Trabajo Fin de Grado

Título del trabajo:

Sistema de reconocimiento de piezas de LEGO para soporte en el juego

Tittle:

LEGO piece recognition system for in-game support

Autora

Marta Martín-Albo Cortijo

Directora

Ana María López Torres

Titulación

Grado en Ingeniería Electrónica y Automática

Escuela Universitaria Politécnica de Teruel

2024

ÍNDICE

1. INTRODUCCIÓN 1
  1.1. Objetivo 1
  1.2. El problema de clasificación 1
2. MARCO TEÓRICO 2
  2.1. Descripción de red neuronal: Imitando al cerebro 2
  2.2. Tipos de redes neuronales artificiales 4
  2.3. Redes convolucionales 4
    2.3.1. Capas convolucionales 5
    2.3.2. Capas de agrupamiento (Pooling) 6
    2.3.3. Capas de aplanamiento (Flatten) 7
    2.3.4. Capas totalmente conectadas (Dense) 7
    2.3.5. Capas dropout 7
  2.4. Aprendizaje supervisado 7
  2.5. Aprendizaje por transferencia 10
  2.6. Segmentación de objetos 11
  2.7. Librerías y recursos 11
    2.7.1. Python 11
    2.7.2. Librerías 12
    2.7.3. Recursos 13
3. DESARROLLO DEL PROYECTO 15
  3.1. Diseño general 15
  3.2. Organización del trabajo realizado 18
    3.2.1. Entrenamiento de la red neuronal (programa 1). 18
    3.2.2. Aplicación de selección de piezas para la construcción de figuras LEGO (programa 2) 29
4. RESULTADOS Y PROBLEMAS 44
  4.1 Resultados 44
  4.2 Problemas 46
5. CONCLUSIONES 49
6. REFERENCIAS BIBLIOGRÁFICAS Y WEBGRAFÍA 50
ANEXOS 54

RESUMEN

El objetivo de este TFG fue desarrollar una aplicación capaz de reconocer en tiempo real siete tipos diferentes de piezas LEGO e indicar al usuario su localización facilitándole la construcción de una figura seleccionada por él mismo. Este sistema fue diseñado para operar en un entorno controlado, con fondo oscuro y una iluminación homogénea. Dentro de esta aplicación se integra una red neuronal convolucional entrenada para el reconocimiento de dichas piezas junto a un menú para la selección de las figuras necesarias para realizar tres diferentes construcciones. Además de la localización a tiempo real, la ubicación de estas piezas también será indicada al mismo tiempo sobre la imagen.

Para conseguir dicho objetivo se entrenó una red neuronal convolucional (CNN) utilizando un conjunto de imágenes de las piezas con las que se iba a trabajar. Se aplica aprendizaje automático supervisado para la obtención del modelo de reconocimiento de piezas partiendo de una red neuronal existente (aprendizaje por trasferencia).

Palabras Clave: Reconocimiento de patrones, red neuronal convolucional, visión por computador, segmentación/recorte de imagen

ABSTRACT

The objective of this Final Degree Project (TFG) was to develop an application capable of recognizing seven different types of LEGO pieces in real-time. This system was designed to operate in a controlled environment, with a dark background and uniform lighting. The application integrates a convolutional neural network (CNN) trained to recognize these pieces, along with a menu for selecting the necessary pieces to create three different constructions. In addition to real-time detection, the location of these pieces is also indicated simultaneously on the image.

To achieve this objective, a convolutional neural network (CNN) was trained using a dataset of images of the pieces involved in the project. Supervised machine learning was applied to obtain the piece recognition model, starting from an existing neural network through transfer learning.

Key Words: Pattern recognition, convolutional Neural Network, computer Vision, image segmentation/cropping

1. INTRODUCCIÓN

1.1. Objetivo

El objetivo de este TFG fue desarrollar una aplicación capaz de reconocer en tiempo real siete tipos diferentes de piezas LEGO. Este sistema fue diseñado para operar en un entorno controlado, con fondo oscuro y una iluminación homogénea. Dentro de esta aplicación se integra una red neuronal convolucional entrenada para el reconocimiento de dichas piezas junto a un menú para la selección de las figuras necesarias para realizar tres diferentes construcciones. Además de la localización a tiempo real, la ubicación de estas piezas también será indicada al mismo tiempo sobre la imagen.

1.2. El problema de clasificación

El problema que se desea resolver es la clasificación de siete tipos distintos de piezas LEGO de colores claros, bajo unas condiciones determinadas. Estas piezas son elementos comunes en las construcciones LEGO y cada una presenta unas características específicas que permite diferenciarlas entre sí, como la forma, el tamaño y sus proporciones. Se pretende que el modelo de clasificación identifique correctamente la categoría a la que pertenece cada pieza en tiempo real, bajo un entorno controlado.

Las condiciones para resolver este problema han sido:

- La iluminación: debe ser suave y homogénea para evitar sombras pronunciadas o reflejos sobre las piezas.
- El fondo: se utilizará un fondo oscuro para maximizar el contraste con las piezas de colores claros y facilitar su segmentación en la imagen.
- La captura en tiempo real: se procesarán las imágenes capturadas en tiempo real mediante una cámara integrada en un teléfono smartphone móvil, obteniendo una respuesta rápida.
- El entorno estático: la distancia de la cámara móvil a las piezas será constante y se reducirá la presencia de elementos externos para minimizar el ruido en las imágenes.

2. MARCO TEÓRICO

Antes de describir el trabajo realizado es importante presentarla introducción teórica a los conceptos necesarios para su realización. Describiremos los conceptos básicos relativos sobre en qué consiste una red neuronal, sus características principales, cómo se entrenan, así como de las herramientas más utilizadas en este campo, destacando aquellas que son necesarias para la consecución del objetivo del presente trabajo.

2.1. Descripción de red neuronal: Imitando al cerebro

La necesidad de entender cómo funciona el cerebro humano, considerado desde siempre la máquina más compleja de la naturaleza, ha llevado a desarrollar máquinas "inteligentes" para imitarlo. Así, desde los autómatas mecánicos ya descritos por Herón de Alejandría (10 d.C- 70 d.C) hasta la inteligencia artificial, capaz de ganar al ser humano en una partida de ajedrez, el esfuerzo para comprenderlo e imitarlo sigue vigente.

Pero nos tenemos que remontar al año 1943 cuando surge la primera propuesta de red neuronal basándose en el funcionamiento del cerebro. La propuesta es realizada por Warren McCullough y Walter Pitts (1943) [1], que consiste en tomar como objeto de estudio las capacidades computacionales de las neuronas a través de un modelo matemático, prescindiendo de sus aspectos fisiológicos y morfológicos. De esta forma, los autores proponen una neurona artificial con comportamiento binario (excitada o inhibida).

Partiendo de la propuesta de Warren McCullough y Walter Pitts, el psicólogo Frank Rosenblatt desarrolló el Perceptron (1958) [2]. Esta neurona artificial consistía en un clasificador binario que generaba una predicción basada en un algoritmo que tenía en cuenta el valor o peso asignado a las diferentes entradas, como podemos observar en la figura 2.1.

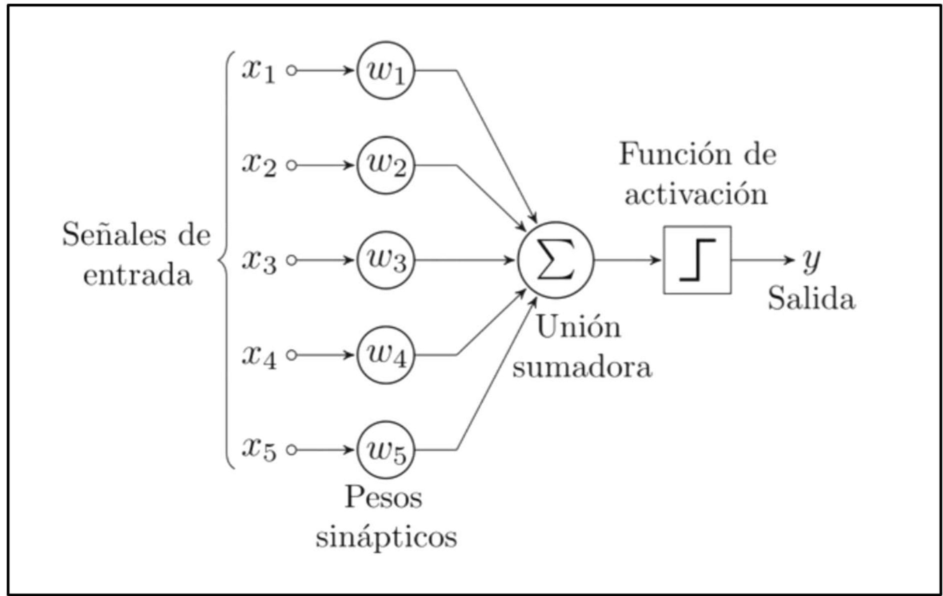


Figura 2.1 *Modelo del Perceptrón* [3]

Pero, ¿qué es una neurona? Nuestro cerebro está formado por una inmensa red de neuronas, las cuales se encargan de procesar todo tipo de entradas sensoriales, ya sean eléctricas o químicas, y generar una respuesta ante esos estímulos. El proceso para obtener esta respuesta comienza con las dendritas, las receptoras de estas entradas eléctricas, que se encargan de transmitir toda la información hasta el soma. Esta parte es el cuerpo principal de la neurona, donde dependiendo de la potencia del estímulo, puede activar la célula si pasa el umbral necesario y así proseguir con el envío de dicha información a través del axón (la parte emisora de la neurona, donde se envía la información) de esta neurona mediante las sinapsis, que son las uniones con las dendritas (receptores de las siguientes neuronas).

En conclusión, una neurona es un componente imprescindible en el procesamiento del cerebro, pues recibe, interpreta y envía información a otras neuronas conectadas, siendo necesario un conjunto de ellas para poder llevar a cabo nuestras acciones y pensamientos, desde los más simples hasta los más complicados. Es decir, una neurona está conectada a un conjunto de otras neuronas configurando lo que se ha denominado red neuronal.

Una red neuronal y una red neuronal artificial no son tan distintas, ambas están formadas por neuronas, cuyo comportamiento ya ha sido descrito anteriormente. Ambas requieren de una sinapsis o unión para transmitir la información de una a otra, una función para sumar todos los estímulos obtenidos de distintas neuronas, una función de activación y una salida.

Véase en la imagen 2.2 la comparación entre una neurona biológica y una artificial:

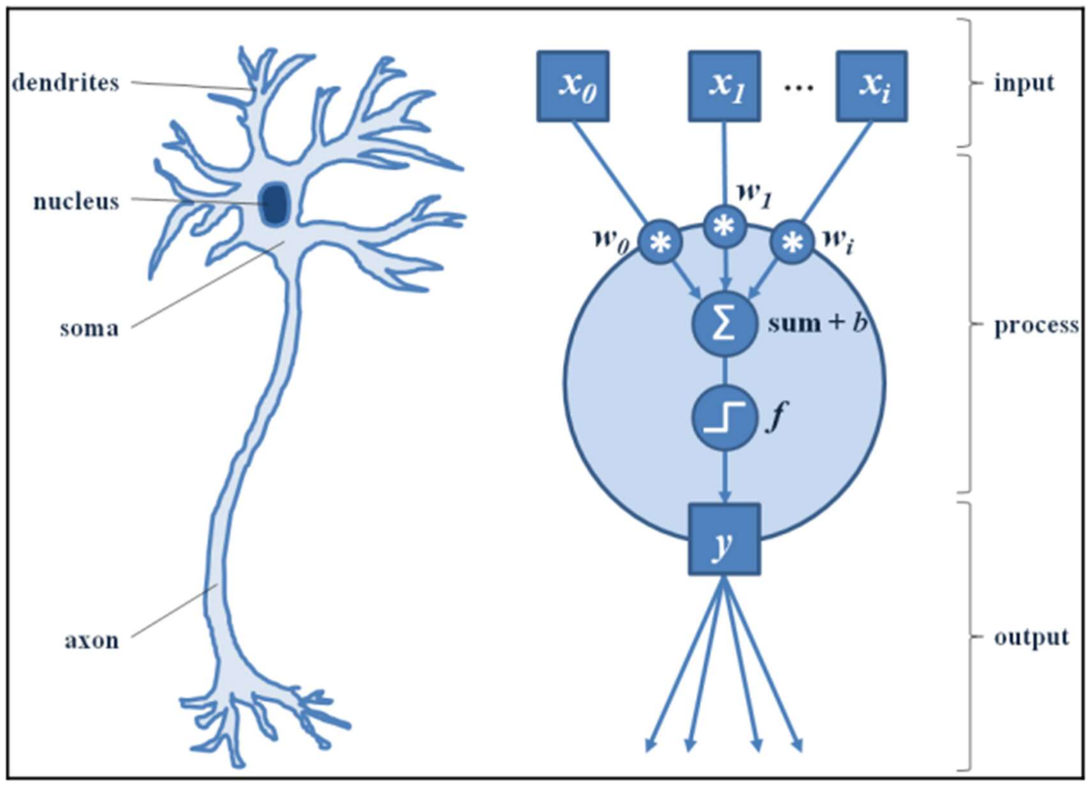


Figura 2.2 *A la derecha, una neurona biológica simplificada. A la derecha, una neurona artificial* [4]

Las dendritas conforman la entrada, el núcleo y el soma son los componentes que se encargan del sumatorio de la información y la activación, siendo el axón la salida de la información.

2.2. Tipos de redes neuronales artificiales

Podemos encontrar diferentes criterios a la hora de clasificar las redes neuronales artificiales (RNA): en torno a su número de capas, tipos de conexiones o grado de las conexiones. A continuación, vamos a describirlas brevemente.

Respecto al número de capas podemos distinguir las monocapas o las multicapas. Atendiendo al tipo de conexiones existen las no recurrentes y las recurrentes. En relación al grado de conexión encontramos las totalmente conectadas y las parcialmente conectadas.

Sin embargo, en relación al objetivo de este trabajo, nos interesa destacar un tipo de red neuronal muy efectiva en las tareas de visión artificial: las redes neuronales convolucionales.

2.3. Redes convolucionales

A diferencia de las redes neuronales tradicionales, que suelen procesar datos unidimensionales, las Redes Neuronales Convolucionales (Convolutional Neuronal Network, CNN) [5] son especialmente efectivas para tareas de visión artificial debido a la estructura de las imágenes, que generalmente tienen una organización bidimensional (ancho y alto) o tridimensional (ancho, alto y profundidad). Estas disposiciones espaciales se adaptan mal a las capas completamente conectadas de las redes neuronales tradicionales. Como para este proyecto vamos a trabajar con imágenes bidimensionales, nos centraremos en las redes neuronales convolucionales.

Las redes neuronales convolucionales están formadas por un subconjunto de capas ocultas cuyo trabajo consiste en detectar distintas características en las entradas. Cada una de estas capas ocultas forma parte de una jerarquía según el peso de las características a identificar. Las primeras capas comienzan con la detección de propiedades simples, por ejemplo, formas básicas de objetos, hasta llegar a las capas más profundas y de mayor peso asociadas a la detección de propiedades más complejas como, por ejemplo, rostros humanos. Además, estas redes neuronales son capaces de trabajar con datos multidimensionales. Estas capas mencionadas anteriormente están conectadas por neuronas que únicamente son capaces de acceder a sus elementos vecinos, es decir, elementos próximos a la capa anterior (parcialmente conectadas). Dicha región de la que se puede acceder a la información se llama el campo receptivo de las neuronas o más conocido como el tamaño del filtro (kernel).

A modo de ejemplo, el esquema en la figura 2.3 representaría lo explicado con anterioridad:

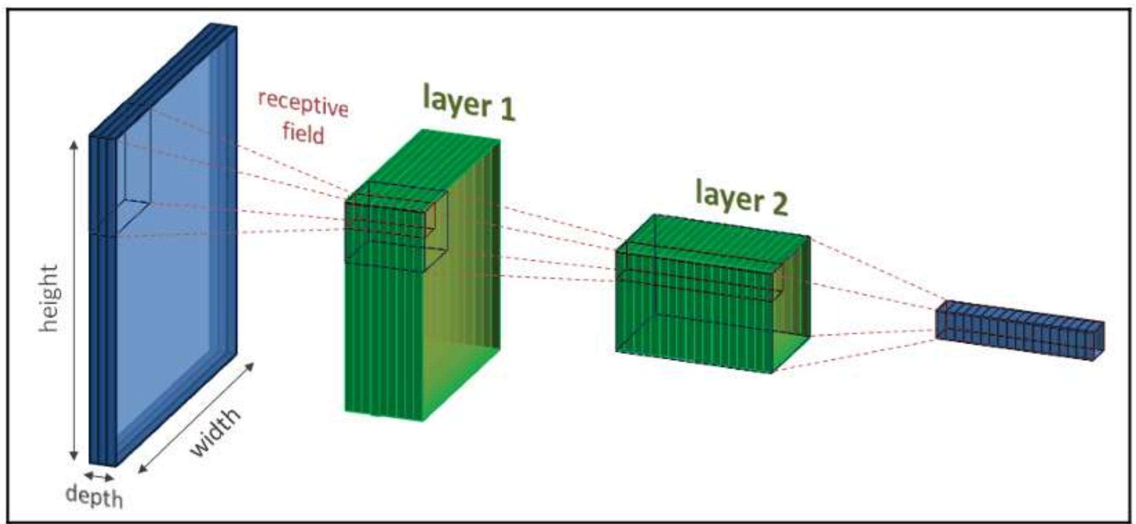


Figura 2.3 *Representación de una CCN. Demostración de los campos receptivos de las neuronas desde la primera capa hasta la última* [6]

2.3.1. Capas convolucionales

Como se ha explicado con anterioridad, las CNN son capaces de trabajar con objetos multimensionales. Las capas convolucionales conforman el núcleo de estas redes, recibiendo de ellas su nombre. Sus entradas están organizadas en tres dimensiones: ancho, alto y profundidad o espesor (HxWxD). Estas capas se utilizan principalmente para la detección de características locales como texturas, patrones, bordes o esquinas en las imágenes de entrada mediante un proceso de convolución. Este proceso consiste en aplicar a la imagen de entrada un filtro o kernel de dimensiones cuadradas o más específicas. Este filtro realiza un barrido por la imagen y va multiplicando los valores de los píxeles con los de este filtro, posteriormente suma los resultados para la obtención de una nueva matriz que destaca los aspectos específicos de la entrada.

Para una mejor comprensión, se muestra en la figura 2.4 un ejemplo de convolución entre la entrada y el kernel.

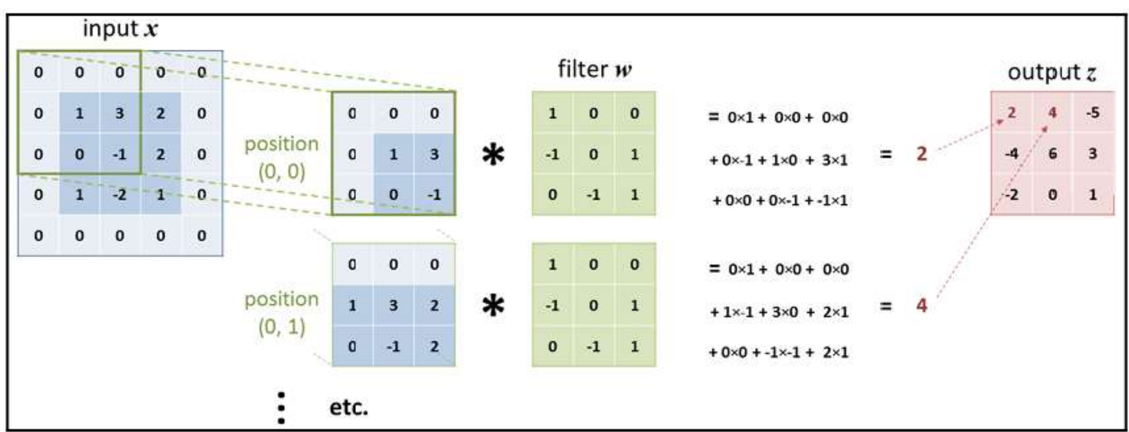


Figura 2.4 *Demostración de una convolución [7]*

En resumen, las capas convolucionales [8] son muy efectivas, y al apilar varias capas en modelos profundos, las capas cercanas a la entrada aprenden características de bajo nivel (como líneas), mientras que las capas más profundas capturan características de mayor nivel o más abstractas, como formas u objetos específicos. Generan mapas de características de datos para resumir la información de la entrada, sin embargo, esto conlleva una limitación: las posiciones de estas características son precisas, por lo que cualquier distorsión o pequeño cambio en la entrada generará otro mapa de características diferente.

