ESCUELA POLITÉCNICA NACIONAL

FACULTAD DE INGENIERÍA ELÉCTRICA Y ELECTRÓNICA

DESARROLLO DE UN SISTEMA DISTRIBUIDO PARA CLASIFICACIÓN DE FICHAS LEGO BASADO EN IMÁGENES

SUBSISTEMA DE CLASIFICACIÓN

TRABAJO DE INTEGRACIÓN CURRICULAR PRESENTADO COMO REQUISITO PARA LA OBTENCIÓN DEL TÍTULO DE INGENIERO EN TECNOLOGÍAS DE LA INFORMACIÓN

ERNESTO SANTIAGO GANGOTENA TORRES

ernesto.gangotena@epn.edu.ec

DIRECTOR: RAÚL DAVID MEJÍA NAVARRETE, M.Sc.

david.mejia@epn.edu.ec

DMQ, Febrero 2022

CERTIFICACIONES

Yo, ERNESTO SANTIAGO GANGOTENA TORRES declaro que el trabajo de integración curricular aquí descrito es de mi autoría; que no ha sido previamente presentado para ningún grado o calificación profesional; y, que he consultado las referencias bibliográficas que se incluyen en este documento.

ERNESTO SANTIAGO GANGOTENA TORRES

Certifico que el presente trabajo de integración curricular fue desarrollado por ERNESTO SANTIAGO GANGOTENA TORRES, bajo mi supervisión.

RAÚL DAVID MEJÍA NAVARRETE, M.Sc.
DIRECTOR

DECLARACIÓN DE AUTORÍA

A través de la presente declaración, afirmamos que el trabajo de integración curricular aquí descrito, así como el producto resultante del mismo, son públicos y estarán a disposición de la comunidad a través del repositorio institucional de la Escuela Politécnica Nacional; sin embargo, la titularidad de los derechos patrimoniales nos corresponde a los autores que hemos contribuido en el desarrollo del presente trabajo; observando para el efecto las disposiciones establecidas por el órgano competente en propiedad intelectual, la normativa interna y demás normas.

ERNESTO SANTIAGO GANGOTENA TORRES

ING. RAÚL DAVID MEJÍA NAVARRETE, M.Sc.

DEDICATORIA

Este trabajo de integración curricular está dedicado a mi familia que ha estado cerca de mí y me ha apoyado en este proceso.

AGRADECIMIENTO

Quiero agradecer a mis papás y hermanos que han estado conmigo en este proceso y me han apoyado.

Quiero agradecer al director de este trabajo el ingeniero David Mejía M.Sc. que ha sido guía en la realización de este proyecto.

Quiero agradecer a mis compañeros de la carrera Jhimmy González, Cristian Rodríguez, Alex Paredes, Belén Cuesta, Cristina Cuesta que por alguna extraña razón nos hicimos amigos, resultado todo mejor cuando hay gente cerca y también más divertido.

ÍNDICE DE CONTENIDO

RESUMEN

En este Trabajo de Integración Curricular se realizó la implementación del subsistema para clasificación de fichas Lego usando atributos como el color o el tipo de ficha. Para la clasificación de las fichas Lego por tipo se implementó un modelo de red neuronal convolucional, y la transferencia de aprendizaje se realizó a través de otra red neuronal pre entrenada mediante la herramienta MobileNetV2. Para determinar el color de la ficha Lego se utilizó la herramienta OpenCV con la que se realizó la detección de bordes para localizar y obtener un punto de la ficha Lego en la imagen, y así extraer el color de un píxel en dicha posición. Además, se implementó un programa que hace uso del modelo de la red neuronal y la detección de color para realizar la clasificación de otras imágenes que contienen fichas Legos, el cual dispone de *stubs* que simulan el funcionamiento del subsistema de adquisición y el subsistema de almacenamiento. Este subsistema realizará la clasificación de fichas Lego que son recibidas desde el subsistema de adquisición y el resultado de la clasificación se envía al subsistema de almacenamiento.

PALABRAS CLAVE: Red Neuronal Convolucional, Aprendizaje por Transferencia, OpenCV, MobileNetV2.

ABSTRACT

In this Final Integration Work, the implementation of the subsystem for classifying Lego tiles was carried out using attributes such as color or type of tiles. For the classification of the Lego tiles by type, a convolutional neural network model was implemented, and the learning transfer was carried out through another pre-trained neural network using the MobileNetV2 tool. The color of the Lego tile was determined using the OpenCV tool with which edge detection was performed to locate and obtain a point of the Lego tile in the image, and thus extract the color of a pixel in that position. In addition, a program was implemented which makes use of the neural network model and color detection to classify other images containing Legos, which has *stubs* that simulate the operation of the acquisition subsystem and the storage subsystem. This subsystem will perform the classification of Lego tiles that are received from the acquisition subsystem and the result of the classification is sent to the storage subsystem.

KEYWORDS: Convolutional Neural Network, Transfer Learning, OpenCV, MobileNetV2.

1. INTRODUCCIÓN

En este Trabajo de Integración Curricular se desarrolló un subsistema que permite clasificar imágenes que contienen fichas Lego por color y por tipo. Primero fue necesario crear una base de datos, la misma que se almacenó en un directorio con subdirectorios, en cada subdirectorio se colocó un conjunto de imágenes con características similares. La base de datos cuenta con aproximadamente 5.800 imágenes.

Se definieron subconjuntos para entrenamiento, validación y pruebas para definir el modelo. Los dos primeros subconjuntos se emplearon para el entrenamiento y la validación. Durante el proceso de pruebas se encontró que el modelo no cumplía con el rendimiento deseado debido a que la cantidad de imágenes obtenidas no era suficiente, por lo que se tuvo que buscar métodos que ayuden a solucionar este problema. Se encontró que se podía mejorar el rendimiento del modelo de la red neuronal al realizar modificaciones en las imágenes de forma aleatoria; por ejemplo, mediante rotaciones, desplazamientos o distorsiones. Así también, se empleó transferencia de aprendizaje con una red neuronal pre entrenada generada en la herramienta MobileNetV2. Esta red neuronal pre entrenada fue elegida porque tiene un tamaño reducido de apenas 17 megabytes.

Una vez entrenado y validado el modelo, se lo almacenó para poder usarlo en el subsistema. El subsistema fue desarrollado mediante .NET Framework con el lenguaje de programación C#. Debido a que, la red neuronal estaba implementada usando el paquete de Keras de TensorFlow y escrita en el lenguaje de programación Python, se requirió instalar los paquetes de TensorFlow.Net, Keras.Net y Numpy.Net para que sea posible emplear la red en el subsistema.

El proceso de clasificación que realiza el subsistema permite predecir el tipo de ficha Lego que hay en la imagen, el subsistema de adquisición de proporcionar la imagen, para lo cual se desarrolló un stub que simula esta opción. Las imágenes proporcionadas deben tener las dimensiones definidas por el modelo implementado.

El modelo hace la predicción y devuelve el tipo de ficha Lego estimado que el subsistema encontró en la imagen, para lo cual emplea un conjunto de probabilidades que fueron aprendidas con base en las imágenes usadas para entrenar al modelo implementado.

Por último, se implementó un algoritmo de detección de contornos de objetos para obtener la posición de la ficha Lego en la imagen, y de esta forma poder extraer un píxel central en

la ficha Lego para poder detectar su color. Para realizar las pruebas del subsistema se utilizaron las imágenes del conjunto de pruebas.

1.1 Objetivo general

Desarrollar un subsistema de clasificación para imágenes que contienen una ficha Lego mediante el uso de redes neuronales convolucionales y aprendizaje por transferencia para realizar clasificación de las fichas usando características como el color y el tipo.

1.2 Objetivos específicos

Los objetivos de este trabajo de integración curricular son:

1. Desarrollar un modelo de red neuronal, entrenar y validar el mismo.
2. Diseñar el subsistema de clasificación con base en los requerimientos definidos considerando el modelo entrenado, implementarlo y realizar pruebas sobre el mismo.
3. Analizar los resultados del subsistema de clasificación al procesar imágenes nuevas.

1.3 Alcance

En este Trabajo de Integración Curricular se estudiarán las redes neuronales y su funcionamiento, haciendo énfasis en las redes neuronales convolucionales, así como también en el aprendizaje por transferencia de aprendizaje [1].

Por otro lado, también se definirán las tareas para el subsistema, las cuales serán establecidas en el tablero Kanban. Como parte de estas tareas será necesario obtener datos de entrenamiento, definir el tipo de red neuronal a emplearse, el número de capas de la red, la función de activación y las métricas para su evaluación. Posteriormente se procederá a elegir la red pre entrenada y se realizarán pruebas para determinar la eficacia en la clasificación. También se establecerá un mecanismo que permita utilizar el modelo entrenado para su uso en el ambiente de .NET.

En el proceso de desarrollo de la red neuronal se requerirá un conjunto de imágenes con fichas Lego. Este conjunto deberá ser separado en grupos que se emplearán para el entrenamiento del modelo, su validación y las pruebas. Para mejorar los resultados, se usará una red neuronal pre entrenada usando la herramienta MobileNetV2 [2]. En el modelo implementado se lo entrenará hasta que mejore tomando en cuenta si mejora o no con respecto a los valores de pérdida en el aprendizaje.

El subsistema realizará únicamente la clasificación de las fichas Lego mediante la detección de los colores y el tipo de Lego, se lo realizará en la plataforma .NET Framework con el lenguaje de programación C#. La detección de colores tiene como paso intermedio la detección de los bordes o contornos de una ficha Lego dentro de una imagen mediante el algoritmo Canny de identificación de bordes que está disponible dentro de la herramienta OpenCV. Al haber detectado la ficha Lego se obtiene su punto central, con el cual se obtienen los valores RGB (*red blue green*) de un píxel del punto central obtenido y en consecuencia el color aproximado.

1.4 Marco teórico

En esta sección se presentan las bases teóricas sobre el aprendizaje de máquina (*Machine Learning*), las redes neuronales, las redes neuronales convolucionales, el aprendizaje por transferencia, así como las tecnologías usadas para desarrollar este Trabajo de Integración Curricular.

1.4.1 Aprendizaje Automático

El aprendizaje automático (*Machine Learning*) es una rama de la inteligencia artificial, que permite a las máquinas usar la experiencia adquirida o el conocimiento previo para mejorar el proceso de predicción. Usualmente, en el aprendizaje máquina se emplean los siguientes pasos [4]:

1. Alimentar con datos al algoritmo, es decir proveer de información al modelo.
2. Con una parte de los datos se realiza el entrenamiento del modelo.
3. Con la otra parte de los datos se realizan pruebas con el modelo.
4. Finalmente, se usa el modelo entrenado para tareas de predicción.

1.4.2 Redes Neuronales Artificiales

Las redes neuronales artificiales (ANN – *Artificial Neural Networks*) han sido inspiradas en las redes neuronales biológicas presentes en animales y humanos, las redes neuronales bilógicas se conectan a cientos de miles de neuronas en la red. Cada neurona recibe señales de entrada que se combinan para generar señales de salida las que serán enviadas a otras neuronas conectadas. La red neuronal puede hacer funciones complejas las que depende de cómo están establecidas las conexiones de la cantidad de existente de estas entre neuronas de la red, por tanto, la idea clave de las redes neuronales artificiales es la de construir modelos matemáticos que simulen el comportamiento de una red neuronal biológica para lograr inteligencia artificial.

En la Figura 1.1 se muestra el modelo matemático de una neurona. Como se puede observar, una neurona puede tener múltiples entradas (ejemplo: x_1, x_2, …, x_m), en la cual se realiza una operación como por ejemplo una suma, con unos parámetros ajustables (ejemplo: w_1, w_2, …, w_m que son denominados pesos), así como el parámetro b (*bias*). El parámetro *bias* permite para ajustar el resultado a la salida de la neurona, para obtener mejores resultados. El resultado de la operación se remite a una función de activación Ø(.), la cual genera la salida de la neurona [2].

