Tecnológico de Costa Rica

Escuela de Ingeniería Electrónica

Optimización de redes neuronales para ejecución eficiente de aplicaciones de inteligencia artificial en GPUs embebidos.

Documento de tesis sometido a consideración para optar por el grado académico de Maestría en Electrónica con Énfasis en Sistemas Embebidos

Oscar Segura Quesada

Agosto de 2021

Declaro que el presente documento de tesis ha sido realizado enteramente por mi persona, utilizando y aplicando literatura referente al tema e introduciendo conocimientos y resultados experimentales propios.

En los casos en que he utilizado bibliografía he procedido a indicar las fuentes mediante las respectivas citas bibliográficas. En consecuencia, asumo la responsabilidad total por el trabajo de tesis realizado y por el contenido del presente documento.

Oscar Segura Quesada

Agosto de 2021

Céd: 1-1208-0575

Instituto Tecnológico de Costa Rica
Escuela de Ingeniería Electrónica
Maestría Académica en Electrónica
Trabajo Final de Graduación
Tribunal Evaluador
Acta de Aprobación

Defensa del Trabajo Final de Graduación
Requisito para optar por el título de Máster en Ingeniería Electrónica
Grado Académico de Magister Scientiae

El Tribunal Evaluador aprueba la defensa del Trabajo Final de Graduación denominado "Optimización de redes neuronales para ejecución eficiente de aplicaciones de inteligencia artificial en GPUs embebidos", realizado por el señor **Oscar Segura Quesada** Carné: **200236803**, y hace constar que cumple con las normas establecidas por la Unidad Interna de Posgrados de la Escuela de Ingeniería Electrónica del Instituto Tecnológico de Costa Rica.

Miembros del Tribunal Evaluador

M.Sc. María Haydeé Rodríguez Blanco
Profesora Lectora

Dr. Ing. Pablo Mendoza Ponce
Profesor Lector

Dr. Ing. Miguel Ángel Aguilar Ulloa
Director de tesis

Dra. Ing. Laura Cabrera Quirós
Evaluadora Independiente

Dr. Ing. Johan Carvajal Godínez
Coordinador de la Maestría

Cartago, 23 de Agosto de 2021

Instituto Tecnológico de Costa Rica
Escuela de Ingeniería Electrónica
Maestría Académica en Electrónica
Trabajo Final de Graduación
Acta de Evaluación

Defensa del Trabajo Final de Graduación
Requisito para optar por el título de Máster en Ingeniería Electrónica
Grado Académico de Magister Scientiae

Estudiante: Oscar Segura Quesada
Carné: 200236803

Nombre del proyecto:" Optimización de redes neuronales para ejecución eficiente de aplicaciones de inteligencia artificial en GPUs embebidos"

Los miembros de este Tribunal hacen constar que este trabajo final de graduación ha sido aprobado y cumple con las normas establecidas por la Unidad Interna de Posgrados de la Escuela de Ingeniería Electrónica del Instituto Tecnológico de Costa Rica y es merecedor de la siguiente calificación: **95**

Miembros del Tribunal Evaluador

M.Sc. María Haydeé Rodríguez Blanco
Profesora Lectora

Dr. Ing. Pablo Mendoza Ponce
Profesor Lector

Dr. Ing. Miguel Angel Aguilar Ulloa
Director de tesis

Dra. Ing. Laura Cabrera Quirós
Evaluadora Independiente

Dr. Ing. Johan Carvajal Godínez
Coordinador de la Maestría

Cartago, 23 de Agosto de 2021

Resumen

El área de la inteligencia artificial ha tenido un gran desarrollo en los últimos años, por lo cual se han logrado grandes avances y mejoras que han llevado inclusive a la sustitución de algoritmos clásicos para la solución de ciertos problemas específicos. Esto ha provocado que las redes neuronales al evolucionar puedan llegar a ser computacionalmente intensivas y lleguen a requerir gran cantidad de recursos con los que no siempre se puede contar. Con el fin de implementar eficientemente en aplicaciones de la vida real los modelos entrenados, estos deben ser optimizados para las diferentes arquitecturas. Para estudiar las optimizaciones de redes neuronales en sistemas embebidos, se diseña un modelo de detección de placas oficiales de vehículos de Costa Rica, el cual se optimiza para su ejecución eficiente en un GPU móvil. Las optimizaciones que se aplican en este proyecto corresponden a Cuantización y Pruning y se aplican a diferentes frameworks con el fin de observar los efectos en varias configuraciones.

Palabras clave: VGG16, Mobilenet, Inception, Optimización, set de datos, Inteligencia Artificial, Red Neuronal, GPU, Cuantización, Pruning.

Abstract

The area of Artificial intelligence has had a big development in the last few years, in consequence, many great advances and improvements have caused even the substitution of classic algorithms for the resolution of specific problems. As a result of this, the architectures may become computationally intense and start requiring a big amount of resources that are not always available. In order to implement those efficiently in real life applications, the models require to be optimized for the different architectures. To study the neural network optimization in embedded systems, a detection model for official Costa Rican vehicles plates has been designed, this is being also optimized for its efficient use on a mobile GPU. The optimizations to be applied on this project are Quantization and Pruning, those are applied to different frameworks to observe the effects on the different configurations.

Keywords: VGG16, Mobilenet, Inception, Optimization, dataset, Artificial Intelligence, neural network, GPU, Quantization, Pruning.

a mi amada esposa Lyna, mi hijo Felipe, mis queridos padres, hermana y sobrina…

Agradecimientos

El resultado de este trabajo no hubiese sido posible sin el apoyo incondicional de mi esposa Lyna Hylton, mi familia y la gran motivación al saber que mi hijo Felipe está en camino… y a todos mis amigos que colaboraron para poder obtener las fotos necesarias para completar el set de datos.

Oscar Segura Quesada

Agosto de 2021

Contenidos

Índice de figuras

Índice de tablas

Capítulo 1. Introducción

En la última década, y en especial en los últimos años, la inteligencia artificial (IA) se ha transformado, para algunos, esta fase se inició en 2007 con la llegada de los teléfonos inteligentes. En palabras muy generales, la inteligencia es solo inteligencia, ya sea orgánica o artificial. Se trata de una forma de computación y, como tal, de transformación de la información. La abundancia de información personal, resultado de la vinculación voluntaria de la sociedad a internet, nos ha permitido trasladar un gran caudal de conocimiento explícito e implícito de la cultura humana obtenido por medio de cerebros humanos a formato digital. Una vez ahí, podemos utilizarlo no solo para funcionar con una competencia propia de humanos, sino también para generar más conocimiento y acciones mediante la computación automatizada.

Durante décadas, incluso antes de la creación del término, la IA suscitó tanto miedo como interés, cuando la humanidad contemplaba la posibilidad de crear máquinas a su imagen y semejanza. La expectativa de que los artefactos inteligentes tenían que ser, necesariamente, humanoides nos ha distraído de un hecho importante: hace ya algún tiempo que hemos logrado la IA. Los avances de la IA a la hora de superar la capacidad humana en actividades como el ajedrez y la traducción llegan ahora a ser de conocimiento público en los titulares, pero la IA está presente en la industria desde, al menos, la década de 1980. Los sistemas de normas de producción o sistemas "expertos" se convirtieron en la tecnología estándar para comprobar circuitos impresos y detectar el fraude con tarjetas de créditos. De modo similar, hace algún tiempo que se emplean estrategias de aprendizaje automático (AA) como los algoritmos genéticos para problemas computacionales de muy difícil resolución o la planificación de sistemas operativos informáticos y redes neuronales para modelar y comprender el aprendizaje humano, pero también para tareas básicas de control y supervisión básicos en la industria.

Durante la década de 1990, los métodos probabilísticos y bayesianos revolucionaron el AA y sentaron las bases de algunas de las tecnologías de IA predominantes hoy, como por ejemplo la búsqueda a través de grandes masas de datos. Esta capacidad de búsqueda incluía la posibilidad de hacer análisis semánticos de textos en bruto que permiten a los usuarios de la red encontrar los documentos que buscan entre billones de páginas web con solo escribir unas cuantas palabras.