Para abordar este problema, se puede agregar una capa de agrupamiento (que se describe a continuación) después de la capa convolucional.

Una vez que las capas convolucionales y de agrupamiento han extraído y resumido las características relevantes de los datos de entrada, es necesario transformar esta información en un formato que pueda ser interpretado por las capas finales encargadas de realizar la clasificación o regresión. Con el fin de alcanzar este objetivo, se utilizan las capas completamente conectadas y las de aplanamiento (flatten) que se explicarán más adelante.

2.3.2. Capas de agrupamiento (Pooling)

Las capas de agrupamiento [9] son un tipo de capa utilizadas en CNN, especialmente después de la capa convolucional. Su objetivo es reducir la dimensión espacial de los datos y extraer la información más relevante, descartando la menos importante.

Estas capas [10] operan sobre los mapas de características de forma individual y generan nuevos mapas con las características más relevantes del mapa anterior. De esta forma, se consigue un modelo robusto ante las alteraciones de las posiciones de las características en la entrada.

Dentro de las capas de agrupamiento, existen dos tipos muy utilizados [11]:

- Max Pooling: Es el tipo de capa de agrupamiento más conocida. Se utiliza para seleccionar los elementos máximos del mapa de características y reducir la cantidad de datos.

Es decir, la imagen de entrada se divide en regiones de dimensiones 2x2 o 3x3 y para cada una de estas regiones, se obtiene el valor máximo. De esta forma, se consigue una entrada de menor tamaño y más resumida, tal y como se esquematiza en la figura 2.5.

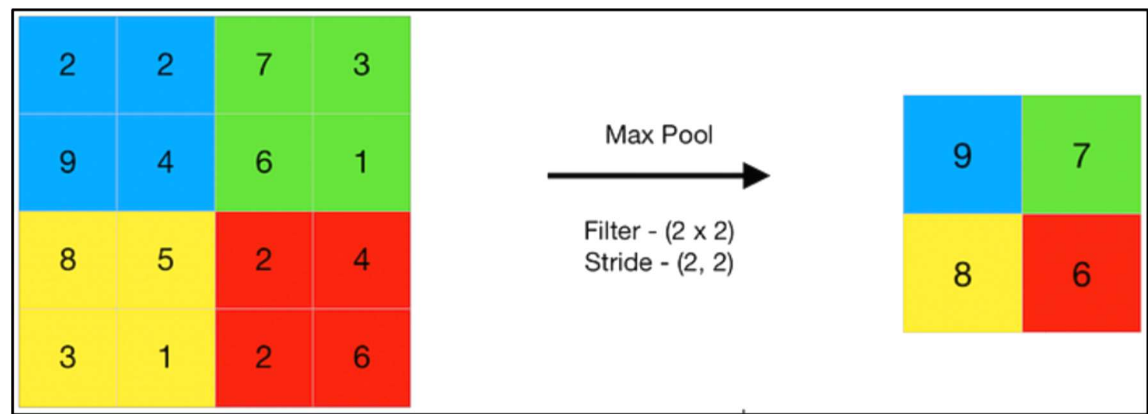


Figura 2.5 *Max Pooling*

- Average Pooling: Este tipo de capa de agrupamiento es similar al Max Pooling. Sin embargo, realiza el valor promedio de cada región 2x2 o 3x3 en lugar de seleccionar el valor máximo (figura 2.6).

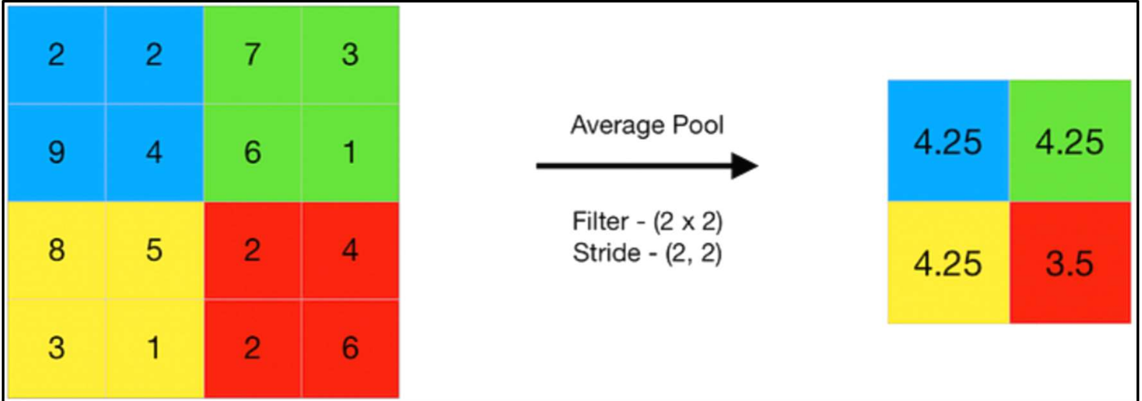


Figura 2.6 *Average Pooling*

2.3.3. Capas de aplanamiento (Flatten)

Las capas Flatten [12] se colocan después de las capas convolucionales y de agrupamiento, que generan salidas en forma de matrices 3D o tensores multidimensionales. Estas capas son utilizadas para transformar la salida de la capa anterior en un vector unidimensional, lo cual es necesario para que la información pueda ser procesada por la capa posterior, que son las capas totalmente conectadas (dense).

2.3.4. Capas totalmente conectadas (Dense)

Como indica su nombre, una capa Dense [13] está completamente conectada con la capa anterior, es decir, todas sus neuronas se encuentran conectadas a las neuronas de la capa anterior. Cuando una capa Dense es la capa final de la red neuronal, su número de neuronas corresponde al número de salidas de la red.

En redes neuronales de clasificación, cada neurona final representa una clase de dato y su valor de salida, generalmente después de una función de activación [14] como Softmax o Sigmoid.

2.3.5. Capas dropout

Estas capas tienen como función evitar el sobreajuste durante el entrenamiento, es decir, que el modelo de red no dependa demasiado de los datos usados durante el entrenamiento y así obtener un modelo más robusto. Esta capa dropout [15][16] funciona desactivando de forma aleatoria una fracción de las neuronas durante el entrenamiento para obligar a las neuronas a adaptarse y aprender patrones más generales. De esta forma se mejora la capacidad del modelo para generalizar, es decir, reconocer patrones con los que no ha sido entrenada.

2.4. Aprendizaje supervisado

Para obtener la red neuronal adecuada a nuestro problema de clasificación, se van a utilizar herramientas de aprendizaje automático, en concreto, aprendizaje de tipo supervisado.

En el aprendizaje supervisado [17] se utiliza un conjunto de datos etiquetados para modelar una tarea. Es decir, el modelo dispone de información sobre las categorías o los valores asociados a

los datos de entrada y va ajustándose para aprender la relación entre las entradas y sus salidas esperadas en el nuevo dominio, partiendo de ejemplos específicos.

Como se muestra en figura 2.7, se muestra el flujo general del proceso de entrenamiento de un modelo supervisado.

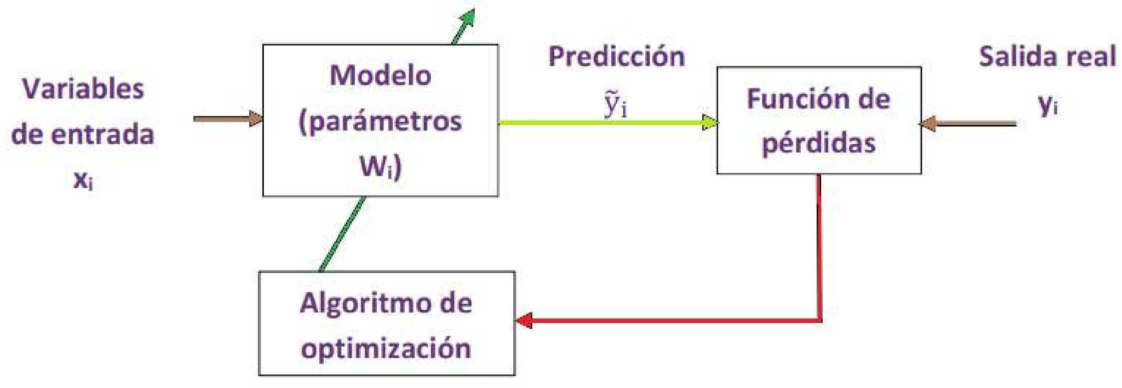


Figura 2.7 *Flujo del proceso de entrenamiento de un modelo supervisado* [18]

En este esquema, vemos los elementos y bloques que definen el proceso de entrenamiento supervisado:

Variables de entrada x_i

Las variables de entrada representan los datos de entrada que el modelo recibe para procesar. En el caso de este proyecto, las imágenes de piezas LEGO.

Etiquetas o salida real y_i

Al tratarse de aprendizaje supervisado, se debe conocer la clase a la que pertenece cada conjunto de variables de entrada; por tanto, para entrenar el modelo se debe disponer de un conjunto de datos (dataset) formado por vectores de variables de entrada y su correspondiente etiqueta o salida real. Estos datos se deben organizar en dos partes.

- Las imágenes de entrenamiento [19] son el conjunto de datos que la red neuronal usa para aprender, es decir, son los datos que procesa repetidamente. Durante el entrenamiento, la red calcula los errores en sus predicciones con una función de pérdidas y, de esta forma, ajusta los parámetros del modelo (como los pesos de las conexiones entre las neuronas) con un algoritmo de optimización para reducirlos, obteniendo así mejores predicciones. Este es el proceso representado en la figura 2.7.

- Las imágenes de validación [20]. Un segundo conjunto sirve para verificar si la red está o no aprendiendo de forma efectiva sin ajustar los parámetros internos de dicha red. Simplemente se usa el subconjunto de los datos de validación para evaluar su precisión o pérdidas.

Modelo (parámetros)

El modelo es el algoritmo matemático que, a partir de las variables de entrada x_i debe proporcionar la salida y_i. Una vez definido el modelo (en nuestro caso una CNN) durante el

entrenamiento, los parámetros del modelo (Wi), son ajustados para que la salida que proporciona $\tilde{y}_i$ sea lo más próxima posible a la salida real y_i.

Función de pérdidas

Con la función de pérdidas se comparan las predicciones del modelo ($\tilde{y}_i$) con las salidas reales (y_i) para obtener una medida del error cometido. De esta forma se sabe si el modelo está entrenado correctamente.

En este trabajo se ha escogido la función Sparse Categorical Crossentropy [21], pues se está trabajando con un número entero de etiquetas (indicado más adelante con Sparse al cargar las imágenes de los directorios) y se dispone de 7 clases de etiquetas distintas. De esta forma no se obtiene un vector de siete elementos, sino un número entero. Por ejemplo, en lugar de usar [1, 0, 0] para "Pieza 1", [0, 1, 0] para "Pieza 2", etc., se utiliza 0, 1, 2, etc.

Fórmula matemática de la función Sparse Categorical Crossentropy:

$$L_{CE}(y_i, \tilde{y}_i) = -\sum_{i=1}^{n} y_i \log(\tilde{y}_i)$$

Esta función compara las probabilidades estimadas ($\tilde{y}_i$) con las etiquetas reales (y_i) para n clases.

Si el modelo asigna una probabilidad alta ($\tilde{y}_i$ cercano a 1) a la clase correcta (y_i =1), el término L_{CE} será pequeño, lo que significa que la pérdida (error) en esa predicción es ínfima.

Por el contraria, si el modelo asigna una probabilidad baja ($\tilde{y}_i$ cercano a 0) a la clase correcta, el valor de L_{CE} será mayor. Cuanto más baja sea la probabilidad estimada, mayor será la pérdida.

Algoritmo de optimización

Una vez calculada la pérdida, el algoritmo de optimización ajusta los parámetros del modelo (Wi) para reducir esta pérdida. Esto se realiza mediante la actualización de dichos parámetros del modelo en base a los resultados obtenidos en la función de pérdidas.

En este trabajo se utiliza el algoritmo de optimización conocido como Adamax. Perteneciente al conjunto de técnicas de optimización basadas en el descenso de gradiente, el optimizador Adamax [22] es una variante del optimizador Adam muy robusta. Está basado en la norma infinito, lo que lo hace particularmente útil en problemas donde los gradientes son esporádicos o características específicas que dificultan el uso de optimizadores estándar.

Tasa de aprendizaje

En todos los algoritmos basados en el descenso de gradiente la tasa de aprendizaje (learning rate) es un parámetro que controla el tamaño de los pasos que el optimizador toma para ajustar los pesos del modelo en cada iteración. En este caso, no se seleccionó una tasa de aprendizaje de manera manual. En su lugar, se utilizó el optimizador Adamax, el cual incluye un valor predeterminado para la tasa de aprendizaje (0.002) [23]. Esto permitió que el modelo ajustara sus pesos automáticamente utilizando esta configuración por defecto.

Además de los parámetros del modelo que son las variables que se deben calcular como resultado del entrenamiento, existen otro conjunto de parámetros denominados hiperparámetros que determinan cómo se va a realizar éste. La tasa de aprendizaje ya

mencionada es uno de estos hiperparámetros que va a determinar la rapidez y estabilidad a la hora de aproximarse a los valores óptimos del modelo. Otros hiperparámetros son:

Épocas

El entrenamiento se desarrolla en un número de épocas (epochs). Denominamos época al periodo que utiliza el modelo para procesar todas las imágenes de entrenamiento una vez.

El tamaño de lote identifica el número de muestras de entrada procesadas en cada actualización de los parámetros W_i. Los parámetros internos del modelo no se ajustan para cada muestra de entrada, sino para un paquete o lote de ellas cuyo tamaño se define con este hiperparámetro. Cuanto más pequeño es el tamaño del lote, requerirá menos memoria durante el entrenamiento, aunque el coste de tiempo será mayor. De esta forma, con los lotes se divide el conjunto de datos en fragmentos manejables y se facilita el entrenamiento en hardware con memoria limitada.

Es decir, el modelo procesa cada lote de manera independiente, actualizando sus parámetros internos al final del procesamiento de cada lote. Este proceso se conoce como actualización de parámetros.

Una vez decididas las épocas y el tamaño del lote, hay que calcular el número de lotes necesarios para que en una sola época se lleguen introducir todas las imágenes durante el entrenamiento. Tanto para el entrenamiento como para la validación es necesario calcular los lotes por época necesarios (steps per epoch), ambos se calculan con la siguiente fórmula:

$$Steps\ per\ epoch/validation\ steps = \frac{n^{\underline{o}}\ total\ de\ imágenes\ entrenamiento/validación}{n^{\underline{o}}\ de\ batch\ size}$$

Un factor importante a tener en cuenta en el proceso de entrenamiento es el ya mencionado sobreajuste (overfitting). Aunque el entrenamiento en múltiples épocas y lotes mejora la capacidad del modelo para identificar patrones, existe el riesgo de que el modelo aprenda a memorizar características específicas de las imágenes del conjunto de entrenamiento, en lugar de generalizar bien para datos nuevos. Para mitigar este riesgo, es fundamental implementar técnicas como las capas de dropout, anteriormente explicadas, o realizar unas transformaciones aleatorias a las imágenes usadas para el entrenamiento, también conocida como augmentación de datos, que modifican las imágenes de entrenamiento antes de cada época y así incrementar su variabilidad

2.5. Aprendizaje por transferencia

También conocido como Transfer Learning [24] es una técnica de Machine Learning en la que se reutiliza o se adapta un modelo pre-entrenado para realizar una nueva tarea. Esta técnica es especialmente útil en escenarios donde hay pocos datos disponibles para entrenar desde cero o cuando se busca ahorrar tiempo y recursos computacionales. Se aprovecha el conocimiento previamente adquirido por un modelo (como patrones aprendidos en grandes conjuntos de datos) para aplicarlo en un dominio relacionado o una tarea similar.

En este caso se mantienen los valores de los parámetros de las primeras capas del modelo existente y solo se modifican los correspondientes a las capas más próximas a la salida usando la información del conjunto de entrenamiento del problema. De esta manera se reduce el número de parámetros a determinar y, por tanto, la necesidad de datos de referencia.

2.6. Segmentación de objetos

Una de las principales limitaciones al trabajar con redes neuronales entrenadas de la manera descrita es que suelen estar diseñadas para identificar un solo objeto a la vez en cada predicción. Sin embargo, dado que en este trabajo se busca identificar múltiples piezas simultáneamente, es necesario realizar una segmentación previa. Esto permite aislar cada pieza individualmente, asegurando que la red procese y clasifique cada objeto de manera independiente y precisa.

Para abordar este problema, se ha realizado un trazado de contornos mediante la función cv2.findContorus() de la biblioteca OpenCV2. Como se sabe, los contornos son los límites de los objetos o regiones dentro de una imagen y, mediante esta función, se consiguen extraer los contornos de dicha imagen para posteriormente procesarlos. La función cv2.findContours() utiliza la información de gradiente para seleccionar los píxeles conectados entre sí que formarán parte de un contorno

Para poder extraer los contornos, dicha entrada debe ser binaria, es decir, debe tratarse de una imagen en escala de grises con valores de píxel comprendidos entre el 0 y el 255.

También, dentro de la función deben indicarse unos parámetros [25]: en primer lugar, el modo de recuperación de contornos [26], el cual se escogió la opción de cv2.RETR_TREE, pues recupera todos los contornos detectados en la imagen y genera una jerarquía completa de los contornos detectados; en segundo lugar, el método de aproximación de contornos [27], donde se escogió la opción de mantener solo los puntos más esenciales (cv2.CHAIN_APPROX_SIMPLE). Por ejemplo, se codificará con 4 puntos en el caso de un contorno rectangular vertical, dejando solo los puntos finales en vez de todos los que componen dicho rectángulo.

2.7. Librerías y recursos

En este apartado se van a explicar las herramientas software utilizadas en el desarrollo del proyecto.

2.7.1. Python

Python [28] es un lenguaje de programación de alto nivel, de código abierto y multiplataforma, ampliamente reconocido por su flexibilidad, simplicidad y potencia. Este lenguaje permite a los desarrolladores enfocarse en la lógica del problema en lugar de los detalles técnicos como la gestión de memoria. Aunque soporta varios paradigmas de programación, como la programación funcional y la estructurada, Python está principalmente orientado a objetos, organizando el código en clases y objetos para facilitar su reutilización y mantenimiento. Esto lo hace más accesible y fácil de usar en comparación con lenguajes más estrictos como C++ o Java.

Su diseño se centra en la legibilidad del código y su simplicidad, permitiendo una programación más rápida y eficiente, reduciendo la probabilidad de cometer errores.

Además, se trata de un lenguaje interpretado, es decir, el código fuente se convierte en bytecode y es posteriormente ejecutado a través de una máquina virtual de Python o PVM (Python Virtual Machine).

Las aplicaciones más comunes de Python son el desarrollo web, la ciencia y análisis de datos, así como el aprendizaje automático, la inteligencia artificial y la automatización.

2.7.2. Librerías

Para la realización de diferentes tareas, además de las funciones que forman parte del paquete básico del lenguaje Python es necesario utilizar un conjunto de librerías.

- TensorFlow [29]

Es una librería de código libre desarrollada por Google para construir y entrenar redes neuronales capaces de asemejarse al razonamiento humano, pudiendo reconocer patrones. Nos permite crear e implementar modelos de aprendizaje automático (Machine Learning) y nos proporciona las herramientas necesarias para llevarlo a cabo de manera eficiente. Ofrece también APIs (interfaces de uso intuitivo) intuitivas que facilitan el desarrollo y entrenamiento de modelos de redes neuronales, acelerando el proceso de automatización de diversas tareas.

    - Keras [30]

Es una librería de código abierto antiguamente perteneciente a la librería de TensorFlow, que actualmente se está desarrollando como software independiente. Fue creada en 2015 por un desarrollador de Google llamado François Chollet. Su objetivo principal es agilizar y simplificar la creación de redes neuronales, abstrae la complejidad de los detalles más técnicos y ofrece una experiencia más accesible tanto para principiantes como para expertos en la programación.