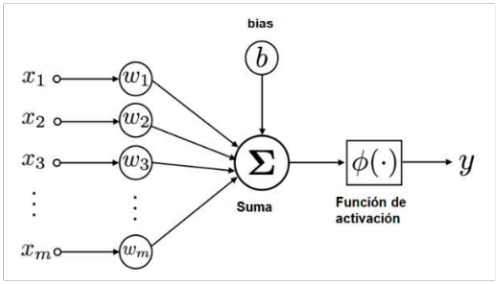

Figura 1.1 Modelo de neurona artificial [2]

La salida de la neurona puede representarse usando la siguiente ecuación:

$$y = \emptyset(w_1x_1 + w_2x_2 + \cdots + w_mx_m + b)$$

La misma que puede ser descrita como:

$$y = \emptyset(WX + b)$$

El parámetro W representa los pesos y b el parámetro *bias*. En una red neuronal el parámetro de los pesos se aprende mediante entrenamiento, así como el parámetro *bias*.

En la Figura 1.2 se presenta la arquitectura de una red neuronal con múltiples entradas y múltiples salidas, así también como múltiples capas ocultas.

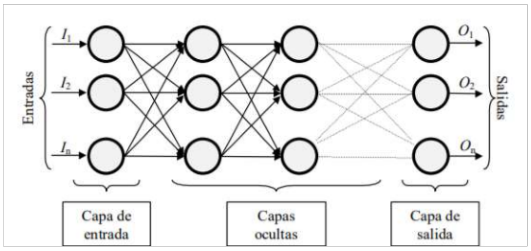

Figura 1.2 Red neuronal completamente conectada [3]

1.4.2.1 Función de Activación

En la Figura 1.1 se puede observar que antes de la salida de una neurona existe una función de activación la cual sirve para ayudar a la red neuronal a aprender patrones complejos de datos.

Función de activación ReLu: En la Figura 1.3 se presenta la función de activación ReLu. Esta es la función de activación más usada en las capas intermedias de una red neuronal, se caracteriza por ser eficiente [4].

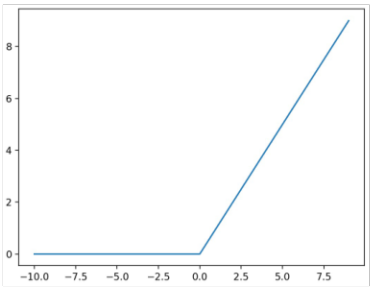

Figura 1.3 Función de activación ReLu

Función de activación SoftMax: La función de activación Softmax funciona como una distribución de probabilidad discreta para n clases, esta función de activación generalmente se emplea en la última capa porque está hecha para generar probabilidades de salida. En la Figura 1.4 se presenta una gráfica de esta función.

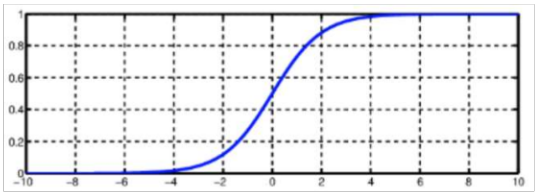

Figura 1.4 Función de activación Softmax [4]

1.4.3 Redes Neuronales Convolucionales

Una red neuronal convolucional (CNN – *Convolutional Neural Network*) es un tipo de red neuronal artificial que dentro de su estructura tiene capas convolucionales, capas de agrupamiento, capas de aplanamiento, y capas densas. La capa convolucional sirve para disminuir el tamaño de, por ejemplo, una imagen, y reúne toda la información del campo en un solo píxel, la capa de agrupamiento permite resumir las características presentes en una región generada por una capa de convolución. La capa de aplanamiento permite que los datos de las capas convolucionales y de agrupamiento sean transferidos a un vector de una sola dimensión, y las capas densas consisten en neuronas totalmente conectadas que permiten continuar con el aprendizaje después de los procesos de convolución y agrupamiento. Las redes neuronales convolucionales difieren de otros tipos de redes neuronales en la forma en la que se organizan las capas de agrupamiento y las capas convolucionales, así como en el entrenamiento de estas [5].

1.4.3.1 Capa Convolucional

Esta capa presenta una arquitectura única, puesto que sus entradas están organizadas en tres dimensiones: ancho, alto y espesor. En esta capa se especifican filtros o *kernels*. Los filtros funcionan como detectores de patrones. La función de estas capas convolucionales es la de detectar patrones como bordes y esquinas, pero hoy en día, están incluso en capacidad de detectar animales como gatos o perros, entre otros.

La capa convolucional usualmente se implemente mediante una matriz de filtro o *kernel* que puede ser cuadrada o de una dimensión específica. Se puede empezar por valores iniciales de 3x3 elementos o 5x5 elementos. Además, los datos que se emplean en estas matrices son aleatorios inicialmente, y la tarea de la red neuronal es aprender los valores óptimos de esta matriz.

En la Figura 1.5, la Figura 1.6 y la Figura 1.7 se presenta un ejemplo del funcionamiento de la operación de convolución [6].

En la Figura 1.5 la segunda matriz que se aprecia es el filtro o *kernel*, la misma que se irá sobreponiendo sobre la matriz con datos, para determinar con que filas y columnas de esta matriz se operará para obtener la matriz de filtro. Se realizan operaciones matemáticas de suma y multiplicación entre los valores del *kernel* con los de la matriz que resulta de la superposición, el resultado de estas operaciones se ubica en la primera celda de la matriz de filtro definitiva.

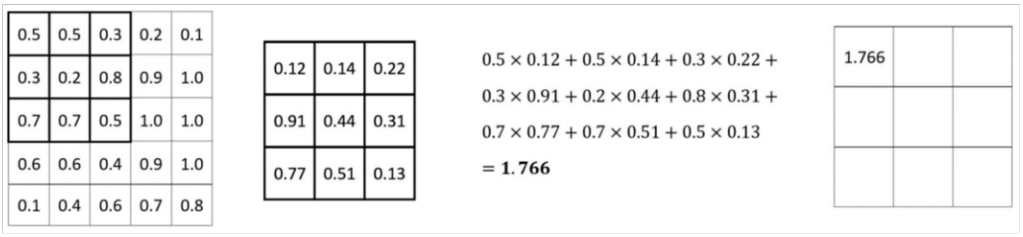

Figura 1.5 Primer paso de la operación de convolución

En la Figura 1.6 se observa cómo la matriz del filtro se ha movido una posición a la derecha para realizar las operaciones de suma y multiplicación entre la matriz de *kernel* con la correspondencia en filas y columnas de la matriz de datos, para obtener como resultado el segundo elemento de la matriz.

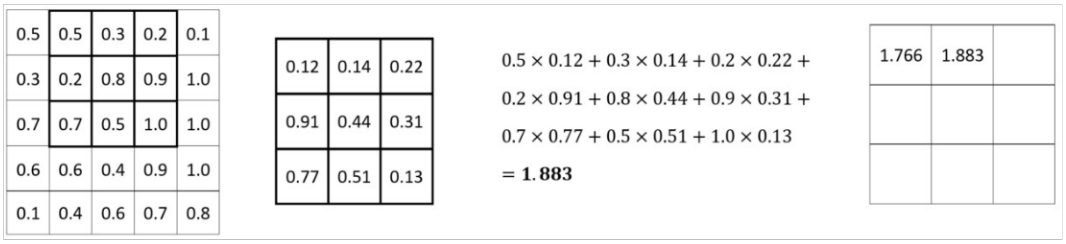

Figura 1.6 Segundo paso de la convolución

En la Figura 1.7 se ve el paso final de la convolución, la matriz de kernel se ha superpuesto en las últimas filas y columnas de la matriz de datos, por lo que, una vez que se realizan las operaciones con estas matrices se obtiene el último elemento de la matriz.

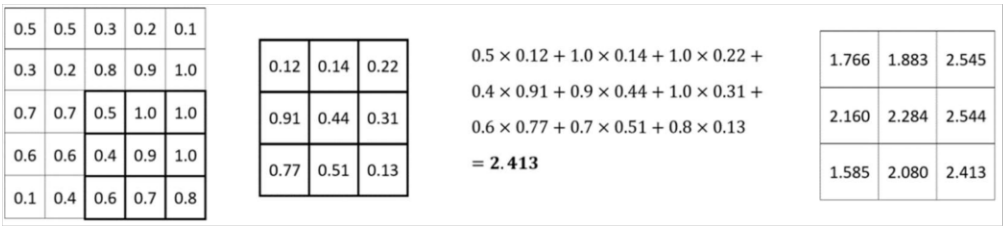

Figura 1.7 Último paso de la operación de convolución [6]

1.4.3.2 Capas de agrupamiento (*pooling*)

Las capas convolucionales resultan efectivas en modelos de aprendizaje profundo. Estas permiten que las capas cercanas a la entrada aprendan características de bajo nivel, por ejemplo, líneas; mientras que las capas más profundas en el modelo aprendan características de alto orden o más abstractas, como formas u objetos específicos.

Una limitación de la salida de las capas convolucionales es que registran la posición precisa de las características en la entrada. Esto significa que pequeños movimientos en la posición de la característica en la imagen de entrada darán como resultado un mapa de características diferente. Esto puede pasar con cambios en las imágenes como recortes, rotaciones, desplazamientos, entre otros.

Para solucionar este problema se implementaron las capas de agrupamiento. Por un lado, las capas de agrupamiento reducen el tamaño de las matrices resultantes de las capas convolucionales. La agrupación consiste en pasar las matrices de convolución por un filtro, generalmente de dimensiones 2x2 con pasos de 2, lo que reduce el tamaño de las matrices de convolución en un factor de 2. Las funciones de agrupamiento más comunes son agrupación promedio (*Average Pooling*) y agrupación máxima (*Max Pooling*) [7].

Average Pooling: como se muestra en la Figura 1.8, se calcula el valor promedio para cada filtro de dimensiones 2x2 de los datos de entrada, usando un paso de dos elementos, cada promedio pasa a formar parte de la matriz reducida. La matriz original tiene dimensiones de 4x4 y luego de aplicar el filtro se obtiene una matriz de 2x2.

Max Pooling: En este caso, en lugar de obtener un promedio como en *Average Pooling*, se obtiene el valor máximo. En la Figura 1.9, se presenta una matriz que será procesada usando un filtro *max pooling*, así como el resultado del filtrado.

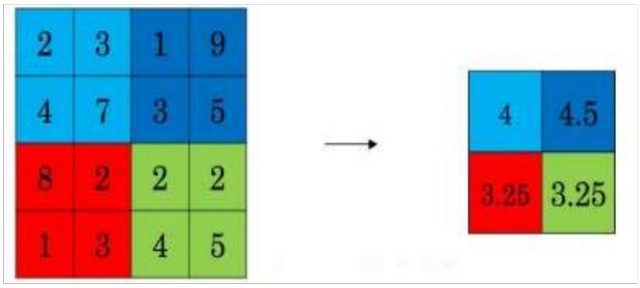

Figura 1.8 *Average Pooling* con pasos de 2 [6]

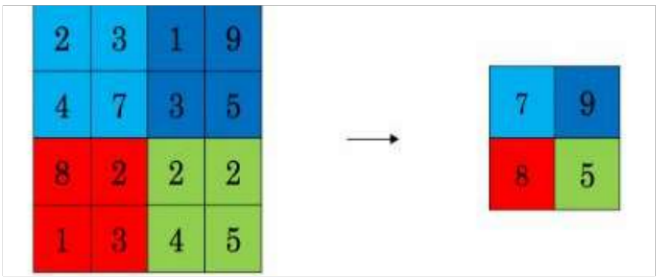

Figura 1.9 Max Pooling con pasos de 2 [8]

1.4.3.3 Capas de aplanamiento *(flatten)*

La capa de aplanamiento, en una red neuronal convolucional, generalmente va ubicada luego de las capas convolucionales y de las capas de agrupamiento. Esta capa se encarga de convertir a las matrices resultantes de las capas anteriores en vectores planos, para luego pasar la información a las capas densas.

1.4.3.4 Capas densas (*dense layers*)

En una red neuronal una capa densa es la que tiene todas sus neuronas conectadas entre sí mismas, y también se encuentran conectadas entre las capas precedentes y subsecuentes. Las capas densas reciben sus entradas de las capas anteriores y sus salidas van a las capas siguientes, así como también definen el número de salidas de la red neuronal, estas constituyen las capas más comunes en una red neuronal [9].

1.4.3.5 Capas *dropout*

Estas capas implementan un método para desactivar aleatoriamente neuronas de una red neuronal para evitar el sobre entrenamiento[1].