Esta capacidad de descubrimiento de la IA se ha ampliado no solo por el incremento masivo de los datos digitales y la capacidad de computación, también por las innovaciones en los algoritmos de IA y AA. En definitiva, la IA ya está aquí a nuestra disposición y nos beneficia a todos. Sin embargo, sus consecuencias para nuestro orden social no se entienden y, además, hasta hace poco apenas eran objeto de estudio. Pero también hoy, gracias a los avances en la robótica, la IA está entrando en nuestro espacio físico en forma de vehículos, armas, drones y dispositivos domésticos autónomos, incluidos los "altavoces inteligentes" y consolas de videojuegos. Cada vez estamos más rodeados e integrados de percepciones, análisis y acciones automatizadas generalizadas. A lo largo de las últimas décadas, el aprendizaje de máquina se ha convertido en un tema de investigación muy importante en IA. Fundamentalmente, se implementa utilizando técnicas como las redes neuronales y los algoritmos genéticos.

¿Cómo puede entonces la inteligencia artificial beneficiar a la sociedad? Si abordamos el tema específicamente para nuestro país, Costa Rica, una aplicación de utilidad para la IA es la

identificación de las placas de los vehículos dentro del área de circunvalación. Como es conocido, en los últimos años ha aumentado en gran manera el número de vehículos que circulan en la gran área metropolitana, por lo que el gobierno de la república ha establecido una restricción vehicular en el anillo de circunvalación según el último dígito de la placa de los vehículos.

Parte de la problemática con esta medida implementada por las autoridades, es el bajo número de oficiales de tránsito disponibles para atender, no solo los accidentes de tránsito y operativos policiales rutinarios, sino también el control para asegurar que los conductores con placas restringidas se abstengan de circular por las zonas en las cuales no se les permite el ingreso.

Con el uso de una red neuronal, es posible encontrar una solución al inconveniente planteado. Es mediante el uso de una red neuronal convolucional que se puede diseñarse un sistema que permita identificar la placa en cada vehículo y a la vez, identificar el número de esta.

Mediante el despliegue de la red neuronal en un sistema empotrado, este puede ser ubicado en puntos estratégicos en el anillo de circunvalación de forma que los vehículos que ingresen a la zona restringida en el día de su restricción puedan ser debidamente identificados y multados de acuerdo con lo establecido por la ley vigente.

Para el desarrollo del dispositivo, se requiere de un orden lógico que permita realizar las tareas aquí mencionadas. Para poder cumplir con ese orden que permita alcanzar el propósito, se discute en la siguiente sección los objetivos y la estructura que seguirá este documento para alcanzar los mismos.

1.1 Objetivos

El progreso del proyecto se divide en dos etapas principales donde inicialmente se debe de definir el área de aplicación de la red neuronal, definir un set de datos apropiado, entrenar diversos modelos de redes neuronales, comparar el desempeño de los modelos y seleccionar el modelo más adecuado para el proyecto. En la segunda etapa, se requiere aplicar optimización al modelo seleccionado en la primera etapa y luego desplegar el modelo ya optimizado en el hardware adecuado para que pueda ser utilizado en el campo.

Primera etapa:

A continuación, se lista el objetivo principal para la primera etapa de este proyecto:

- Desarrollar una red neuronal que permita la identificación de una placa de circulación vehicular oficial de Costa Rica para vehículos de uso particular y que logre identificar además el último dígito de la placa y el día de restricción

Los objetivos adicionales para alcanzar el objetivo principal son los siguientes:

- Evaluar diferentes arquitecturas de redes neuronales para ser utilizadas en la primera parte del proyecto y seleccionar las 3 consideradas más adecuadas.
- Elaboración de un set de datos construido en su totalidad con imágenes de placas oficiales de vehículos de Costa Rica.
- Entrenar los tres modelos seleccionados con el set de datos elaborado.

- Realizar una comparación de los modelos entrenados en términos de tiempo, perdida y precisión.
- Definir el modelo más apropiado para este proyecto como base para la segunda etapa del proyecto.
- Utilizando imágenes de prueba con el modelo seleccionado, extraer el número de placa e identificar el último digito de la placa y el día de la restricción vehicular.

Para lograr desarrollar el proyecto, se hace necesario utilizar un set de datos personalizado ya que en diferentes sitios de Internet se encuentran disponibles algunos sets de datos de placas vehiculares de diferentes latitudes y por tanto de diferentes formas a las placas costarricenses, razón por la que es necesario crear dicho set de datos para el entrenamiento adecuado de la red neuronal.

A partir de esto, se entrenarán tres modelos diferentes, en los cuales se debe evaluar ventajas y desventajas y seleccionar el que mejor se adecue al proyecto. Esta comparación tendrá como métrica principal la exactitud del modelo (definida mediante la IoU) una vez que el mismo sea entrenado y el tiempo total de entrenamiento. La selección adecuada del modelo es uno de los puntos más importantes de esta primera etapa, ya que de esto dependerá el desarrollo de la segunda etapa y será la base para la implementación en hardware del proyecto.

Una vez seleccionado el modelo con el mejor desempeño, se debe identificar tanto la placa, la cual se resalta por medio de un rectángulo que encierra los números de esta, como el último dígito de esta. A partir de esta información, se debe mostrar entonces el número de placa completo, el último dígito y el día de restricción.

Segunda etapa:

Para la segunda etapa, el objetivo principal es:

- Optimizar el modelo seleccionado en la primera etapa del proyecto

Con el fin de cumplir este objetivo, se establecen los siguientes objetivos adicionales:

- Utilizar diferentes métodos de optimización en el modelo seleccionado.
- Comparar el modelo antes y después de ser optimizado.
- Implementar el modelo en el GPU de NVIDIA.

En la segunda etapa del proyecto, el objetivo principal será **la Optimización del modelo** seleccionado en la primera etapa y su implementación en un GPU. Se utilizan las técnicas de optimización de Cuantización (FP16 e INT8) y Pruning para reducir el tamaño y los requerimientos de computación del modelo.

1.2 Estructura del documento

La estructura de este documento consta de un primer capítulo introductorio, en el cual se expone la motivación del proyecto y cuáles serán los objetivos principales del mismo en el momento de plantearlo. Se explica además cómo va a llevarse a cabo dicho proyecto, es decir, las herramientas y métodos que se utilizarán durante el desarrollo de este.

En el segundo capítulo, se presentan conceptos técnicos que es necesario conocer a fondo a la hora de resolver el problema propuesto. Es decir, se expone todo el trabajo de estudio previo a la realización práctica del proyecto, y que por ello constituye el eje central del mismo. Entre estos conceptos se encuentran los de reconocimiento de objetos en imágenes, deep learning y la explicación general del funcionamiento de las redes neuronales convolucionales, enfocadas al tratamiento de imágenes.

En el tercer capítulo se exploran las diferentes arquitecturas para utilizar en la primera etapa y que serán candidatas para el desarrollo del modelo. Se crea un set de datos adecuado para la tarea, en este caso, la detección de placas y se establecen las bases para el cálculo de la intersección sobre unión que juega un papel importante en la segunda etapa del proyecto. Aquí se incluyen también tablas y figuras que ayuden a una mejor comprensión de los diversos temas mencionados.

En el capítulo número cuatro, se presenta el análisis de los resultados que llevan al cumplimiento de los objetivos de la primera parte, la identificación de la placa, y se exponen los diseños experimentales realizados para comprobar el funcionamiento correcto del sistema. También se muestran resultados concretos del funcionamiento de los modelos y sus respectivas comparaciones. Esta sección también incluye tablas y figuras presentadas de tal modo que se observe cómo los objetivos específicos y el objetivo general del proyecto se han cumplido.

El capítulo cinco abarca la segunda parte del proyecto, esta consta de las diferentes optimizaciones aplicadas a los modelos, así como la discusión de los resultados obtenidos con estas. Aquí se muestran las métricas a evaluar, como la precisión, la memoria de ejecución, el tamaño del archivo y el tiempo de ejecución, las cuales son utilizadas para tomar una decisión final acerca de lo que se considera la configuración más adecuada para este proyecto.

Finalmente, las conclusiones son mostradas en el capítulo seis del documento, mostrando lo que ha llevado el desarrollo de la tesis, no perdiendo de vista los objetivos planteados desde el principio y los resultados obtenidos. También se hace énfasis de los aportes específicos del trabajo y otras acciones que quedan por hacer, o sugerencias para mejorar los resultados.

Capítulo 2. Marco teórico

En este capítulo se presenta una explicación de los conceptos principales necesarios para el desarrollo de este proyecto, tales como reconocimiento de imagen y la detección de objetos. Se centrará en el Deep Learning, principalmente en el área de las Redes Neuronales Convolucionales que serán la principal herramienta matemática que se utilizará para resolver los problemas de reconocimiento de placas y sus respectivos dígitos que se plantean en este proyecto.

