sea uno u otro cambia los valores de sus atributos por los indicados en la llamada a la función.

```
def camara(name, sensor_width, ortho_scale, lens)
```

También tenemos que crear una función que agregue un punto de luz a la escena. Debemos indicar la posición, la rotación y la energía lumínica emitida por el área completa de la luz.

```
def agregar_luz(pos_x, pos_y, pos_z, rot_x, rot_y, rot_z, energy)
```

Después de crear la luz hacemos una función que la modifique aleatoriamente. Como parámetros tiene el nombre del objeto, los límites de las tres coordenadas para la posición y rotación del objeto, los valores mínimo y máximo para la energía lumínica, los rangos para modificar el color en formato HSV y los límites para el tamaño de la luz.

```
def luz(name, x_min, x_max, y_min, y_max, z_min, z_max, w_min, w_max,
↪  H_min, H_max, S_min, S_max, V_min, V_max, r_min, r_max)
```

En primer lugar indicamos que la luz sea una fuente de luz puntual omnidireccional que proyecta sombras, para cualquiera de los dos motores de renderizado. Después, se procede de forma similar para los atributos posición, potencia, color y radio. Se elige aleatoriamente con una distribución uniforme un valor entre el rango de cada parámetro especificado en la llamada de la función y dicho valor se le asigna a la luz mediante su atributo correspondiente.

En las tres funciones `agregar_plano`, `agregar_camara` y `agregar_luz` especificamos el nombre con el que se crea el objeto, ya que dependiendo del idioma que esté configurado en Blender por defecto se crea en ese idioma y nos daría problemas si no nos damos cuenta de cambiar los nombres de los objetos en las

llamadas a las funciones.

3.1.3. Reiniciar objetos

Ahora, creamos unas funciones que nos permitan realizar las modificaciones siempre al objeto inicial, ya que los límites de los parámetros de la función `objeto_transformacion` están obtenidos a partir de la posición y rotación del objeto original, y si le pasamos un objeto en otra localización este puede salirse del campo de visión de la cámara.

La función `objeto_original` selecciona los objetos indicados en el parámetro lista `names` y realiza una copia.

```
def objeto_original(names)
```

Para comenzar en cada iteración con el objeto inicial creamos la función `reiniciar_objeto_unico`. Como en las imágenes que íbamos a descargar de Google y COCO no iba a aparecer un único objeto por imagen decidimos crear la función `reiniciar_objeto`, para añadir la posibilidad de que de manera aleatoria unas veces hubiese un objeto en la escena y otras veces añadiese más objetos.

```
def reiniciar_objeto_unico(names)
```

La función `reiniciar_objeto_unico` selecciona los objetos indicados en el parámetro lista `names`, les elimina y pega lo copiado anteriormente con la función `objeto_original`.

Con la función `reiniciar_objeto` lo que queremos conseguir es pegar los objetos copiados con la función `objeto_original`, pero en algunos casos eliminando todos los objetos y en otros casos eliminando únicamente los objetos con el nombre especificado en el parámetro `names`. Así, tendremos unas imágenes con un objeto y otras en las que aparezcan varios objetos en distintas posiciones y con diferentes texturas.

```
def reiniciar_objeto(names)
```

La función `reiniciar_objeto` elige aleatoriamente un número entero entre 0 y 9. Si dicho número es mayor de 3 entonces para cada objeto de la lista `names` selecciona todos los objetos de la escena que en su nombre contienen los nombres de dicho objeto, ya que cuando se pega un objeto el nombre de este es el mismo que el que tenía pero se van numerando. Si en cambio el valor aleatorio es menor o igual a 3, en primer lugar, se seleccionan los objetos que se encuentran en la lista `names` y se duplican, y después, se seleccionan de nuevo los objetos de la lista `names`, para así mantener en la escena los nuevos objetos con nombre

Figura 3.5: Donuts superpuestos.

diferente a los objetos originales. Cuando sale del condicional elimina los objetos seleccionados y posteriormente pega los objetos que se han copiado con la función `objeto_original`. Por tanto, entrando en la primera parte del condicional el resultado sería una imagen con un solo objeto, mientras que si entra en la segunda parte el resultado sería varios objetos en la imagen. En particular, cuando hay dos objetos uno de ellos es el mismo que en la imagen anterior y el otro es la nueva copia del original a la que se le hacen las modificaciones en esa iteración.

Aquí nos encontramos con un problema, cuando hay más de un objeto en la escena estos se pueden superponer y no queda realista, como se ve en la Figura 3.5. Para solucionarlo creamos la función `cuerpo_rigido`, para que los objetos sean cuerpos rígidos, posteriormente caerán desde una altura y así no se mezclarán los unos con los otros.

```
def cuerpo_rigido(names, tipo, altura=0)
```

Esta función toma como parámetros los nombres de los objetos padre, el tipo de cuerpo rígido que queremos que sea el objeto y la altura que añadiremos al objeto. En primer lugar, seleccionamos los objetos indicados, les añadimos la altura a su coordenada z y le añadimos la propiedad de cuerpo rígido, con un peso pequeño y el tipo especificado. En el caso del plano haremos que sea un cuerpo rígido pasivo, es decir, que los demás objetos no puedan atravesarle pero que este no se caiga hacia abajo en la escena. Además, la altura indicada para el plano será la de por defecto, 0, ya que no queremos moverlo de su posición. En cambio, los otros objetos como el donut y la manzana les diremos que sean cuerpos rígidos activos y les daremos una altura mayor a la suya, para que así dichos objetos caigan sobre el plano con la fuerza de la gravedad. El parámetro `names` debe contener solo nombres de los objetos padre, además, es necesario tener los objetos con la estructura padre-hijo ya que si tenemos dos objetos independientes y les aplicamos la propiedad de cuerpo rígido al caer dichos objetos se separaran.

3.1.4. Renderizar imágenes y etiquetas

La siguiente función sirve para renderizar imágenes. Indicamos el nombre con el que queremos guardar la imagen, el motor de renderizado que queremos emplear (*Eevee* o *Cycles*), el número de rayos que se lanzarán por pixel y en caso de renderizado *Cycles*, el dispositivo que queremos utilizar para el procesamiento (CPU o GPU). El parámetro `animacion` permite renderizar la imagen una vez

que ya hayan caído los objetos, en el caso de que tengamos varios objetos en la escena y les dejemos caer desde una altura.

```
def guardar_imagen(name, animacion, motor, muestras, dispositivo=None)
```

En primer lugar esta función tiene un condicional, si se ha elegido el motor *Cycles* se pone ese tipo de motor de renderizado y los demás parámetros especificados. En caso de que se quiera usar el motor *Eevee* no hay que especificar el parámetro `dispositivo` en la llamada a la función ya que ese motor de renderizado no tiene ese atributo. Indicamos la ruta donde se va a guardar la imagen con el nombre indicado en el parámetro `name`. Si queremos guardar un `frame` diferente al inicial, como es en el caso en el que tengamos más de un objeto en la escena, creamos la animación y la movemos al `frame` que consideremos adecuado, para asegurarnos que los objetos ya han caído. Finalmente, se guarda la imagen renderizada en formato jpg.