1.4.4 Aprendizaje por transferencia

El aprendizaje por transferencia consiste en el mejoramiento del aprendizaje de una nueva tarea a través de la transferencia de conocimiento de tareas relacionadas que ya han sido aprendidas previamente. En el aprendizaje por transferencia se han desarrollado métodos para transferir el conocimiento en una o más fuentes de trabajo y esta información puede usarse para mejorar el aprendizaje en tareas relacionadas. Las técnicas que permiten la transferencia de conocimiento tienen como objetivo hacer que el aprendizaje de máquina sea tan eficiente como el aprendizaje humano [10].

1.4.5 Detección de bordes y contornos

La detección de bordes es una técnica de procesamiento de imágenes para encontrar los límites de los objetos dentro de las imágenes. Funciona detectando discontinuidades en el brillo. La detección de bordes se utiliza para la segmentación de imágenes y la extracción de datos en áreas como el procesamiento de imágenes, y la visión artificial.

Entre algoritmos populares para la detección de bordes y contornos se tienen los de Sobel, Canny, Prewitt, Roberts. [11]. En la Figura 1.10 se presenta el algoritmo Sobel de detección de bordes de una imagen de un paisaje. Mientras que en la Figura 1.11 se puede observar el algoritmo de Canny aplicado en una imagen con monedas.

Figura 1.10 Algoritmo Sobel de detección de contornos

[1] El sobre entrenamiento significa que la red neuronal después de un determinado número de entrenamientos comienza a memorizar patrones, lo cual puede derivar en predicciones erróneas con datos desconocidos.

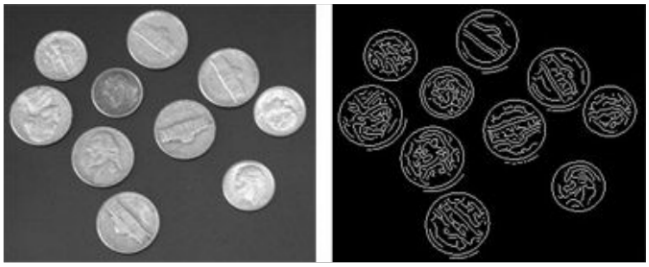

Figura 1.11 Algoritmo Canny de detección de contornos

1.4.5.1 Detección de contornos de Canny

El algoritmo de contornos Canny es un algoritmo de múltiples etapas que utiliza los siguientes pasos:

1. Reducir ruido: Debido a que la detección de bordes es susceptible al ruido, el primer paso es remover el ruido mediante un filtro Gaussiano de 5x5.
2. Encontrar el gradiente de intensidad de la imagen: La imagen suavizada se filtra con un filtro Sobel en dirección horizontal y vertical para obtener la primera derivada en dirección horizontal y en dirección vertical y se combinan ambos resultados. En la Figura 1.12 se presenta una imagen a la que se le ha realizado este procedimiento, se muestran los resultados después del filtro horizontal y vertical y la combinación de ambos filtros.
3. Suprimir pixeles: Después de realizado el proceso anterior se realiza un escaneo completo de la imagen para eliminar los píxeles no deseados que pueden no constituir los contornos [12]. En la Figura 1.13 se presenta el resultado de este proceso.

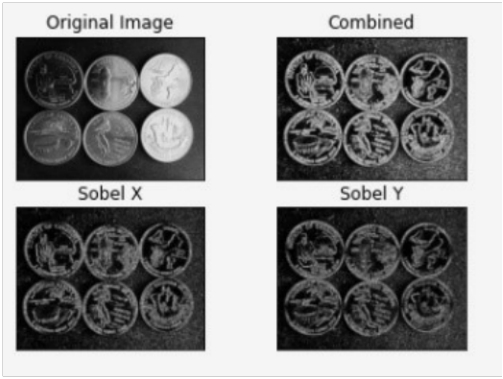

Figura 1.12 Algoritmo Canny aplicado en la detección de contornos de monedas

Figura 1.13 Algoritmo Canny de detección de contornos

1.4.6 Tecnologías utilizadas

1.4.6.1 Python

Python es un lenguaje de programación de código abierto, simple y minimalista, con naturaleza de pseudocódigo que permite al programador concentrarse en la solución más que en el lenguaje, lo que significa que no tiene que preocuparse de detalles de bajo nivel como la memoria. Python está disponible en varios sistemas operativos como Windows, Linux y Mac. Python es un lenguaje interpretado, esto quiere decir que Python no requiere compilación[2]. Python tiene un intérprete que convierte el código en *bytecodes*, que luego son convertidos en código que puede entender la computadora a través de la PVM (Python *Virtual Machine*). Python es orientado a objetos, esto significa que está construido alrededor de métodos que son piezas de código reusable. Python tiene una forma más simple y potente de hacer programación orientada a objetos comparado a lenguajes de programación como C++ o Java. Python tiene una gran cantidad de librerías y documentación que permiten hacer gran variedad de funciones como: pruebas unitarias,

[2] Compilación significa que el código escrito, por ejemplo, en C o C++, es convertido en código binario que es entendido por la máquina. Cuando el programa se ejecuta, su código se copia desde el disco duro a la memoria y desde ahí se ejecuta.

bases de datos, CGI[3], FTP[4], correo electrónico, XML[5], XML-RPC[6], HTML[7], archivos WAV[8], criptografía, GUI (interfaces gráficas de usuario), etc. [13].

1.4.6.2 Numpy

Numpy es una librería[9] fundamental de Python, que permite el manejo de arreglos y matrices, así como realizar operaciones matemáticas, lógicas, transformadas discretas de Fourier, álgebra lineal básica, operaciones estadísticas básicas y mucho más [14].

1.4.6.3 TensorFlow

TensorFlow es una plataforma de código abierto para el aprendizaje máquina. Comprende un ecosistema que tiene herramientas y librerías que facilitan el desarrollo de aplicaciones para aprendizaje automático. TensorFlow permite el entrenamiento y la implementación de modelos de forma fácil, ya sea en servidores, en la web, sin importar el lenguaje o la plataforma que se utilice. En TensorFlow se tiene una API[10] funcional de Keras y una API de subclases de modelos para la creación de topologías complejas [15].

1.4.6.4 Keras

Creada en 2015 por un desarrollador de Google. Keras está integrada en la plataforma TensorFlow. El propósito de Keras es el de desarrollar redes neuronales, para lo cual proporciona abstracciones esenciales y bloques de construcción para desarrollar soluciones de aprendizaje de máquina con alta velocidad de iteración [16], [17]. Keras provee una API que reduce el número de acciones requeridas para usos comunes. Keras se encuentra en el centro de un amplio ecosistema de proyectos estrechamente

---

[3] CGI: *Computer Generated Imagery*, permite crear imágenes que parecen reales, se emplea principalmente en películas.
[4] FTP: *File Transfer Protocol,* protocolo de transferencia de archivos entre dispositivos conectados a una red.
[5] XML: *Extensible Markup Language*, es un lenguaje de marcado para intercambio de datos.
[6] XML-RPC: *Remote Procedure Call,* protocolo de llamada remota que utiliza XML para codificar llamadas y HTTP como protocolo de transporte.
[7] HTML: *HyperText Markup Language*, es un lenguaje de marcado que define la estructura de una página web.
[8] WAV: *Waveform Audio file format*, es un formato de audio digital que puede o no tener compresión.
[9] Librería: Conjunto de código pre escrito que permiten a los programadores optimizar un programa.
[10] API: *Application Programming Interface,* es conjunto de protocolos y definiciones que permiten la comunicación entre diferentes aplicaciones.

conectados que juntos cubren cada paso del flujo de trabajo de aprendizaje automático, en particular para [18]:

- Entrenamiento de modelos escalables en GCP[11] a través de TF Cloud,
- Ajuste de parámetros con KerasTuner,
- Capas adicionales, pérdidas, métricas, devoluciones de llamada, a través de TensorFlow Addons.
- Implementación de modelos en dispositivos móviles o integrados con TF *Lite*.

1.4.6.5 MobileNetV2

MobileNetV2 es una red neuronal convolucional pre entrenada desarrollada por Google, la cual es utilizada en el aprendizaje por transferencia, esta red es el resultado del mejoramiento de la precisión de las redes neuronales en las cuales se tiene el costo de alta capacidad computacional. Con MobileNetV2 se puede mejorar modelos de visión por computadora adaptados a dispositivos móviles, al disminuir significativamente el número de operaciones y memoria necesaria manteniendo la misma precisión. MobileNetV2 toma como entrada una representación comprimida de baja dimensión que primero se expande a una dimensión alta y se filtra con una convolución ligera en profundidad. Posteriormente, las características se vuelven a proyectar en una representación de baja dimensión con una convolución lineal.

MobileNetV2 se puede implementar en cualquier *framework* moderno y permite que se mejore el rendimiento con respecto a un modelo estándar. Este modelo es particularmente apropiado para diseños móviles, porque permite reducir significativamente la memoria necesitada durante la interferencia por medio de nunca materializar completamente los tensores. Esto reduce la necesidad de acceso a la memoria [19].

1.4.6.6 .NET Framework

Es un entorno de ejecución desarrollado por Microsoft, consta de dos componentes principales *Common Language Runtime* (CLR) que controla la ejecución de las aplicaciones y el .NET Framework que contiene la biblioteca de código probado y funcional para las aplicaciones. .NET Framework incluye bibliotecas para determinadas áreas de desarrollo de aplicaciones, como ASP.NET para aplicaciones web, ADO.NET para el

[11] GCP: *Google Cloud Platform,* es un servicio de computación en la nube proporcionado por Google, para empresas o personas.

acceso a los datos, Windows *Communication Foundation* (WCF) para las aplicaciones orientadas a servicios y Windows *Presentation Foundation* (WPF) para las aplicaciones de escritorio de Windows. Los compiladores de lenguajes cuya plataforma es .NET Framework generan un código intermedio denominado *Common Intermediate Language* (CIL), que permite que las rutinas escritas en un lenguaje sean accesibles para otros lenguajes, lo cual hace que los programadores puedan usar su lenguaje de programación preferido [20].

1.4.6.7 Visual Studio

Visual Studio es un entorno de desarrollo integrado (IDE - *Integrated Development Environment*) que soporta muchos aspectos del desarrollo de software. Permite editar, depurar y ejecutar el código y luego publicar una aplicación. Incluye compiladores, herramientas para para completar código, diseñadores gráficos, y muchas características para enriquecer el proceso de desarrollo de software.

Visual Studio está disponible para Windows y Mac y está optimizada para trabajar en desarrollo multiplataforma y aplicaciones móviles [21]. En la Figura 1.14 se presenta la interfaz de usuario de Visual Studio para Windows.

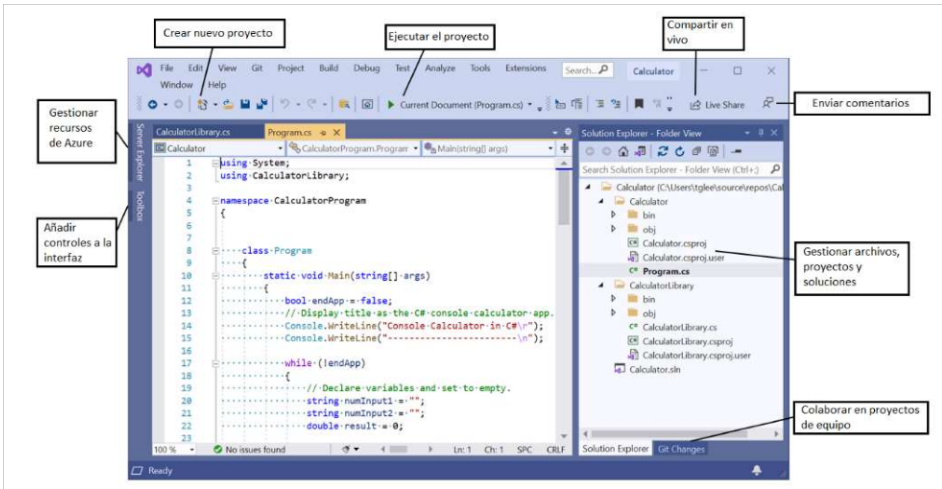

Figura 1.14 Interfaz de usuario de Visual Studio en Windows [21]

1.4.6.8 Tensorflow.Net

Tensorflow.Net es un proyecto de código abierto que implementa TensorFlow en ambientes de .NET. TensorFlow.NET usa el estándar .NET Standard 2.0, por lo que puede usarse mediante .NET Framework o .NET Core/.NET 5 [22].