- Matplotlib [31]

Es una librería de código abierto creada en 2002 por el neurobiólogo John Hunter. Originalmente, su propósito era visualizar las señales eléctricas del cerebro de personas con epilepsia. Sin embargo, tras el fallecimiento de Hunter, la comunidad continuó su desarrollo, ampliando sus capacidades. Hoy en día, la librería se utiliza ampliamente para generar gráficas, histogramas, diagramas y visualizaciones de alta calidad en diversos campos.

- Numpy [32]

Numpy es una librería de Python esencial para el procesamiento de datos y cálculos matemáticos. Permite crear y manipular colecciones de datos de un mismo tipo dispuestos en varias dimensiones (arrays), los cuales son más rápidos y eficientes que las listas predefinidas de Python.

- OpenCV2 [33]

OpenCV2 es la versión más actualizada de OpenCV (Open Source Computer Vision Library). Es una librería muy utilizada en el procesamiento de imágenes y visión por computador. Proporciona herramientas que nos permite realizar desde la detección de rostros hasta el análisis en vídeo en tiempo real y el reconocimiento de objetos.

Gracias a los filtros, transformaciones geométricas, manipulación de colores y detección de bordes que nos proporciona, se pueden realizar una gran variedad de operaciones con las imágenes. También incluye herramientas [34] como:

    - Reconocimiento de objetos
    - Seguimiento de movimiento
    - Realidad aumentada (AR)
    - Procesamiento de vídeo en tiempo real

  - Detección de rostros y reconocimiento facial [35][36]

2.7.3. Recursos

- MobileNetV2 [37][38]

Presentada en 2018 como una versión mejorada de MobileNetV1, MobileNetV2 es una red neuronal convolucional optimizada para dispositivos móviles perteneciente a TensorFlow. Se centra en equilibrar precisión y eficiencia, permitiendo con bajo consumo de recursos, la ejecución de modelos de visión por computador en tiempo real.

Su arquitectura cuenta con 53 capas de profundidad y es capaz de clasificar objetos en 1000 categorías distintas al haber sido entrenada con más de un millón de imágenes.

En este proyecto, MobileNetV2 se ha utilizado como modelo base para la red neuronal. Esto significa que la arquitectura preentrenada de MobileNetV2 sirve como punto de partida, permitiendo transferir conocimientos previamente adquiridos (transfer learning) para entrenar un modelo adaptado a los objetivos específicos del trabajo. Este enfoque facilita el desarrollo de un modelo más eficiente y especializado, reduciendo la necesidad de entrenar desde cero.

- Visual studio [39]

Es un potente entorno de desarrollo integrado (IDE) desarrollado por Microsoft que permite escribir, editar, depurar y crear código; todo dentro de una misma herramienta. Facilita la creación de aplicaciones en una gran variedad de plataformas, como dispositivos móviles, escritorios, webs y en la nube. Además, es compatible con diversos lenguajes de programación, como C++, C#, JavaScript y Python. En la figura 2.8 se muestra la interfaz de este entorno de desarrollo.

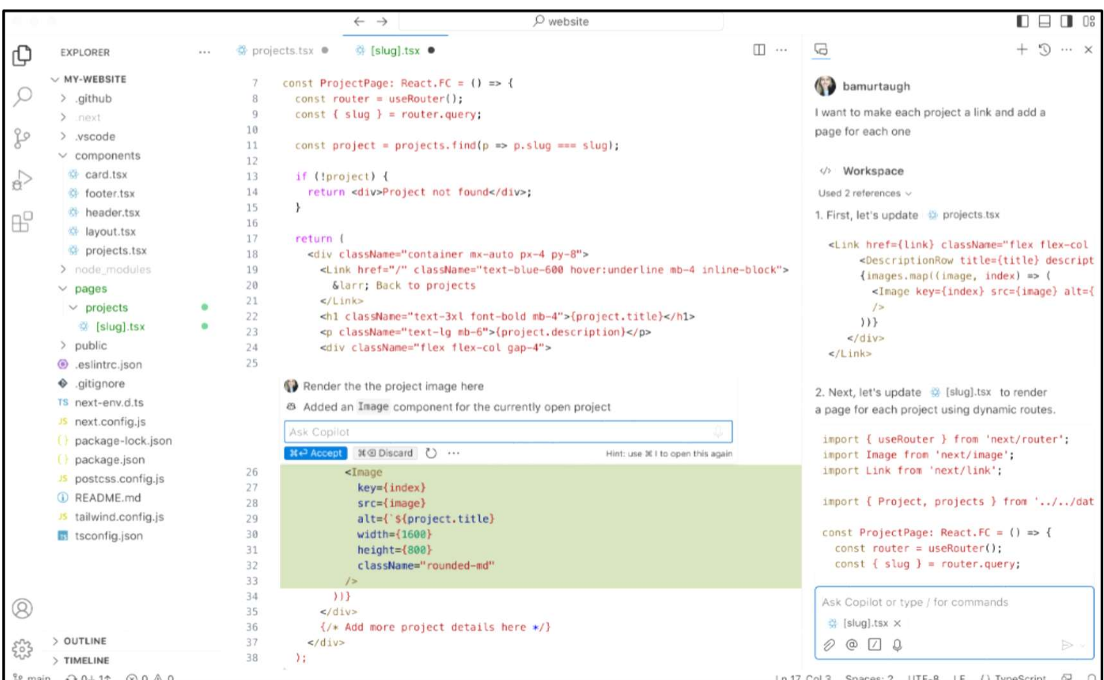


Figura 2.8 *Interfaz de Visual Studio* [40]

- Droidcam [41]

Es una aplicación gratuita que permite convertir un dispositivo Android en una cámara web para un ordenador. Esta conexión puede realizarse vía USB o mediante conexión Wi-Fi. Además, permite utilizar el micrófono del móvil como fuente de audio.

Permite ajustes de video, como el brillo y el contraste, y soporta resoluciones de hasta 720p si la cámara del dispositivo Android lo permite.

- Página base de imágenes Kaggle [42]

Kaggle es una plataforma en línea especializada en datos y aprendizaje automático. Ofrece un entorno colaborativo para que desarrolladores y entusiastas de la inteligencia artificial puedan trabajar en proyectos, participar en competiciones y mejorar sus habilidades a través de datasets y notebooks gratuitos. Es decir, nos proporciona conjuntos de datos para el entrenamiento de una red neuronal en este trabajo.

Para este proyecto se han extraído dos bases de datos:

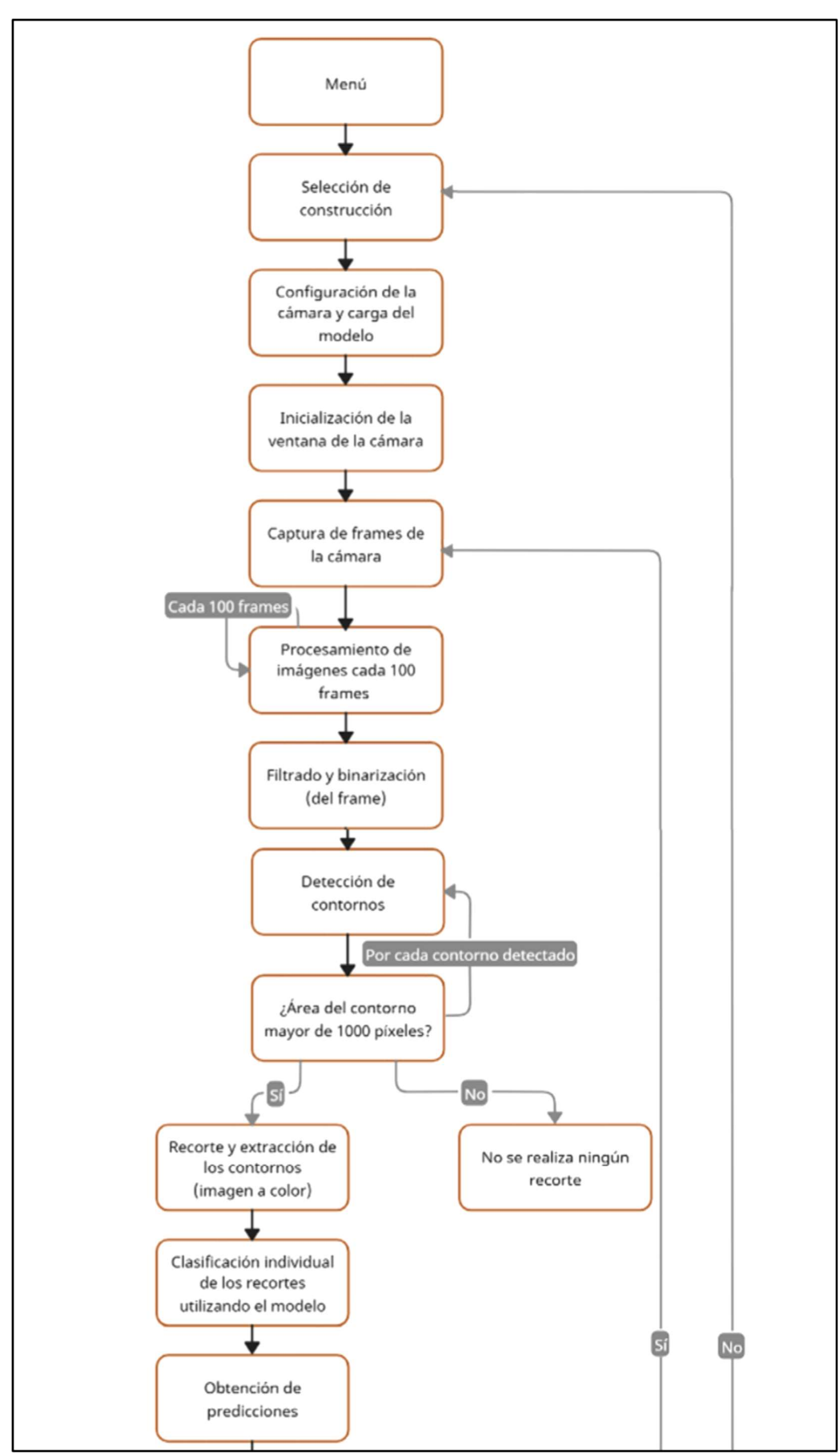
- B200C LEGO Classification Dataset [43]
- Images of LEGO Bricks [44]

3. DESARROLLO DEL PROYECTO

3.1. Diseño general

En este trabajo ha sido necesario la creación de dos programas. El primero es el entrenamiento del modelo de una red neuronal, para ello, se ha tenido que seleccionar unos parámetros adecuados para que este entrenamiento resultase más preciso a la hora de obtener los resultados. Dichos parámetros se han ido variando a medida que se iban obteniendo las gráficas de los resultados. Como se ha mencionado anteriormente, las imágenes utilizadas para esta primera parte del trabajo se han extraído de la página web Kaggle.com. En este caso, se han utilizado dos bases de datos distintas de donde se han seleccionado las imágenes necesarias para el entrenamiento. Una vez obtenidas y seleccionadas las imágenes, se ha procedido con el entrenamiento de la red neuronal en el lenguaje de Python. Tras el entrenamiento del modelo, se guarda como un archivo '.h5' para su posterior uso en el siguiente programa.

El segundo programa corresponde al uso del modelo entrenado para el reconocimiento de piezas en un proceso de selección. Primero se escogerá una de las tres construcciones disponibles para obtener el listado de piezas que la aplicación debe localizar. Entonces, el programa iniciará la cámara, viendo desde arriba las piezas. Después se tomarán automáticamente capturas de la cámara para su posterior tratamiento con filtros y así, proceder con la detección de contornos. Una vez detectados los contornos, se guardan los recortes de los objetos detectados en una lista de la cual, se irá pasando la red neuronal elemento por elemento. Así, se van detectando uno a uno los objetos LEGO y se muestra posteriormente por pantalla las piezas que faltan para completar la figura y se marcan las piezas ya localizadas en la ventana que muestra la imagen de la cámara en tiempo real. En la figura 3.1 se muestra el diagrama de flujo de este segundo programa. Este diagrama resume los pasos clave que toma el programa en tiempo real, desde la entrada de los datos hasta la obtención de la predicción final.


Menú
Selección de construcción
Configuración de la cámara y carga del modelo
Inicialización de la ventana de la cámara
Captura de frames de la cámara
Cada 100 frames
Procesamiento de imágenes cada 100 frames
Filtrado y binarización (del frame)
Detección de contornos
Por cada contorno detectado
¿Área del contorno mayor de 1000 píxeles?
Sí
No
Recorte y extracción de los contornos (imagen a color)
No se realiza ningún recorte
Clasificación individual de los recortes utilizando el modelo
Obtención de predicciones
Sí
No

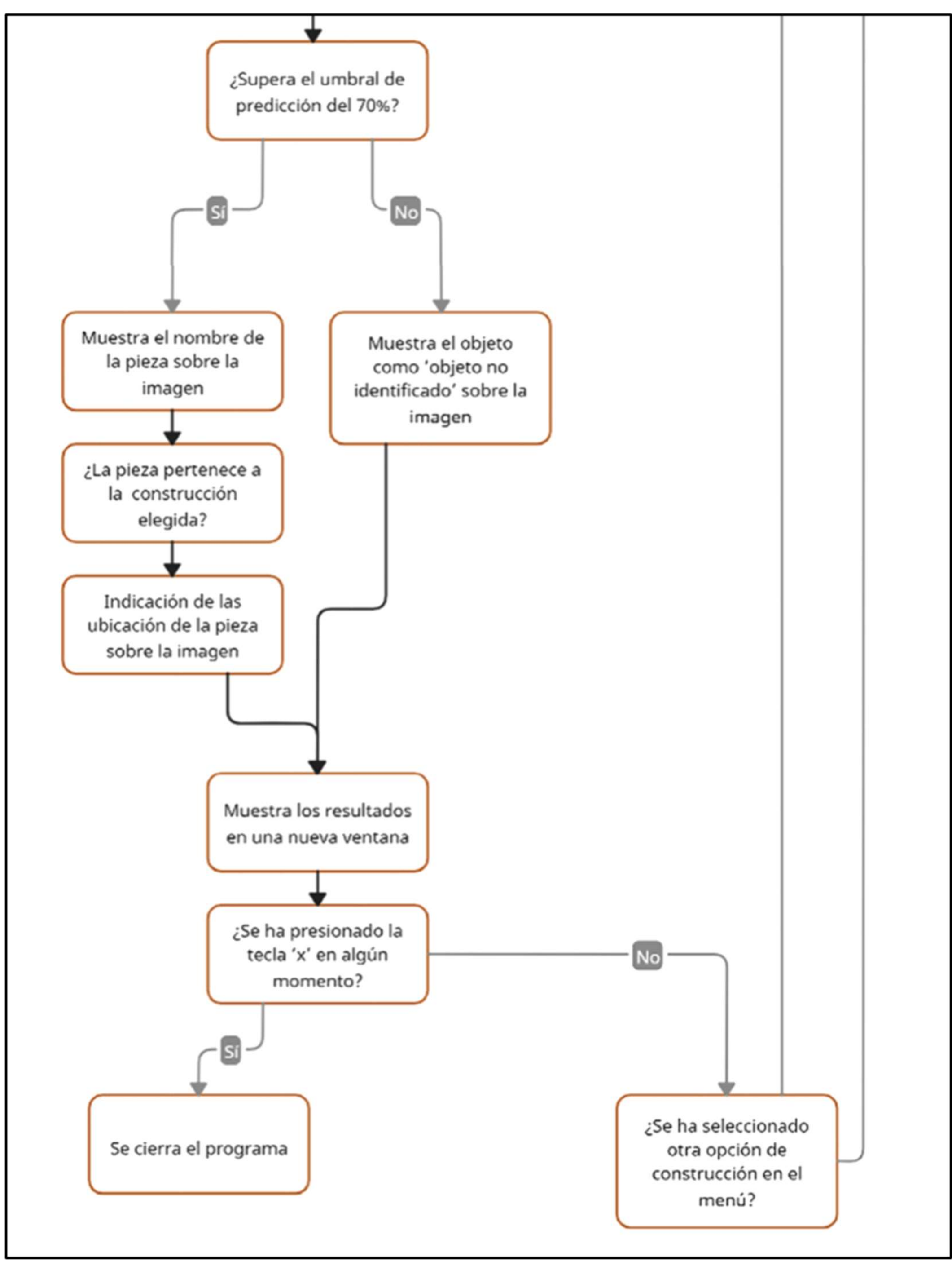


Figura 3.1 *Diagrama de flujo del programa principal*

3.2. Organización del trabajo realizado

Para la consecución de los objetivos, se ha procedido a diseñar un plan de trabajo que seguirá los siguientes pasos:

- Entrenamiento de la red neuronal (programa 1). El desarrollo de este programa incluye:
    - Selección de piezas LEGO.
    - Búsqueda de imágenes de las piezas LEGO seleccionadas en Kaggle.es.
    - Organización de las imágenes para su posterior entrenamiento de red neuronal.
    - Diseño del programa de entrenamiento y validación.
    - Muestra de resultados del modelo neuronal.
    - Validación con cámara real del modelo neuronal pieza a pieza.
- Aplicación de selección de piezas para la construcción de figuras LEGO (programa 2). El desarrollo de este programa incluye:
    - Diseño programa aplicación de selección de piezas para la construcción de figuras LEGO

3.2.1. Entrenamiento de la red neuronal (programa 1).

I. Selección de piezas LEGO

Como se ha mencionado en diversas ocasiones, para este trabajo se ha decidido utilizar siete piezas LEGO de distintas dimensiones. Para escoger entre las diferentes piezas existentes en las bases de datos, se ha tenido en cuenta diversos criterios:

- La disponibilidad de imágenes de estas piezas, buscando una gran diversidad entre estas imágenes, pues de esta forma evitaremos que la red neuronal se aprenda las imágenes de forma mecánica (o de memoria).