En un principio en vez de crear toda la animación, que es más costoso, lo que hicimos fue simplemente especificar el `frame` que queríamos renderizar. Esto no funcionaba porque hacía falta iniciar la animación. Intentamos iniciar la animación, saltar al `frame` indicado y renderizar. Pero tampoco funcionó. El motivo es que no le dábamos tiempo a que la animación llegase al `frame` correcto y lo que nos renderizaba era la mesa vacía. Una solución hubiese sido darle algo de tiempo antes de renderizar para que llegase al `frame`, pero optamos por probar a crear toda la animación desde el principio ya que era más simple.

Ahora vamos a construir una función que dada la escena, la cámara y un objeto nos devuelva las coordenadas bidimensionales del *bounding box* del objeto en la imagen renderizada, basándonos en [23].

```
def obtener_coordenadas_2d(scene, cam_ob, me_ob)
```

Esta función calcula las coordenadas del cuadro delimitador del marco de la cámara y calculando el tamaño de la imagen renderizada y la traslación, rotación y escala del objeto en la escena, nos devuelve las coordenadas de los vértices superior izquierdo e inferior derecho del rectángulo que contiene al objeto en la imagen 2D. Sirve para los dos tipos de cámara utilizados, ortogonal y perspectiva.

Una vez creada la función que calcula el *bounding box* de un objeto dado, construimos una función para guardar las etiquetas en un archivo xml. Utilizaremos el formato Pascal VOC [5] pero solo incluiremos la información necesaria para el modelo de detección: el nombre de la imagen, la clase a la que pertenece el objeto y el *bounding box*.

```
def guardar_etiquetas(name, objects, camera)
```

La función `guardar_etiquetas` recibe como parámetros el nombre con el que queremos guardar el archivo xml, una lista con los nombres de los objetos principales de la imagen y el objeto cámara de la escena. En primer lugar, obtenemos una lista con los nombres de todos los objetos de la escena cuyo nombre contenga a alguno del parámetro `name`. Es una lista de sublistas, en la que cada sublista es una pareja padre-hijo de los objetos de la escena. Después creamos la estructura del xml y añadimos en ella el nombre de la imagen.

Recorremos todos los pares de objetos de la lista y para cada objeto del par llamamos a la función `obtener_coordenadas_2d` para calcular su *bounding box*, indicamos como parámetros la escena, la cámara y el objeto. Con la función `obtener_coordenadas_2d` hemos obtenido un *bounding box* para cada objeto de la pareja, por tanto para calcular el *bounding box* que contenga a los objetos padre-hijo calculamos el mínimo o el máximo de cada esquina, según corresponda. Añadimos la clase del objeto al archivo xml y las coordenadas del *bounding box*. Por último, guardamos las etiquetas en el directorio correspondiente, siempre y cuando el objeto esté dentro del campo de visión de la cámara.

3.2. Ejecución del script

Finalmente, procedemos a llamar a las funciones en un orden específico para cada modelo. Los parámetros para cada modelo y de cada función se obtienen experimentalmente. Ajustamos los rangos de los valores para que los objetos no se salgan del campo visual de la cámara o que las imágenes tengan mucha o poca luz. Con el modelo del donut, por ejemplo, mostramos la llamada a las funciones para obtener imágenes con un objeto en la escena y para renderizar con el motor *Eevee*. Para el modelo de la manzana mostramos el proceso para obtener imágenes con varios objetos y renderizando con el motor *Cycles*.

Para el caso del donut comenzamos llamando a la función `objeto_original` con los objetos `Donut` y `Glaseado`, y así creamos una copia del objeto. Este modelo ya tiene un plano, una cámara y un punto de luz, por tanto no necesitamos llamar a las funciones `agregar_plano`, `agregar_camara` y `agregar_luz`. Después creamos un bucle con el que obtendremos tantas imágenes como iteraciones realice, y en cada iteración se hará lo siguiente:

1. Llamamos a la función `objeto_transformacion` con el objeto `Donut`, el objeto padre, y sus parámetros correspondientes, es decir, siempre le aplicaremos las transformaciones de posición, rotación y escalado al objeto que tenga como nombre Donut.
2. Llamamos a la función `camara` con el objeto `Camera` y sus parámetros correspondientes para ajustar los atributos de la cámara.
3. Llamando a la función `luz` modificamos las propiedades del objeto `Light` según sus parámetros.

4. Con la función `textura` añadimos una textura a los objetos `Donut`, `Glaseado` y `Plane`. Para el caso del `Donut` y el `Glaseado` especificamos que queremos que la envoltura de la textura al objeto se haga desde la vista.

5. Renderizamos la imagen actual mediante la función `guardar_imagen` indicando los valores de los parámetros que queramos. Como `name` ponemos el nombre de la clase del objeto, en este caso `Donut`, y separado por dos barras bajas añadimos el número de iteración, así las imágenes se guardarán en una carpeta llamada Donut y estas estarán numeradas, de esta forma obtenemos las etiquetas de las imágenes para clasificación. Ponemos dos barras bajas ya que las imágenes que se descargan de Google se guardan numeradas y con una barra baja. Otros parámetros que indicamos son el motor de renderizado *Eevee*, el número de rayos que queremos que se lancen por pixel y el booleano le ponemos a falso para que no cree una animación, ya que solo tenemos un objeto en la escena y no hace falta dejarle caer desde arriba.

6. Llamamos a la función `guardar_etiquetas` con el mismo nombre con el que hemos guardado la imagen y como objetos le pasamos el `Donut` y el `Glaseado` para obtener su *bounding box*, así como la cámara de la escena.

7. Llamamos a la función `reiniciar_objeto_unico` con la que eliminamos el `Donut` y el `Glaseado` actual y obtenemos el que teníamos inicialmente.

Por último, mostramos el tiempo que ha tardado en ejecutarse todo el script. Dicho tiempo junto con el tiempo que tarda en renderizarse cada imagen se pueden ver desde la consola del sistema.

```
start_time = time()
objeto_original(['Donut', 'Glaseado'])
for i in range(0,200):
    objeto_transformacion(['Donut'], -0.2, 0.2, -0.1, 0.1, 0.005, 0.005,
    ↪  -3, 3, -3, 3, 0, 360, 1, 2)
    camara('Camera', 80, 0.7, 50)
    luz('Light', -2, -1, -1.7, 1.7, 1.2, 1.7, 100, 500, 0.1, 1, 0, 0.5,
    ↪  1, 1, 0.3, 1.5)

    textura('Donut', True)
    textura('Glaseado', True)
    textura('Plane')

    guardar_imagen('Donut__'+ str(i), False, 'eevee', 128)
    guardar_etiquetas('Donut__'+ str(i), ['Donut', 'Glaseado'], 'Camera')
    reiniciar_objeto_unico(['Donut', 'Glaseado'])

elapsed_time = time() - start_time
print("Tiempo de ejecucion:", datetime.timedelta(seconds=elapsed_time))
```

El proceso con el modelo de la manzana es similar. En primer lugar con la función `objeto_padre` indicamos que el objeto `Manzana` sea el padre del objeto `Rabo`. Después, hacemos que la `Manzana` sea un cuerpo rígido activo añadiendo

una altura de 0.1 metros. Hacemos una copia del objeto `Manzana` mediante la función `objeto_original`. Como este modelo no tiene plano, cámara, ni luz, tenemos que añadirlo llamando a las funciones `agregar_plano`, `agregar_camara` y `agregar_luz`. Por último, hacemos que el plano sea un cuerpo rígido pasivo.