2.1 Reconocimiento de imágenes y detección de objetos

La definición más simple de reconocimiento de imágenes dice que es la tarea para encontrar e identificar objetos en una imagen o secuencia de video. Los humanos reconocemos una multitud de objetos en imágenes con poco esfuerzo, a pesar del hecho que la imagen del objeto puede variar un poco en diferentes puntos de vista, en diferentes tamaños o escala e incluso cuando están trasladados o rotados. Los objetos pueden ser reconocidos cuando están parcialmente obstruidos desde una vista. No obstante, esta tarea es un desafío para los sistemas de visión por computadoras. Para este problema se han implementado muchos métodos durante múltiples décadas.

El reconocimiento y detección de objetos en imágenes tomados por un sistema de visión por computadora, tiene siempre la misma estructura de trabajo:

- Adquisición de la imagen.
- Digitalización
- Extracción de características, búsqueda de patrones en la imagen, ya sean de tipo geométrico, estadístico, topológico, etc.
- Sistema de clasificación: se trata de asignar a cada clase de objeto que se busca reconocer, un patrón o conjunto de patrones característicos, de manera que podamos asociarlos y poder así clasificar a cada uno de ellos.

Así, cada vez que se tenga una nueva imagen por clasificar, se obtendrán sus características principales y éstas serán comparadas con aquellas que ya se habían estudiado para cada clase de objeto. En función de la similitud encontrada, la imagen se asocia a una clase u otra, definiendo así la probabilidad que tendría de pertenecer a dicha clase, cuanto mayor sea la similitud, mayor es la probabilidad que se obtiene. Cuando se asocian patrones de características a cada objeto, es usual que se defina un vector de características. Se define de esta manera un patrón como el conjunto de características de una imagen:

$$\boldsymbol{x} = [x_1, x_2, \dots, x_n]^{\mathrm{T}} \tag{2.1}$$

Donde x representa el patrón que se estudia mientras que x_i son las características que lo componen. Para cada clase hay que encontrar una función de decisión (clasificador) de manera que si x pertenece a la clase w y no a la clase v entonces:

$$(x) > d(x) \tag{2.2}$$

La clasificación del objeto se puede realizar mediante dos métodos diferenciados: *clasificación supervisada* o *no supervisada*. La clasificación supervisada es aquella en la cual se utiliza un gran conjunto de datos de aprendizaje, a partir del cual se entrenará el sistema para asociar patrones y clases. En la clasificación no supervisada en cambio, no se utiliza un conjunto de datos para aprender a clasificar la información, sino que se usan cálculos estadísticos para realizarla.

Adicional a la detección de objetos, hay otras tareas que pueden ser realizadas por diversos métodos, entre ellos clasificación, localización y segmentación. La Figura 1 muestra un ejemplo de las tareas mencionadas. Además, estas tareas pueden ser ejecutadas en imágenes con uno o más objetos de interés.

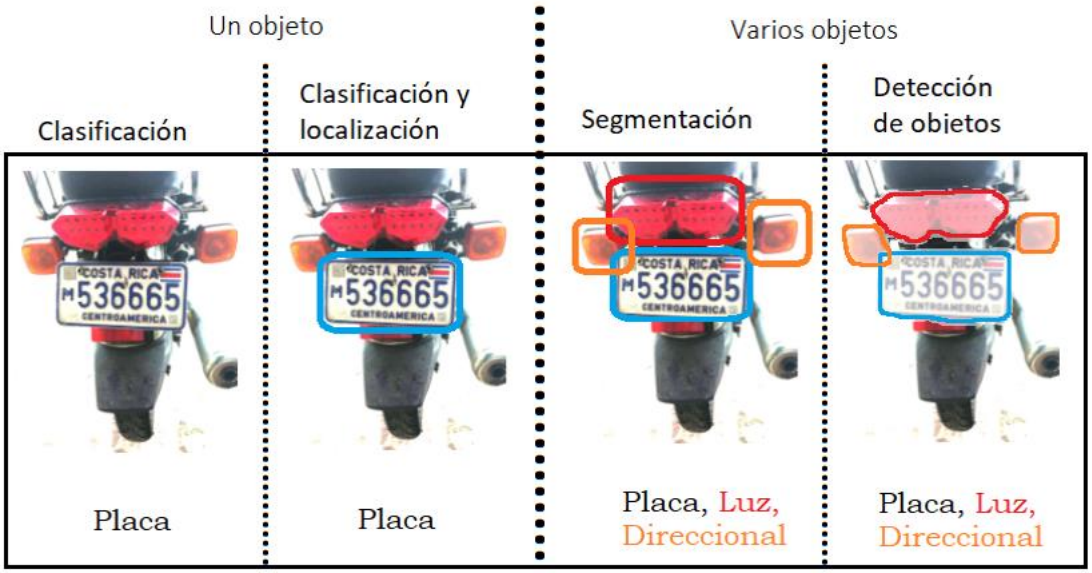

Figura 1. Diferentes tareas realizadas con visión por computadora.

Existen actualmente diversas herramientas o librerías en el campo de la visión por computadora, sin embargo, una de las más populares y utilizadas actualmente es OpenCV (Open Source Computer Vision). OpenCV es una librería con funciones que apuntan principalmente a la visión por computadora en tiempo real. Soporta también Deep Learning frameworks como Caffe, Tensorflow, Torch/PyTorch [1].

2.2 Machine learning e inteligencia artificial

En los últimos años, la Inteligencia Artificial se ha desarrollado enormemente, utilizando técnicas que, aunque no son novedosas, se han podido llevar a cabo con mayor facilidad en parte gracias al uso de GPUs para el procesamiento en paralelo, que hace que sea mucho más rápido y económico. Las técnicas de Machine Learning ya supusieron un gran avance en el desarrollo de la Inteligencia Artificial. El deep learning conforma una subcategoría del aprendizaje automático. Este aprendizaje trata el uso de redes neuronales para mejorar tareas automáticas como la visión por ordenador, el reconocimiento de voz, etc. Para diferenciarlo del aprendizaje de máquina o Machine Learning, se puede decir que el deep learning está menos sometido a supervisión, utilizando redes neuronales a gran escala que permiten que la máquina aprenda y reconozca patrones por sí misma, al basar si código en el funcionamiento del cerebro humano.

2.3 Redes neuronales y Deep learning

Hay tres razones por las cuales el Deep Learning ha cobrado popularidad:

1. Disponibilidad de los datos.
2. El incremento en el poder computacional, GPUs, CPUs mejorados, ASICs, etc.
3. La aparición de nuevos y creativos algoritmos que han cambiado la forma en que trabajan las redes neuronales.

Las Redes Neuronales Artificiales son un sistema de procesamiento de información mediante unidades básicas de procesamiento que se basan en el funcionamiento de la célula fundamental del sistema nervioso humano: la neurona. Estas se interconectan entre sí, intercambiando información entre ellas y formando así las redes neuronales, las cuales producirán una salida a partir de un conjunto de datos de entrada a la misma. En las neuronas humanas, estas conexiones son la clave para el procesamiento de información y son llamadas sinapsis. En la Figura 2 se muestra una comparación de ambos tipos de neuronas.