- Una diferencia visual de las piezas, puesto que si se tiene en cuenta que buscamos que las piezas serán vistas desde arriba, no podemos escoger piezas que tengan sus características más destacables en algún perfil que no se muestre posteriormente al enfocar la pieza, véase el ejemplo en la figura 3.2:

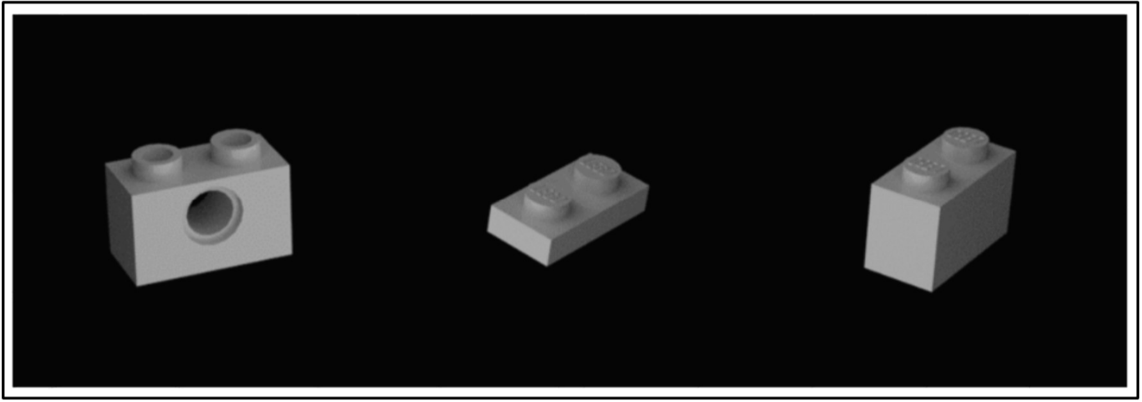
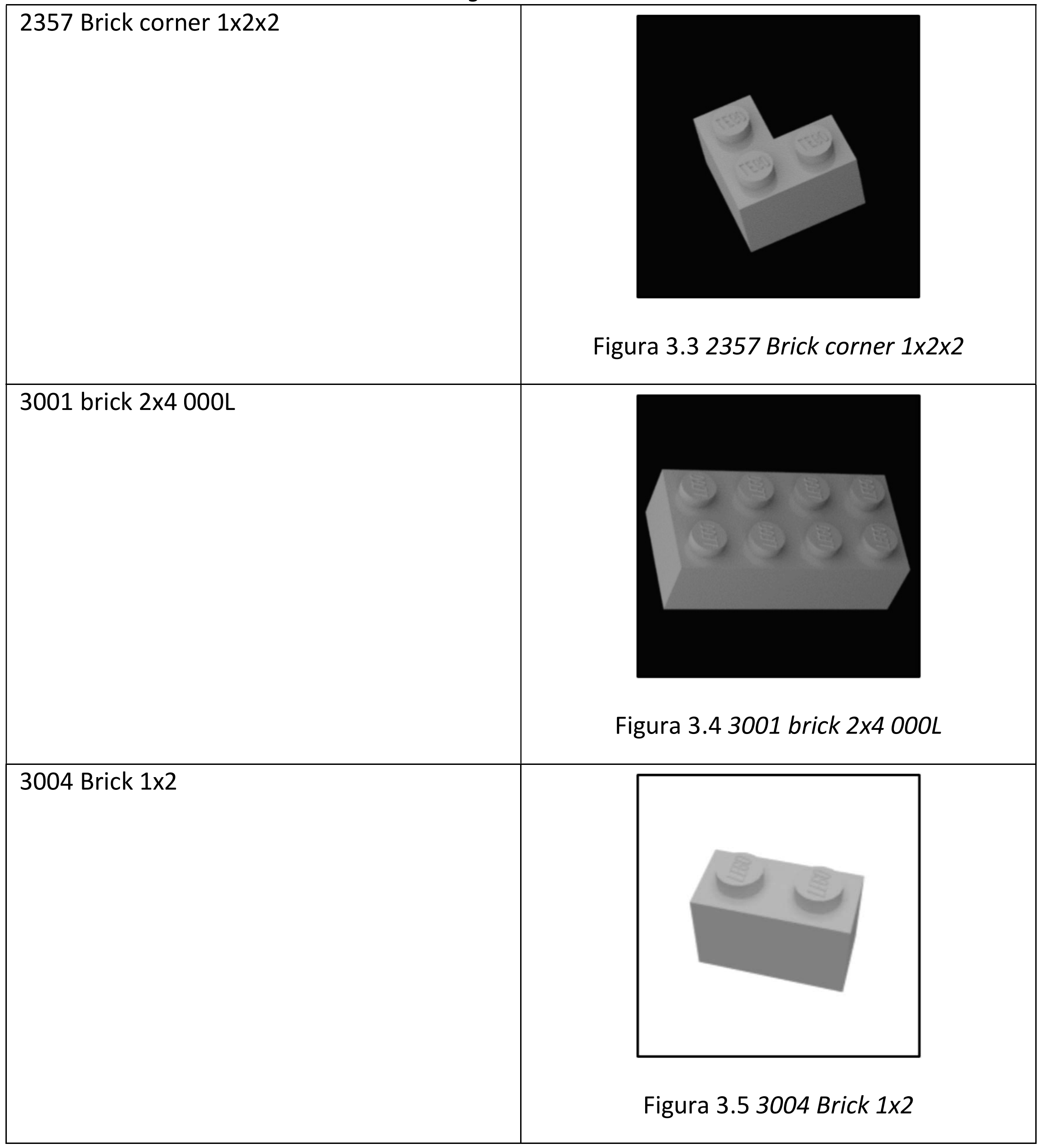


Figura 3.2 *De izquierda a derecha: 3700 Technic Brick 1x2 058L, 3023 Plate 1x2 221R y 3004 brick 1x2 047R*

A simple vista, desde esta perspectiva, diferente a la que se piensa utilizar para la detección de las piezas, se ve claramente la diferencia de estas. Pero visto desde arriba no, por lo que hemos evitado escoger piezas cuyas características principales se encuentren en los laterales de las piezas o en su altura.

Como resultado, se han escogido las siguientes piezas (ver Tabla 1):

Tabla 1. *Figuras seleccionadas*



2357 Brick corner 1x2x2	Figura 3.3 *2357 Brick corner 1x2x2*
3001 brick 2x4 000L	Figura 3.4 *3001 brick 2x4 000L*
3004 Brick 1x2	Figura 3.5 *3004 Brick 1x2*

3005 Brick 1x1	Figura 3.1 *3005 Brick 1x1*
3010 brick 1x4 000L	Figura 3.7 *3010 brick 1x4 000L*
3062 Round Brick 1x1 390L	Figura 3.8 *3062 Round Brick 1x1 390L*
43093 Bush 2M friction - Cross axle 380L	Figura 3.9 *43093 Bush 2M friction - Cross axle 380L*

II. Búsqueda de imágenes de las piezas LEGO seleccionadas en Kaggle.es

Para la obtención de imágenes de estas piezas, se ha escogido la página web de Kaggle.com, de donde se han sacado dos bases de datos esenciales para el entrenamiento de la red neuronal.

Se ha tenido en cuenta la existencia de diversos formatos de imágenes de estas piezas en diversas bases de datos de Kaggle:

Pese a que existen múltiples bases de datos, la mayoría de las imágenes de una misma pieza son exactamente iguales con fondos distintos como se muestra en la figura 3.10:

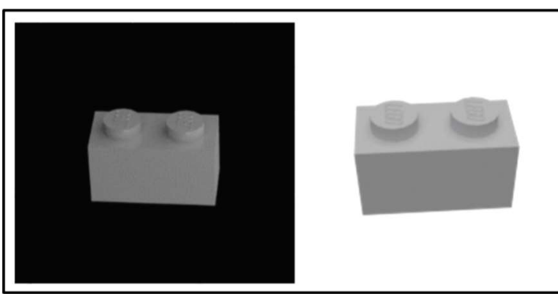


Figura 3.10 *Comparación entre dos imágenes de la misma pieza de dos distintos dataset*

Y para el entrenamiento óptimo de una red neuronal, se busca más diversidad de imágenes.

En la búsqueda de estas imágenes, finalmente se han escogido dos bases de datos distintas (una donde las imágenes de ven con claridad y otra donde existe más variedades de una misma pieza en cuanto a colores y fondos).

III. Organización de las imágenes para su posterior entrenamiento de red neuronal

Una vez obtenidas todas las imágenes de las piezas, se procede a otra parte muy importante, previa al uso de las imágenes en el programa. Para el siguiente paso, el diseño del programa de entrenamiento y validación, debemos tener organizadas las imágenes en dos carpetas principales, una llamada "Entrenamiento" y otra "Validación". Dentro de cada una de estas dos carpetas, habrá siete subdirectorios más, cada uno con el nombre correspondiente a una de las siete piezas y, dentro de estas, colocaremos las imágenes que hemos seleccionado en el paso anterior.

Es importante tener en cuenta en este caso que las siete carpetas, tanto dentro de la carpeta de entrenamiento como en la de validación han de tener los mismos nombres para la misma pieza. Es decir, si para la pieza con forma de esquina, la carpeta de entrenamiento donde se encuentran las imágenes de esa pieza se llama 'esquina', en la carpeta de validación donde se encuentre otra para esa misma pieza, deberá tener también el nombre de 'esquina'.

También, es importante no introducir las mismas imágenes para una pieza en las carpetas de validación y de entrenamiento. Para las carpetas de una misma pieza, si se ha colocado una imagen en la carpeta de la pieza dentro del entrenamiento, esta no podrá ser usada en la carpeta dentro de la validación.

Y, por último, en las carpetas situadas dentro del directorio de entrenamiento, deberá haber más imágenes respecto al directorio de validación, siendo al menos el doble; intentando que

exista aproximadamente un mismo número de imágenes entre los subdirectorios dentro de 'Validación' y 'Entrenamiento'. La estructura de carpetas y el número de imágenes en cada una de ellas se muestra en la figura 3.11.

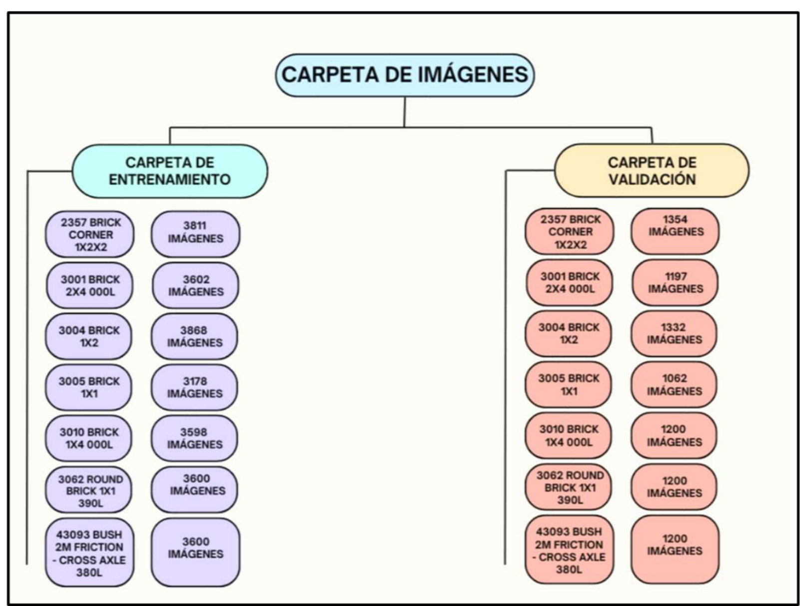


Figura 3.21 *Esquema de la distribución de las imágenes*

IV. Diseño del programa de entrenamiento y validación

A continuación, se explicarán las diferentes partes que componen este programa.

A. Preparación del Dataset de imágenes

Una vez realizada la selección de imágenes, se colocaron sobre sus respectivas carpetas. En este apartado disponemos de un total de 33.801 imágenes, las cuales han sido divididas entre 7 clases diferentes de piezas LEGO, las cuales hemos mostrado anteriormente:

- 2357 Brick corner 1x2x2
- 3001 brick 2x4 000L
- 3004 Brick 1x2
- 3005 Brick 1x1
- 3010 brick 1x4 000L
- 3062 Round Brick 1x1 390L
- 43093 Bush 2M friction - Cross axle 380L

Como también se ha mencionado anteriormente, se han separado las imágenes en una parte de entrenamiento y en otra de validación, resultando en 25.256 imágenes para el entrenamiento y 8.545 imágenes para la validación.

Para adaptar las imágenes al proceso de entrenamiento, es necesario introducir previamente una librería necesaria para la preparación de las imágenes.

```
13    #Para la preparación de las imágenes
14    from tensorflow.keras.preprocessing.image import ImageDataGenerator
```

Figura 3.12 *Librerías utilizadas*

En el código se define el directorio de trabajo, dentro del cual se encontrarán las dos carpetas de trabajo. La figura 3.13 muestra el fragmento de código empleado para ello:

```
20    dir_base="C:\\Users\\marta\\Desktop\\TFG\\Imagenesgris"
21
22    dir_train="C:\\Users\\marta\\Desktop\\TFG\\Imagenesgris\\Entrenar"
23    dir_valid="C:\\Users\\marta\\Desktop\\TFG\\Imagenesgris\\Validar"
```

Figura 3.13 *Directorios de las carpetas de trabajo*

La línea 20 corresponde al directorio principal donde vamos a trabajar y las líneas 22 y 23 indican las carpetas de entrenamiento y validación respectivamente.

Después de indicar la ubicación de las carpetas de trabajo, se prepara el conjunto de datos para el proceso de entrenamiento y validación. Para esta parte, se aplica una *augmentation* al conjunto de imágenes para el entrenamiento, es decir, se ajustan los valores de los píxeles entre 0 y 1, se genera una rotación aleatoria de hasta 7 grados (ayudando a generalizar el modelo al entrenarlo con imágenes ligeramente diferentes) y se aplica una inversión horizontal aleatoria a las imágenes (figura 3.14). Para el conjunto de validación, no se realizará ninguna *augmentation* dado que se busca evaluar el modelo sin modificaciones en los datos.

```
25    #Preparación de los datos (para entrenamiento y validación)
26    #Directorios a los que python va acceder para las imágenes
27    train_datagen = ImageDataGenerator(rescale=1./255, rotation_range=7, horizontal_flip=True)
28    valid_datagen = ImageDataGenerator(rescale=1./255)
```

Figura 3.14 *Preparación de los conjuntos de datos*

Posteriormente se crea, finalmente, los paquetes de imágenes con el método flow_from_directory de los objetos train_datagen y valid_datagen. Este método necesita los directorios de las imágenes. Se indica que las imágenes cargadas desde sus respectivos directorios tengan unas dimensiones de 224x224 (dimensiones que coinciden con el tamaño de entrada de la red), junto con un *batch_size* de 64, indicando el número de imágenes por lote para el entrenamiento y generando con *sparse* números de clases dependientes del directorio. Es decir, cada subdirectorio dentro de los directorios de entrenamiento y validación representará una clase diferente (en total 7 clases), y el generador asignará un número entero a cada clase, en función del orden de los directorios. Véase la figura 3.15

```
30  train_generator = train_datagen.flow_from_directory(dir_train,target_size=(224,224),
31                                                      batch_size=64,
32                                                      class_mode='sparse')
33
34  validation_generator = valid_datagen.flow_from_directory(dir_valid,target_size=(224,224),
35                                                           batch_size=64,
36                                                           class_mode='sparse')
```

Figura 3.15 *Cargar las imágenes de los directorios*

B. Definición del modelo

Para este apartado, será necesaria la utilización de las librerías mostradas en la figura 3.16.

```
2   #Creación del modelo de red neuronal
3   from tensorflow.keras import layers #Posibles capas de una CNN
4   from tensorflow.keras import models #Posibles modelos ML
5   from tensorflow.keras import optimizers #Posibles métodos de optimización
6   from tensorflow.keras.applications import MobileNetV2
7   from tensorflow.keras.layers import Input
8   from tensorflow.keras.layers import MaxPooling2D
9   from tensorflow.keras.layers import Dropout
10  from tensorflow.keras.layers import Flatten
11  from tensorflow.keras.layers import Dense
12  from tensorflow.keras.models import Model
```

Figura 3.16 *Librerías utilizadas para el entrenamiento*

Es necesario obtener la red MobileNetV2 que se va a usar como base del modelo. Como se trata de un modelo preentrenado, ya tiene unos pesos existentes obtenidos a partir del conjunto de imágenes 'imagenet' [45]. Además, se omiten las últimas capas de clasificación del modelo, dejando solo las capas convolucionales, para posteriormente agregar nuestras propias capas de clasificación personalizadas y así adaptar el modelo para una tarea específica.

```
35  mobile=MobileNetV2(weights='imagenet', include_top = False)
```

Figura 3.173 *Preparación para la capa de transferencia*

Más tarde se congelan todas las capas del modelo de MobileNetV2 a excepción de las diez últimas capas, esto significa que solo dichas 10 capas y las capas añadidas posteriormente serán ajustadas durante el entrenamiento.

Después, se procede con la creación de un modelo secuencial vacío llamado 'modeloBase'. Un modelo secuencial [46] es un conjunto de capas apilado en las que cada capa tiene una entrada y una salida. En la línea 42 de la figura 3.18 se indica también el tamaño de la capa de entrada (ancho y alto de 224 píxeles) y sus canales de color, en este caso los tres canales para los colores RGB.

En la siguiente línea se agrega finalmente el modelo preentrenado de MobileNetV2 como la capa inicial del 'modeloBase' y en las líneas 46 a la 49 se añaden las capas adicionales a esta red para adaptar las salidas del modelo a la tarea específica de reconocer piezas LEGO.

Por orden de capas:

Introduciremos una capa MaxPooling2D con un filtro de tamaño de 2 píxeles de ancho y alto; seguida de una capa de aplanamiento para la capa densa de la línea 49. Esta capa dense tiene una salida de 7 neuronas, correspondiente a las siete clases de objetos (Piezas LEGO) con las que se pretende trabajar luego, y usa la función de activación *softmax*.

Para prevenir el sobreajuste, se ha añadido una capa *dropout* en la línea 48, que desactiva aleatoriamente el 30% de las neuronas durante el entrenamiento. Ayudando al modelo a generalizar mejor y evitar que dependa demasiado de combinaciones específicas de neuronas.

```
41    modeloBase = models.Sequential()
42    modeloBase.add(layers.Input(shape=(224,224,3)))
43    modeloBase.add(mobile)
44
45    #Capas a usar en el entrenamiento del modelo
46    modeloBase.add(layers.MaxPooling2D((2, 2)))
47    modeloBase.add(layers.Flatten())
48    modeloBase.add(layers.Dropout(0.3))
49    modeloBase.add(layers.Dense(7,activation='softmax'))
```

Figura 3.18 *Sección de código para definir las capas del modelo*

Una vez definidas las capas del modelo, se realiza su compilación. En esta línea de código se configura el modelo con una función de pérdida, un optimizador y una métrica de evaluación.

Como ya se ha mencionado anteriormente, para la función de pérdida se ha escogido *sparse_categorical_crossentropy*, como optimizador de ha utilizado *Adamax* y la métrica de evaluación de *acc para* monitorear el entrenamiento.

La métrica acc (exactitud) se utiliza para evaluar el porcentaje de predicciones correctas del modelo, pero no influye directamente en el ajuste de los pesos ni en la tasa de aprendizaje.

```
51    #Compilación del modelo
52    modeloBase.compile(loss='sparse_categorical_crossentropy',
53                       optimizer=optimizers.Adamax(),
54                       metrics=['acc'])
```

Figura 3.19 *Compilación del modelo*

C. Entrenamiento del modelo

El entrenamiento del modelo se lleva a cabo con el método *fit_generator* del modelo generado (figura 3.20).

Con el fin de optimizar la red, se han realizado dos modelos con diferentes pasos de época.

Sabiendo la cantidad de imágenes para el entrenamiento (25.256) y para la validación (8.545), junto a sus respectivos tamaños de lote (64), se aplicó la fórmula proporcionada en el apartado 2.4 para calcular los pasos por época y de validación.

$$Steps\ per\ epoch = \frac{25.256}{64} = 394'625 = 394$$

$$validation\ steps = \frac{8.545}{64} = 133'5156 = 133$$

Para el primero se han definido 394 pasos por época para el entrenamiento y 133 para validación, es decir, indicamos cuántas veces el generador producirá datos en cada época. Como se ha explicado anteriormente, una época es un ciclo completo en el que el modelo procesa todo el conjunto de datos una vez, en este caso, se realizarán 60 épocas o ciclos para maximizar el aprendizaje.

Finalmente, el modelo entrenado se guarda con el nombre de 'RedNeuronal' para su posterior uso.

```
56    #Entrenamos el modelo y lo guardamos
57    Salidas = modeloBase.fit_generator(train_generator,steps_per_epoch=394,
58                                       epochs=60,
59                                       validation_data=validation_generator,validation_steps=133,
60                                       verbose=1)
61
62    modeloBase.save('RedNeuronal.h5') #Guardamos la red neuronal entrenada, ahora denominado modelo
```

Figura 3.20 *Entrenamiento del modelo (Modelo1)*

Para el segundo modelo, se probó una red neuronal modificando a la baja los pasos por época del entrenamiento y la validación (en este 59 tanto para el entrenamiento como para la validación), con el fin de comparar ambos modelos.

```
56    #Entrenamos el modelo y lo guardamos
57    Salidas = modeloBase.fit_generator(train_generator,steps_per_epoch=59,
58                                       epochs=60,
59                                       validation_data=validation_generator,validation_steps=59,
60                                       verbose=1)
61
62    modeloBase.save('RedNeuronal.h5') #Guardamos la red neuronal entrenada, ahora denominado modelo
```

Figura 3.21 *Entrenamiento del modelo (Modelo2)*

V. Muestra de resultados del modelo neuronal

Una vez finalizado el proceso de entrenamiento, se obtienen los resultados que se presentan en la Figura 3.22. Estos resultados reflejan el desempeño del modelo utilizado en este trabajo, incluyendo métricas clave que evalúan su precisión, eficacia y capacidad de generalización. Además, junto con los resultados, se muestran los parámetros calculados durante el entrenamiento, los cuales son fundamentales para interpretar y analizar el rendimiento del modelo.

```
Epoch 60/60
394/394 [==============================] - 494s 1s/step - loss: 0.0175 - acc: 0.9941 - val loss: 0.3554 - val acc: 0.9253
```

Figura 3.22 *Resultado final del entrenamiento*

En la figura 3.22 se muestran las épocas correspondientes (60), los pasos por época (394), junto con el tiempo utilizado para el entrenamiento durante esa época, sus respectivos valores de pérdidas (loss), la exactitud obtenida (acc) y los valores obtenidos de la validación de las pérdidas (val_loss) y exactitud (val_acc).