A partir de aquí el bucle es similar al utilizado en el modelo del donut, pero con los parámetros correspondientes para la manzana. La única diferencia es el final del bucle. Al guardar la imagen con la función `guardar_imagen` el parámetro booleano le asignamos a verdadero para crear una animación en la que los objetos caigan sobre el plano y así poder renderizar un `frame` en el que los objetos ya estén abajo. Además, indicamos que se rendericen las imágenes con el motor *Cycles* y en caso de tener GPU utilizamos ese dispositivo, si no se pondría CPU. El uso de GPUs es preferible porque acelera los tiempos necesarios para renderizar en comparación con CPUs. En este caso para crear varios objetos en la escena llamamos a la función `reiniciar_objeto` con los objetos `Manzana` y `Rabo`.

```
start_time = time()
objeto_padre('Manzana', ['Rabo'])
cuerpo_rigido(['Manzana'], 'ACTIVE', 0.1)
objeto_original(['Manzana', 'Rabo'])
agregar_camara(0.038, 0.3211, 0.289, 43, 0, 177)
agregar_luz(-1.297, -0.397, 1.381, 37.3, 3.16, 107, 60)
agregar_plano(2, 0, 0, 0, 0, 0, 0)
cuerpo_rigido(['Plane'], 'PASSIVE')
for i in range(0, 200):
    objeto_transformacion(['Manzana'], -0.1, 0.1, -0.2, 0.2, 0, 0, -27,
    ↪  -18, 0, 27, 18, 360, 0.8, 1.2)
    camara('Camera', 80, 0.6, 50)
    luz('Light', -1, 1, -1.7, 1.7, 1.2, 1.7, 30, 100, 0.1, 1, 0, 1, 1,
    ↪  0.5, 0,1)

    textura('Manzana', True)
    textura('Rabo')
    textura('Plane')

    guardar_imagen('ManzanaVariosCycles__'+ str(i), True, 'cycles', 128,
    ↪  'GPU')
    guardar_etiquetas('ManzanaVariosCycles__'+ str(i), ['Manzana',
    ↪  'Rabo'], 'Camera')
    reiniciar_objeto(['Manzana', 'Rabo'])

elapsed_time = time() - start_time
print("Tiempo de ejecucion:", datetime.timedelta(seconds=elapsed_time))
```

En los casos en los que creemos varios objetos por imagen revisaremos las imágenes resultantes porque puede suceder que al caer sobre el plano los objetos rueden y se salgan de la vista de la cámara. En el caso de que todos los objetos se salgan del campo de visión de la cámara eliminaremos dichas imágenes y sus correspondientes etiquetas. Mientras que en el caso de que algunos de ellos se salgan no hay problema porque la función `guardar_etiquetas` lo tiene en cuenta

	CPU	RAM	GPU
Ordenador 1	i5 – 4210 2.60Hz	8GB	Intel(R) HD Graphics 4600
Ordenador 2	i7 – 4790 3.6GHz	16GB DDR3	Nvidia Quadro K620

Tabla 3.1: Diferentes ordenadores empleados para renderizar.

	Ordenador 1	Ordenador 2
Un donut con *Eevee*	7 seg	6 seg
Una manzana con *Eevee*	3 seg	1 seg
Un donut con *Cycles*	2 min 30 seg	40 seg
Una manzana con *Cycles*	1 min 40 seg	20 seg
Varios donuts con *Eevee*	15 seg	10 seg
Varias manzanas con *Eevee*	6 seg	3 seg
Varios donuts con *Cycles*	4 min	54 seg
Varias manzanas con *Cycles*	27 min	23 seg

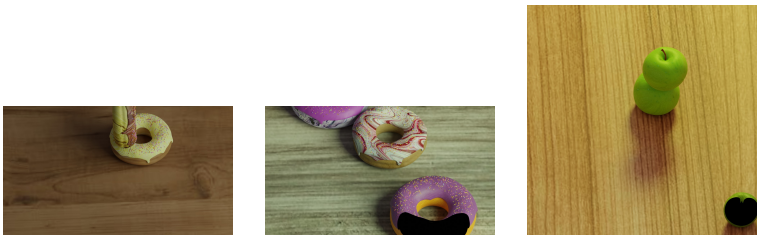
Tabla 3.2: Tiempos de renderizado en Blander.

y no añade su *bounding box* al archivo xml.

3.2.1. Imágenes generadas

Vamos a generar imágenes de los dos modelos, con los dos motores de renderizado y con uno o varios objetos en la imagen. Dependiendo del motor de renderizado que se use y de la potencia del ordenador, los tiempos en renderizar las imágenes pueden variar. Además, a mayor cantidad de objetos en la escena el tiempo de renderizado también aumenta. Estas diferencias se pueden ver en la tabla 3.2, donde las propiedades de cada ordenador se ven en la tabla 3.1. Para renderizar muchas imágenes hemos empleado el ordenador 2 (ordenador del tutor) para acelerar el proceso.

Para los casos en los que solo hay un objeto en la imagen hemos renderizado 250 imágenes y con varios objetos 450 imágenes, ya que al dejar caer los objetos sobre la mesa, en casi la mitad de imágenes estos se salían del campo visual de la cámara y obteníamos una imagen únicamente con la mesa. Dichas imágenes sin objetos las eliminábamos posteriormente. Además, al lanzar los objetos desde una altura, estos pueden caer de cualquier forma o incluso acercarse demasiado a la cámara (Figura 3.6).



(a) Donut torcido. (b) Donut cerca de la cámara. (c) Manzana cerca de la cámara.

Figura 3.6: Imágenes con fallos.

Al generar tanta cantidad de imágenes, en el caso de renderizar 450, nos hemos encontrado con un problema, renderizaba cierto número de imágenes y luego saltaba un error. Vimos que el error saltaba porque se llenaba la memoria RAM ya que Blender cachea las texturas, partículas, objetos y demás cosas (incluso aunque los borres de la escena) en memoria RAM tras cada render para ahorrar tiempo al volver a renderizar. Y como renderizábamos un montón de imágenes de golpe, al final fallaba. Esto ocurre por el tipo de script que hemos creado, que sobre la misma escena se van añadiendo y borrando objetos con distintas texturas, partículas y propiedades, y haciendo renders distintos desde Python. Para solucionarlo, añadimos el comando `bpy.ops.outliner.orphans_purge()` en el código para que así el uso de RAM por parte de Blender se mantuviese estable y no creciese tras cada renderizado.

Como se ve en la tabla 3.2 la mayor diferencia de tiempos entre los ordenadores se encuentra con el renderizado *Cycles* que necesita mayor potencia del ordenador. Aunque sea más lento las imágenes obtenidas son de mejor calidad que las que se obtienen con el renderizado *Eevee*, la mayor diferencia se observa en las sombras 3.7. En la Figura 3.8 podemos ver imágenes generadas con su *bounding box* obtenido automáticamente con Blender.

(a) Donut con renderizado *Cycles*

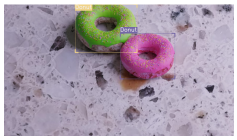
(b) Donut con renderizado *Eevee*

(c) Manzana con renderizado *Cycles*

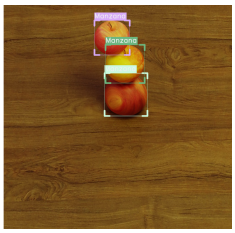
(d) Manzana con renderizado *Eevee*

Figura 3.7: Diferencia de resultados con los motores de renderizado.