1.4.6.9 Numpy.Net

Es una implementación de C# para NumPy, lo que permite usar en .NET la biblioteca fundamental para computación e inteligencia artificial. Numpy.Net no requiere instalación local de Python [23].

1.4.6.10 OpenCV

OpenCV es una biblioteca de código abierto que incluye varios cientos de algoritmos de visión artificial. OpenCV incluye varias bibliotecas compartidas o estáticas. Entre todas las bibliotecas que existen se tienen por ejemplo las siguientes:

- Procesamiento de imágenes (`imgproc`): módulo de procesamiento de imágenes que incluye filtrado de imágenes lineales y no lineales, transformaciones de imágenes geométricas (cambio de tamaño, deformación afín y perspectiva, reasignación genérica basada en tablas), conversión de espacio de color, histogramas, etc.
- Análisis de video (`video`): módulo de análisis de video que incluye estimación de movimiento, sustracción de fondo y algoritmos de seguimiento de objetos.
- Calibración de cámara y reconstrucción 3D (`calib3d`): algoritmos básicos de geometría de vista múltiple, calibración de cámara única y estéreo, estimación de pose de objeto, algoritmos de correspondencia estéreo y elementos de reconstrucción 3D.
- Detección de objetos (`objdetect`): detección de objetos e instancias de las clases predefinidas (por ejemplo, rostros, ojos, tazas, personas, automóviles, etc.). [25].

1.4.7 Metodología Kanban

La metodología Kanban se utiliza para programar la fabricación justo a tiempo (JIT – *Just InTime*). Kanban significa literalmente cartel o letrero. El nombre se deriva de las tarjetas que la fábrica inventora de esta metodología usó para hacer un seguimiento de la producción. La metodología Kanban se centra en las tareas que tiene que realizar un equipo de trabajo, cuando una tarea se termina se hacen pruebas para luego hacer la implementación. Kanban permite a los equipos e individuos administrar múltiples proyectos

o tareas al mostrarlas en lo que se conoce como un tablero Kanban. El uso de un tablero Kanban, permite a los equipos mantenerse actualizados sobre el progreso general del proyecto y el estado de la tarea. Lo más importante de Kanban es el tablero Kanban, el cual se utiliza para visualizar el trabajo que debe realizarse al presentar la información del progreso y el estado de las tareas. Como se puede ver todas las tareas en un tablero Kanban, funciona como una visión general de alto nivel del trabajo.

El tablero contiene tres columnas organizadas de izquierda a derecha:

- Pendiente: esta es la columna donde colocará todo el trabajo próximo. Son tareas que no han iniciado.
- En curso: en esta columna se encuentran todas las tareas en las que se está trabajando actualmente.
- Completado: aquí es donde se colocarán todas las tareas terminadas.

Las tareas se colocan en tarjetas, y se las mueve de izquierda a derecha con respecto a las columnas del tablero, de esta forma se asegura de que todas las tareas terminen antes de continuar con la siguiente. De esta manera, mejorará su eficiencia [26].

2 METODOLOGÍA

El presente Trabajo de Integración Curricular fue realizado con el enfoque cuantitativo y deductivo, fue necesario entrenar un modelo, en el cual se estimaron los parámetros requeridos, así como se realizaron pruebas con el modelo de red neuronal con transferencia de aprendizaje y por medio de porcentajes cuantificables se fue mejorando el modelo hasta tener resultados adecuados.

Este trabajo es de tipo experimental y la información se recolectó a través la búsqueda de imágenes por medio de Google y una base de datos de imágenes del sitio web Kaggle.com [27]. Previo al diseño del modelo, se realizó limpieza de las imágenes; es decir, se descartó aquellas que poseían varios tipos de Legos y se recortó aquellas que contenían texto.

Para el diseño del modelo se utilizaron redes neuronales con *transfer learning* escritas mediante el lenguaje de programación Python. El modelo, después de que se terminó de entrenar se lo guardó, para después utilizarlo en una aplicación desarrollada en Visual Studio mediante el lenguaje de programación C# para disponer del subsistema de clasificación de fichas Lego.

El subsistema también clasifica las fichas Lego por color mediante la detección de contornos y la extracción de un píxel central del cual se identifica el color.

El subsistema de clasificación implementa un *stub* que simula recibir las imágenes en formato matrices RGB redimensionadas a 224x224 píxeles de alto por ancho desde subsistema de adquisición. El subsistema está conformado de la parte de detección de color y de la detección del tipo de ficha Lego para imágenes que contienen únicamente un objeto. Con los resultados de la clasificación el *Stub* simula enviar al subsistema de almacenamiento.

2.1 Diseño

El diseño consistió en realizar los siguientes pasos:

- Determinación de los requisitos funcionales y no funcionales a implementarse.
- Determinación de la arquitectura a utilizarse.
- Implementación del subsistema.
- Pruebas de funcionamiento del subsistema.

2.2 Historias de Usuario

Para los requerimientos funcionales y no funcionales se generaron historias de usuario, las cuales luego fueron colocadas en el backlog. Las historias de usuario tienen de identificación la letra H seguida del número de historia ejemplo: H1.

Para definir las historias de usuario se realizó una entrevista con el Director del Trabajo de Integración Curricular.

La Tabla 2.1 presenta la historia de usuario relativa al reconocimiento de fichas Lego en imágenes, la Tabla 2.2 presenta la historia de usuario relativa al tipo de reconocimiento requerido, la Tabla 2.3 presenta la historia de usuario relativa al reconocimiento del color.

Tabla 2.1 Historia de usuario H1

Tareas:	Reconocer fichas Lego
Identificación	H1
Descripción:	Se requiere reconocer fichas Lego mediante la implementación de una red neuronal.

Tabla 2.2 Historia de usuario H2

Tareas:	Reconocer fichas Lego por factor y forma
Identificación	H2
Descripción:	Se requiere que en el reconocimiento se tome en consideración la forma y el factor de la ficha Lego.

Tabla 2.3 Historia de usuario H3

Tareas:	Establecer el color de la ficha lego
Identificación	H3
Descripción:	Una vez definido el tipo de ficha Lego, es necesario reconocer el color de esta.

Lo establecido en la historia de usuario H2 considera el tipo de ficha Lego, para lo cual se requiere reconocer tanto la forma como el factor de la ficha Lego. Debido a la gran cantidad de tipos de ficha Lego, se decidió emplear solo 14 tipos de fichas.

Después de obtenidas las historias de usuario se procedió a generar los requerimientos funcionales y no funcionales del subsistema. En la Tabla 2.4 se los resume.

Tabla 2.4 Requerimientos

Requerimientos funcionales	Requerimientos no funcionales
El subsistema debe emplear el formato JPG para las imágenes. El subsistema debe manejar imágenes de tamaño 224x224 pixeles. El subsistema debe emplear las matrices RGB de cada imagen.	El modelo será implementado mediante el lenguaje de programación Python y las herramientas TensorFlow y Keras. El modelo será almacenado en un formato para ser utilizado en el subsistema. El subsistema será implementado en el lenguaje de programación C#, con el uso de los paquetes Nuget Keras.NET, TensorFlow.NET, Numpy.NET y OpenCVSharp.

Una vez definidos los requerimientos se establecieron las tareas que deben ser realizadas. Estas tareas fueron colocadas en el tablero Kanban.

2.3 Arquitectura

La arquitectura del subsistema consiste en un *stub* que implementa la clasificación de imágenes que contienen una solo ficha Lego de un solo color. El subsistema cuenta con un modelo de red neuronal convolucional con aprendizaje por transferencia mediante la herramienta MobileNetV2. El modelo también clasifica las imágenes por color mediante la extracción de un píxel para obtener su color, para este proceso primero de detecta el contorno de la ficha lego para poder obtener un píxel central. El *stub* simula la recepción de imágenes del subsistema de adquisición al tener que el mismo realizar el preprocesamiento de las imágenes JPG para hacer las predicciones, los resultados se envían al subsistema de almacenamiento.

2.3.1 Arquitectura de la red neuronal

La arquitectura del subsistema consiste en un modelo de red neuronal convolucional secuencial desarrollada en Keras. En esta red neuronal se utiliza transferencia de aprendizaje con la herramienta MobileNetV2. Para la transferencia de aprendizaje solo fue necesario importar esta herramienta que está disponible como una librería de Keras. A esta red MobileNetV2 se descongelaron las últimas 26 capas para reentrenarlas con una tasa de entrenamiento baja. Las primeras capas de esta red neuronal no se sometieron al proceso de entrenamiento, esto es debido a que MobileNetV2 ya ha sido entrenada previamente. La función de la red neuronal es la de clasificar los 14 tipos de fichas Lego que se definieron en las historias de usuario. El modelo después de haber sido entrenado, se lo sometió a pruebas para comprobar su eficacia, cuando estas pruebas fueron satisfactorias se almacenó el modelo para su posterior uso en el subsistema. En la Figura 2.1 se puede observar la arquitectura de la red neuronal.

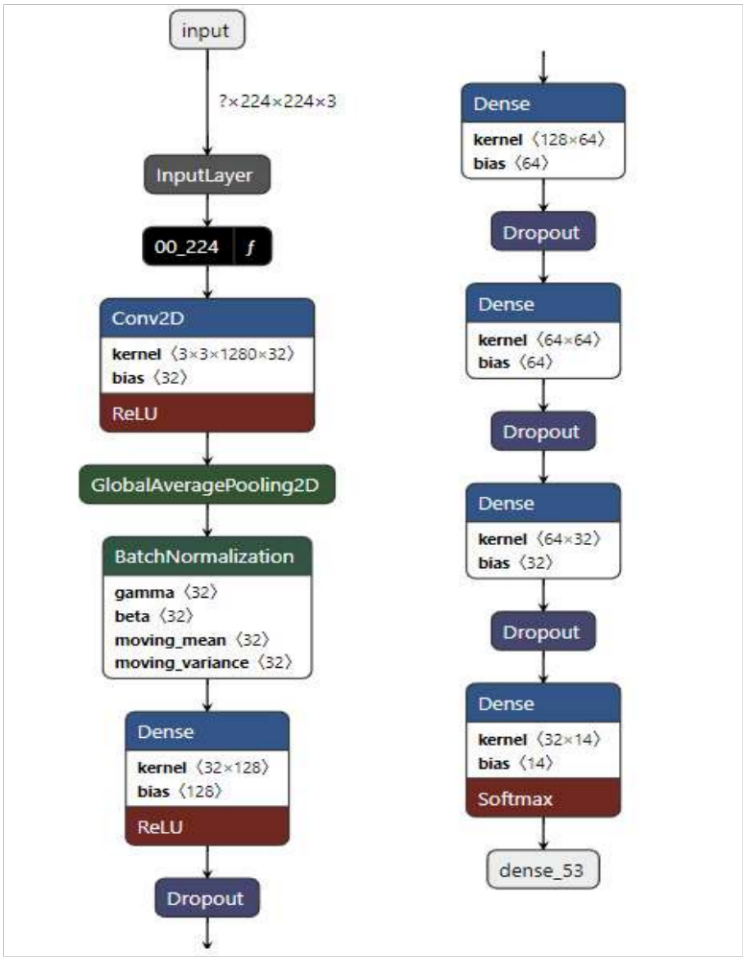

Figura 2.1 Arquitectura de la red neuronal

La red neuronal está conformada por 14 capas: la primera capa de la red neuronal corresponde a la de transferencia de aprendizaje, la cual fue implementada mediante la herramienta MobileNetV2, las siguientes capas son una de convolución, seguida de una de agrupamiento, luego de lo cual se tienen 5 capas densas y 4 capas de *dropout* intercaladas; y la última capa de la red neuronal define la cantidad de salidas por cada tipo de ficha Lego a predecir, que en este caso corresponde a 14 tipos.

La red neuronal dispone de las siguientes capas:

Se tiene primero la entrada de la red neuronal un arreglo de matrices RGB, cada matriz de dimensiones 224x224, esta entrada está definida de esta manera porque la primera capa corresponde a la transferencia de aprendizaje, y en esta capa solo se acepta este formato. Esta capa tiene como nombre 00_224 y fue implementada mediante la herramienta MobileNetV2.

La capa siguiente de nombre Conv2D es una capa de convolución que emplea un *kernel* de tamaño 3x3 con 32 tipos de filtros, con un *bias* de 32, lo que indica que está conectada a 32 entradas de la capa siguiente. Esta capa usa una función de activación ReLU.