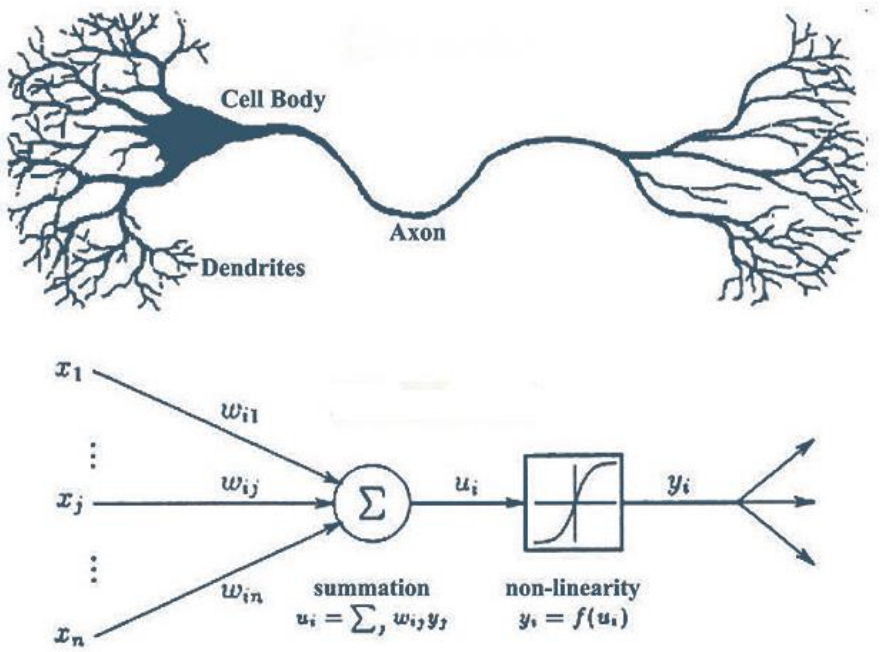

Figura 2. Neurona natural vs neurona artificial.
Imagen tomada de https://magiquo.com/

Las neuronas artificiales se comportan de una forma similar a las naturales, ya que recibirán una serie de señales de entrada que harán activar la neurona según una determinada función de activación. Cuando esto ocurra, se generará una señal de salida, que podrá ser transmitida hacia otra neurona. La unión de dos o más neuronas conforma una red neuronal, a la cual se ingresa información que se irá procesando en cada una de sus neuronas de una manera diferente y finalmente aportará una información en su salida, esto se representa en la Figura 3.

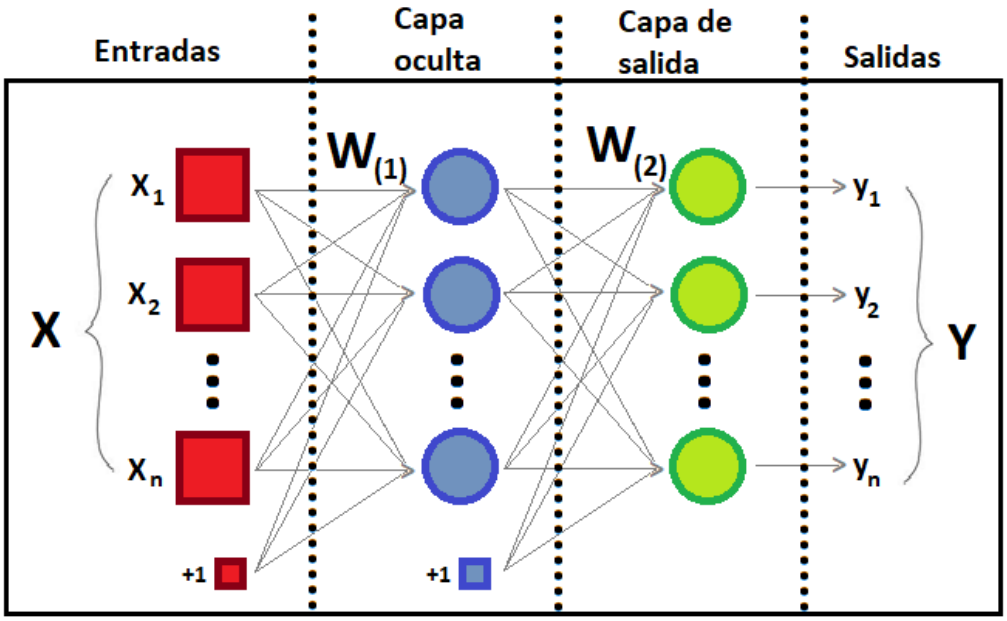

Figura 3. Representación de una red neuronal artificial.

Las unidades de procesamiento se organizan en capas. Hay tres partes normalmente en una red neuronal: una capa de entrada, con unidades que representan los campos de entrada; una o varias capas ocultas; y una capa de salida, con una unidad o unidades que representa el campo o los campos de destino. Las unidades se conectan con fuerzas de conexión variables (o ponderaciones). Los datos de entrada se presentan en la primera capa, y los valores se propagan desde cada neurona hasta cada neurona de la capa siguiente. Al final, se envía un resultado desde la capa de salida.

La red aprende examinando los registros individuales, generando una predicción para cada registro y realizando ajustes a las ponderaciones cuando realiza una predicción incorrecta. Este proceso se repite muchas veces y la red sigue mejorando sus predicciones hasta haber alcanzado uno o varios criterios de parada. Al principio, todas las ponderaciones son aleatorias y las respuestas que resultan de la red son, posiblemente, disparatadas. La red aprende a través del entrenamiento. Continuamente se presentan a la red ejemplos para los que se conoce el resultado, y las respuestas que proporciona se comparan con los resultados conocidos. La información procedente de esta comparación se pasa hacia atrás a través de la red, cambiando las ponderaciones gradualmente.

A medida que progresa el entrenamiento, la red se va haciendo cada vez más precisa en la replicación de resultados conocidos. Una vez entrenada, la red se puede aplicar a casos futuros en los que se desconoce el resultado.

Cada una de las entradas de una neurona tiene diferente factor de importancia a la hora de procesar la información que le llega. Por ello cada entrada tiene asociada un peso, que cambia la medida de influencia que tienen los valores de entrada. Estos pesos se definen mediante la matriz de pesos que se muestra en la Figura 4.

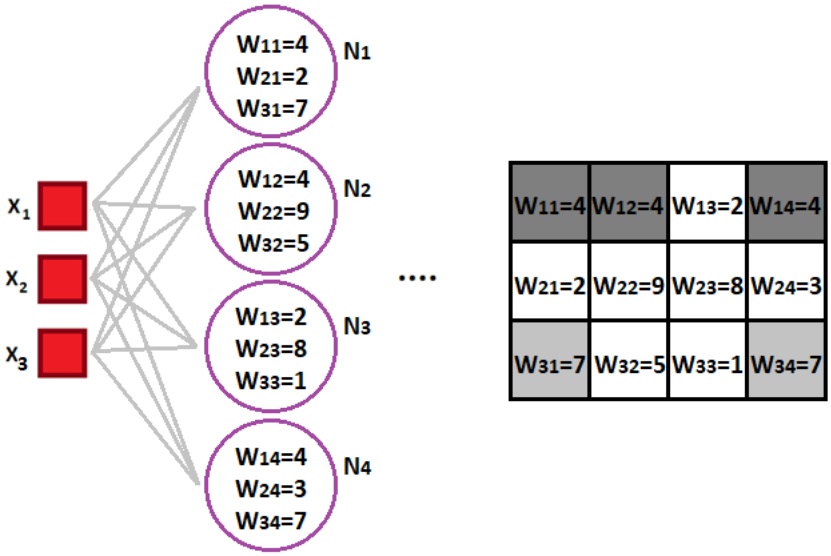

Figura 4. Matriz de pesos de una red neuronal [13].

La primera línea de la matriz contiene los pesos de las conexiones hacia la primera neurona, y la primera columna contiene los pesos de las conexiones que salen desde la primera entrada. Mediante este aprendizaje automático no es necesario extraer características concretas de los datos de entrada, sino que la propia red se encarga de obtener estos patrones o atributos de cada entrada. Por tanto, se puede decir que el proceso de aprendizaje de una red neuronal se basa en encontrar los valores correctos de los pesos de las entradas a la misma, de manera que se obtenga la salida que se desea.

A pesar de la gran ventaja que supone el uso de dichas redes neuronales, existe un inconveniente en la implementación de las mismas, y es la dificultad que supone el hecho de que las neuronal estén completamente interconectadas entre sí cuando se trata con un conjunto de datos de entrada de gran tamaño, es aquí donde surgen las Redes Neuronales Convolucionales.

2.4 Redes neuronales convolucionales

Una red neuronal convolucional es un tipo de red neuronal artificial donde las neuronas corresponden a campos receptivos de una manera muy similar a las neuronas en la corteza visual primaria de un cerebro biológico. Este tipo de red es una variación de un perceptron multicapa, sin embargo, debido a que su aplicación es realizada en matrices bidimensionales, son muy efectivas para tareas de visión artificial, como en la clasificación y segmentación de imágenes, entre otras aplicaciones. El trabajo realizado por Hubel y Wiesel en [2] jugó un papel importante en la comprensión sobre cómo funciona la corteza visual, particularmente las células responsables de la selectividad de orientación y detección de bordes en los estímulos visuales dentro de la corteza visual primaria. Dos tipos de células principales fueron identificadas aquí, teniendo éstas campos receptivos alargados, con lo cual tienen una mejor respuesta a los estímulos visuales alargados como las líneas y los bordes. Estas se denominan células simples y células complejas.