(a) Donuts.



(b) Donuts.



(c) Manzanas.

(d) Manzanas.

Figura 3.8: Imágenes etiquetadas para detección.

Capítulo 4

Creación de modelos de redes neuronales

En este capítulo se explicará el proceso de descarga de imágenes reales para entrenar y validar los modelos, y los Notebooks de Jupyter que se han empleado para la creación de los modelos de clasificación de imágenes y detección de objetos. Dichos Notebooks están basados en las prácticas realizadas en la asignatura de Aprendizaje profundo [33].

4.1. Modelo de clasificación de imágenes

Para crear un modelo de clasificación de imágenes utilizaremos la red neuronal *ResNet-18* a la que entrenaremos la última capa con imágenes descargadas de Google e imágenes sintéticas creadas en Blender con el script explicado en el capítulo 3. Para entrenar el modelo utilizaremos las imágenes reales y sintéticas, mientras que para comprobar el modelo utilizaremos solo imágenes reales.

En primer lugar, desde un Notebook de Jupyter en Anaconda nos descargamos las imágenes de Google [14]. Gracias a la librería `simple_image_download` podemos conectarnos a Google y descargar mediante peticiones HTTP varias imágenes. A la función le indicamos las clases de imágenes que queremos descargar, el número de imágenes por clase y el formato de estas. El proceso de descarga será más costoso cuanto mayor sea el número de clases o de imágenes a descargar. Revisaremos las imágenes descargadas manualmente por si alguna es incorrecta o está duplicada, y eliminaremos dichas imágenes.

Una vez que tenemos las imágenes descargadas y comprobadas construimos una función para separarlas en carpetas de entrenamiento y test. Dentro de cada carpeta las imágenes se encontrarán distribuidas en carpetas según la clase a la que pertenecen, así tendremos las imágenes etiquetadas para clasificación.

```
import os
from sklearn.model_selection import train_test_split
import shutil

def train_test_google(directorio)
```

Esta función tiene como parámetro el nombre del directorio en el que se encuentran las imágenes descargadas, cuando se descargan las imágenes de Google se crea una carpeta llamada *simple_images*. Para cada clase de imágenes descargadas la función crea los directorios correspondientes de train y test y carga las imágenes descargadas, con la librería `os`. Separa las imágenes en train y test mediante la función `train_test_split` con una proporción de 80 % para train y 20 % para test. Por último para cada imagen de train y test las copia, mediante la librería `shutil`, de su directorio original a su carpeta correspondiente dentro de los directorios train y test, respectivamente. Como la separación en train y test lo hacemos para cada clase, el conjunto de test estará equilibrado, es decir, habrá el mismo número de muestras de cada clase.

Construimos una función similar para guardar automáticamente las imágenes sintéticas obtenidas con Blender en sus carpetas correspondientes. En este caso, como las imágenes sintéticas solo las utilizaremos para entrenar el modelo, añadimos todas las imágenes a las carpetas correspondientes según la clase del objeto dentro de la carpeta train. Como parámetro de entrada tiene el nombre del directorio donde se encuentran las imágenes generadas mediante modelos 3D.

```
def train_blender_clasificacion(directorio)
```

A partir de aquí continuamos en un Notebook de Jupyter pero en Google Colab, ya que permite entrenar los modelos mediante GPU y esto hace que el entrenamiento sea más rápido. Para construir la red neuronal utilizaremos la librería `fastAI`. Antes de empezar, subimos a Google Drive la carpeta `Imagenes` creada antes, donde se encuentran las imágenes de train y test, para poder cargar las imágenes en Google Colab de manera rápida. Obtenemos los directorios (path) de las imágenes mediante el siguiente código.

```
from fastai.vision.all import *

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

path = Path('/content/drive/MyDrive/TFM/Clasificacion/Imagenes')
trainPath = path/'train'
testPath = path/'test'
```

A continuación cargamos el dataset para poder posteriormente crear el modelo. Este proceso se hace en dos pasos. Primero se construye un objeto `DataBlock` y a continuación se construye un objeto `DataLoader` a partir del

`DataBlock`.

Explicamos las distintas componentes del `DataBlock`:

- El atributo `blocks` sirve para indicar el tipo de nuestros datos. Como estamos en un problema de clasificación de imágenes, tenemos que la entrada del modelo será una imagen, es decir un `ImageBlock`, y la salida será una categoría, es decir un `CategoryBlock`.
- El atributo `get_items` debe proporcionar una función para leer los datos. En nuestro caso queremos leer una serie de imágenes que estarán almacenadas en un path. Para ello usamos la función `get_image_files`.
- El atributo `splitter` nos indica cómo partir el dataset para tener un conjunto de validación. Mediante la función `RandomSplitter` separamos un 20 % aleatorio del conjunto de entrenamiento para validar el modelo.
- El atributo `get_y` sirve para indicar cómo extraer la clase a partir de nuestros datos. La función `get_image_files` proporciona una lista con los paths a las imágenes de nuestro dataset. Como la clase de cada imagen viene dada por la carpeta en la que está contenida, usamos el método `parent_label` para obtener la clase de la misma.
- El atributo `item_tfms` sirve para escalar todas las imágenes al mismo tamaño.

El `DataLoader` se construye a partir del `DataBlock` construido anteriormente, indicándole el path donde se encuentran las imágenes y el tamaño del `batch` que queremos utilizar. Para comprobar que se han cargado correctamente las imágenes y sus anotaciones se puede mostrar un `batch` del `DataLoader` mediante el comando `show_batch`.

Definimos una serie de `callbacks` que sirven para modificar el proceso de entrenamiento. En este caso vamos a utilizar 3 `callbacks`:

- `ShowGraphCallback`: sirve para mostrar las curvas de entrenamiento y validación.
- `EarlyStoppingCallback`: permite aplicar la técnica de early stopping. Para ello debemos indicarle la métrica que queremos monitorizar para saber cuándo parar, y la paciencia, es decir, cuántos `batchs` dejamos que el modelo continúe entrenando si no ha habido mejora.
- `SaveModelCallback`: sirve para guardar el mejor modelo encontrado durante el proceso de entrenamiento y lo carga al final del mismo. Como parámetro se le puede indicar el nombre con el que debe guardar el modelo.

A continuación construimos el modelo, un objeto de la clase `Learner`, utilizando el método `cnn_learner` que toma como parámetros el `DataLoader`, la arquitectura que queremos entrenar (*resnet18*), los callbacks y la métrica que

usaremos para evaluar el modelo, como es un problema de clasificación utilizaremos la `accuracy` [1]. Además, incluimos la transformación del modelo a `mixed precision`, que sirve para acelerar el entrenamiento del modelo.

Con la función `lr_find()` buscamos un `learning rate` [2] adecuado para posteriormente entrenar el modelo. Para entrenar el modelo ejecutamos el comando `fine_tune` con 10 `batchs` y un `learning rate` de 10^{-3}. El proceso que sigue para entrenar consiste en:

- Congelar todas las capas salvo la última, y entrenar esa parte del modelo durante un `batch`.
- Descongelar la red, y entrenar el modelo por el número de `batchs` indicado.