Con los datos obtenidos se realizó una tabla de pérdidas (*loss)* y precisión (*accuracy*) para comparar varios modelos entrenados. Con la precisión se observa el porcentaje de predicciones correctas hechas por el modelo en relación con el total de predicciones; mientras que con las pérdidas se observará el porcentaje de error que comete el modelo en sus predicciones.

Tabla 2. *Modelos de red entrenados*

Modelo	Precisión	Precisión Validación	Pérdidas	Pérdidas validación
Modelo1	0’9941	0’9253	0’0175	0’3554
Modelo2	0’968	0’905	0’032	0’223

Modelo 1

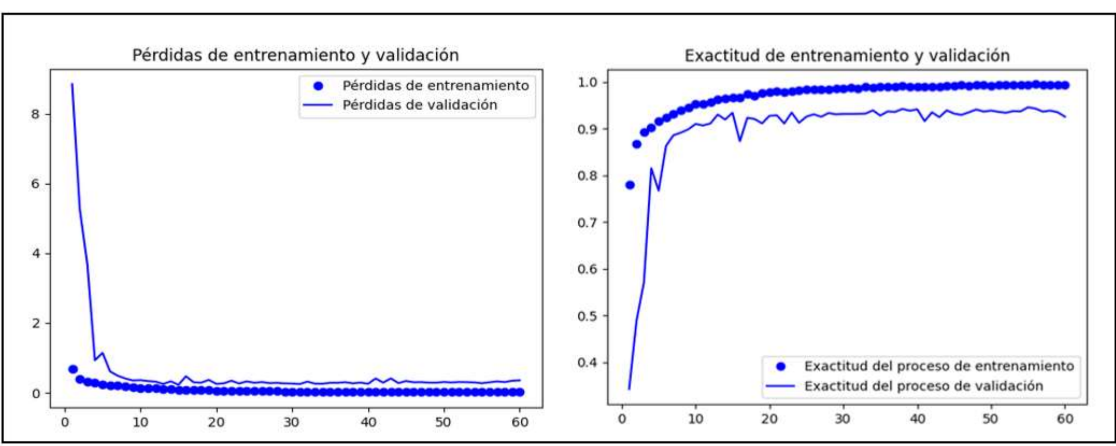


Figura 3.234 *Gráficas de exactitud y pérdidas del modelo 1*

En el Modelo 1, las pérdidas de entrenamiento disminuyen rápidamente y se estabilizan alrededor de 0’0175, mostrando que el modelo está aprendiendo de manera efectiva desde las primeras épocas. De manera similar, las pérdidas de validación también convergen adecuadamente, indicando una buena consistencia entre el entrenamiento y la validación. Por otro lado, la exactitud de entrenamiento alcanza valores cercanos a 1’0, demostrando que el modelo logra capturar casi por completo los patrones presentes en los datos de entrenamiento. La exactitud de validación, aunque ligeramente menor, sigue siendo alta, lo que sugiere que el modelo logra una buena generalización y es capaz de desempeñarse correctamente en datos no vistos.

Modelo 2

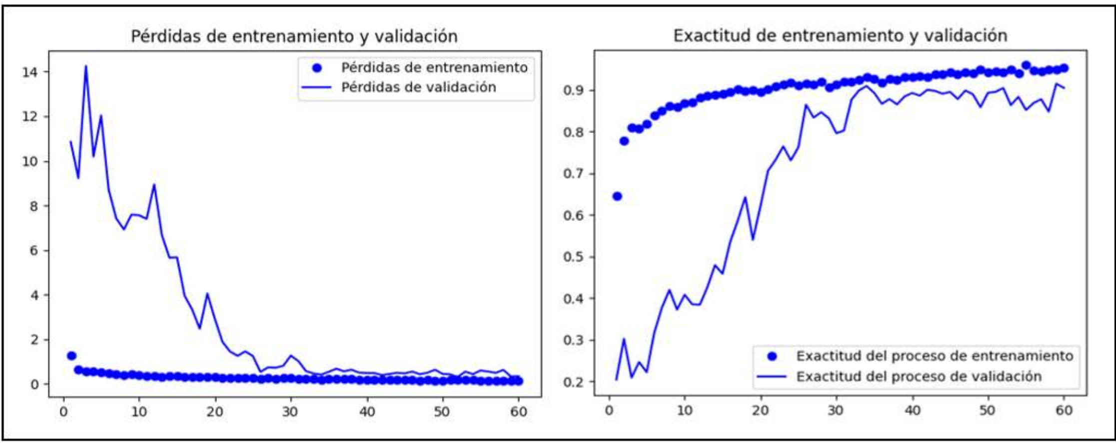


Figura 3.24 *Gráficas de exactitud y pérdidas del modelo 2*

El Modelo 2 muestra un inicio más complicado durante el entrenamiento, ya que las pérdidas comienzan significativamente más altas (14 frente a 8 en el Modelo 1). A pesar de esto, las pérdidas convergen de manera adecuada al final, aunque en el proceso se observan más oscilaciones, especialmente en el conjunto de validación. Este comportamiento podría reflejar que el modelo está ajustándose a patrones más diversos o explorando características más complejas en los datos.

En cuanto a la exactitud, se observa un rápido incremento en el desempeño del modelo durante las primeras épocas, aunque con oscilaciones iniciales más marcadas en comparación con el Modelo 1. La exactitud de validación alcanza valores cercanos a 0’9, lo cual es un excelente resultado, pero este logro ocurre de manera más lenta y con menos estabilidad.

En general, aunque el Modelo 2 es más variable durante el entrenamiento, esta variabilidad parece ser una ventaja, ya que permite que el modelo ajuste mejor a los datos específicos de la tarea. Las oscilaciones observadas podrían ser indicativas de que el modelo está capturando patrones más complejos que lo hacen más efectivo en la práctica, incluso si las métricas estándar sugieren una menor estabilidad en comparación con el Modelo 1.

Comparando las gráficas, se puede comprobar que en el caso del ‘Modelo1’, las pérdidas de validación y la exactitud del proceso de validación evoluciona de una forma más progresiva y sin tantos cambios bruscos en su progresión, mostrando una mayor estabilidad. Aunque el ‘Modelo 2’ alcanza también unos valores similares en cuanto a la precisión final.

VI. Validación con cámara real del modelo neuronal pieza a pieza.

Como se puede observar, el modelo entrenado más preciso es el ‘Modelo1’; sin embargo, se tuvo que comprobar su efectividad en la práctica. Entre el ‘Modelo2’ y ‘Modelo1’ no hubo ninguna diferencia notable a la hora de reconocer las piezas, teniendo una precisión del 85% en ambos casos para la detección individual de las piezas, fallando en la detección de la pieza 1x1 cuadrada.

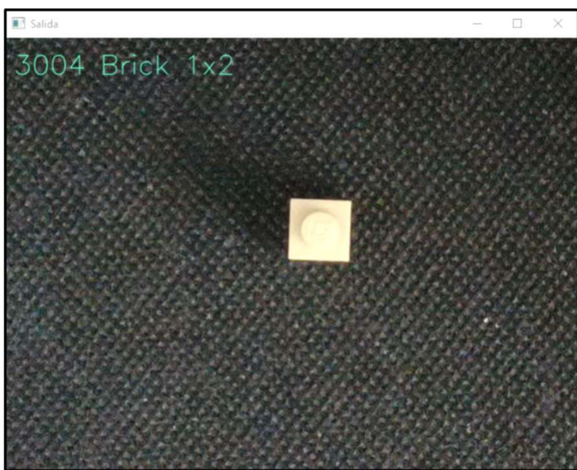


Figura 3.255 *Resultado de la detección de la pieza 1x1, mismo resultado con ambos modelos*

3.2.2. Aplicación de selección de piezas para la construcción de figuras LEGO (programa 2)

VII. Diseño programa aplicación de selección de piezas para la construcción de figuras LEGO

Este programa se ha dividido especialmente en la creación de un menú, la detección de los contornos, la aplicación de la red neuronal.

A. Selección de figura (menú)

En esta parte, para poder escoger entre tres construcciones distintas, se ha utilizado especialmente la librería de *OpenCV2*.

Figura 3.26 *Librería utilizada*

Para el fondo del menú hemos utilizado una imagen plana de color gris, a la cual hemos guardado como ‘Fondo’ para poder usarla en el programa.

Primero hemos abierto esta imagen y la hemos guardado en una variable denominada ‘fondo’ y la hemos abierto en escala de grises, además de realizar posteriormente una copia (‘Imagen_Modificada’) de esa misma imagen para poder utilizarla más adelante.

```
13    #Abrimos imagen que será el fondo del menú
14    fondo=cv2.imread('Fondo.jpg',1) #Abrimos en niveles de gris
15    Imagen_Modificada=fondo.copy()
```

Figura 3.27 *Segmento de código que muestra las imágenes a utilizar*

Al haber tres opciones a escoger, hemos generado tres rectángulos sobre la ‘Imagen_Modificada’, que se puede ver en la imagen 3.28, de la línea 19 hasta la 21. De la línea 24 a la 26 hemos introducido, como texto dentro de estos rectángulos, los nombres de las construcciones: ‘Figura 1’, ‘Figura 2’ y ‘Figura 3’.

```
18    #Añadimos el texto y los recuadros paras las opciones que aparecerán en el menú
19    cv2.rectangle(Imagen_Modificada, (500, 700), (3500,400), (255, 0, 0), 20, cv2.LINE_8)
20    cv2.rectangle(Imagen_Modificada, (500, 1100), (3500,800), (255, 0, 0), 20, cv2.LINE_8)
21    cv2.rectangle(Imagen_Modificada, (500, 1500), (3500,1200), (255, 0, 0), 20, cv2.LINE_8)
22
23    cv2.putText(Imagen_Modificada, 'Seleccione figura a construir:',(150,300), cv2.FONT_ITALIC,7, (255,0,0),10,cv2.LINE_8)
24    cv2.putText(Imagen_Modificada, 'Figura 1',(1550,635), cv2.FONT_ITALIC,8, (255,0,0),10,cv2.LINE_8)
25    cv2.putText(Imagen_Modificada, 'Figura 2',(1550,1035), cv2.FONT_ITALIC,8, (255,0,0),10,cv2.LINE_8)
26    cv2.putText(Imagen_Modificada, 'Figura 3',(1550,1435), cv2.FONT_ITALIC,8, (255,0,0),10,cv2.LINE_8)
```

Figura 3.286 *Código para la creación de los recuadros y textos sobre la Imagen Modificada*

Como se quiere realizar una selección dentro de la ventana del mencionado menú, se requerirá una función que habilite el uso del ratón dentro de esa ventana, además de que, dependiendo de donde que clique, se seleccionará una opción u otra.

Antes de comenzar con esta función, vamos a inicializar las variables que pretendemos usar.

Las variables a inicializar serán:

- Los puntos de coordenadas en el eje x e y de la posición al clicar sobre la pantalla del menú. (p1_x y p1_y)
- Una variable que detecta si se ha presionado el botón del ratón o no, inicialmente puesto como falso, indicando que no se ha presionado el botón izquierdo del ratón. (mouse_pressed)
- Una variable auxiliar que cuenta la cantidad de veces que se ha clicado sobre el menú. (Punto1)
- La lista donde se guardarán los nombres de las piezas necesarias para su construcción.

```
26    #Inicializamos las variables y la lista
27    p1_x=p1_y=0
28    mouse_pressed=False
29    Punto1=0
30    Lista_LEGOS=[]
```

Figura 3.29 *Inicializar las variables para la función del ratón*

Una vez inicializadas las variables, podemos iniciar con la función de selección con el ratón.

Se define la función llamada ‘puntos_imagen’ en la línea 33 del código y se introducen los parámetros de los que va a depender, en este caso será el evento de presionar el botón izquierdo del ratón, las posiciones x e y en las que ocurre el evento (clicar sobre el fondo), el estado de las teclas modificadoras como Ctrl o Shift mientras se hace el clic y un parámetro adicional que no se usa directamente en este caso, pero es necesario indicarlo para la función (param).

Esta función denominada ‘puntos_imagen’ utiliza varias variables globales, indicadas en la línea 34, que han sido definidas fuera de la función. Estas variables son necesarias para controlar el estado de la imagen y realizar cambios en ella cuando se detectan clics del ratón.

Dentro de la función, se detecta si el evento corresponde a un clic izquierdo del ratón en la línea 36. Para esto, se usa la constante cv2.EVENT_LBUTTONDOWN, que indica que el botón izquierdo del ratón ha sido presionado; en el caso de ser cierto, se incrementará el contador ‘Punto1’.

```
33  def puntos_imagen(event, x, y, flags, param):
34      global Imagen_Modificada, Punto1, p1_x, p1_y, mouse_pressed,Lista_LEGOS,img_copy
35
36      if event == cv2.EVENT_LBUTTONDOWN:
37          mouse_pressed = True
38          Punto1+=1
```

Figura 3.307 *Definimos la función del ratón junto a sus variables globales y el evento que la activa*

Este contador volverá a tomar el valor de 0 cuando su valor sea mayor de 0, es decir, cuando escojamos una opción, el contador volverá a un valor nulo y podremos seleccionar otra construcción en ese mismo instante.

```
57          if Punto1>0:
58              Punto1=0
```

Figura 3.31 *Contador*

Después de detectar que el ratón ha sido presionado, el código guarda las coordenadas del clic en las variables ‘p1_x’ y ‘p1_y’. Luego, se comprueba en qué área de la imagen se ha hecho clic. Dependiendo de la zona seleccionada, se muestra una lista diferente de piezas de LEGO.

Las áreas de la imagen están determinadas por los rangos de las coordenadas x e y, y cada área corresponde a una lista distinta de piezas. Si las coordenadas x e y están dentro de un rango específico, se seleccionará la primera, la segunda o la tercera lista de piezas.

```
42          if Punto1==1: #Si se ha pulsado una vez el botón, se guardan los puntos de la posición del ratón
43              p1_x, p1_y = x, y
44
45              #si p1_x y p1_y está dentro de esos tres posibles rangos, seleccionará una de las tres listas de piezas
46              if p1_x<3500 and p1_x>500 and p1_y<700 and p1_y>400:
47                  print('Opcion 1')
48                  Lista_LEGOS=['2357 Brick corner 1x2x2',
49                               '3001 brick 2x4 000L',
50                               '3010 brick 1x4 000L',
51                               '3005 Brick 1x1']
52
53              if p1_x<3500 and p1_x>500 and p1_y<1100 and p1_y>800:
54                  print('Opcion 2')
55                  Lista_LEGOS=['3062 Round Brick 1x1 390L',
56                               '43093 Bush 2M friction - Cross axle 380L',
57                               '3010 brick 1x4 000L','3005 Brick 1x1']
58
59              if p1_x<3500 and p1_x>500 and p1_y<1500 and p1_y>1200:
60                  print('Opcion 3')
61                  Lista_LEGOS=['2357 Brick corner 1x2x2',
62                               '3001 brick 2x4 000L','3010 brick 1x4 000L',
63                               '3004 Brick 1x2']
64
65          if Punto1>0: #Si el contador es mayor a 1, se reinicia el contador
66              Punto1=0
```

Figura 3.32 *Código de las áreas determinadas junto a sus listas de piezas*

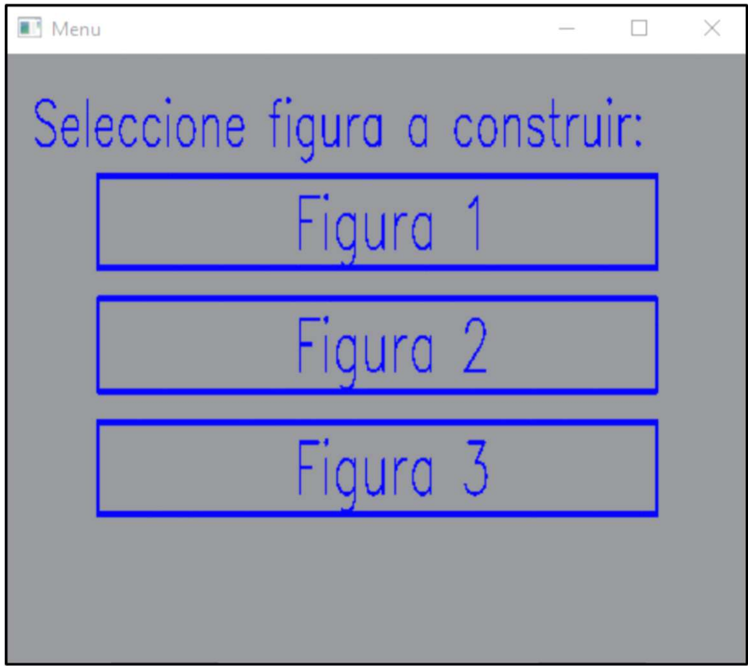


Figura 3.33 *Interfaz del menú*

Una vez seleccionada la figura a construir, sobre el menú (figura 3.33), se generará una ventana con el nombre de 'Piezas requeridas', que mostrará sobre una imagen los nombres de las piezas necesarias para la construcción escogida.

Dentro de cada opción, se introdujo una lista de objetos diferente:

- Figura 1: 2357 Brick corner 1x2x2, 3001 brick 2x4 000L, 3010 brick 1x4 000L y 3005 Brick 1x1.
- Figura 2: 3062 Round Brick 1x1 390L, 43093 Bush 2M friction - Cross axle 380L, 3010 brick 1x4 000L, 3005 Brick 1x1.
- Figura 3: 2357 Brick corner 1x2x2, 3001 brick 2x4 000L, 3010 brick 1x4 000L y 3004 Brick 1x2.

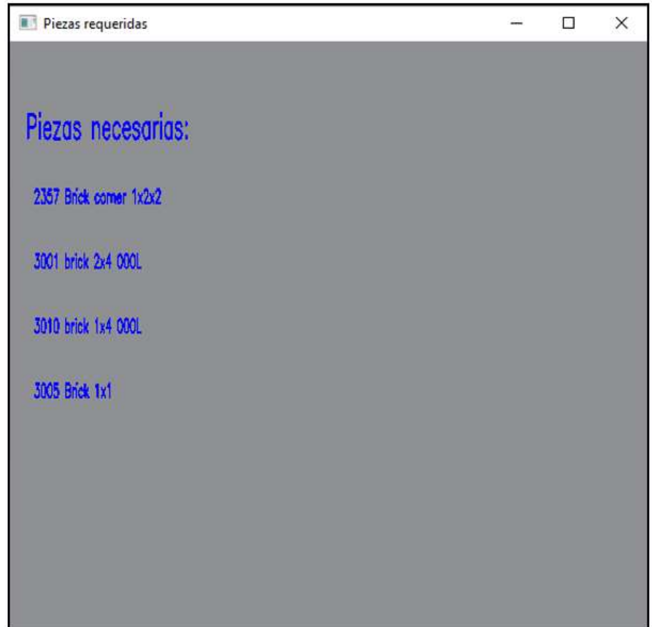


Figura 3.34 *Ejemplo de la lista de piezas para la figura 1 del menú*

Una vez definida la función del ratón, mostramos la venta correspondiente al menú y establecemos dicha función como el *callback* para manejar los eventos del ratón en esa ventana, la cual será llamada cada vez que el ratón interactúe con la ventana del menú.

```
74    #Finalmente, mostramos el menú
75    cv2.namedWindow('Menu', cv2.WINDOW_NORMAL)
76    cv2.setMouseCallback('Menu', puntos_imagen)
77    cv2.imshow('Menu', Imagen_Modificada)
```

Figura 3.35 *Segmento de código de la generación de la ventana para el menú*

B. Conexión de la cámara del móvil con la aplicación DroidCam

Como se ha explicado anteriormente, se pretende reconocer las piezas LEGO en tiempo real a través de la cámara de un teléfono móvil.

Antes de comenzar con el código, hablaremos de cómo conectar nuestro teléfono móvil al ordenador con la aplicación DroidCam.