Las células simples tienen regiones excitadoras e inhibitorias, ambas formando patrones elementales alargados en una dirección, posición y tamaño en particular en cada célula. Si un estímulo visual llega a la célula con la misma orientación y posición, de tal manera que ésta se

alinea perfectamente con los patrones creados por las regiones excitadoras y al mismo tiempo se evita activar las regiones inhibitorias, la célula es activada y emite una señal.

Las células complejas operan de una manera similar. Como las células simples, éstas tienen una orientación particular sobre la cual son sensibles. Sin embargo, éstas no tienen sensibilidad a la posición. Por ello, un estímulo visual necesita llegar únicamente en la orientación correcta para que esta célula sea activada.

Otro punto importante sobre las células en la corteza visual es la estructura que éstas forman. A lo largo de la jerarquía de la corteza, comenzando por la región V1 de la corteza visual, luego siguiendo a las regiones V2, V4 e IT, se encuentra que la complejidad de los estímulos ideales incrementa cada vez más. Al mismo tiempo, las activaciones de las células se hacen menos sensibles a la posición y tamaño de los estímulos iniciales. Esto sucede como resultado de las células activando y propagando sus propios estímulos a otras células conectadas a esta jerarquía, principalmente gracias a la alternación entre células simples y complejas.

Cualquier algoritmo de detección de objetos conlleva dos fases diferenciadas: la extracción de características del contenido de la imagen y la posterior búsqueda de objetos basada en dichas características para su correcta clasificación. Las CNN realizan estas dos fases sin necesidad de separarlas, por lo que esto nos facilita tener un aprendizaje de principio a fin.

Las CNN convolucionan las características aprendidas de los datos de entrada y emplea capas convolucionales 2D, lo cual hace que esta arquitectura resulte adecuada para procesar datos 2D, tales como imágenes. Por ello serán ideales para la aplicación concreta que se necesita.
Estas convoluciones resuelven por ejemplo el problema de tener un objeto con las mismas características, pero desplazado dentro de una imagen. Aplicando estos filtros de convolución se transforma la matriz de pixeles (imagen de entrada) en una matriz de características. Cada píxel contiene información no solo ya de sí mismo, sino de los pixeles que le rodean.

La arquitectura general de las redes neuronales convolucionales está conformada por cuatro capas principales que se muestran a continuación:

- Capa convolucional.
- Reducción.
- Capa completamente conectada.

2.4.1 Capa convolucional

Esta capa aplica a la imagen una operación de convolución con un filtro determinado. La operación de convolución tiene la siguiente expresión matemática:

$$G(i,j) = \sum_{u=-k}^{k} \sum_{v=-k}^{k} F(u,v) \cdot I\,(i-u, j-v) \qquad (2.3)$$

Donde F es el filtro que se aplica, e I será la imagen a la cual se aplica la operación de convolución. En este caso, ilustrado en la Figura 5, no se desea aplicar a la imagen un único filtro, sino un banco de filtros para conseguir detectores de distintas características de la imagen.

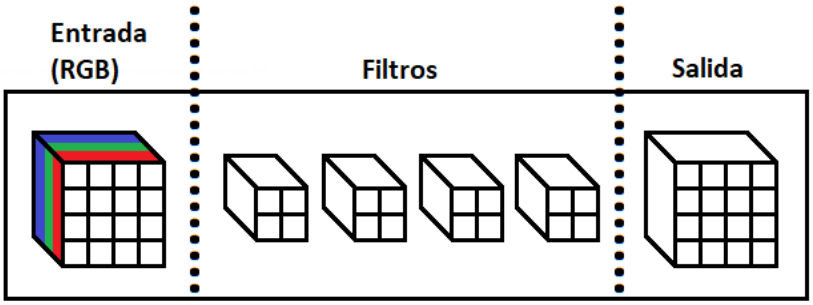

Figura 5. Aplicación de convolución con diferentes filtros.

Al aplicar convolución con un banco de filtros, a la salida se obtendrá un volumen 3D en el que cada canal es una salida de la convolución de la imagen con uno de los filtros que componen el banco. Así, las dimensiones del volumen de salida serán WxHxF, donde WxH es el tamaño de la imagen original y F es el número de filtros que conforman el banco. Los hiperparámetros que controlan el tamaño del volumen de salida de cada capa son:

- **Profundidad**: corresponde al número de filtros que se quiere usar, cada uno aprendiendo a buscar algo diferente en la entrada.

- **Stride**: Es el deslizamiento del filtro, mientras mayor sea el stride, el volumen será menor

- **Zero-padding**: En algunos casos será conveniente o necesario colocar ceros alrededor del borde del volumen de entrada.

Para ilustrar el funcionamiento de los filtros y la convolución, a continuación, se muestra en la Figura 6 un ejemplo de un filtro 3 x 3 con las siguientes características:

$$\begin{bmatrix} -1 & 2 & -1 \\ -1 & 2 & -1 \\ -1 & 2 & -1 \end{bmatrix}$$

Figura 6. Ejemplo de filtro de 3 x 3

Si aplicamos este filtro a una imagen en blanco y negro, podremos obtener los valores verticales de la misma. A continuación, se presenta la imagen de la Figura 7 a la cual se va a aplicar el filtro:

Figura 7. Imagen de muestra para aplicación del filtro de 3x3.

Como se mencionó anteriormente, al aplicar el filtro a la imagen, se obtendrán los valores verticales de la misma, y cuando se ejecuta un CNN se aprenden los valores del filtro. Vale la pena señalar nuevamente que el tamaño del stride y el filtro son hiperparámetros, lo que significa que no son aprendidos por el modelo. Así que el diseñador debe concretar qué valores de esas cantidades funcionarán mejor para su aplicación.

En la Figura 8 se muestra el resultado de la aplicación del filtro mostrado en la Figura 6 sobre la Figura 7, evidenciando la detección de los valores verticales encontrados en la imagen original.

Figura 8. Resultado de la aplicación del filtro de 3 x 3 en imagen blanco y negro.

Un concepto final que hay que entender acerca de la convolución es la idea de normalización. Si la imagen no encaja con el filtro en un número de veces entero (teniendo en cuenta el stride), hay que normalizar la imagen. Hay dos formas de hacer esto: la normalización válida y la normalización igual. La primera básicamente elimina los valores faltantes del borde de la imagen. Es decir, si el filtro es 2 x 2 con un stride de 2, y la imagen tiene una anchura de 3, entonces la normalización válida ignora la tercera columna de valores de la imagen. Entre tanto, la normalización igual añade valores (normalmente ceros, zero-padding) a los bordes de las imágenes para incrementar sus dimensiones hasta que el filtro pueda encajar un número de veces entero. Esta normalización normalmente se realiza de forma simétrica (es decir, intenta añadir el mismo número de columnas/filas a cada lado de la imagen).

También es interesante apuntar que las convoluciones de la imagen tienen aplicaciones que no se limitan a la visión por computadora. Muchas técnicas de filtrado de imágenes se pueden implementar a través de la convolución, como, por ejemplo, el desenfoque y el enfoque.

2.4.2 Reducción (Pooling)

La capa de reducción o pooling se coloca generalmente después de la capa convolucional. Su utilidad principal radica en la reducción de las dimensiones espaciales (ancho x alto) del volumen de entrada para la siguiente capa convolucional. No afecta a la dimensión de profundidad del volumen. La operación realizada por esta capa también se llama reducción de muestreo, ya que la reducción de tamaño conduce también a la pérdida de información. Sin embargo, una pérdida de este tipo puede ser beneficioso para la red por dos razones:

1. La disminución en el tamaño conduce a una menor sobrecarga de cálculo para las próximas capas de la red

2. También trabaja para reducir el sobreajuste.

La operación que se suele utilizar en esta capa es max-pooling, que divide a la imagen de entrada en un conjunto de rectángulos y, respecto de cada subregión, se va quedando con el máximo valor como se observa en la Figura 9.