Finalmente, evaluamos el modelo en el conjunto de test. Para ello debemos crear unos nuevos `DataBlock` y `DataLoader` y modificar el objeto `Learner`. La única diferencia con el `DataBlock` utilizado previamente es que para hacer la partición del dataset usamos un objeto de la clase `GrandparentSplitter` indicando que el conjunto de validación es nuestro conjunto de test. Para que en este punto no salte un error las carpetas que contienen las imágenes deben llamarse train y test, en minúsculas. En el caso del `DataLoader`, la diferencia con el definido anteriormente es que cambiamos la ruta al path.

Por último, evaluamos el modelo usando el método `validate`, el cual devuelve dos valores: el valor de la función de pérdida [3], y el valor de la métrica utilizada, en este caso la `accuracy`. También podemos construir un objeto `ClassificationInterpretation` para mostrar alguna de las imágenes que ha clasificado incorrectamente, es decir, aquellas en las que se produce un mayor error.

4.2. Modelo de detección de objetos

Partiremos de la arquitectura *Faster R-CNN*, entrenada con imágenes de *ImageNet*, para crear el modelo de detección de objetos, es decir, un modelo para clasificar y localizar algunos objetos en una imagen. Entrenaremos la última capa de la red neuronal con imágenes reales obtenidas de datasets de COCO e imágenes sintéticas creadas con Blender y validaremos los resultados únicamente con las imágenes reales.

Comenzamos descargando un dataset de COCO [6]. En este caso hemos descargado el dataset *2014 Train/Val annotations* y utilizaremos únicamente

[1] La `accuracy` mide el número de muestras que han sido correctamente clasificadas por el modelo

[2] La tasa de aprendizaje (`learning rate`) es un parámetro de ajuste en un algoritmo de optimización que determina el tamaño del paso en cada iteración mientras se mueve hacia un mínimo de una función de pérdida.

[3] La función de pórdida es una función que evalúa la desviación entre las predicciones realizadas por la red neuronal y los valores reales de las observaciones utilizadas durante el aprendizaje

los datos para validación ya que el dataset es muy grande y nosotros queremos pocas instancias. Mediante un Notebook de Jupyter, basado en [8], cambiamos el formato del fichero *instances* de anotaciones, de formato COCO a formato Pascal VOC, formato utilizado por ImageNet. El formado COCO es un json en el que aparecen las anotaciones de todas las imágenes, en cambio el formato Pascal VOC consta de un fichero de anotaciones para cada imagen, con el mismo nombre que la imagen [5].

Definimos una función que dados los datos que nos interesan del json los convierta en un archivo xml para cada imagen.

```
import os
import xml.etree.ElementTree as ET
import pandas as pd
import cv2
import json


def write_to_xml(image_name, image_dict, data_folder, save_annotations,
↪  save_images)
```

La manera de crear el xml es similar a la utilizada en Blender. Vamos creando los elementos y asignándoles su valor correspondiente del diccionario para la imagen indicada. Por último, cargamos la imagen y se guarda en el directorio indicado en los parámetros. De igual forma, se guarda el archivo xml en la carpeta especificada. Esta función será llamada varias veces dentro de un bucle.

En primer lugar manualmente buscamos en el archivo json los id de las clases que queremos utilizar y las añadimos a una lista. Definimos los `path` del archivo de anotaciones y de las imágenes, así como donde se van a guardar los datos resultantes. Creamos un diccionario en el que las claves son los nombres de las diferentes imágenes y como valores aparece la clase y las coordenadas de los píxeles de las esquinas superior izquierda e inferior derecha de los diferentes objetos que aparecen en dicha imagen. Es importante que las clases de los objetos se llamen siempre de la misma forma para que el modelo reconozca los objetos como una sola clase. Por tanto, almacenamos las clases en castellano ya que en Blender tenemos definidas las clases de esta manera y en COCO aparecen en inglés.

Por último, mediante un bucle, para cada imagen del diccionario llamamos a la función `write_to_xml` anterior indicándole el nombre de la imagen, el diccionario, la carpeta donde se encuentran las imágenes y las rutas donde se quieren guardar las imágenes y anotaciones resultantes. No recorremos todas las imágenes del dataset de COCO, sino que nos quedamos con cierto número de ellas para poder comprobar si al añadir las imágenes sintéticas los resultados finales mejoran.

(a) Imagen de la clase Donut.

(b) Imagen de la clase Ciruela.

Figura 4.1: Imágenes de Google etiquetadas con *labelImg*.

A continuación, construimos una función para separar un 80 % de los datos para train y el 20 % restante para test. En este caso no separamos las imágenes según su clase ya que en una misma imagen podemos tener objetos de diferentes clases. Por tanto, obtendremos las carpetas de train y test y dentro de estas sus respectivas imágenes y etiquetas.

```
def train_test_coco(directorio)
```

Igual que para las funciones de clasificación de la sección 4.1, esta función toma como parámetro el directorio en el que se encuentra el dataset descargado, separamos en train y test y copiamos los datos a las carpetas images y labels correspondientes. Esta vez el número de imágenes de cada clase no será el mismo en cada conjunto.

Como veremos en el siguiente capítulo, los resultados obtenidos con el dataset COCO no son buenos porque los objetos no están en primer plano o no se ven correctamente, por lo que probaremos el modelo de detección de objetos con las imágenes descargadas de Google anteriormente. Para ello, primero tenemos que etiquetarlas creando un *bounding box* para los objetos de cada imagen (Figura 4.1). Las imágenes las etiquetamos manualmente con la ayuda de la herramienta *labelImg* mediante la consola de Anaconda [16] y generamos las etiquetas en formato Pascal VOC.

Una vez etiquetadas las imágenes creamos una función similar a la anterior para dividir los datos en train y test. En este caso, las imágenes se encuentran separadas en carpetas según la clase a la que pertenecen y las etiquetas están todas en una misma carpeta. Leemos las imágenes y las etiquetas, separamos en train y test y las guardamos en sus respectivas carpetas. Igual que con el dataset COCO, las carpetas train y test puede que no estén equilibradas para las clases. Además, una diferencia respecto al dataset COCO es que en este caso solo tenemos una clase de objeto por imagen.

```
def train_test_google_deteccion(dir_img, dir_lab)
```

De manera similar, construimos una función para guardar automáticamente las imágenes creadas en Blender en sus respectivas carpetas de train para entrenar el modelo de detección.

```
def train_blender_deteccion(directorio)
```

Igual que hemos hecho en el modelo de clasificación subimos las carpetas obtenidas a Google Drive y comenzamos a construir el modelo de detección de objetos. Creamos un Notebook de Jupyter en Google Colab. Utilizaremos la librería `IceVision`. En primer lugar definimos el `path` en el que se encuentran las imágenes.

```
from icevision.all import *

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

path=Path('/content/drive/MyDrive/TFM/Deteccion/dataset')
```

Creamos un `parser` para leer las imágenes y las anotaciones. Para ello, lo primero que hacemos es construir un objeto de la clase `ClassMap` que contiene las clases de objetos de nuestro dataset. A continuación, definimos los `parsers` mediante el método `parsers.voc`, uno para leer el conjunto de entrenamiento, y otro para el de test, indicando los directorios de las imágenes y las etiquetas.