Una vez descargado en ambos dispositivos, se abrirá la aplicación tanto en el ordenador como en la del teléfono.

Primero se procederá a instalar la aplicación en nuestro teléfono móvil y en nuestro ordenador; se podrá observar en el teléfono la IP del Wifi y el código del puerto; estos códigos deberán colocarse en nuestro ordenador para conectar ambos dispositivos y darle finalmente a *Start*. De esta forma, en nuestro ordenador aparecerá una ventana con el vídeo que se está mostrando a través de la cámara del móvil.

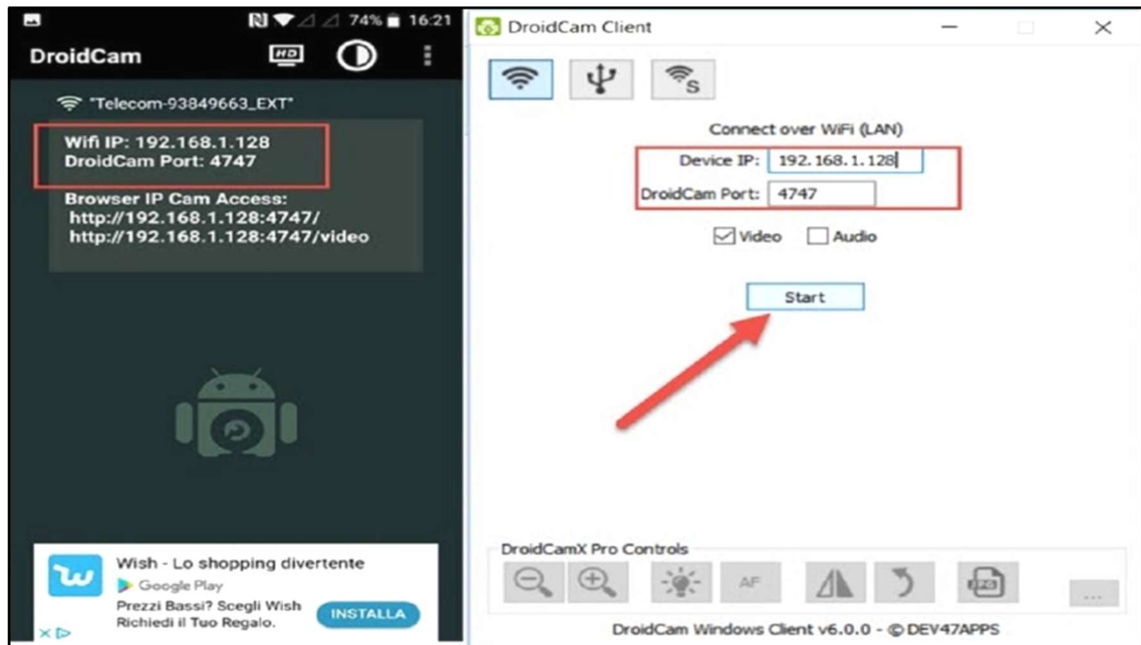


Figura 3.36 *A la izquierda DroidCam en el móvil y a la derecha en el ordenador. Indicando cómo conectarlos entre sí* [47]

Ahora que la cámara del teléfono móvil está conectada al ordenador, se puede proceder con la programación del código para poder identificar las piezas a tiempo real.

C. Detección de contornos

Como el modelo de red neuronal no es capaz de detectar varias piezas en una misma imagen, se debe realizar una separación de estos objetos para poder utilizar el modelo de red entrenado sobre estos objetos de forma individual.

Para ello, es necesario cargar el modelo entrenado e introducir el directorio donde se ha guardado. También es necesario definir el objeto 'cámara' para posteriormente poder activarla y trabajar con ella. De esta forma, se generará un bucle donde la cámara permanecerá abierta hasta que se pulse la tecla 'x' en el teclado. Dentro de dicho bucle, se realizará una captura de imagen de la cámara cada 100 frames y se realizará una detección de contornos que se explicará a continuación.

```
 99  while True:
100      ret,frame= camara.read() #lee la camara
101
102      #Si la cámara funciona y la detecta:
103      if (ret==True):
104          cv2.imshow('Camara', frame) #mostrar el frame en la ventana generada
105          frame_count+=1
106          Key=cv2.waitKey(1) & 0xFF #Tecla 'x' para cerrar el bucle
107
108          #Cada 100 frames, guarda el frame en la variable Imagen para su posterior análisis
109          if frame_count%100==0:
110              Imagen=frame
```

Figura 3.378 *Segmento de código donde se abre la cámara y se hace captura cada 100 frames*

La detección de contornos debe de realizarse sobre una imagen en escala de grises, por lo que se guardará también una copia del frame capturado en escala de grises.

```
110            Imagengris=cv2.cvtColor(Imagen, cv2.COLOR_BGR2GRAY)
```

Figura 3.38 *Código donde se guarda en una variable la imagen en escala de grises*

En esta sección de detección de contornos se han realizado varias pruebas, pues se debe de tener en cuenta el color del fondo, el color de las piezas, la iluminación y el tamaño de la matriz *kernel* a la hora de aplicar el filtro *Gaussian Blur.* Este filtro es aplicado en el procesamiento de imágenes para suavizar la imagen y reducir el ruido. Es importante antes de detectar contornos porque el ruido y los pequeños detalles innecesarios pueden generar contornos no deseados o imprecisos.

En el código de a continuación cabe resaltar que inicialmente se prueba sobre un fondo blanco y piezas LEGO de diversas formas y colores, además, se utilizó sobre la imagen un tamaño de *kernel* o matriz del filtro Gaussiano de tamaño 25 de ancho y de alto junto a una desviación estándar de 25 en la distribución Gaussiana para suavizar más la imagen, ayudando a reducir el ruido y los detalles pequeños.

También se realizó una binarización con umbral Otsu invertido sobre la imagen desenfocada anteriormente. Se combinan dos métodos [48], THRESH_BINARY_INV invierte el resultado (los objetos en blanco y fondo negro) y THRESH_OTSU calcula automáticamente el valor del umbral óptimo para separar los objetos del fondo.

```
7    #Abrimos la imagen
8    Imagen=cv2.imread('ImagenP.jpg',0)
9    Imagencolor=cv2.imread('ImagenP.jpg',3)
10
11   #Aplicamos un filtro
12   Sal_Med1= cv2.GaussianBlur(Imagen,(25,25), 25)
13
14   #Binarizamos
15   Umbral=0
16   Umb1,Binario1=cv2.threshold(Sal_Med1,Umbral,255,type=cv2.THRESH_BINARY_INV+cv2.THRESH_OTSU)
17   cv2.namedWindow('D1',cv2.WINDOW_NORMAL)
18   cv2.imshow('D1', Binario1)
```

Figura 9.39 *Código de prueba número 1 sobre una imagen*

Visualizamos los resultados a partir de una imagen tomada de las piezas sobre un fondo blanco:

Figura 3.40 *Imagen en escala de grises a la izquierda y a la derecha el resultado obtenido*

Como se puede comprobar, las sombras distorsionan los contornos, algunas piezas no aparecen debido a los suaves colores que provocan un bajo contraste con el color blanco del fondo, conllevando a su nula detección.

Dado el resultado, se prueba otra imagen sobre el mismo fondo, pero con una mejor iluminación para evitar la proyección de sombras:

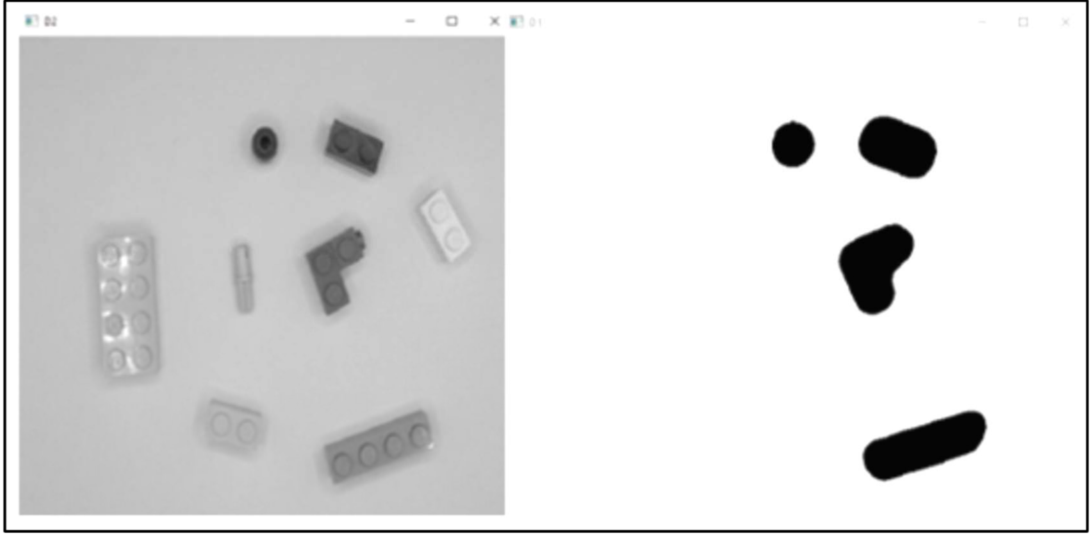


Figura 3.41 *A la izq. imagen en escala de grises sobre fondo blanco. A la dcha. el resultado obtenido*

Con estos resultados se observa que las piezas más claras no son detectadas, por lo que se prueba a continuación con un fondo negro, cambiando también la función THRESH_BINARY_INV por THRESH_BINARY para ver los objetos en blanco y el fondo en negro para una mejor comprobación entre la imagen en canal de grises y su resultado.

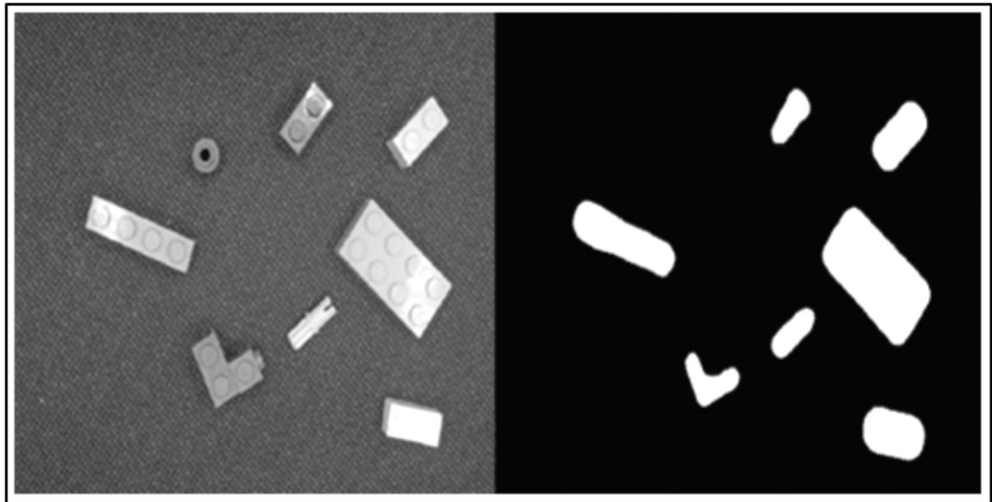


Figura 3.42 *A la izq. imagen en escala de grises sobre fondo negro. A la dcha. el resultado obtenido*

Con este método, al haber muchas piezas de color claro, se detectan con mayor facilidad a excepción de la pieza 1x1 redonda, por lo que se decide usar una de color más claro.

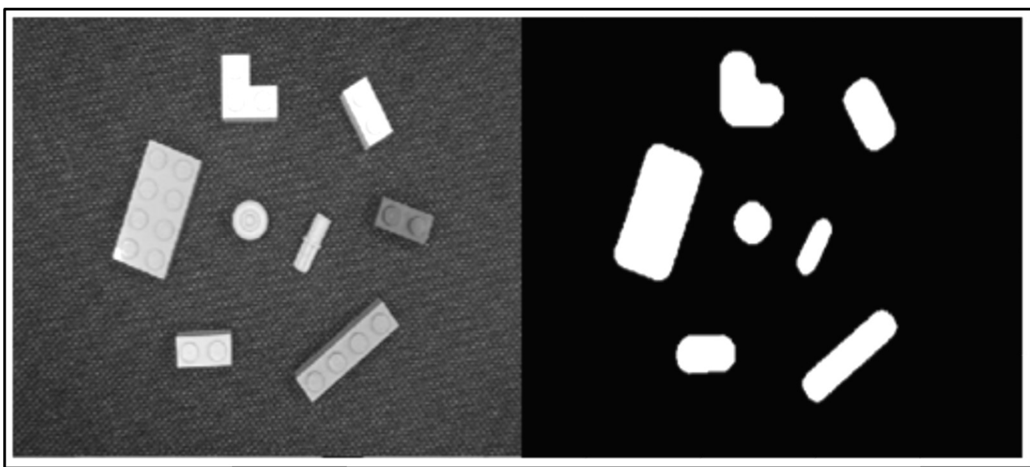


Figura 3.43 *A la izq. imagen en escala de grises con pieza 1x1 redonda de color claro. A la dcha. el resultado obtenido*

Como se mencionó anteriormente, la iluminación juega un papel fundamental, pues la figura 1x2 de color más oscuro anteriormente fue detectada al reflejar el brillo de la iluminación, sin embargo, en este caso no fue detectado debido a que la iluminación resultó más homogénea y no produjo reflejos.

Se decidió utilizar el último método, poner las piezas sobre un fondo oscuro, debido a que se producían menos sombras y a las piezas disponibles, pues la mayoría son de colores suaves y se detectan mejor sobre un fondo oscuro.

Tras obtener unas imágenes con las piezas notoriamente distinguibles, se utilizó la función de encontrar los contornos sobre las imágenes binarizadas en la figura 3.39.

```
#Encontrar contornos
contornos1,jerarquia1=cv2.findContours(Binario1,mode=cv2.RETR_TREE,method=cv2.CHAIN_APPROX_SIMPLE)
```

Figura 3.44 *Segmento de código para la detección de los contornos*

A continuación, se procede con el dibujo de los contornos de las piezas LEGO, para lo que se utiliza el código de la figura 3.40.

```
31    #Recorremos los contornos
32    for i in range(len(contornos1)):
33        Contornos_Externos=cv2.drawContours(ext,contornos1,i,255,1)
34        cont=contornos1[i]
35
36    cv2.namedWindow('Cont_Externos', cv2.WINDOW_NORMAL)
37    cv2.imshow('Cont_Externos', Contornos_Externos)
```

Figura 3.45 *Segmento de código para el dibujo de los contornos*

De la imagen anterior, obtenemos los contornos de cada una de las piezas aplicando la función cv2.findContours que se mencionó en el apartado 2.6. Dibujamos esos contornos utilizando la función cv2.drawContours() (figura 3.40)

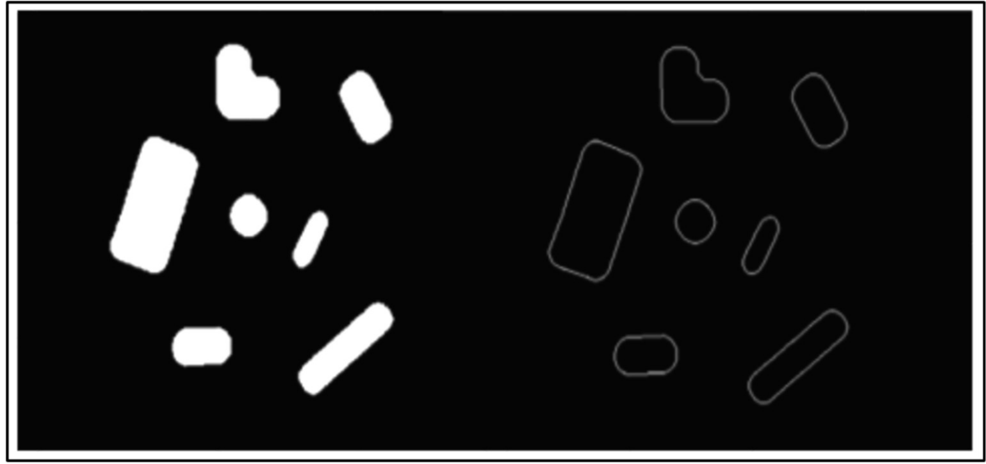


Figura 3.46 *A la izq. Las figuras detectadas. A la dcha. sus respectivos contornos*

Observando los contornos, las esquinas están redondeadas, esto provoca que se pueda perder información, especialmente entre la pieza 1x1 redonda y la pieza 1x1 cuadrada, junto a la pieza cross angle y el 1x2. Para evitar obtener las piezas con los bordes redondeados, cambiamos el tamaño de *kernel* del filtro aplicado por unas dimensiones más pequeñas, en este caso 3 píxeles de ancho y de alto.

```
11    #Aplicamos un filtro
12    Sal_Med1= cv2.GaussianBlur(Imagen,(3,3), 25)
```

Figura 3.47 *Nuevo tamaño para el kernel*

De esta forma, se ha obtenido unos contornos más fieles a la imagen original.

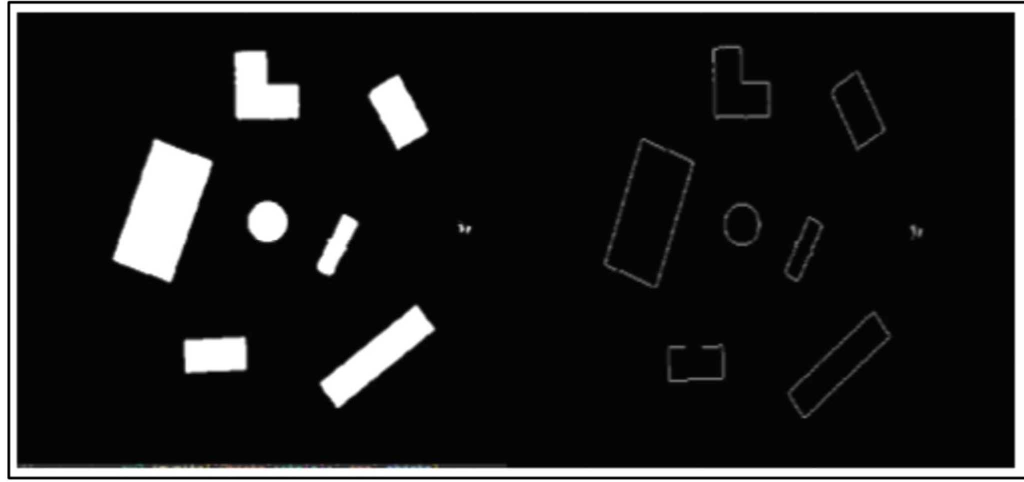


Figura 3.48 *A la izq. figura detectas con el nuevo tamaño de kernel. A la dcha. sus respectivos contornos*

Una vez obtenidos el filtro adecuado y haber decidido el color del fondo sobre el que trabajar, se genera una lista vacía sobre la que se guardarán las imágenes recortadas de las piezas detectadas. Puesto que se ha explicado anteriormente que la red neuronal entrenada no puede trabajar sobre una imagen con todas las piezas a la vez.