La capa siguiente se denomina GlobalAveragePooling2D, y es una capa de agrupamiento.

La capa siguiente tiene por nombre BatchNormalization y sirve para normalizar a la salida de la capa previa usando valores entre 0 y 1, el beneficio que brinda este proceso es el de facilitar el proceso de aprendizaje. Esta normalización depende en parámetros estadísticos aprendidos y no aprendidos. En esta capa se pueden observar 4 parámetros los que son: *gamma*, *beta, moving mean* y *moving variance.* Estos parámetros sirven para hacer las normalizaciones que son el resultado de la salida de esta capa. *Gamma*, *beta* son dos vectores que se aprenden y que se van ajustando por cada época[12] del entrenamiento. Los parámetros *moving mean* y *moving variance* son vectores que contienen valores estadísticos que se calculan al inicio del entrenamiento y no entran en el proceso de aprendizaje.

Después de la capa de BatchNormalization se tienen las capas densas intercaladas con capas de *Dropout.* La primera capa densa emplea un *kernel* 32x128 lo que significa que esta capa tiene 32 entradas conectadas a 128 neuronas y un *bias* de 128 para ajustar la función de activación ReLU. La capa subsiguiente es una capa de *dropout.* La segunda

[12] Época: se denomina época cuando un conjunto de datos completo se pasa hacia adelante y hacia atrás a través de la red neuronal una vez.

capa densa emplea un *kernel* 128x64 lo que significa que esta capa tiene 128 entradas conectadas a 64 neuronas. A continuación de esta capa se tiene otra capa de *dropout*. La tercera capa densa emplea un *kernel* 64x64, esta capa tiene 64 neuronas con 64 salidas que se conectan a la capa siguiente, con un *bias* de 64 para ajustar los valores de salida que van a la siguiente capa. A continuación de esta capa se tiene otra capa de *dropout.* La cuarta capa, es una capa densa que emplea un *kernel* 64x32 lo que significa que esta capa tiene 64 entradas conectadas a 32 neuronas de la capa siguiente, con un *bias* de 32 para ajustar las salidas a la siguiente capa. Después de esta capa se tiene otra capa *dropout.* La última capa de la red neuronal es otra capa densa con 32 entradas que se conectan a 14 neuronas, con una función de activación Softmax que define las clases de salida de la red neuronal. Para esta red neuronal se definieron 14 clases de salida las cuales corresponden a los tipos de fichas Legos para las cuales el modelo podrá realizar la predicción. La salida de la red es un vector con 14 probabilidades, donde el elemento de ese vector con mayor valor corresponde al tipo de ficha Lego que el modelo predecirá.

2.3.2 Arquitectura del subsistema

El subsistema de clasificación es un *stub* que recibe imágenes del Subsistema de almacenamiento para realizar la clasificación de imágenes que contienen una sola ficha Lego; los resultados de la clasificación se envían al subsistema de almacenamiento. La clasificación calcula el tipo y color de una ficha Lego. El *stub* es una aplicación de consola escrita en el lenguaje de programación C# y que emplea la versión de .NET Framework 4.7.2. En la aplicación de consola están incluidos los métodos para hacer predicciones con el modelo que se implementó y almacenó. El modelo fue realizado con el lenguaje de programación Python con la ayuda de la librería TensorFlow y los modelos de Keras, el modelo fue almacenado en el formato nativo. En el subsistema se instaló los paquetes Nuget de Keras.NET, TensorFlow.NET, Numpy.NET para realizar las predicciones. Para el reconocimiento de los colores se usó la herramienta OpenCVSharp.

2.3.3 Diagramas de clase

Para la implementación del subsistema, el modelo de red neuronal implementado fue almacenado en el formato .H5, que es el tipo de modelo nativo de Keras.

Para el subsistema se definieron tres clases: `Program`, `RealizarPrediccion` y `ObtenerCentro`. En la Figura 2.2 se presenta el diagrama de clases.

Para poder utilizar el modelo se creó la clase `RealizarPrediccion` la cual contiene el método estático `Prediccion`. El resultado de este método es un vector con 14 elementos, cada elemento corresponde a una clase o tipo de ficha Lego específico, el valor de cada elemento corresponde a la probabiliad de que sea ese tipo de ficha Lego.

La clase `ObtenerCentro` dispone de metodos para realizar la detección de contornos con OpenCVSharp, una vez realizada la detección de contornos permite encontrar el centro de la ficha Lego, y el píxel central del cual se extrae su color y de esa manera determina el color de la ficha Lego.

La clase `Program` dispone de métodos para determinar el color de la ficha Lego.

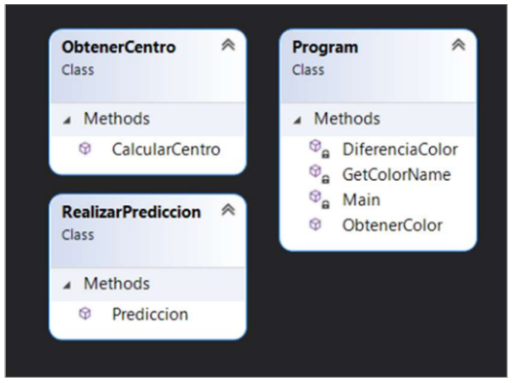

Figura 2.2 Diagrama de clases

2.4 Implementación del subsistema

En esta sección se presentará un resumen de los pasos que se usaron para la implementación del subsistema.

2.4.1 Preparación del set de datos con imágenes de fichas Lego

Después de realizada la limpieza de las imágenes a estas se las colocó en sus respectivas carpetas. La base de datos dispone de una cantidad aproximada de 5.800 imágenes. Estas imágenes de fichas Lego fueron divididas en 14 clases, considerando los siguientes tipos de ficha Lego: `lego_1x1`, `lego_1x1_Circular`, `lego_1x2`, `lego_1x2_Pendiente`, `lego_1x3`, `lego_2x2`, `lego_2x2_L`, `lego_2x2_Pendiente`, `lego_2x3`, `lego2x3_Plato`, `lego2x3_Pendiente`, `lego_2x4`, `lego2x4_Plato`,

`lego2x4_Pendiente`. Luego de lo cual se separaron las imágenes en 3 conjuntos: entrenamiento, validación y pruebas.

En la Figura 2.3 se aprecia el fragmento de código que permitió separar las imágenes en los 3 conjuntos. En la línea 1 se importó la librería `splitfolders`, en la línea 2 se incluyó la carpeta que contiene las imágenes y en la línea 6 se creó un directorio nuevo que contiene 3 subcarpetas, una para entrenamiento, una para validación y una para pruebas. El subconjunto de entrenamiento tiene el 80% de las imágenes, el de validación 10%, y el de pruebas el 10% restante.

```
1 import splitfolders
2
3 input_folder="Images"
4 #Se separa las imágenes con tasas de 80% para pruebas, 10% validación y 10%
5 para pruebas.
6 splitfolders.ratio(input_folder, output="Data2",
  seed=1337, ratio=(.8, .1, .1))
```

Figura 2.3 Segmento de código para dividir la base de datos de imágenes

Para comprobar la separación, se utilizó el segmento de código presentado en la Figura 2.4, el cual permite abrir una imagen en particular, transformarla a un arreglo mediante la librería Numpy y graficarla. El resultado de la imagen se la puede apreciar en la Figura 2.5.

```
1 from PIL import Image
2 import numpy as np
3 img = Image.open('Data2/train/Brick_1x1/3005.png')
4 img = np.array(img)
5 plot_image(img=img)
```

Figura 2.4 Segmento de código para presentar una imagen

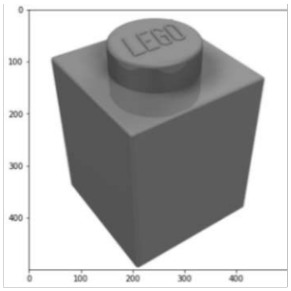

Figura 2.5 Resultado

Después de realizada la división de la base de datos de imágenes se realizó una preparación del conjunto de datos para su uso en la red neuronal.

Mediante Keras se procedió a crear el *dataset* para entrenamiento y para validación. En la Figura 2.6 se presenta un fragmento de código para preparar las imágenes con *data augmentation* (añadir distorsiones, rotaciones y desplazamientos) y crear el *dataset* para el entrenamiento del modelo con la carpeta de imágenes de entrenamiento.

```
1 ancho = 224
2 alto = 224
3 num_clases = 10
4 epochs = 50
5 batch_size = 32
6 train_datagen = tf.keras.preprocessing.image.ImageDataGenerator(
7     rescale=1/255.0,
8     rotation_range=20,
9     width_shift_range=0.2,
10    height_shift_range=0.2,
11    shear_range=0.2,
12    zoom_range=0.2,
13    horizontal_flip=True,
14    fill_mode='nearest'
15 )
16 train_data = train_datagen.flow_from_directory(
17    directory=train_path,
18    target_size=(ancho,alto),
19    batch_size=batch_size,
20    #save_to_dir='/content/drive/MyDrive/train_data',
21    class_mode='categorical'
22 )
```

Figura 2.6 Código para hacer data augmentation y detectar clases antes del entrenamiento

Las líneas de código de la 1 a la 15 del fragmento de código de la Figura 2.6 permite preparar las imágenes que van a conformar el *dataset* de entrenamiento. En las líneas 1 y 2 se ingresa el ancho y alto de los pixeles para redimensionar las imágenes a 224x244 pixeles. Mediante la variable `batch_size` se define la cantidad de imágenes por lote para los entrenamientos. En la línea 7, se normaliza el valor de cada píxel de datos para que sea menor a 1 debido a que así requiere la entrada de la red neuronal. En la línea 8 se añade rotación a las imágenes, en las líneas 9 y 10 se añaden desplazamientos verticales

y horizontales en las imágenes. En las líneas 11 y 12 se añaden distorsiones y acercamientos a las imágenes.

Por último, desde la línea 16 hasta la línea 22 se crea el *dataset*. En la línea 16 se ingresa el directorio con las imágenes de fichas Lego para el entrenamiento. Como en este directorio se encuentran las fichas Lego clasificadas por tipo en diferentes carpetas, se crean las clases dependiendo de la cantidad de subdirectorios para el proceso del entrenamiento. En la línea 17 se ingresa el directorio que contiene las fichas Lego de entrenamiento, en la línea 18 se ingresan las dimensiones de las imágenes. En la línea 19 se define la cantidad de imágenes por lote que se van a entrenar, en este caso son 32 imágenes, según lo definido en la variable `batch_size`.

En la línea 21 se define un parámetro con `class_mode='categorical'`. Este parámetro se utiliza cuando se tienen más de dos clases a entrenar

Cuando ya se ha realizado la creación del *dataset* de entrenamiento, se continúa de la misma manera con los datos de validación, pero sin agregar distorsiones, desplazamientos o acercamientos. El código de este proceso se lo presenta en la Figura 2.7.

```
1 valid_data = valid_datagen.flow_from_directory(
2     directory='/content/drive/MyDrive/Data/val/',
3     target_size=(224, 224),
4     class_mode='categorical',
5     batch_size=64,
6     seed=42
7 )
```

Figura 2.7 Generación de datos de validación

2.4.2 Implementación de la red neuronal

En esta sección se describen los pasos que se siguieron para implementar la red neuronal.

Para crear la red neuronal se utilizaron varias librerías de Python, las cuales se importaron como se indica en el segmento de código de la Figura 2.8.

```
1  import numpy as np
2  import tensorflow as tf
3  from tensorflow.keras import layers
4  import matplotlib.pyplot as plt
5  from PIL import Image
6  import numpy as np
7  from tensorflow.keras import regularizers
8  from tensorflow.keras.callbacks import ModelCheckpoint, EarlyStopping,
   ReduceLROnPlateau
9  from tensorflow.keras import regularizers
10 from tensorflow.keras.preprocessing import image
```

Figura 2.8 Librerías utilizadas

En el segmento de código de la Figura 2.9 se importa la red neuronal MobileNetV2 y se le da un nombre, en este caso `base_model`. En la línea 2 se definen los pesos preexistentes de la red MobileNetV2, los cuales tienen el nombre `imageset`, estos pesos son los que se recomienda en la documentación para imágenes. Luego, en la línea 4 se procede a descongelar las últimas 26 capas de esta red.

```
1 base1_model = tf.keras.applications.MobileNetV2(input_shape=IMG_SHAPE,
2 include_top=False, weights='imagenet')
3
4 for layer in base1_model.layers[:-26]:
5     layer.trainable = False
```

Figura 2.9 Segmento de código para la preparación de la capa de transferencia MobileNetV2

En el segmento de código de la Figura 2.10 se presenta la implementación de la red neuronal.