Construimos los `records` a partir de los objetos `parser`. Un `record` es un diccionario que contiene todos los campos parseados definidos en el proceso anterior. Para ello debemos llamar al método `parse` e indicarle cómo se van a repartir los datos que se lean. Dividimos el dataset en tres partes: un conjunto de entrenamiento, uno de validación y uno de test. Por lo tanto tendremos que construir tres `records`. Los `records` de entrenamiento y validación los construiremos a partir de los datos de entrenamiento usando una partición 90/10, mientras que el `record` de test se construye a partir del conjunto de test usándolo completamente.

Podemos comprobar que los datos se han cargado correctamente ejecutando la función `show_records`. Aplicamos una transformación a los datos con el método `tfms.A.Adapter`, escalamos todas las imágenes al mismo tamaño y las normalizamos, ya que es necesario normalizar las imágenes al rango de las imágenes de *ImageNet*. Para ello, utilizamos la clase `Dataset` que combina los `records` y las transformaciones. Debemos crear un dataset para nuestro conjunto de entrenamiento, otro para el conjunto de validación y otro para el de test.

Al igual que para el modelo de clasificación de `FastAI`, el último paso es crear el `DataLoaders` a partir de los datasets construidos anteriormente. Para

crear el modelo debemos crear un objeto `Learner` de `FastAI`. Para crear dicho objeto construimos un modelo con la arquitectura *Faster RCNN*.

A continuación proporcionamos las métricas que queremos utilizar para evaluar el modelo. Por el momento la única métrica soportada por `IceVision` es el `mAP` (Mean Average Precision) de COCO, por lo tanto utilizamos dicha métrica. Buscamos un `learning rate` adecuado utilizando la función `lr_find()` y entrenamos el modelo mediante `fine_tune` con 10 `batchs` y un `learning rate` de 10^{-4}.

Por último, para validar el modelo con el conjunto de test construimos un nuevo `dataloader` indicando que el conjunto de validación es el de test, modificamos el `dataloader` del objeto `Learn` que hemos entrenado anteriormente y evaluamos el modelo. Al igual que en el caso de los modelos de clasificación el método `validate` devuelve dos valores, el valor de la pérdida y el valor de la métrica asociada al conjunto de validación, que en este caso es el de test. Para ver el resultado de algunas imágenes de test podemos utilizar la función `show_results`.

Capítulo 5

Validación de resultados

En este capítulo se mostrarán los resultados obtenidos con las diferentes imágenes generadas en Blender para los problemas de clasificación y detección de objetos.

5.1. Clasificación

Para el problema de clasificación utilizamos el modelo descrito en la sección 4.1. Realizamos diferentes pruebas para ir comprobando si los resultados mejoran al añadir imágenes sintéticas. En primer lugar, entrenamos el modelo únicamente con imágenes reales descargadas de Google de las clases donut, manzana, peras, cupcakes, ciruelas y roscón de reyes (Figura 5.1). Hacemos diferentes pruebas y los resultados pueden verse en la tabla 5.1.

Primero utilizamos 500 imágenes para cada clase sin eliminar las duplicadas o incorrectas, vemos que la `accuracy` obtenida es muy alta. Al revisar las imágenes vimos que había muchas de ellas que se encontraban repetidas, lo cual explicaría el resultado tan alto ya que probablemente muchas imágenes empleadas para entrenar el modelo también de encontraban en el conjunto de test. La siguiente prueba que hicimos fue descargar menos imágenes, 100 para cada clase, y la `accuracy` disminuyó un poco. Para la tercera prueba nos descargamos más de 100 imágenes para cada clase ya que posteriormente eliminamos las duplicadas, erróneas o poco claras como la Figura 5.2, hasta quedarnos con 100 de ellas por clase. En este caso obtuvimos una `accuracy` de 0.883.

Para comprobar si los resultados mejoran añadiendo imágenes sintéticas utilizamos las 100 imágenes filtradas descargadas de Google. Realizamos diferentes pruebas añadiendo imágenes sintéticas al conjunto de entrenamiento de las clases donut y manzana. En la tabla 5.2 pueden verse las diferentes pruebas, añadiendo diferentes cantidades de imágenes, con los dos motores de renderizado y con uno o varios objetos por imagen. En los conjuntos de imágenes con varios objetos en cada imagen no todas las imágenes tienen más de un objeto,

(a) Donut

(b) Cupcakes

(c) Roscón de reyes

(d) Manzanas

(e) Ciruelas

(f) Peras

Figura 5.1: Imagenes descargadas de Google.

(a) Donut

(b) Manzana

(c) Ciruelas

Figura 5.2: Ejemplo de imágenes eliminadas.

Nº de imágenes de cada clase	Filtradas	Accuracy	Tiempo de un `batch`
500	no	0.953	49 seg
100	no	0.816	16 seg
100	si	0.883	17 seg

Tabla 5.1: Resultados de clasificación con imágenes reales de Google.

Nº de imágenes de cada clase	Nº de objetos por imagen	Motor de renderizado	Accuracy	Tiempo de un `batch`
100	1	*Eevee*	0.8	20 seg
200	1	*Eevee*	0.875	24 seg
100	1	*Cycles*	0.825	20 seg
200	1	*Cycles*	0.841	25 seg
100	varios	*Eevee*	0.791	20 seg
200	varios	*Eevee*	0.875	26 seg
100	varios	*Cycles*	0.816	20 seg
200	varios	*Cycles*	0.883	25 seg

Tabla 5.2: Resultados de clasificación añadiendo imágenes sintéticas a las imágenes de Google filtradas.

también podemos encontrarnos con imágenes con un solo objeto.

Vemos que los tiempos para cada modelo son muy similares, cada `batch` tarda en realizarse alrededor de 24 segundos. Con respecto a la `accuracy`, no hemos conseguido mejorar la que ya se tenía únicamente con las imágenes de Google, pero tampoco se ha empeorado demasiado. Utilizando 200 imágenes sintéticas con varios objetos en cada imagen y renderizadas con el motor *Cycles* obtenemos el mismo resultado.

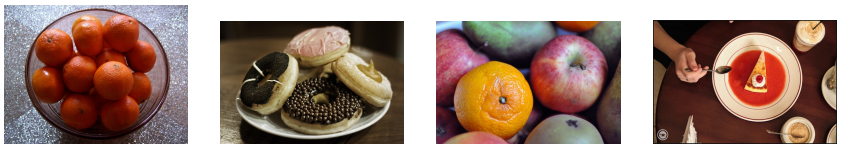
También hemos realizado algunas pruebas añadiendo imágenes sintéticas, renderizadas con el motor *Cycles*, a las 100 imágenes sin filtrar descargadas de Google, como se ve en la tabla 5.3. En esos casos, la `accuracy` obtenida es superior o igual que la obtenida únicamente con las imágenes reales, probablemente porque en este caso las imágenes reales son peores ya que no hemos eliminado las erróneas o duplicadas.

Si nos fijamos en los resultados obtenidos para cada par de pruebas en las que cambian el número de imágenes empleadas vemos que en los casos de 200 imágenes la `accuracy` es mayor que en los casos donde se han usado menos imágenes, 100. El hecho de tener más objetos en la imagen no ha mejorado los resultados con respecto a las pruebas realizadas con imágenes en las que únicamente había un objeto. Con respecto a los motores de renderizado, podemos ver que en casi todos los casos los resultados obtenidos con *Cycles* son mejores que los obtenidos con el motor *Eevee*. En la Figura 5.3 se pueden ver algunas de las predicciones fallidas.