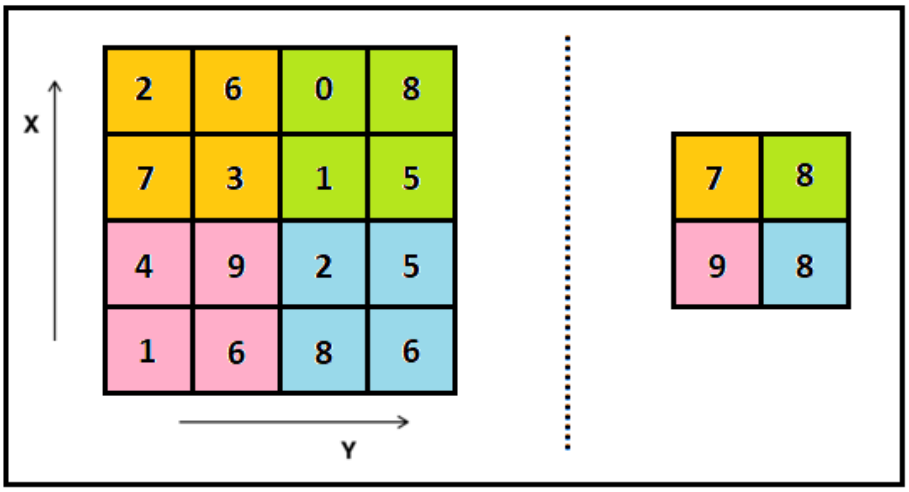

Figura 9. Ejemplo de aplicación de max-pooling.

En resumen, max-pooling es un proceso de desratización basado en muestras cuyo objetivo es reducir la muestra de una representación de entrada (imagen, matriz de salida de capa oculta, etc.), reduciendo su dimensionalidad y permitiendo suposiciones sobre las características contenidas en las subregiones agrupadas.

2.4.3 Capa completamente conectada (Fully connected)

Al final de las capas convolucional y de pooling, las redes utilizan generalmente capas completamente conectadas, similar a la mostrada en la Figura 10, en la que cada píxel se considera como una neurona separada al igual que en una red neuronal regular. Esta última capa clasificadora tendrá tantas neuronas como el número de clases que se debe predecir. En

esta no se realizará convolución, sino el producto lineal entre el filtro y la salida de la capa anterior, que típicamente habrá sido una capa max pooling.

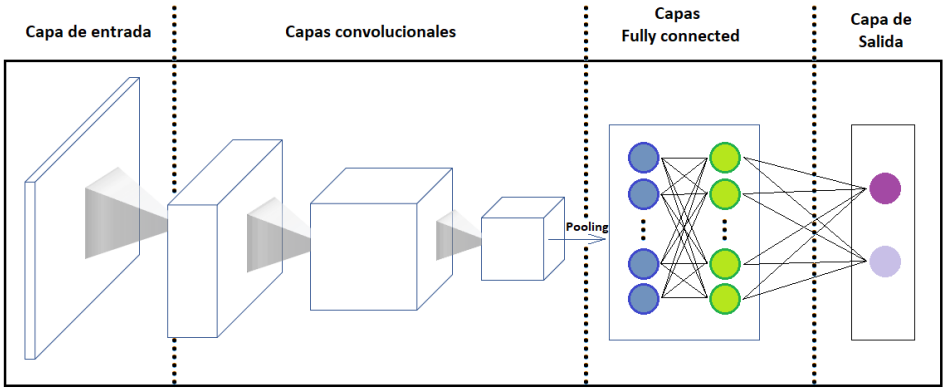

Figura 10. Capa completamente conectada.
Imagen basada en www.analyticssteps.com

2.5 Funciones de activación

Las funciones de activación se encargan de devolver una salida a partir de un valor de entrada, normalmente el conjunto de valores de salida en un rango determinado como (0,1) o (-1,1). En una red neuronal se buscan funciones cuyas derivadas sean simples, para así minimizar el coste computacional.

Para las capas agregadas a los modelos de este proyecto, se utiliza una función de activación del tipo unidad lineal rectificada o "ReLU" por sus siglas en Ingles (Rectified Linear Unit). Esta función de activación básicamente es utilizada en este caso ya que los valores negativos no son importantes en el procesamiento de imágenes y, por lo tanto, se establecen en 0, pero los valores positivos después de la convolución deben pasar a la siguiente capa. La función ReLU es bastante sencilla y está dada de la siguiente manera:

$$R(z) = max\ (0, z) \tag{2.4}$$

ReLU se está utilizando como una función de activación dada su capacidad de eliminar los valores negativos, como se muestra en la Figura 11. Si se utilizara sigmoid o tanh, la información se pierde ya que ambas funciones modificarán las entradas a un rango muy cerrado.

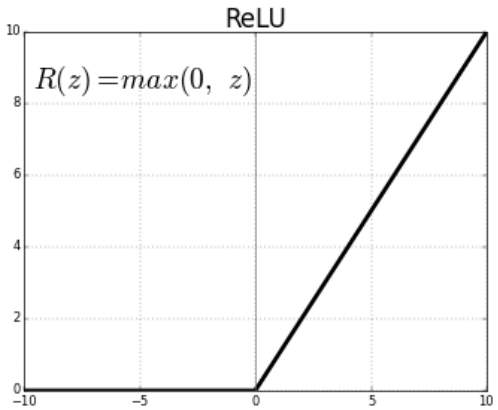

Figura 11. Función de activación ReLU.

Para la última capa agregada ya no se utiliza la función de activación ReLU, sino más bien se utiliza la función sigmoid (o sigmoidea). Esta función es una de las más antiguas y utilizadas y se define de la siguiente manera:

$$\sigma(x) = \frac{1}{1+e^{-x}} \tag{2.5}$$

Al ser esta la última capa de la arquitectura, y debido a que solo se tiene un objeto por identificar (la placa del vehículo), se hace uso de la función sigmoidea en esta capa, ya que esta solo provee valores de salida entre 0 y 1. Esta función es ampliamente utilizada para este tipo de modelos, donde se debe predecir la probabilidad como una salida y dado que la probabilidad de que algo exista está en el rango [0,1], suele ser la mejor opción la activación sigmoidea. En la Figura 12 se observa un ejemplo de la función sigmoidea.

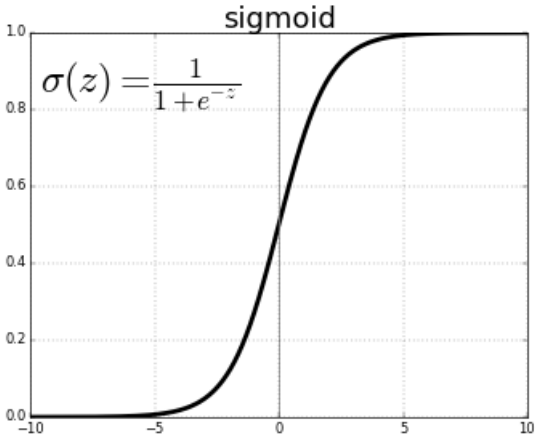

Figura 12. Función de activación Sigmoidea.

2.6 Transfer learning

El concepto de transfer learning hace referencia a un enfoque popular en el deep learning en el cual se utilizan modelos pre-entrenados como punto de partida en tareas de visión por computadora y procesamiento del lenguaje natural, dados los vastos recursos informáticos y de tiempo necesarios para desarrollar modelos de redes neuronales sobre estos problemas y los enormes saltos de habilidad que proporcionan sobre problemas relacionados.

Esto quiere decir que es posible obtener un modelo pre entrenado, por ejemplo, para identificar perros en las imágenes y hacer una transferencia de aprendizaje en ese modelo para reutilizarlo de forma que aprovechando la capacidad que se ha adquirido para identificar algunos patrones en las imágenes, podamos identificar ahora, por ejemplo, gatos en lugar de perros.

En la práctica lo que se hace entonces es tomar el modelo entrenado y guardar en un archivo (h5, pb, etc.) los pesos aprendidos. Después se carga estos pesos en un nuevo modelo y se elimina la última capa que es la capa de predicción que sabía clasificar perros. Luego se hace un “freeze” a las capas en el estado que están, con los pesos que tienen aprendidos, para que los próximos entrenamientos no las tengan en cuenta. Finalmente se agrega una nueva capa dense (o conjunto de capas) para que haga la predicción de si la imagen es un burro o un gato, siendo esta (o estas) únicamente las capas que deben ser entrenadas.

2.7 Optimizaciones

Los dispositivos de última tecnología suelen tener capacidades de computación limitada. Es posible aplicar varias optimizaciones a los modelos con el fin de reducir este inconveniente y que se puedan ejecutar dentro de estas restricciones. Además, algunas optimizaciones permiten el uso de hardware especializado, como por ejemplo GPUs para inferencias aceleradas.