También se añadió un límite de área para evitar la detección de objetos demasiado pequeños, ya sean irregularidades de la textura de la tela del fondo, manchas o sombras indeseadas.

```
131         Lista_OBJETOS=[] #Lista de los objetos encontrados a partir de la detección de contornos
132         Lista_COORDENADAS_PIEZAS=[]
133 
134         #Recorremos los contornos
135         for i in range(len(contornos1)):
136             Contornos_Externos=cv2.drawContours(frame,contornos1,i,255,1)
137             cont=contornos1[i]
138 
139             #Añadimos un límite de área
140             if cv2.contourArea(cont)>1000:
141                 (x,y,w,h)=cv2.boundingRect(cont)
142                 Lista_COORDENADAS_PIEZAS.append((x, y, w, h))
143                 #Recortamos cada contorno detectado
144                 objeto = Imagencolor[max(0, y-40):min(y+h+30, 480), max(0, x-40):min(x+w+30, 640)]
145                 #Guardamos en la lista la imagen de cada contorno recortado
146                 Lista_OBJETOS.append(objeto)
147                 c=c+1
```

Figura 3.49 *Segmento de código explicado*

Ahora es el momento de recorrer esa lista de objetos detectados para introducirlos en la red neuronal para su reconocimiento. Como las imágenes guardadas en la lista 'Lista_OBJETOS' no tienen el tamaño adecuado para ser introducidas en la red neuronal (cuyo tamaño de entrada es de 224 x 224 píxeles), se genera una imagen vacía de las dimensiones de entrada de la red neuronal. Se calcula el centro de cada una de las imágenes de la lista, realizando una redimensión en el caso de sobrepasar el tamaño de la entrada en alguna de sus dimensiones, y se coloca en el centro de la imagen vacía para poder proceder con el reconocimiento individual de las piezas (figura 3.50).

```
for objeto in Lista_OBJETOS:
    #Obtenemos el tamaño de cada imagen recortada
    h, w = objeto.shape[:2]

    ##IMPORTANTE:
    # Si las dimensiones de objeto son mayores a 224x224, redimensionamos proporcionalmente
    if h > 224 or w > 224:
        factor_escala = min(224/h, 224/w) #Factor de reducción que se aplicará para ajustar el tamaño del objeto.
        w_nueva = int(w * factor_escala)
        h_nueva = int(h * factor_escala)
        objeto = cv2.resize(objeto, (w_nueva, h_nueva))

        h, w = objeto.shape[:2] #Actualizamos el tamaño de la imagen redimensionada

    #Calculamos las coordenadas para centrar la imagen recortada en la imagen de 224x224
    x_offset = (224 - w) // 2 #// 2 divide el espacio libre entre ambos lados (izquierda y derecha)
    y_offset = (224 - h) // 2 #// 2 divide el espacio libre entre ambos lados (izquierda y derecha)

    #Copiamos la imagen recortada en el centro de la imagen de fondo de 224x224
    img_centrada = img_background.copy()  # Copia de la imagen de fondo
    img_centrada[y_offset:y_offset + h, x_offset:x_offset + w] = objeto
```

Figura 3.50 *Proceso de redimensión y centralización de las imágenes recortadas sobre la imagen vacía*

Con el tamaño adecuado de las imágenes, se clasifican las piezas utilizando el modelo. Cada objeto se redimensiona y se clasifica utilizando el modelo de aprendizaje automático cargado previamente. Para ello se utiliza el método *predict* del modelo.

```
183            img_tensor = image.img_to_array(img_centrada)
184            img_tensor = np.expand_dims(img_tensor, axis=0) #Adaptar dimensiones
185            img_tensor /= 255. #Normalizar
186            predicciones=model.predict(img_tensor,batch_size=1,steps=1,verbose=0)[0]
```

Figura 3.51 *Se clasifican las piezas*

Con ‘predicciones’, se obtiene un vector de siete componentes, tantas como clases de clasificación y, por tanto, como neuronas de la capa de salida de la red neuronal entrenada. Cada valor representa la probabilidad de pertenecer a cada una de las siete clases posibles. Para identificar la clase más probable, se selecciona el elemento con el valor más alto en el vector, ya que este valor representa la clase con la mayor probabilidad de coincidencia. En otras palabras, la posición del valor más alto en el vector indica la clase más probable para el objeto identificado.

Para ello, se obtiene la posición con mayor porcentaje de probabilidad.

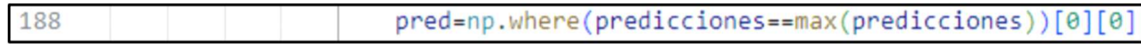


Figura 3.52 *Se obtiene la posición dentro del vector que tiene mayor probabilidad*

Y se asocia la posición del elemento dentro del vector a cada clase, es decir, para saber a qué clase corresponde cada elemento del vector, se debe observar en qué orden se encuentran las carpetas de las imágenes (no importa si se mira en la carpeta de validación o entrenamiento, puesto que tienen el mismo nombre).

Nombre	Fecha de modificación	Tipo	Tamaño
2357 Brick corner 1x2x2	26/10/2024 13:38	Carpeta de archivos	
3001 brick 2x4 000L	26/10/2024 13:38	Carpeta de archivos	
3004 Brick 1x2	18/03/2024 11:29	Carpeta de archivos	
3005 Brick 1x1	24/04/2024 14:40	Carpeta de archivos	
3010 brick 1x4 000L	09/09/2024 20:32	Carpeta de archivos	
3062 Round Brick 1x1 390L	24/04/2024 15:11	Carpeta de archivos	
43093 Bush 2M friction - Cross axle 380L	24/04/2024 15:14	Carpeta de archivos	

Figura 3.53 *Orden de las carpetas de las imágenes*

```
191                 if np.max(predicciones)<0.70:
192                     Titulo='Objeto no identificado'
193                     Lista_PIEZAS.append(Titulo)
194 
195                 elif pred==0:
196                     Titulo='2357 Brick corner 1x2x2'
197                     Lista_PIEZAS.append(Titulo)
198 
199                 elif pred==1:
200                     Titulo='3001 brick 2x4 000L'
201                     Lista_PIEZAS.append(Titulo)
202 
203                 elif pred==2:
204                     Titulo='3004 Brick 1x2'
205                     Lista_PIEZAS.append(Titulo)
206 
207                 elif pred==3:
208                     Titulo='3005 Brick 1x1'
209                     Lista_PIEZAS.append(Titulo)
210 
211                 elif pred==4:
212                     Titulo='3010 brick 1x4 000L'
213                     Lista_PIEZAS.append(Titulo)
214 
215                 elif pred==5:
216                     Titulo='3062 Round Brick 1x1 390L'
217                     Lista_PIEZAS.append(Titulo)
218 
219                 elif pred==6:
220                     Titulo='43093 Bush 2M friction - Cross axle 380L'
221                     Lista_PIEZAS.append(Titulo)
```

Figura 3.54 *Asociación de las posiciones del vector con su respectiva clase*

Se ha utilizado un umbral para la predicción del 70%. Es decir, si la pieza analizada no presenta ningún valor superior del 0,7 en los siete elementos del vector, se considerará que el objeto no ha sido reconocido correctamente.

Los resultados se guardarán en una lista para posteriormente mostrar sobre la imagen de las piezas el nombre de la clase a la que pertenecen y, si coinciden con la lista de piezas escogidas

para la construcción a realizar, se redondeará dicha pieza, mostrando así la ubicación de esta.

```
# Finalmente, vamos a dibujar los resultados (rectángulos y texto) sobre la imagen original
for i, (x, y, w, h) in enumerate(Lista_COORDENADAS_PIEZAS):
    if i<len(Lista_PIEZAS):
        Titulo= Lista_PIEZAS[i]

        # Colocar el texto sobre la pieza
        cv2.putText(frame,str(Titulo),(x-40,y-40),cv2.FONT_HERSHEY_SIMPLEX,0.3,(255,255,255),1,cv2.LINE_AA)

        # Dibujar el rectángulo alrededor de la pieza
        if Titulo in Lista_LEGOS:
            cv2.rectangle(frame, (x, y), (x + w, y + h), (0, 255, 0), 2)
```

Figura 3.55 *Segmento de código para señalizar la pieza y su clase*

También se crean dos ventanas nuevas, una para mostrar el resultado final y otra que muestra la lista de piezas necesarias y a su derecha las piezas encontradas.

```
248            cv2.namedWindow('Nombres piezas', cv2.WINDOW_NORMAL) #generar una ventana
249            cv2.imshow('Nombres piezas', frame)
250
251        for Titulo in Lista_PIEZAS:
252            if Titulo in Lista_LEGOS:
253                cv2.putText(img_copy2, 'Piezas encontradas:',(2050,300), cv2.FONT_ITALIC,3.5, (255,0,0),10,cv2.LINE_8)
254                cv2.putText(img_copy2, str(Titulo),(2050,500+m), cv2.FONT_ITALIC,2, (255,0,0),8,cv2.LINE_8)
255                m=m+200
256                cv2.namedWindow('Piezas encontradas',cv2.WINDOW_NORMAL)
257                cv2.imshow('Piezas encontradas', img_copy2)
258                print('Figura encontrada')
259                print(Titulo)
```

Figura 3.56 *Código de las dos ventanas generadas para los resultados*

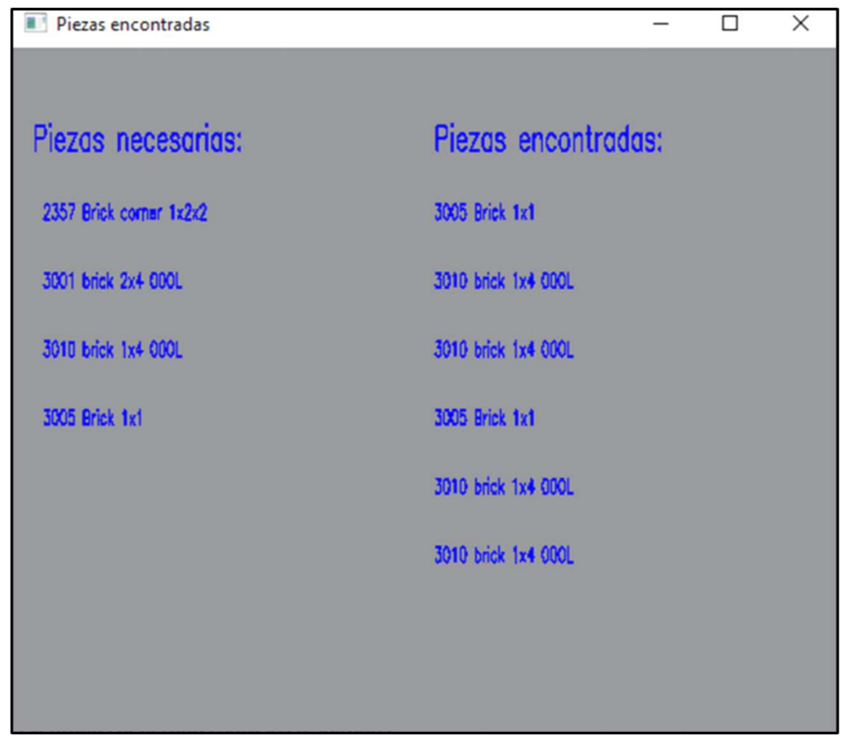


Figura 3.57 Ventana que muestra la lista de piezas necesarias (izq.) y las piezas encontradas (dcha.)

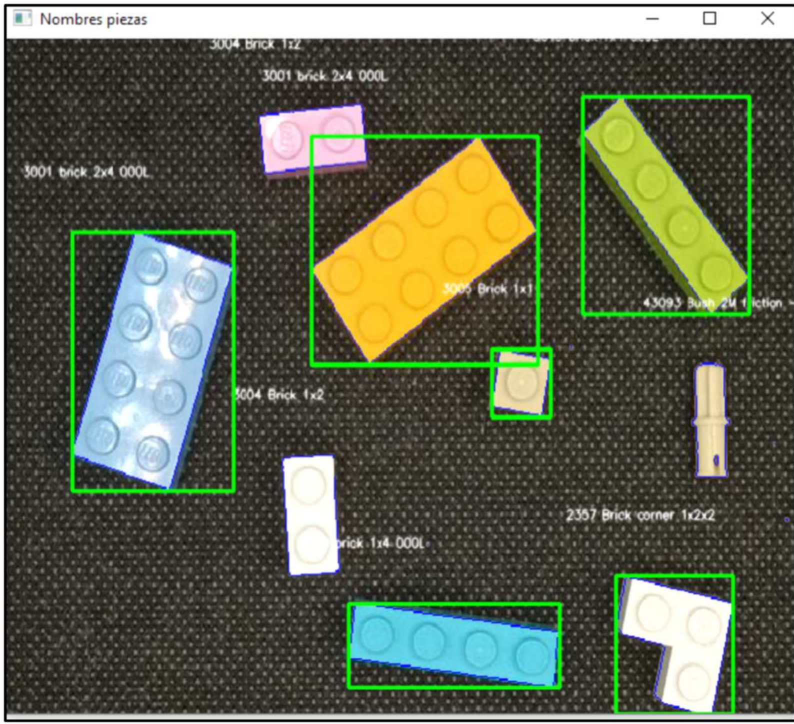


Figura 3.58 Ventana que muestra el nombre de cada pieza detectada y donde se indican las ubicaciones de las piezas necesarias para la construcción

Además, para las piezas no encontradas, se indicarán las piezas que faltan (ejemplo figura 3.60) gracias al código proporcionado en la figura 3.59.

```
264         diferencias1 = set(Lista_LEGOS) - set(Lista_PIEZAS)
265         print(f"Elementos que faltan: {diferencias1}")
```

Figura 3.59 *Código de la indicación de piezas restantes*

```
Elementos que faltan: {'2357 Brick corner 1x2x2', '3001 brick 2x4 000L', '3010 brick 1x4 000L', '3005 Brick 1x1'}
```

Figura 3.60 *Ejemplo de la señalización de piezas restantes*

Para finalizar, indicamos la manera para concluir con el programa, liberando la cámara y cerrando las ventanas creadas. En este caso será presionando la tecla ‘x’, como se mencionó al principio de este apartado.

```
267         if Key==ord('x'):
268             camara.release()
269             cv2.destroyAllWindows()
270             break
```

Figura 3.61 *Código para cerrar el programa*

4. RESULTADOS Y PROBLEMAS

En este apartado se mostrarán los resultados obtenidos en este proyecto, así como las conclusiones y problemas encontrados durante su realización.

4.1 Resultados

Modelo aplicado con detección de contornos en tiempo real

En este apartado se muestra el funcionamiento de la aplicación utilizando los dos modelos entrenados para la predicción de la clase de las diferentes piezas

Para la aplicación del modelo en la red neuronal, se utilizaron los dos modelos entrenados para comparar sus resultados en la detección de piezas de LEGO. Esto permitió analizar y diferenciar el rendimiento de ambos modelos, evaluando su precisión y capacidad para clasificar correctamente las distintas piezas.

Se obtienen estos resultados de un mismo *frame* con los diferentes modelos con la opción 1 escogida en el menú del programa de aplicación de selección de piezas:

<u>Modelo 1</u>

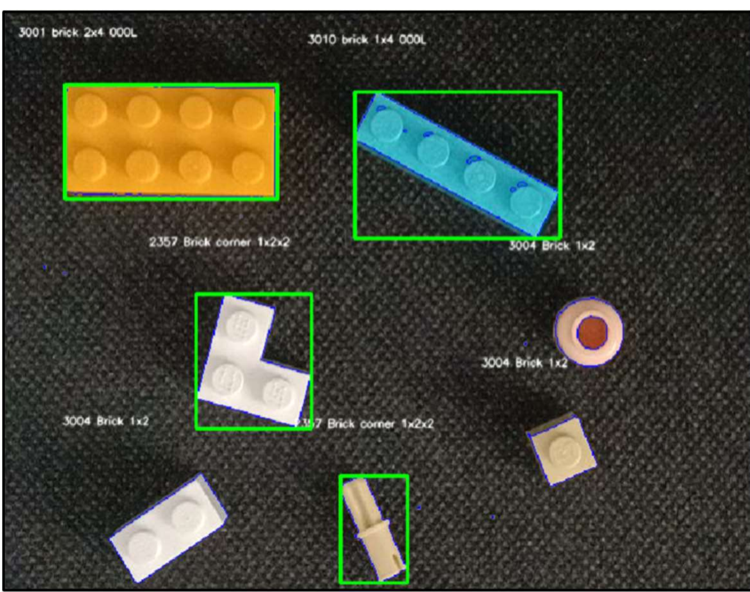


Figura 4.1 *Resultados del programa aplicación de selección de piezas con el Modelo1*

<u>Modelo 2</u>

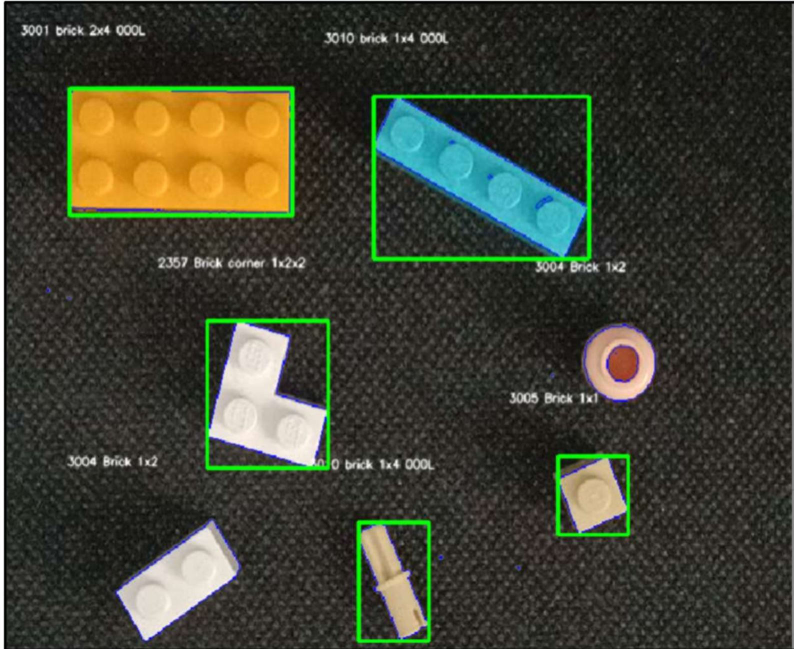


Figura 4.2 *Resultados del programa* aplicación de selección de piezas *con el Modelo2*

Se puede observar, que pese a tener una mejor precisión sobre los gráficos y una menor pérdida, el 'Modelo1' identifica de forma errónea la pieza 1x1 a diferencia del 'Modelo2'.

También, como se puede apreciar, las piezas 3062 Round Brick 1x1 390L y 43093 Bush 2M friction - Cross axle 380L no han sido reconocidas correctamente en ninguno de los dos casos. Sin embargo, de forma individual sí fueron reconocidas.

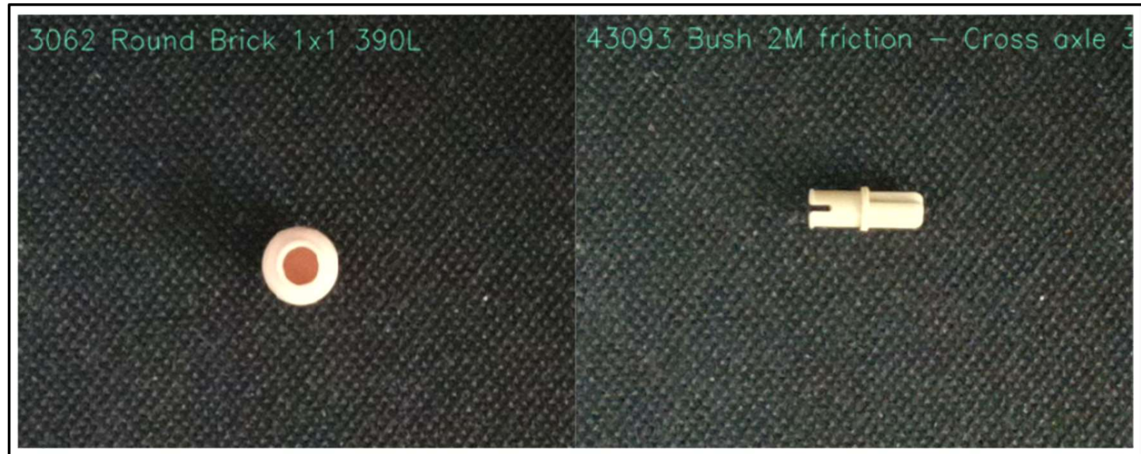


Figura 4.3 *Piezas correctamente reconocidas de forma individual*

En base a estos resultados, el 'Modelo2' ha resultado ser el más preciso pese a tener unas métricas peores en comparación al 'Modelo1', por lo que se ha terminado por escoger este modelo para este trabajo.

4.2 Problemas

En el proceso de desarrollo de la aplicación de reconocimiento de piezas LEGO se encontraron algunos problemas que se explicarán a continuación.

En el proceso de entrenamiento

Al inicio del proyecto, fue necesario definir las perspectivas desde las cuales se capturarían las imágenes de las piezas. Tal como se mencionó en el apartado de metodología, en la sección sobre selección de piezas, la variedad de imágenes utilizadas durante el entrenamiento y la validación puede aumentar la confusión del modelo al identificar correctamente cada clase. Debido a la gran cantidad de imágenes que se revisaron, se optó por capturar las piezas desde una perspectiva superior. Esta vista desde arriba ayuda a evitar que el modelo confunda las piezas debido a variaciones en las perspectivas, mejorando así la precisión en la clasificación.