Nº de imágenes de cada clase	Nº de objetos por imagen	Accuracy	Tiempo de un `batch`
100	1	0.841	20 seg
200	1	0.85	19 seg
100	varios	0.816	21 seg
200	varios	0.858	18 seg

Tabla 5.3: Resultados de clasificación añadiendo imágenes sintéticas con renderizado *Cycles* a las imágenes de Google sin filtrar.

Figura 5.3: Predicciones fallidas de los modelos.



(a) Naranja (b) Donut (c) Diferentes frutas (d) Pastel

Figura 5.4: Ejemplo de imágenes con objetos que se ven bien.

(a) Manzana (b) Diferentes frutas (c) Plátano (d) Manzana

Figura 5.5: Ejemplo de imágenes en las que los objetos no se diferencian bien.

5.2. Detección

Para el problema de detección utilizamos el modelo descrito en la sección 4.2. Igual que para clasificación, realizamos diferentes pruebas para comprobar si los resultados mejoran al añadir imágenes sintéticas a dos tipos de imágenes reales. En primer lugar, utilizaremos las imágenes del dataset COCO utilizando las clases plátano, naranja, manzana, donut y pastel. En estos datasets podemos encontrarnos imágenes en las que el objeto se ve claramente (Figura 5.4), pero la mayoría de imágenes no son tan evidentes (Figura 5.5). Hacemos diferentes pruebas y los resultados pueden verse en la tabla 5.4.

Primero utilizamos 500 imágenes en total, no de cada clase ya que en una misma imagen puede haber varios objetos de diferentes clases. Como la métrica `mAP` obtenida era tan baja probamos a aumentar el número de imágenes a 1000, pero la mejoría fue mínima. Otra prueba fue utilizar solo imágenes con objetos de las clases donut y manzana, pero tampoco mejoró. Aunque los resultados eran tan bajos, probablemente por la mala calidad de las imágenes, probamos a añadir alguna imagen sintética.

Nº de imágenes totales	Nº de clases	`mAP`	Tiempo de un `batch`
500	5	0.101	22 seg
1000	5	0.143	1 min 27 seg
500	2	0.163	43 seg

Tabla 5.4: Resultados de detección con imágenes reales de COCO.

Nº de objetos por imagen	`mAP`	Tiempo de un `batch`
1	0.098	34 seg
varios	0.102	34 seg

Tabla 5.5: Resultados de detección añadiendo a las imágenes de COCO 100 imágenes sintéticas de cada clase renderizadas con *Cycles*.

(a) Manzana (b) Diferentes frutas (c) Plátano (d) Manzana

Figura 5.6: Imágenes de Google etiquetadas manualmente.

Probamos a añadir 100 imágenes de cada clase, donut y manzana, realizadas con el renderizado *Cycles* y tanto para imágenes con un objeto como con varios objetos el resultado fue igual de malo, como se ve en la tabla 5.5. Como los resultados obtenidos eran tan malos nos planteamos utilizar las imágenes descargadas anteriormente de Google para el problema de clasificación. El problema era que había que etiquetar manualmente las imágenes creando el *bounding box* de cada objeto. Para no perder tanto tiempo etiquetando solo utilizamos las clases donut, manzana, ciruelas y roscón de reyes (Figura 5.6).

En la tabla 5.6 vemos los resultados. Utilizando 100 imágenes de cada una de las 4 clases obtenemos una métrica `mAP` de 0.702, mucho mayor que para el dataset COCO. Probando solo con las clases donut y manzana el resultado disminuye un poco. Vemos si los resultados mejoran añadiendo a las imágenes de las 4 clases diferente tipo de imágenes sintéticas de las clases donut y manzana creadas con Blender.

En la tabla 5.7 vemos las diferentes pruebas realizadas. Vemos que en la mayoría de casos el tiempo de ejecución de un `batch` es de aproximadamente un minuto. En todos los casos la métrica `mAP` supera, en 2 o 3 décimas, al modelo entrenado solo con imágenes reales. También observamos que a mejor motor de renderizado, a mayor cantidad de imágenes de cada clase y a mayor cantidad de objetos por imagen los resultados son mejores.

En la Figura 5.7 se pueden ver algunas de las predicciones que realiza

Nº de clases	`mAP`	Tiempo de un `batch`
4	0.702	22 seg
2	0.674	12 seg

Tabla 5.6: Resultados de detección con 100 imágenes reales de Google de cada clase.

Nº de imágenes de cada clase	Nº de objetos por imagen	Motor de renderizado	`mAP`	Tiempo de un `batch`
100	1	*Eevee*	0.724	34 seg
100	1	*Cycles*	0.725	1 min 04 seg
200	1	*Cycles*	0.734	1 min 27 seg
100	varios	*Eevee*	0.722	1 min 2 seg
100	varios	*Cycle*	0.726	1 min 3 seg
200	varios	*Cycle*	0.734	1 min 24 seg

Tabla 5.7: Resultados de detección añadiendo a las imágenes de Google imágenes sintéticas.

(a) Ciruelas (b) Donut

(c) Donut (d) Manzana

Figura 5.7: Imágenes reales (izquierda) y predichas (derecha).

el modelo sobre el conjunto de test de imágenes de Google. En la Figura 5.7a vemos que el modelo predice alguna ciruela como manzana, en la Figura 5.7b observamos que predice más donuts de los que hay y uno de ellos dice que es un roscón de reyes y en las Figuras 5.7c y 5.7d reconoce más donuts y manzanas, respectivamente, de las que aparecen en las imágenes.

Un problema encontrado en este punto fue que al realizar tantas ejecuciones de los modelos en Google Colab saltaba un error de conexión a la GPU debido a los límites de uso de Google Colab. La solución era esperar un tiempo indefinido y variable o ejecutar los modelos utilizando otra cuenta de Google. Además, los tiempos de ejecución de los `batchs` mostrados anteriormente no son muy fiables, ya que si se hacen muchas ejecuciones seguidas, las primeras que se realizan tardan menos que las últimas pruebas, probablemente por sobrecarga de la GPU, y esto hace que los tiempos de los `batchs` sean mayores.

Conclusiones y posibles mejoras

A pesar de que los resultados obtenidos no han sido tan buenos como se esperaba, este proyecto me ha resultado muy interesante ya que me ha permitido aprender sobre el modelado geométrico y el programa Blender. La mayoría de lo realizado en Blender se han hecho mediante un script de Python, pero la interfaz gráfica del programa permite realizar muchas más opciones, y de manera más sencilla de las que se han implementado aquí.

En el proyecto se pretendía comprobar si los resultados mejoraban simplemente añadiendo imágenes sintéticas para ampliar datasets pequeños con imágenes reales, ya que el etiquetado manual de nuevas imágenes reales es costoso, y mediante modelos geométricos se pueden obtener dichas imágenes etiquetadas automáticamente. Probablemente los resultados se puedan mejorar creando imágenes sintéticas de mejor calidad, es decir, más realistas.