Existen diversas técnicas de optimización que pueden ser aplicadas a los modelos, sin embargo, la definición de la técnica más adecuada es totalmente dependiente de lo que se quiere lograr con el modelo, y de cuanto se desee sacrificar uno u otro parámetro para mejorar algún otro.

Las optimizaciones que se aplican en este proyecto se consideran con más profundidad en la sección 5.1 Tipos de optimizaciones. Dichas optimizaciones son:

- Cuantización:
    - Float 32 a Float 16: Algunos frameworks permiten la cuantización de punto flotante de 16 bits. En esta literalmente los pesos se convierten de números de punto flotante de 32 bits a números de punto flotante de 16 bits, logrando una reducción a la mitad del tamaño del modelo.

    - Float 32 a Integer 8: Al igual que en el caso anterior, aquí se hace una conversión de un número de punto flotante de 32 bits, pero esta vez a un número entero de 8 bits, logrando una reducción de x4 en el tamaño del archivo.

- Pruning:
    - Pruning es una técnica de compresión que tiene como objetivo remover partes redundantes del modelo en términos de pesos, canales y capas. Al eliminar esta redundancia, tiene como efectos colaterales la reducción de la computación y memoria. Es importante que el pruning se haga de forma gradual y que haya un reentrenamiento del modelo después de cada aplicación, esto con el fin de evitar impactos severos en la precisión del modelo.

2.7.1 Ventajas de las optimizaciones

Algunas de las ventajas que ofrece la optimización de modelos computacionales son la reducción del tamaño, reducción del tiempo de ejecución, compatibilidad del acelerador entre otras. A continuación, se describen las más comunes utilizadas.

Reducción del tamaño: Se pueden utilizar algunas formas de optimización para reducir el tamaño de un modelo. Los modelos más pequeños tienen algunos beneficios como:

- Tamaño de almacenamiento más pequeño: los modelos más pequeños ocupan menos espacio de almacenamiento en los dispositivos embebidos, en los cuales algunas veces no se cuenta con la capacidad suficiente para almacenar un modelo de tamaño muy grande.

- Tamaño de descarga más pequeño: los modelos más pequeños requieren menos tiempo y ancho de banda para descargarse en los dispositivos.

- Menor uso de memoria: los modelos más pequeños generalmente usan menos RAM cuando se ejecutan, lo que libera memoria para que la utilicen otras partes de la aplicación y puede traducirse en un mejor rendimiento y estabilidad.

Reducción del tiempo de ejecución: Algunas formas de optimización pueden reducir la cantidad de cálculo necesario para ejecutar inferencias utilizando un modelo, lo que resulta en un tiempo de ejecución más bajo, esto puede tener también un impacto en el consumo de energía.

Compatibilidad del acelerador: Algunos aceleradores de hardware, como Edge TPU, pueden ejecutar inferencias extremadamente rápido con modelos que se han optimizado correctamente. Sin embargo, generalmente, este tipo de dispositivos requieren que los modelos se cuantifiquen de una manera específica.

2.7.2 Desventajas de las optimizaciones

Las optimizaciones pueden potencialmente resultar en cambios en la precisión del modelo, que deben ser considerados durante el proceso de desarrollo de la aplicación. Los cambios de precisión dependen del modelo individual que se optimiza y son difíciles de predecir con anticipación. Generalmente, los modelos optimizados para el tamaño o el tiempo de ejecución potencialmente pueden sufrir una degradación en términos de precision. Dependiendo de la aplicación, esto puede afectar o no el funcionamiento adecuado para la tarea específica con la que fue diseñado el modelo. Es importante también aclarar que, en muy raras ocasiones, ciertos modelos pueden mejorar su precisión como resultado del proceso de optimización, ya que, por ejemplo, el weight pruning funciona como una técnica de "regularización" que ayuda a que el modelo generalice mejor por lo que la precisión tiende a mejorar en algunos casos.

Capítulo 3. Etapa I – Exploración de arquitecturas de redes neuronales convolucionales

A continuación, se presentan los diferentes pasos a seguir para la creación apropiada de la red neuronal que permita la identificación de los números de placas oficiales de vehículos particulares de Costa Rica.

Para esto se han realizado diversas tareas entre las cuales se incluyen la identificación de las arquitecturas de redes neuronales disponibles y la selección de las más adecuadas para el propósito de este proyecto. Posteriormente, se crea el set de datos adecuado para la identificación de las placas de los vehículos. Una vez obtenido el set de datos, se procede a entrenar la red neuronal en las diferentes arquitecturas. Con los resultados obtenidos luego del entrenamiento, se procede a seleccionar la arquitectura con los mejores resultados. Adicionalmente, es apropiado realizar algunas pruebas con varias fotografías de prueba. Finalmente, es necesario la identificación del último número de la placa. A continuación, se detallan las actividades mencionadas.

3.1 Arquitectura de la red neuronal

El concepto de arquitectura referida a redes neuronales hace mención no solo al número de capas neuronales o al número de neuronas en cada una de ellas, sino a la conexión entre neuronas o capas, al tipo de neuronas presentes e incluso a la forma en la que son entrenadas.

Las redes neuronales, también son llamadas *Deep Convolutional Neural Networks* (DCNN). Las capas que las componen (no todas formadas por neuronas) pueden dividirse en dos bloques: el primer bloque, formado principalmente por capas convolucionales y de pooling, tienen como objetivo la identificación de patrones gráficos, mientras que el segundo bloque tiene como objetivo la clasificación de los datos que reciben.

Dado a la naturaleza de las convoluciones dentro de las redes neuronales convolucionales, estas son aptas para poder aprender a clasificar todo tipo de datos donde estos estén distribuidos de una forma continua a lo largo del mapa de entrada, y a su vez sean estadísticamente similares en cualquier lugar del mapa de entrada. Por esta razón, son especialmente eficaces para clasificar imágenes.

Las redes neuronales han existido desde ya varias décadas y han evolucionado en capacidad, tamaño y muchas otras características. Es importante anotar que no hay una arquitectura que pueda considerarse mejor que otra de una forma general, sino más bien dependiendo del propósito, una arquitectura puede resultar más conveniente que otra.

Para efectos de este proyecto, solo se consideran las arquitecturas de redes neuronales convolucionales, dada su naturaleza ya mencionada.

La primera arquitectura analizada, es **LeNet-5**, esta es una estructura de red neuronal convolucional propuesta por Yann LeCun et al. en 1998. Es una red neuronal convolucional simple. Como representante de la red neuronal convolucional temprana, LeNet posee las unidades básicas de la red neuronal convolucional, como la capa convolucional, la capa de agrupación y la capa de conexión completa, sentando las bases para el desarrollo futuro de la red neuronal convolucional. Consta de siete capas. Además de la entrada, cualquier otra capa

puede entrenar parámetros. Reconocer imágenes de dígitos simples es la aplicación más clásica de LeNet, ya que se creó debido a eso y si bien esto está relacionado al tema de este proyecto, se requiere identificar también las placas por lo que es mejor considerar otras opciones.

AlexNet compitió en el Desafío de reconocimiento visual a gran escala ImageNet el 30 de septiembre de 2012 y logró un error de top 5 del 15.3%, más de 10.8 puntos porcentuales menos que el del subcampeón. El resultado principal del artículo original fue que la profundidad del modelo era esencial para su alto rendimiento, que era computacionalmente costoso, pero factible debido a la utilización de unidades de procesamiento de gráficos (GPU) durante el entrenamiento. AlexNet contiene ocho capas; las primeras cinco eran capas convolucionales, algunas de ellas seguidos por capas de agrupamiento máximo, y las últimas tres eran capas completamente conectadas. Se utilizó la función de activación ReLU no saturante, que mostró un mejor rendimiento de entrenamiento en tanh y sigmoid.

El modelo **VGG16** es una serie de capas convolucionales seguidas de una o algunas capas densas (o completamente conectadas). Desde la capa de entrada hasta la última capa de agrupación máxima se considera parte de extracción de características del modelo, mientras que el resto de la red se considera parte de clasificación del modelo. Después de definir el modelo, es necesario cargar la imagen de entrada con el tamaño esperado por el modelo, en este caso, 224 × 224. A continuación, el objeto de imagen debe convertirse en una matriz de datos de píxeles y expandirse de una matriz 3D a una matriz 4D con las dimensiones de [muestras, filas, columnas, canales], donde solo tenemos una muestra. Los valores de los píxeles deben ser escalados adecuadamente para el modelo VGG 16.