El proceso comienza con el establecimiento del tipo del modelo (secuencial), este modelo funciona como un arreglo de capas que se van agregando una a continuación de otra. La primera capa que se añadió fue la de MobileNetV2. En la línea 2 se puede observar el nombre de la primera capa, `base_model`, en esta capa se implementa la transferencia de aprendizaje

En la línea 3 se define la capa de convolución. En esta capa se estableció 32 filtros y el tamaño del *kernel* de 3x3. En la línea 5 se define la capa de agrupamiento. En la línea 8 se agregó una capa densa con 128 neuronas, con una función de activación ReLU,

regularizadores de *kernel*, los cuales penalizan la taza de entrenamiento al existir sobre entrenamiento. Se añadieron capas de *dropout* también sirven para solventar este problema. Estas capas desconectan neuronas aleatoriamente de las capas densas, se puede observar que en las líneas 12 y 14 se tiene una probabilidad de desconexión de una neurona del 40%, en las líneas 16, 18 y 20 se definió una probabilidad de desconección del 30%. Por último, en la línea 21 se definió una capa densa con 14 neuronas las cuales representan las clases de fichas Lego definidas en la sección 2.2.1.

```
 1 model_7 = tf.keras.Sequential([
 2     base1_model,
 3     tf.keras.layers.Conv2D(filters=32, kernel_size=(3, 3),
 4 activation='relu'),
 5     tf.keras.layers.GlobalAveragePooling2D(),
 6     tf.keras.layers.BatchNormalization(),
 7     #tf.keras.layers.GlobalAveragePooling2D(),
 8     tf.keras.layers.Dense(128,
 9 activation='relu',kernel_regularizer=regularizers.l1_l2(l1=1e-5, l2=1e-4),
10     bias_regularizer=regularizers.l2(1e-4),
11     activity_regularizer=regularizers.l2(1e-5)),
12     tf.keras.layers.Dropout(rate=0.4),
13     tf.keras.layers.Dense(128, activation='relu'),
14     tf.keras.layers.Dropout(rate=0.4),
15     tf.keras.layers.Dense(64),
16     tf.keras.layers.Dropout(rate=0.3),
17     tf.keras.layers.Dense(64),
18     tf.keras.layers.Dropout(rate=0.3),
19     tf.keras.layers.Dense(32),
20     tf.keras.layers.Dropout(rate=0.3),
21     tf.keras.layers.Dense(14, activation='softmax'),])
```

Figura 2.10 Código para la Implementación de la red neuronal

Después de definir la red neuronal se la compila para poder realizar el proceso de entrenamiento. En el código de la Figura 2.11, entre las líneas 3 y 4 se definió el optimizador con la tasa de aprendizaje y la métrica de precisión (`accuracy`)- La tasa de aprendizaje fue de 0.00015, se tuvo esta tasa porque se utilizó transferencia de aprendizaje, y por tanto no se necesita una tasa alta porque la red MobileNetV2 fue previamente entrenada.

En el código de la Figura 2.12 se crean dos funciones de *callbacks*[13]: `ReduceLROnPlateau` en la línea 7 y `early_stopping` en la línea 9. `ReduceLROnPlateau` se utiliza para definir una tasa de aprendizaje variable, que disminuye a una tasa de 0.5 con el parámetro `patience` con valor 2, este parámetro hace que se implemente penalización en la tasa de aprendizaje que se reduce a la mitad cada dos épocas si la métrica `val_accuracy`[14] no mejora. En la línea 9 se define `early_stopping`, que sirve para parar el entrenamiento si el modelo no mejora.

```
1 model_7.compile(
2     loss=tf.keras.losses.categorical_crossentropy,
3     optimizer=tf.keras.optimizers.Adam(learning_rate=0.00015),
4     metrics=['accuracy']
5 )
```

Figura 2.11 Código para compilar la red neuronal

```
1 reduce_lr = ReduceLROnPlateau(monitor='val_accuracy',
2 factor=0.5, patience=2, verbose=2,
3 mode='max', min_lr=0.00015)
4 checkpoint =
5 ModelCheckpoint(filepath,
6 verbose=2, save_best_only=True)
7 reduce_lr = ReduceLROnPlateau(monitor='val_accuracy',
8 factor=0.5, patience=2, verbose=2, mode='max', min_lr=0.00015)
9 early_stopping=tf.keras.callbacks.EarlyStopping(monitor='val_loss',
10 patience=3)
```

Figura 2.12 Código para definir los callbacks

En el segmento de código de la Figura 2.13 se entrena la red con los *datasets* de entrenamiento en la línea 2, en la línea 3 se agrega el *dataset* de validación, en la línea 4 se definen 30 épocas de entrenamiento. En la línea 5 se llama a los *callbacks.*

[13] Un *callback* es un objeto que puede realizar acciones en varias etapas de entrenamiento, los *callbacks* son de ayuda porque permiten obtener el mejor modelo posible en la fase del entrenamiento.

[14] `val_accuracy`(precisión de validación): la precisión de validación se calcula en el conjunto de datos que no se usan en el entrenamiento, la validación de precisión sirve para probar la capacidad de generalización de un modelo o para la detención temprana del entrenamiento (`early_stopping`).

```
history_7 = model_7.fit(
    train_data,
    validation_data=valid_data,
    epochs=30,
    callbacks=[reduce_lr, early_stopping, checkpoint]
)
```

Figura 2.13 Código para entrenar la red neuronal

Al final de este proceso se guarda el modelo para poder usarlo en el subsistema. En la Figura 2.14 se realiza este proceso.

```
model_7=tf.keras.models.load_model('/content/drive/MyDrive/model7.h5')
```

Figura 2.14 Código para guardar modelo en formato de Keras

2.4.3 Creación del proyecto e instalación de paquetes necesarios

Para poder usar el modelo mediante una aplicación en .NET Framework en Visual Studio se creó un proyecto de consola (ver Figura 2.15).

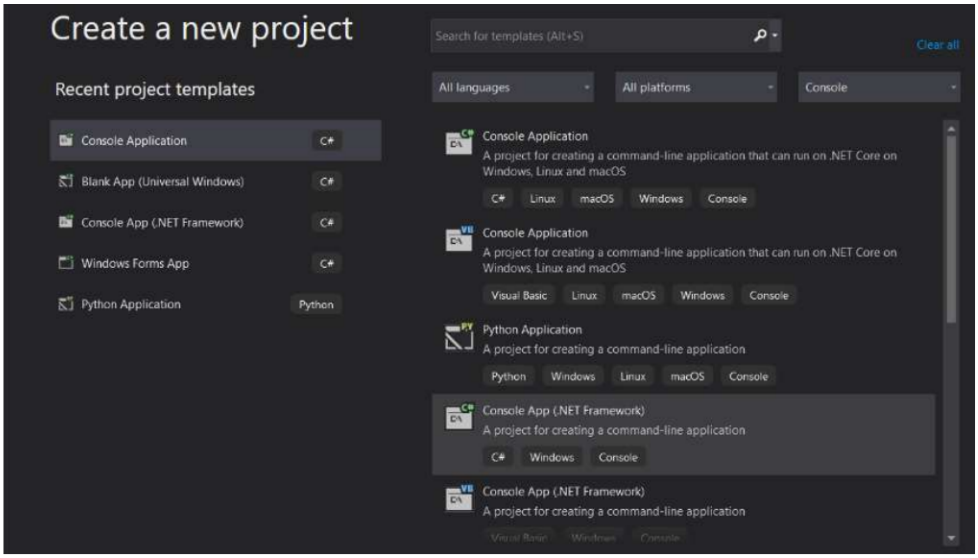

Figura 2.15 Creación de proyecto en Visual Studio

Como parte de la creación del proyecto, se debe colocar un nombre para el mismo y se debe indicar la versión 4.7.2 de .NET Framework (ver Figura 2.16).

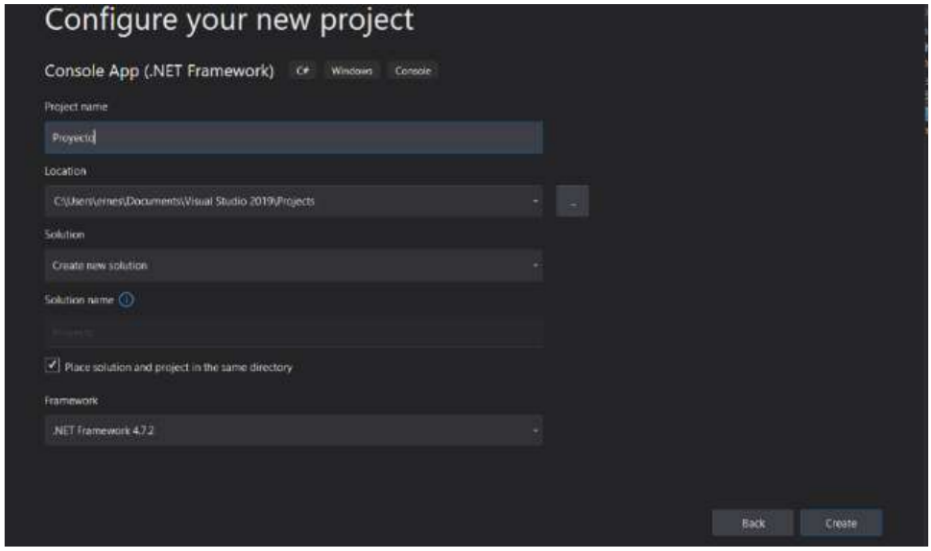

Figura 2.16 Configuración del proyecto

Después crear el proyecto se procede a definir un directorio denominado assets, en el que se colocará el modelo de red neuronal (ver Figura 2.17).

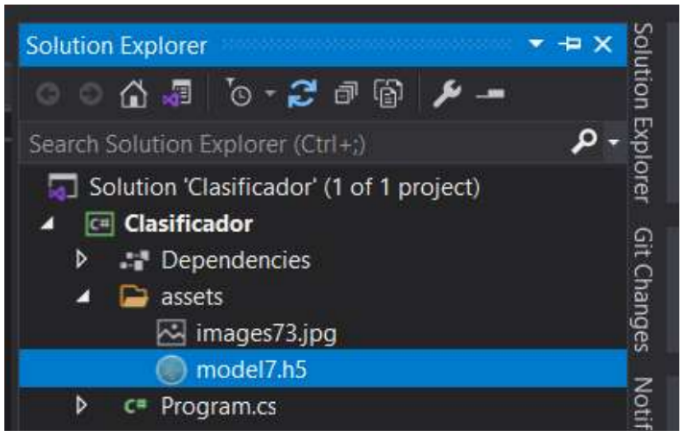

Figura 2.17 Estructura del proyecto

El proyecto para funcionar necesita los paquetes denominados: Tensorflow.Net, Keras.Net, Numsharp, Numpy.Net. Para instalar los paquetes mediante el explorador de soluciones, en el nombre del proyecto se debe hacer clic derecho y escoger la opción *Manage Nuget Packages* para agregar los paquetes NuGet (ver Figura 2.18).

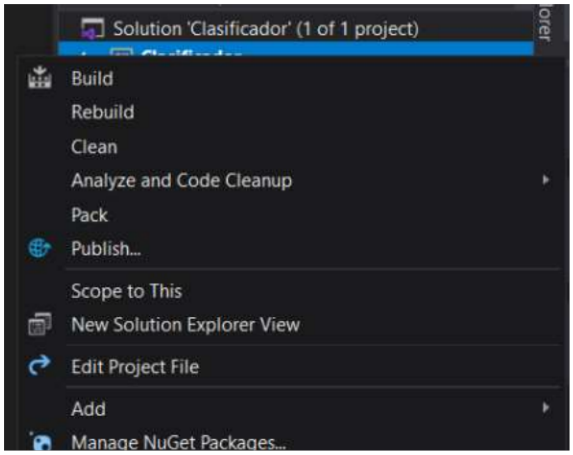

Figura 2.18 Agregar paquetes Nuget

Posteriormente se presentará la ventana del explorador de paquetes Nuget, como se aprecia en la Figura 2.19. En el explorador, en la pestaña Buscar se ingresa el nombre del paquete y se presiona en el botón instalar. Este proceso debe realizarse con cada paquete requerido.