Para que las imágenes fuesen más realistas en vez de buscar tantas imágenes de texturas por internet, se podría utilizar la librería OpenCV de Python para aplicar transformaciones a las texturas más reales, como cambios de contraste, de colores, aplicar filtros, etc. Otra mejora que se podría hacer es utilizar como fondo una imagen real donde haya varios objetos, y sustituir uno de ellos por el objeto sintético creado, ajustando el tamaño y la posición para que sea la misma que la del objeto real que eliminamos. Otro ejemplo de mejora sería aplicar transformaciones a la malla poligonal moviendo vértices o aristas, o incluso esculpiendo la figura, para que la base del objeto no sea exactamente igual en todas las imágenes.

Con estas mejoras sobre las imágenes sintéticas es probable que se consigan mejores resultados, pero no hemos podido probarlas por falta de tiempo y porque muchos de los cambios resultarían más fáciles hacerlos desde la interfaz gráfica de Blender y en este proyecto se quería utilizar scripts de Python para hacer las modificaciones de los objetos de manera más automática.

Bibliografía

[1] Acerca de la nube de puntos. https://knowledge.autodesk.com/es/support/inventor-products/learn-explore/caas/CloudHelp/cloudhelp/2018/ESP/Inventor-Help/files/GUID-BF0DD4A3-4C03-4AF3-88ED-2DD135668D64-htm.html. Acceso: 2021-5-21.

[2] Apple fruit. Blender Boom. 3D model repository. https://www.blenderboom.com/product/apple-fruit/. Acceso: 2021-2-26.

[3] Blender 2.92.0 Python API Documentation. https://docs.blender.org/api/current/index.html. Acceso: 2021-2-26.

[4] Cinta De Moebius modelo 3d. https://free3d.com/es/modelo-3d/mobius-strip-5204.html. Acceso: 2021-5-21.

[5] COCO and Pascal VOC data format for Object detection. https://towardsdatascience.com/coco-data-format-for-object-detection-a4c5eaf518c5. Acceso: 2021-6-17.

[6] COCO datasets. https://cocodataset.org/#download. Acceso: 2021-6-1.

[7] Conejo Geométrico. https://www.informaciondeherramientas.com/producto/conejo-geometrico/. Acceso: 2021-5-21.

[8] Convert COCO annotations to xml format. https://github.com/mhiyer/coco-annotations-to-xml. Acceso: 2021-6-1.

[9] Curvas de Bézier. http://oscarhumbertoramirezcruzgraficacion.blogspot.com/2017/11/curvas-de-bezier.html. Acceso: 2021-5-21.

[10] Deep Learning with Robolink Object Detection App. https://stemkitreview.com/object-detection-feb2021/. Acceso: 2021-5-21.

[11] Donut modelo 3d. Free3D. https://free3d.com/3d-model/donut-716088.html. Acceso: 2021-2-26.

[12] El 3D en el diseño de interiores. https://www.dimensi-on.com/blog/el-3d-en-el-diseno-de-interiores/. Acceso: 2021-5-21.

[13] Evolution of Video Game Graphics 1958-2020. https://www.youtube.com/watch?v=IsPPWWlV-T8&t=2s. Acceso: 2021-5-21.

[14] Google image downloader. https://pypi.org/project/simple-image-download/. Acceso: 2021-5-10.

[15] How to Unwrap project from view with script? Blender stackexchange. https://blender.stackexchange.com/questions/99035/how-to-unwrap-project-from-view-with-script. Acceso: 2021-5-21.

[16] LabelImg 1.8.5. https://pypi.org/project/labelImg/. Acceso: 2021-6-17.

[17] Machine Learning VS DeepLearning: Settling the age-old debate. https://medium.com/featurepreneur/machine-learning-vs-deeplearning-settling-the-age-old-debate-e7b44702c816. Acceso: 2021-5-21.

[18] Modelado CAD 3D. Piezas Mecánicas. https://dolphin-tecnologias.com/portfolio/modelado-cad-3d-piezas-mecanicas/. Acceso: 2021-5-21.

[19] Nurbana. http://www.3drender.com/nurbana/. Acceso: 2021-5-21.

[20] Principales Algoritmos usados en Machine Learning. https://www.aprendemachinelearning.com/principales-algoritmos-usados-en-machine-learning/. Acceso: 2021-5-22.

[21] R-CNN, Fast R-CNN, Faster R-CNN, YOLO — Object Detection Algorithms. https://towardsdatascience.com/r-cnn-fast-r-cnn-faster-r-cnn-yolo-object-detection-algorithms-36d53571365e. Acceso: 2021-5-21.

[22] Receip data augmentation: using virtual people for real image analysis. https://way2vat.com/receipt-data-augmentation-using-virtual-people-for-real-image-analysis/. Acceso: 2021-2-15.

[23] Save the 2D bounding box of an object in rendered image to a text file. Blender stackexchange. https://blender.stackexchange.com/questions/7198/save-the-2d-bounding-box-of-an-object-in-rendered-image-to-a-text-file. Acceso: 2021-6-1.

[24] The SYNTHIA dataset. http://synthia-dataset.net/. Acceso: 2021-2-15.

[25] Toy Story 3. https://www.fotogramas.es/peliculas-criticas/a328876/toy-story-3/. Acceso: 2021-5-21.

[26] Alhaija, H. A., Mustikovela, S. K., Mescheder, L., Geiger, A., and Rother, C. Augmented reality meets computer vision: Efficient data generation for urban driving scenes.

[27] Almezhghwi, K., and Serte, S. Improved classification of white blood cells with the generative adversarial network and deep convolutional neural network. *Computational Intelligence and Neuroscience 2020* (07 2020), 1–12.

[28] Ana Romero Ibáñez. Recuperando la tercera dimensión. Conferencia online del proyecto "Marzo, mes de las matemáticas". Universidad de La Rioja. https://www.youtube.com/watch?v=pWsltJ0NpSg.

[29] Ana Romero Ibáñez y Jose Divasón Mallagaray. *Asignatura de Procesamiento de imágenes digitales.* Máster en Ciencia de Datos y Aprendizaje Automático, Universidad de La Rioja.

[30] IBM Cloud Education. Neural Networks. `https://www.ibm.com/cloud/learn/neural-networks`. Acceso: 2021-5-22.

[31] Iwasaki, M., and Yoshioka, R. Data augmentation based on 3d model data for machine learning. In *2019 IEEE 4th International Conference on Computer and Communication Systems (ICCCS)* (2019), IEEE, pp. 1–4.

[32] James Chen. Neural Network. `https://www.investopedia.com/terms/n/neuralnetwork.asp`. Acceso: 2021-5-22.

[33] Jónatan Heras Vicente. *Asignatura de Aprendizaje profundo*. Máster en Ciencia de Datos y Aprendizaje Automático, Universidad de La Rioja.

[34] Liu, S., and Ostadabbas, S. A semi-supervised data augmentation approach using 3d graphical engines. In *Proceedings of the European Conference on Computer Vision (ECCV) Workshops* (2018), pp. 0–0.

[35] Pixelary, B. T. Blender for Computer Vision Machine Learning. `https://blog.thepixelary.com/post/174286685782/blender-for-computer-vision-machine-learning`. Acceso: 2021-2-15.

[36] Rubén Velasco. Blender: el programa por excelencia de renderizado y creación 3D. `https://www.softzone.es/programas/imagen/blender/`. Acceso: 2021-5-22.