Recorte de las imágenes previa al proceso de reconocimiento

Durante el proceso de detección de contornos y generación de recortes de las piezas surgieron tres problemas. El primero estaba relacionado con la iluminación de las piezas. El segundo consistía en dificultades en el modelo para identificar correctamente la pieza, debido al área del recorte. El tercer problema causaba el cierre inesperado del programa cuando el área del recorte excedía los límites de la ventana de visualización de la pieza.

- En el primer problema se encontró con la identificación de las sombras en conjunto a las figuras, impidiendo su correcto reconocimiento. Para resolverlo se ha utilizado una luz suave que no genera sombras sobre las piezas.
- Para el segundo problema se realizaron varias pruebas con el área más idónea para los recortes.

Inicialmente se probó realizando un recorte sin área, solamente de la pieza y se procedió con la identificación de las piezas:

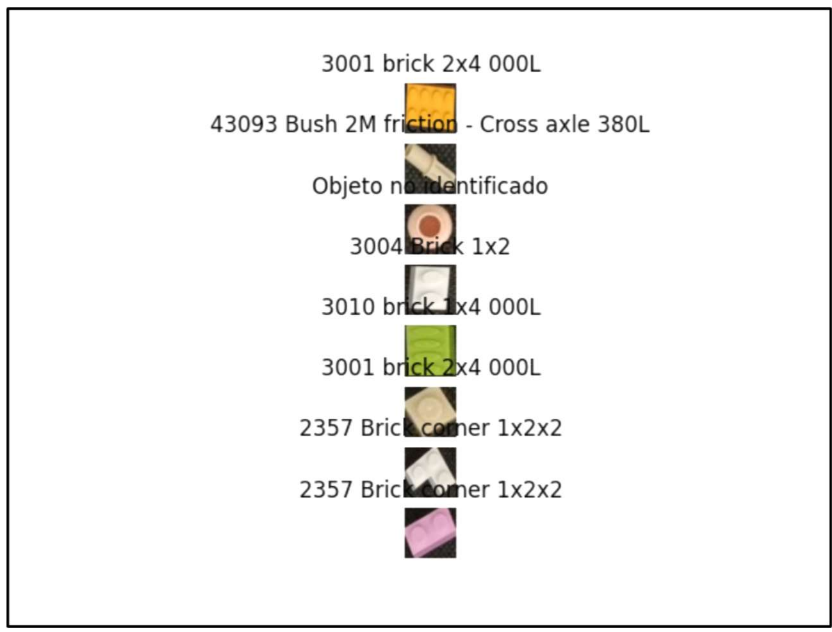


Figura 4.4 *Recortes sin área de las imágenes junto a sus resultados*

Finalmente se probó con otra distancia y se descubrió que se detectaban mejor las piezas cuando existía un área de mayor tamaño que el recorte de las piezas.

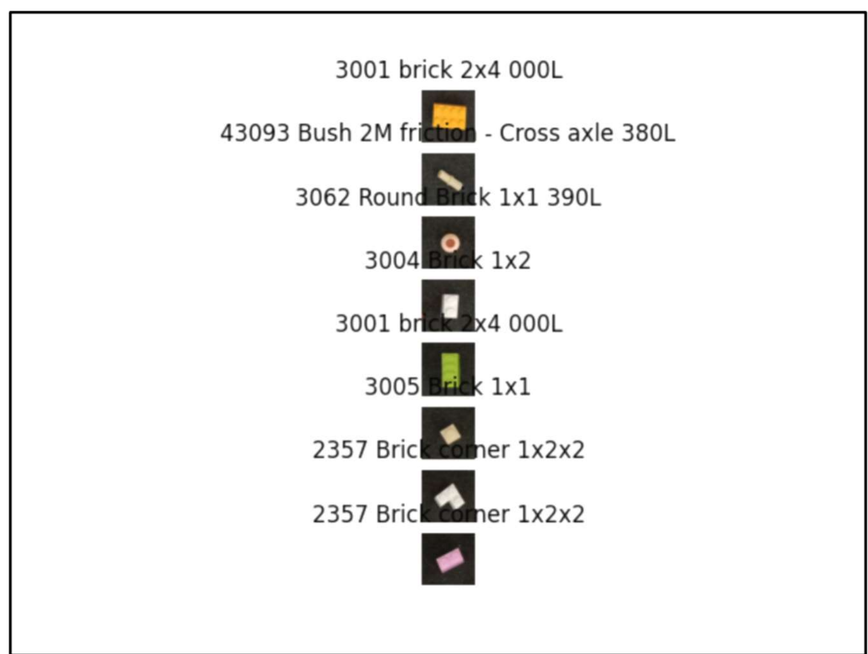


Figura 4.5 *Recortes con área de las imágenes junto a sus resultados*

Se concluyó que, para un mejor resultado, se recortarían las imágenes con una pequeña área alrededor.

- En el tercer problema, se decidió realizar una limitación de las áreas:

```
objeto = Imagencolor[max(0, y-40):min(y+h+30, 480), max(0, x-40):min(x+w+30, 640)]
```

Figura 4.610 *Límites de área para los recortes*

Con *max(0, y-40)* y *max(0, x-40)* se evita que el recorte intente usar coordenadas negativas, lo cual podría dar lugar a un error.

Lo mismo sucede con *min(y+h+30, 480)* y *min(x+w+30, 640)* evitan que el recorte sobrepase los límites de la imagen, pues las dimensiones de imagen obtenidas con DroidCam tienen dimensiones de 480 píxeles de alto y 640 píxeles de ancho.

Redimensionado de las imágenes para su adaptación al tamaño de entrada de la red neuronal

A la hora de aplicar el modelo entrenado en los recortes proporcionados de las piezas durante el programa de identificación, es necesario escalar cada uno de ellos para que su tamaño coincidiera con el esperado por la red neuronal. Dependiendo de cómo se realizará esa adaptación de tamaño se introducía distorsión en las imágenes dificultando su reconocimiento.

Inicialmente se aplicó directamente un escalado (función cv2.resize) sobre las imágenes en vez de colocar las imágenes sobre una imagen vacía de dimensiones 224 x 224.

Se obtuvieron resultados donde las piezas 3010 brick 1x4 000L eran siempre reconocidas como 3001 brick 2x4 000L y las piezas 3004 Brick 1x2 eran reconocidas como 2357 Brick corner 1x2x2.

Esto hacía que, a diferencia del programa utilizado para detectar las piezas de manera individual, dejaran de detectar de forma correcta.

Se llegó a la conclusión de que el *resize* utilizado para ambos programas no se aplicaba de igual manera, ya que en el caso de la presencia de varias piezas los recortes eran más pequeños y con el *resize*, esta se deformaba y podía confundirse con otra pieza similar. Se muestra en la figura 4.7 un ejemplo en el caso de la pieza 3010 brick 1x4 000L:

Figura 4.711 *A la izq. pieza 1x4 con sus dimensiones originales. A la dcha, pieza con el resize aplicado*

Como se puede ver en las imágenes, en el caso de esta pieza, la deformación ocasionada por esta operación puede llevar a un error de detección en la red por otra figura de dimensiones similares, como es el caso del 3001 brick 2x4 000L.

Para solucionar el problema, se decidió eliminar el redimensionamiento (resize) de las imágenes. En su lugar, las imágenes se colocaron en su tamaño original sobre un lienzo o fondo de dimensiones adecuadas. De esta manera, las imágenes se ajustan al formato necesario para ser introducidas en el modelo entrenado, sin distorsionar ni modificar sus proporciones.

5. CONCLUSIONES

El objetivo de este trabajo consistió en desarrollar una aplicación capaz de localizar las piezas LEGO necesarias para el montaje de una figura seleccionada previamente a partir de la identificación de las piezas deseadas en una imagen en tiempo real de un conjunto de ellas.

Para lograr este objetivo se diseñó una red neuronal convolucional junto con un programa de detección de piezas donde se utilizó la red entrenada.

Se han obtenido dos modelos utilizando la misma estructura de red neuronal, pero con diferente valor del hiperparámetro pasos por época y pasos de validación durante su entrenamiento. Como podemos comprobar en los resultados, a la hora de probar sus prestaciones, que ambas fueron bastante similares en cuanto a los valores de error y precisión tras el proceso de entrenamiento y para la identificación de piezas individuales; sin embargo, el Modelo2 entrenado destacó sobre el Modelo1 cuando se probaron con el programa de aplicación de identificación de varias piezas a la vez. Este resultado podría implicar que el Modelo1 resultó en una mala generalización de patrones de las piezas LEGO pese a que fue entrenado con más imágenes que el Modelo2.

Atendiendo al objetivo principal del trabajo, se puede concluir que el modelo entrenado ha cumplido con el objetivo esperado; sin embargo, es necesario realizar algunas matizaciones.

La primera cuestión a tener en cuenta es la relativa al efecto del tamaño de las imágenes de cada pieza en el proceso de identificación. Para las piezas más pequeñas, la detección ha sido mejor cuando se ha ampliado el *zoom* sobre la pieza, siendo indiferente para las más grandes, lo que explicaría que la detección fuera cercana al 100% cuando se realizaba la identificación de las piezas de forma individual.

La segunda cuestión está relacionada con el tamaño de entrada de la red neuronal a la hora de introducir una pieza a reconocer. El tamaño de la entrada del modelo es de 224 píxeles de ancho y de alto. Tal y como se ha explicado en la parte de resultados, la red ha proporcionado mejores resultados cuando se ha eliminado el *resize* y se han colocado con su tamaño original sobre una imagen vacía de dimensiones 224 x 224. Este puede ser uno de los motivos por los que se ha perdido precisión a la hora de reconocer las piezas más pequeñas.

Como perspectiva futura del trabajo, las propuestas serían:

- Calcular de forma automática con la librería Optuna los hiperparámetros para entrenar una red neuronal mucho más precisa.
- Utilizar el sistema de código abierto YOLO (You Only Look Once) para la detección de objetos en imágenes e implementarlo a tiempo real.

6. REFERENCIAS BIBLIOGRÁFICAS Y WEBGRAFÍA

[1] McCullough, W.S, & Pitts, W. (1943). A logical calculus of the ideas immanent in neurons activity. Bull. *Math. Biophys., 5,*115-133.

[2] Rosenblatt, F. (1958). The perceptron: a probalistic model for information storage and organization in the brain. *Psychological review, 65*(6), 386.

[3] Wikimedia Foundation. (2005, August 30). *Perceptrón.* Wikipedia. https://es.wikipedia.org/wiki/Perceptr%C3%B3n

[4] Planche, B., & Andres, E. (2019). *Hands-on Computer Vision with tensorflow 2: Leverage deep learning to create powerful image processing apps with tensorflow 2. 0 and keras (pag. 28)*. Packt Publishing, Limited.

[5] EITCA, A. (2023, August 8). *¿En qué se diferencia una red neuronal convolucional 3d de una red 2D en términos de dimensiones y pasos? - academia eitca*. EITCA Academy. https://es.eitca.org/artificial-intelligence/eitc-ai-dltf-deep-learning-with-tensorflow/3d-convolutional-neural-network-with-kaggle-lung-cancer-detection-competiton/running-the-network-3d-convolutional-neural-network-with-kaggle-lung-cancer-detection-competiton/examination-review-running-the-network-3d-convolutional-neural-network-with-kaggle-lung-cancer-detection-competiton/how-does-a-3d-convolutional-neural-network-differ-from-a-2d-network-in-terms-of-dimensions-and-strides/

[6] Planche, B., & Andres, E. (2019). *Hands-on Computer Vision with tensorflow 2: Leverage deep learning to create powerful image processing apps with tensorflow 2. 0 and keras (pag. 77)*. Packt Publishing, Limited.

[7] Planche, B., & Andres, E. (2019). *Hands-on Computer Vision with tensorflow 2: Leverage deep learning to create powerful image processing apps with tensorflow 2. 0 and keras (pag. 79)*. Packt Publishing, Limited.

[8] Brownlee, J. (2019, July 5). *A gentle introduction to pooling layers for Convolutional Neural Networks*. MachineLearningMastery.com. https://machinelearningmastery.com/pooling-layers-for-convolutional-neural-networks/

[9] Planche, B., & Andres, E. (2019). *Hands-on Computer Vision with tensorflow 2: Leverage deep learning to create powerful image processing apps with tensorflow 2. 0 and keras (pag. 87)*. Packt Publishing, Limited.

[10] GeeksforGeeks. (2024, September 4). *CNN: Introduction to pooling layer*. https://www.geeksforgeeks.org/cnn-introduction-to-pooling-layer/

[11] DhanushKumar. (2023, November 29). *Max pooling*. Medium. https://medium.com/@danushidk507/max-pooling-ef545993b6e4

[12] GeeksforGeeks. (2024a, February 13). *What is a neural network flatten layer?* https://www.geeksforgeeks.org/what-is-a-neural-network-flatten-layer/

[13] Verma, Y. (2024, August 1). *What is dense layer in neural network?* Analytics India Magazine. https://analyticsindiamag.com/topics/what-is-dense-layer-in-neural-network/

[14] Team, K. (n.d.). *Keras Documentation: Activation Layers*. https://keras.io/api/layers/activation_layers/

[15] Team, K. (n.d.). *Keras documentation: Dropout layer*. https://keras.io/api/layers/regularization_layers/dropout/

[16] Yadav, H. (2023, May 31). *Dropout in neural networks*. Medium. https://towardsdatascience.com/dropout-in-neural-networks-47a162d621d9

[17] Valenzuela González, G. (2022). *Aprendizaje Supervisado: Métodos, Propiedades y Aplicaciones.*

[18]López Torres, A. M. (2023). *Apuntes asignatura Visión por computador.* Universidad de Zaragoza

[19]Goodfellow, I., Bengio, Y., & Courville, A. (2016). *Deep Learning (pag.109)*. MIT Press.

[20] Gillis, A. S. (2024, July 29). *What is a validation set? how do they compare to test, train data sets?* WhatIs. https://www.techtarget.com/whatis/definition/validation-set

[21] *Tf.keras.losses.SparseCategoricalCrossentropy : tensorflow V2.16.1*. TensorFlow. (n.d.). https://www.tensorflow.org/api_docs/python/tf/keras/losses/SparseCategoricalCrossentropy

[22] Astitwa. (2024, May 26). *Comparative analysis of adam optimizer and its variants: Adamax, RMSProp, and Adagrad-addressing...* Medium. https://medium.com/@astitwa15108824/comparative-analysis-of-adam-optimizer-and-its-variants-adamax-rmsprop-and-adagrad-addressing-02889adc8735

[23] *Adamax.* Adamax - PyTorch 2.5 documentation. (n.d.). https://pytorch.org/docs/stable/generated/torch.optim.Adamax.html

[24] Daniel. (2023, October 30). *¿Qué es el transfer learning?* Formación en ciencia de datos | DataScientest.com. https://datascientest.com/es/que-es-el-transfer-learning

[25]*Structural analysis and shape descriptors*. OpenCV. (n.d.). https://docs.opencv.org/4.x/d3/dc0/group__imgproc__shape.html#gadf1ad6a0b82947fa1fe3c3d497f260e0

[26] *Structural analysis and shape descriptors*. OpenCV. (n.d.-a). https://docs.opencv.org/4.x/d3/dc0/group__imgproc__shape.html#ga819779b9857cc2f8601e6526a3a5bc71

[27] *Structural analysis and shape descriptors*. OpenCV. (n.d.-a). https://docs.opencv.org/4.x/d3/dc0/group__imgproc__shape.html#ga4303f45752694956374734a03c54d5ff

[28] Porto, J. P. (2024, March 18). *Qué es, historia, características y aplicaciones.* Definición de Python. https://definicion.de/python/

[29] Larkin Alonso, J. (2022, June 16). *¿Qué es tensorflow y para qué sirve?* https://www.incentro.com/es-ES/blog/que-es-tensorflow

[30] IONOS, E. editorial de. (2020, October 8). *Keras: Biblioteca de Código Abierto Para Crear redes neuronales*. IONOS Digital Guide. https://www.ionos.es/digitalguide/online-marketing/marketing-para-motores-de-busqueda/que-es-keras/

[31] Daniel. (2023a, October 30). *Matplotlib: Todo Lo que tienes que saber sobre la librería Python de Dataviz*. Formación en ciencia de datos | DataScientest.com. https://datascientest.com/es/todo-sobre-matplotlib

[32] Alberca, A. S. (2022, May 12). *La librería numpy*. Aprende con Alf. https://aprendeconalf.es/docencia/python/manual/numpy/

[33] *Procesamiento de Imágenes Con opencv en python*. Imagina Formación: Cursos Online y Presencial, bonificados. (n.d.). https://imaginaformacion.com/tutoriales/opencv-en-python

[34] *OpenCV-python tutorials*. OpenCV. (2023, December 27). https://docs.opencv.org/4.x/d6/d00/tutorial_py_root.html

[35] w3compadmin. (2023, September 5). *Using python and opencv for face recognition & detection*. W3computing.com - A Developer Reference Website. https://www.w3computing.com/articles/python-opencv-face-recognition-detection/

[36] *DNN-based Face Detection And Recognition*. OpenCV. (2024, June). https://docs.opencv.org/4.x/d0/dd4/tutorial_dnn_face.html

[37] Hernández González, N. (2023). *Análisis de rendimiento de modelos basados en aprendizaje por transferencia con TensorFlow y TensorRT.*

[38] *mobilenetv2*. MathWorks. (n.d.). https://es.mathworks.com/help/deeplearning/ref/mobilenetv2.html

[39] What is the visual studio Ide? What is the Visual Studio IDE? | Microsoft Learn. (2024, June 19). https://learn.microsoft.com/en-us/visualstudio/get-started/visual-studio-ide?view=vs-2022

[40] Microsoft. (2021, November 3). *Visual studio code - code editing. redefined*. RSS. https://code.visualstudio.com/

[41] Softonic. (n.d.). Droidcam. Droidcam - Download. https://droidcam.en.softonic.com/

[42] Author links open overlay panelCasper Solheim Bojer, AbstractWe review the results of six forecasting competitions based on the online data science platform Kaggle, AthanasopoulosG., CroneS.F., DarinS.G., FildesR., FryC., GillilandM., HongT., HyndmanR.J., KangY., MakridakisS., Montero-MansoP., PetropoulosF., SalinasD., & SmylS. (2020, September 2). *Kaggle forecasting competitions: An overlooked learning opportunity*. International Journal of Forecasting. https://www.sciencedirect.com/science/article/abs/pii/S0169207020301114

[43] P, K. (2021, August 9). *B200C lego classification dataset*. Kaggle. https://www.kaggle.com/datasets/ronanpickell/b200c-lego-classification-dataset

[44] Hazelzet, J. (2019, December 31). *Images of lego bricks*. Kaggle. https://www.kaggle.com/datasets/joosthazelzet/lego-brick-images

[45] Team, K. (n.d.). *Keras Documentation: Mobilenet, mobilenetv2, and MobileNetV3.* https://keras.io/api/applications/mobilenet/

[46]García Ferreira, I. (2022, April 26). *Como Hacer redes neuronales complejas - garcía*. Ferreira. https://www.garcia-ferreira.es/como-hacer-redes-neuronales-complejas/#:~:text=Un%20modelo%20secuencial%20es%20cuando,por%20ejemplo%20cuando%20hicimos%20LSTM

[47] Softonic. (n.d.). Droidcam . Droidcam - Download. https://droidcam.en.softonic.com/

[48] *Image Thresholding*. OpenCV. (2024b, June). https://docs.opencv.org/4.x/d7/d4d/tutorial_py_thresholding.html

[49] Gangotena Torres, E. S. (2022). *Subsistema de clasificación: desarrollo de un sistema distribuido para clasificación de fichas Lego basado en imágenes.* TFG defendido en la Universidad Politécnica Nacional. Quito: EPN.

Todas las referencias a las páginas web utilizadas estaban operativas en diciembre de 2024.

ANEXOS

ANEXO A: Código Programa de entrenamiento y validación

ANEXO B: Código Programa aplicación de selección de piezas para la construcción de figuras LEGO