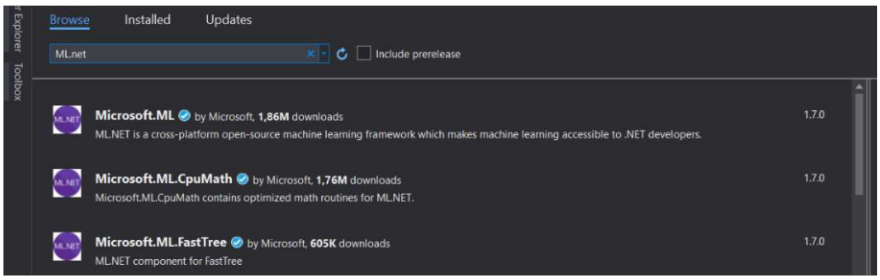

Figura 2.19 Instalación de paquetes NuGet

2.4.4 Importación del modelo en .NET Framework

Para emplear el modelo en .NET Framework fue necesario importar las librerías Nuget: NumSharp, Pythonnet, OpenCvSharp4, y Keras.Net, como se ve en la Figura 2.20.

```
1 using Keras.PreProcessing.Image;
2 using System;
3 using Numpy;
4 using System.Drawing;
5 using System.Linq;
```

Figura 2.20 Librerías necesarias para usar el modelo en .NET Framework

Para probar el funcionamiento del modelo, se debe cargar una imagen, la imagen debe ser convertida en una matriz mediante Numpy, y luego debe ser redimensionada a 224x224 píxeles de alto y ancho (ver Figura 2.21).

```
1 //Cargar imagen y transformar a numpy array
2             Image imagen = Image.FromFile(path);
3             var img = ImageUtil.LoadImg(path, target_size: (224, 224));
4             var x = ImageUtil.ImageToArray(img)/255.0;
```

Figura 2.21 Cargar imagen y transformar a matriz

Para cargar el modelo se usa el código presentado en la Figura 2.22, en la línea 1 se inserta una nueva dimensión al arreglo definido en el código de la Figura 2.2. En la línea 2 se carga el modelo previamente creado, el cual fue colocado en el directorio `assets`. En la línea 3 se realiza la predicción del modelo, para lo cual se pasa como argumento la imagen, el resultado es un vector que contiene 14 valores con probabilidades de que la imagen pertenezca a un tipo específico de Lego, la probabilidad mayor es el tipo de ficha Lego resultante.

```
1 x = np.expand_dims(x, axis: 0);
2 var model = Keras.Models.Model.LoadModel(@"assets/model7.h5");
3 var preds = model.Predict(x);
```

Figura 2.22 Convertir imagen a matriz y crear predicción

En el fragmento de código de la Figura 2.23 se realiza el procesamiento de la predicción. En la línea 1 se creó un arreglo de nombre `labels` con los tipos de fichas Lego. En la línea 6 se obtiene el tipo de ficha Lego cuya probabilidad de clase fue la más alta. En la línea 7 se procesa la cadena que contiene el resultado, separando en caso de ser necesario el factor y forma de la ficha Lego.; por ejemplo, si la predicción dio como resultadoes `lego_2x2_L`, se separa: Lego, factor 2x2 y forma L.

```
String[] labels = {"lego_1x1","lego_1x1_Circular","lego_1x2",
"lego_1x2_Pendiente", "lego_1x3", "lego_1x3", "lego_2x2", "lego_2x2_L",
"lego_2x2_Pendiente",
"lego_2x3", "lego_2x3_Plato","lego_2x3_Pendiente",
"lego_2x4", "lego_2x4_Plato","lego_2x4_Pendiente"};
string aux = labels[(int)np.argmax(preds)];
string[] resultado = aux.Split("_");

if (resultado.Length==2)
        {
        Console.WriteLine("El resultado del modelo fue un lego rectangular
 con factor de: "+resultado[1]);
        }
else
        {
                Console.WriteLine("El resultado del modelo fue un lego
con factor de: "+resultado[1]);
        Console.WriteLine("La caracteristica del lego es: "+resultado[2]);
}

```

Figura 2.23 Procesar la predicción

2.4.5 Reconocimiento de color en .NET Framework

Para realizar el reconocimiento del color de la ficha Lego se realizó el proceso que se representa en la Figura 2.24.

Figura 2.24 Proceso para obtener el color de la ficha Lego

En el código de la Figura 2.25 se presenta el método `CalcularCentro` que devuelve el centro del objeto Lego que se encuentra en una imagen. En la línea 5 y 6 se definen las variables que almacenarán los valores de las coordenadas del centro de la ficha Lego. En la línea 7 se lee la imagen que será procesada. En la línea 8 se transforma la imagen a escala de grises. En la línea 10 se aplica un filtro Gaussiano de 5x5. En la línea 14 se calculan los contornos con el algoritmo de Canny. Después de calcular los contornos entre las líneas 18 a 36 se calculan las coordenadas del centro del objeto Lego iterando entre todos los puntos del contorno.

Una vez obtenido el punto central, mediante el código de la Figura 2.26 se obtiene el color de dicho punto, para lo cual se emplearon 2 métodos, el primer método denominado `ObtenerColor`, el cual se observa en la línea 15, este acepta como argumento de entrada un objeto de tipo `Color`, el cual está conformado por los valores RGB de un punto en particular. Como los valores RGB no son exactos a los valores de color que se tiene en el arreglo de colores, se creó un método de nombre `DiferenciaColor` (línea 1), el cual sirve para sacar la diferencia entre el tono y la saturación entre el color del arreglo y el valor de RGB obtenido, con el valor de esta diferencia se selecciona el color más cercano definido en el arreglo de colores.

```
1  public static int[] CalcularCentro(string ImagePath)
2  {
3      //código para obtener
4      //el centro de un contorno
5      int cx = 0, cy = 0;
6      int[] centro = { 0, 0 };
7      Mat image = Cv2.ImRead(ImagePath);
8      image = image.CvtColor(ColorConversionCodes.BGR2GRAY);
9      Mat blurred = new Mat();
10     Cv2.GaussianBlur(image, blurred, new OpenCvSharp.Size(5, 5), 0);
11     Mat canny = new Mat();
12     //Usamos el proceso de detección de bordes de Canny
13     Cv2.Canny(blurred, canny, 30, 300);
14     Point[][] contour;
15     HierarchyIndex[] hierarchyIndexes;
16     // se busca el contorno
17     // y se lo guarda en forma de coordenadas
18     Cv2.FindContours(canny, out contour,
19     out hierarchyIndexes, RetrievalModes.External,
20     ContourApproximationModes.ApproxSimple);
21     foreach (var c in contour)
22     {
23         Moments M = Cv2.Moments(c);
24         if (M.M00!=0)
25         {
26                cx=(int)(M.M10/M.M00);
27                cy=(int)(M.M01/M.M00);
28         }
29         else
30         {
31                cx=0;
32                cy=0;
33         }
34      }
35             centro[0]=cx;
36             centro[1]=cy;
37             return centro;
38 }
```

Figura 2.25 Método que retorna la matriz de color al ingresar mapa de bits

```
1 private static int DiferenciaColor(Color c1, Color c2)
2 {
3     int distance;
4     if (c2.GetSaturation() < 0.1)
5     {
6         distance = (int)Math.Abs(c1.GetSaturation()
7             -c2.GetSaturation());
8    }
9    else
10   {
11       distance = (int)Math.Abs(c1.GetHue()- c2.GetHue());
12   }
13   return distance;
14 }
15 public static Color ObtenerColor(Color colorBase)
16         {
17     Color[] colores =
18     {
19         Color.Red,
20         Color.Black,
21         Color.White,
22         Color.Blue,
23         Color.Yellow,
24         Color.Green,
25         Color.Gray,
26         Color.Brown,
27         Color.SkyBlue,
28         Color.Orange
29     };
30
31   var colors = colores.Select(x => new { Value = x,
32       Diff = DiferenciaColor(x, colorBase) }).ToList();
33   var min = colors.Min(x => x.Diff);
34   return colors.Find(x => x.Diff == min).Value;
35 }
```

Figura 2.26 Fragmento de código para obtener el nombre del color

3 RESULTADOS, CONCLUSIONES Y RECOMENDACIONES

En esta sección se presentan los resultados obtenidos en las pruebas realizadas, así como las conclusiones y recomendaciones producto de este Trabajo de Integración Curricular

3.4 Resultados

En la Figura 3.1 se pueden observar los resultados del entrenamiento del último modelo creado, así como se presentan las métricas obtenidas. Para este caso, se puede observar que el entrenamiento llegó hasta la época 14 ya que los *callbacks* de la fase de implementación hizo que se detenga el entrenamiento al no haber mejora.

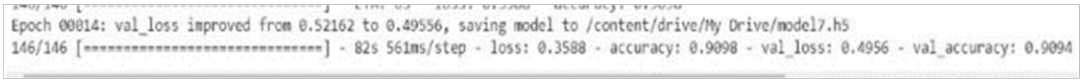

Figura 3.1 Resultado del entrenamiento de la red neuronal

En la Tabla 3.1 se presenta un resumen de las métricas obtenidas en el proceso de entrenamiento y validación. Se definieron dos métricas: pérdida (`loss`) y precisión (`accuracy`). La métrica de precisión describe exactamente qué porcentaje de los datos de prueba se clasifican correctamente, mientras que la métrica de pérdida describe el porcentaje de predicciones erradas.

Tabla 3.1 Resultados de las métricas de los modelos generados

Modelo	Loss	accuracy
model_4	1.7874	0.833
model_5	2.2535	0.9285
model_6	0.6971	0.971
model_7	0.3588	0.9098

De los 4 modelos, se empleó el denominado `model_7` dados los resultados obtenidos en validación, luego de lo cual empleando este modelo se realizó la clasificación de las

imágenes del *dataset* de pruebas. Para presentar los resultados del modelo se creó un método que realiza predicciones usando todas las imágenes de prueba. Se determinó la eficiencia considerando las predicciones obtenidas versus la clase de cada imagen. En la Tabla 3.2 se pueden observar los resultados de las pruebas del modelo denominado `model_7`, con el cual se obtuvo una eficiencia de predicción del 86%.

Tabla 30.2 Eficiencia de pruebas de predicción

Modelo	**model_7**
clase 0	93.15%
clase 1	97.44%
clase 2	90.74%
clase 3	82.35%
clase 4	95.00%
clase 5	92.50%
clase 6	87.18%
clase 7	81.18%
clase 8	92.75%
clase 9	88.24%
clase 10	62.50%
clase 11	88.89%
clase 12	93.75%
clase 13	67.86%

Para descartar sobre entrenamiento en la red neuronal implementada se usaron *callbacks* que detectan el valor de pérdida y pérdida de validación[15]. Con la ayuda de los *callbacks* se detiene el entrenamiento y se guarda el modelo generado si se determina que no hubo mejoras en los valores de las métricas. En la Figura 3.3 se puede observar cómo cambia el valor de pérdida con respecto a las épocas. Se puede observar como el valor de pérdida del entrenamiento va disminuyendo hasta la época 12, pero como la perdida de validación

[15] `val_loss`: pérdida de validación, indica los valores de pérdida calculados en datos desconocidos

comienza a aumentar desde aproximadamente la época 9 se ve que hay sobre entrenamiento y por los tanto se detiene este proceso.

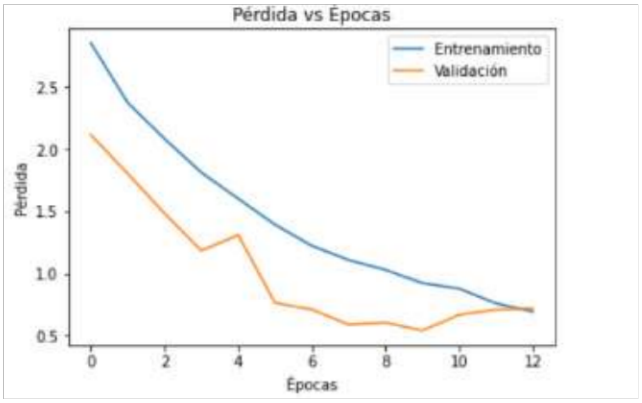

Figura 3.2 Gráfica de pérdida en función de las épocas

3.1.1 Resultados de la detección de contornos

En la Figura 3.3 se observa el resultado de obtener bordes en una imagen en particular:

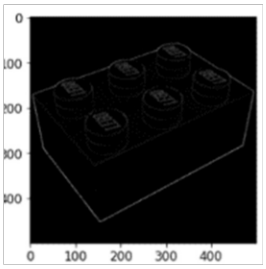

Figura 3.3 Detección de Bordes de una imagen de ficha Lego

En la Figura 3.4 se presenta el resultado de la obtención del contorno de la ficha Lego.

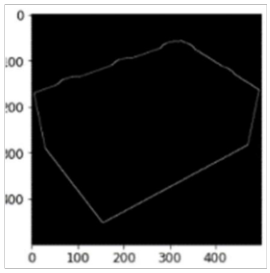

Figura 3.4 Contorno de la ficha Lego

En la Figura 3.5 se puede observar el resultado de la obtención del punto central.

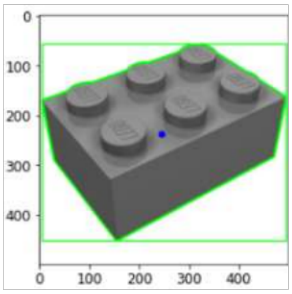

Figura 3.5 Ficha original con el centro calculado y dibujado

3.1.2 Resultados del Subsistema

El subsistema emplea el `model_7` entrenado, así como los métodos para determinar el color de la ficha Lego. Para probar el subsistema se empleó la ficha Lego que se presenta en la Figura 3.6 y el resultado de su clasificación se muestran en la Figura 3.7.

Figura 2.6 Ficha Lego de forma rectangular de factor 2x2

```
To enable them in other operations, rebuild TensorFlow with the appropriate compiler flags.
1/1 [==============================] - 1s 1s/step
Color [A=255, R=72, G=72, B=72]
El color resultante de la clasificación del lego es Black


El resultado del modelo fue un lego rectangular con factor de: 2x2
```

Figura 3.7 Resultados del subsistema de clasificación

Se puede observar que, el subsistema clasificó la imagen indicando que contiene una ficha Lego de color negro con 2x2, con forma rectangular, lo cual coincide con lo presentado en la Figura 3.6.

En la Figura 3.8 y en la Figura 3.9 se presenta otro proceso de clasificación

Figura 3.8 Ficha Lego de forma rectangular de factor 2x2 de color azul

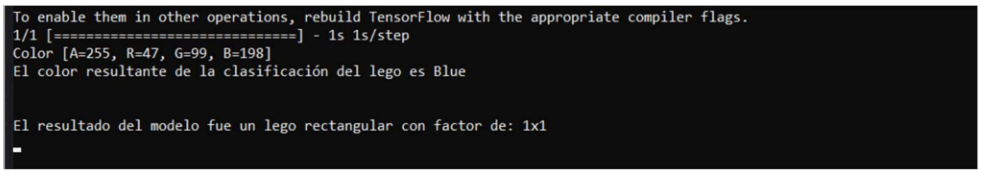

Figura 3.9 Ficha Lego de forma rectangular de factor 1x1 de color azul

Como se aprecia en la Figura 3.9 el subsistema clasificó el contenido de la imagen como una ficha Lego de color azul con factor 1x1 de forma rectangular, lo cual coincide con lo presentado en la Figura 3.8.

3.2 Conclusiones

- En el presente trabajo de Integración Curricular se desarrolló un subsistema de clasificación de fichas Lego por tipo y color. La clasificación funciona con imágenes de formato JPG con una sola ficha Lego y de un solo color.
- Para realizar la clasificación de fichas Lego por tipo se desarrolló un modelo de red neuronal convolucional con transferencia de aprendizaje mediante la herramienta MobileNetV2 la cual se la descargó y se la añadió al modelo secuencial que se estaba desarrollando para mejorar los resultados de la clasificación.

- Para reconocer los colores de la ficha Lego se implementó con la ayuda del paquete Nuget OpenCVSharp. El primer paso fue el de usar un método presente en este paquete de detección de contornos con el algoritmo Canny para localizar la ficha Lego, luego del cual se pudo obtener un píxel para obtener los colores y de ahí identificar el color aproximado.

- El subsistema de clasificación implementa un stub en .NET Framework con el lenguaje de programación C# el cual simula la clasificación de imágenes tipo lego que son recibidas desde el subsistema de adquisición. La clasificación cuenta de dos partes, la primera es la de la aplicación del modelo que fue creado y almacenado en Python y la segunda del reconocimiento de color con la herramienta de OpenCVSharp. El subsistema después de clasificar la imagen por tipo y color envía los resultados al subsistema de almacenamiento.

- En el proceso de implementación del subsistema se crearon 7 modelos, por tanto, para seleccionar el mejor se realizaron pruebas realizadas con la base de datos de imágenes de prueba y en el modelo que se implementó se obtuvo una eficiencia del 86%. Las pruebas consistieron en hacer predicciones para cada imagen de ficha Lego presente en el *dataset* y sacar el porcentaje de eficiencia al dividir las predicciones cantidad de predicciones acertadas para la cantidad de predicciones realizadas.

3.3 Recomendaciones

- Se recomienda crear un modelo de red neuronal que sea capaz de reconocer varios tipos de fichas Lego en una sola imagen porque este modelo solo acepta imágenes con una única ficha Lego.

- Se recomienda utilizar la librería Microsoft.ML que permite crear modelos directamente en Visual Studio con ambientes .NET.

- Se recomienda utilizar la librería Microsoft.ML para modelos de diferente formato en comparación al empleado en este subsistema, por ejemplo se puede utilizar con modelos ONNX(*Open Neural Network Exchange*) el cual es un formato multiplataforma desarrollado por Microsoft. Este formato tiene la ventaja que el cual tiene mejor rendimiento al hacer predicciones comparándolo a modelos H5 en ambientes .NET

- Se recomienda crear un modelo que sea capaz de reconocer más tipos de fichas Lego.
- Se recomienda usar otras arquitecturas de redes de transferencia como ResNet, Inception para comparar resultados de clasificación con la red MobileNetV2.
- Se recomienda utilizar OpenCV para poder reconocer colores en imágenes con multiples objetos de tipo Lego.
- En futuros trabajos se recomienda utilizar la herramienta de TensorFlow con TFLite para implementar modelos de reconocimiento de imágenes en aplicaciones móviles

REFERENCIAS BIBLIOGRÁFICAS

[1] «Deep Lizzard,» 28 9 2020. [En línea]. Available: https://deeplizard.com/learn/video/Zrt76AIbeh4. [Último acceso: 21 12 2021].

[2] H. Jiang, «Machine Learning Fundamentals - A concise Introduction,» York, Cambridge University Press, pp. 151-154.

[3] D. J. Matich. [En línea]. Available: https://www.frro.utn.edu.ar/repositorio/catedras/quimica/5_anio/orientadora1/monograias/matich-redesneuronales.pdf. [Último acceso: 8 1 2022].

[4] M. Chaudhary, «Medium,» [En línea]. Available: https://medium.com/@cmukesh8688/activation-functions-sigmoid-tanh-relu-leaky-relu-softmax-50d3778dcea5. [Último acceso: 26 Febrero 2022].

[5] U. M. J. M. L. M. G. J. S. Dan C. Cires¸an, «Flexible, High Performance Convolutional,» Proceedings of the Twenty-Second International Joint Conference on Artificial Intelligence, Galleria 2, 6928 Manno-Lugano, Switzerland, 2010.

[6] D. Radečić, «Towards Data Science,» 20 11 2021. [En línea]. Available: https://towardsdatascience.com/tensorflow-for-computer-vision-how-to-implement-convolutions-from-scratch-in-python-609158c24f82. [Último acceso: 22 12 2021].

[7] J. Brownlee, «Machine Learning Mastery,» 22 Abril 2019. [En línea]. Available: https://machinelearningmastery.com/pooling-layers-for-convolutional-neural-networks/. [Último acceso: 8 enero 2022].

[8] [En línea]. Available: https://programmerclick.com/article/2420177865/. [Último acceso: 8 Enero 2022].

[9] Y. Verma, 19 9 2021. [En línea]. Available: https://analyticsindiamag.com/a-complete-understanding-of-dense-layers-in-neural-networks/. [Último acceso: 8 1 2022].

[10] J. Brownlee, «Machine Learning Mastery,» 16 Septiembre 2019. [En línea]. Available: https://machinelearningmastery.com/transfer-learning-for-deep-learning/. [Último acceso: 2022 1 7].

[11] MathWorks, [En línea]. Available: https://www.mathworks.com/discovery/edge-detection.html#:~:text=Edge%20detection%20is%20an%20image,computer%20vision%2C%20and%20machine%20vision.. [Último acceso: 12 01 2022].

[12] OpenCV, [En línea]. Available: https://docs.opencv.org/4.x/da/d22/tutorial_py_canny.html. [Último acceso: 12 Enero 2022].

[13] «Byte of Python,» [En línea]. Available: https://python.swaroopch.com/about_python.html. [Último acceso: 7 1 2022].

[14] Numpy Developers, «Numpy,» [En línea]. Available: https://numpy.org/doc/stable/user/whatisnumpy.html. [Último acceso: 15 Febrero 2022].

[15] «Tensorflow,» [En línea]. Available: https://www.tensorflow.org/learn. [Último acceso: 8 Enero 20221].

[16] «Digital Guide Ionos,» 10 Agosto 2020. [En línea]. Available: https://www.ionos.es/digitalguide/online-marketing/marketing-para-motores-de-busqueda/que-es-keras/. [Último acceso: 8 Enero 2022].

[17] «Keras,» [En línea]. Available: https://keras.io/about/. [Último acceso: 8 Enero 2022].

[18] Keras, «Keras,» [En línea]. Available: https://keras.io/why_keras/#:~:text=Keras%20follows%20best%20practices%20for,learn%20and%20easy%20to%20use.. [Último acceso: 22 Enero 2022].

[19] A. G. Z. M. C. B. K. D. W. W. W. T. .. &. A. H. Howard, «Mobilenets: Efficient convolutional neural networks for mobile vision applications.,» 2017. [En línea]. Available: arXiv:1704.04861. [Último acceso: 2 1 2022].

[20] Microsoft, [En línea]. Available: https://docs.microsoft.com/es-es/dotnet/framework/get-started/. [Último acceso: 8 01 2022].

[21] Microsoft, [En línea]. Available: https://docs.microsoft.com/en-us/visualstudio/get-started/visual-studio-ide?view=vs-2019. [Último acceso: 8 Enero 2022].

[22] «Tensorflow.NET,» [En línea]. Available: https://tensorflownet.readthedocs.io/en/latest/HelloWorld.html. [Último acceso: 21 enero 2022].

[23] «Github,» [En línea]. Available: https://github.com/SciSharp/Numpy.NET. [Último acceso: 21 enero 2022].

[24] «GitHUb,» [En línea]. Available: https://github.com/SciSharp/Keras.NET. [Último acceso: 21 enero 2022].

[25] «OpenCV,» [En línea]. Available: https://docs.opencv.org/4.x/d1/dfb/intro.html. [Último acceso: 5 enero 2022].

[26] J. Edge, «Kanban, La guía definitiva de la metodología Kanban para el desarrollo de software ágil,» Scribd, pp. 6-7,46-48.

[27] F. García, «Kaggle,» 2019. [En línea]. Available: https://www.kaggle.com/pacogarciam3/lego-brick-sorting-image-recognition?select=ImageSetKey.csv. [Último acceso: 6 12 2021].

[28] Deeplizard, «https://deeplizard.com/learn/video/Zrt76AIbeh4,» [En línea]. Available: https://deeplizard.com/learn/video/Zrt76AIbeh4. [Último acceso: 2021 12 21].

[29] Microsoft, [En línea]. Available: https://docs.microsoft.com/en-us/azure/machine-learning/concept-deep-learning-vs-machine-learning. [Último acceso: 4 01 2021].

[30] L. Carvajal, Metodología de la Investgación Científica. Curso general y aplicado, 28 ed., Santiago de Cali: U.S.C., 2006, p. 139.

ANEXOS

ANEXO A. Código

ANEXO A: Código

El código generado se encuentra disponible en el repositorio de GitHub:

https://github.com/ernestogt/clasificadorLego.git