Inception v3 es una red neuronal convolucional para ayudar en el análisis de imágenes y la detección de objetos. Es una red neuronal convolucional que tiene 48 capas de profundidad. Es posible cargar una versión pre entrenada de la red capacitada en más de un millón de imágenes de la base de datos ImageNet. La red pre entrenada puede clasificar imágenes en 1000 categorías de objetos, como teclado, mouse, lápiz y muchos animales. Como resultado, la red ha aprendido representaciones de características ricas para una amplia gama de imágenes.

Se consideró además **ResNet**, esta es una red neuronal que se basa en construcciones conocidas de las células piramidales en la corteza cerebral. Las redes neuronales residuales hacen esto utilizando conexiones de omisión o atajos para saltar sobre algunas capas. Los modelos típicos de ResNet se implementan con saltos de doble o triple capa que contienen no linealidades (ReLU) y normalización por lotes en el medio. Se puede usar una matriz de peso adicional para aprender los pesos de salto. En el contexto de las redes neuronales residuales, una red no residual puede describirse como una red simple. Una motivación para saltar sobre las capas es evitar el problema de la desaparición de los gradientes, reutilizando las activaciones de una capa anterior hasta que la capa adyacente aprenda sus pesos.

MobileNetV2 es una mejora significativa sobre MobileNetV1 y empuja el estado del arte para el reconocimiento visual móvil que incluye clasificación, detección de objetos y segmentación semántica. MobileNetV2 se lanza como parte de la Biblioteca de clasificación de imágenes TensorFlow-Slim. MobileNetV2 también está disponible como módulos en TF-Hub, y se pueden encontrar puntos de verificación previamente entrenados en github.

Para este proyecto, se han seleccionado tres tipos de arquitecturas pre entrenados. La primera de ellas es VGG16, la segunda es InceptionV3 y la tercera es MobilenetV2 entre todas las diferentes arquitecturas disponibles, como se muestra en la Figura 13.

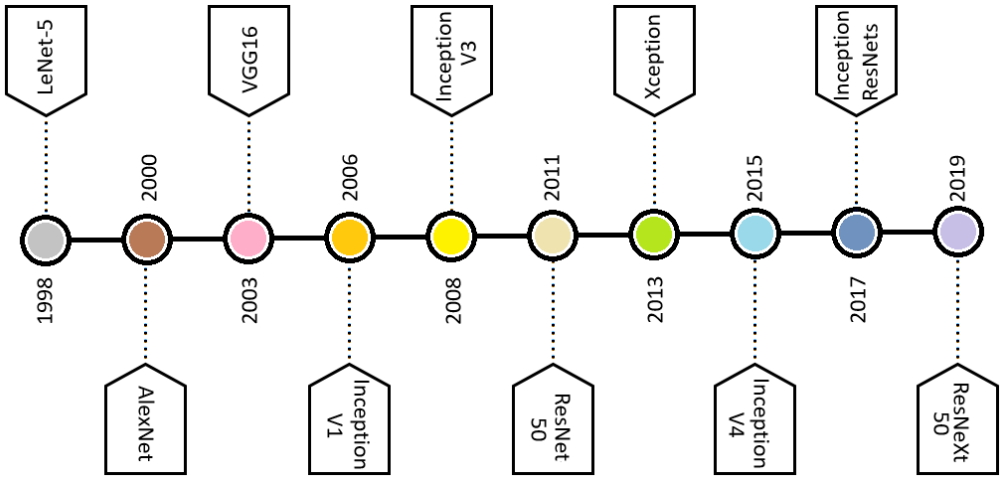

Figura 13. Arquitecturas de redes neuronales convolucionales (1998 - 2019)
Imagen basada en https://www.aismartz.com/

3.1.1 InceptionV3

Inceptionv3 es una red neuronal convolucional para ayudar en el análisis de imágenes y la detección de objetos, y comenzó como un módulo para Googlenet. Es la tercera edición de la red neuronal convolucional de inicio de Google, presentada originalmente durante el ImageNet Recognition Challenge y su código puede encontrarse en [4]. En la Figura 14 se presenta la arquitectura de Inception V3.

Así como ImageNet puede considerarse como una base de datos de objetos visuales clasificados, Inception ayuda a clasificar los objetos en el mundo de la visión por computadora. Una de sus aplicaciones científicas es en la ayuda en la investigación de la leucemia. Fue denominado “Inception” después de la película del mismo nombre.

Inception V3 fue entrenado usando un conjunto de datos de 1,000 clases del conjunto de datos original de ImageNet que fue entrenado con más de 1 millón de imágenes de entrenamiento, la versión de Tensorflow tiene 1,001 clases que se debe a una clase adicional de "fondo" no utilizado en el ImageNet original. Inception V3 fue entrenado para el Desafío de Reconocimiento Visual Grande ImageNet donde fue el primer finalista.

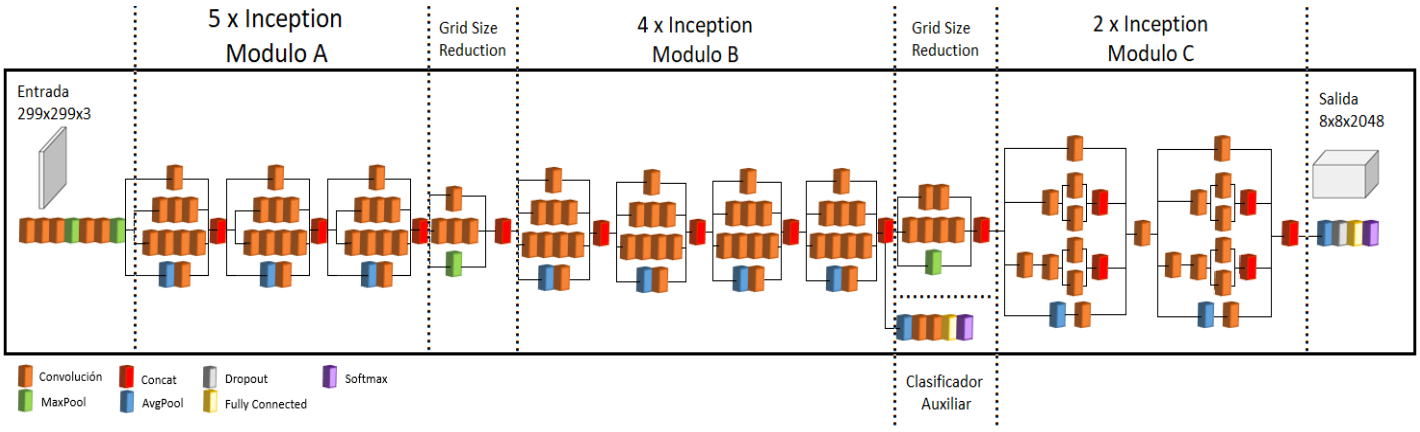

Figura 14. Arquitectura de Inception V3.

Debido a su característica de ser un modelo de reconocimiento de imágenes ampliamente utilizado que se ha demostrado que alcanza una precisión superior al 78,1% en el conjunto de datos ImageNet, se ha decidido utilizar este modelo para efectos de este proyecto. El modelo es la culminación de muchas ideas desarrolladas por múltiples investigadores a lo largo de los años. Se basa en el documento original de Szegedy, et. Al mencionado en [5].

3.1.2 MobilenetV2

MobileNetV2 es una mejora significativa de MobileNetV1 y acelera el estado del arte para el reconocimiento visual móvil que incluye clasificación, detección de objetos y segmentación semántica. Por esta razón es que también se ha tomado la decisión de implementar esta arquitectura para evaluar el proyecto.

Esta arquitectura es un extractor de funciones muy eficaz para la detección y segmentación de objetos. Por ejemplo, para la detección cuando se combina con SSDLite, el nuevo modelo es aproximadamente un 35% más rápido con la misma precisión que MobileNetV1, como se muestra en el artículo de Sandler, et. Al en [6]. La arquitectura de Mobilenet V2 se muestra en la Figura 15. Este es una representación a grandes rasgos del modelo. De forma más detallada, puede verse en la Figura 16 los detalles de cada bloque.

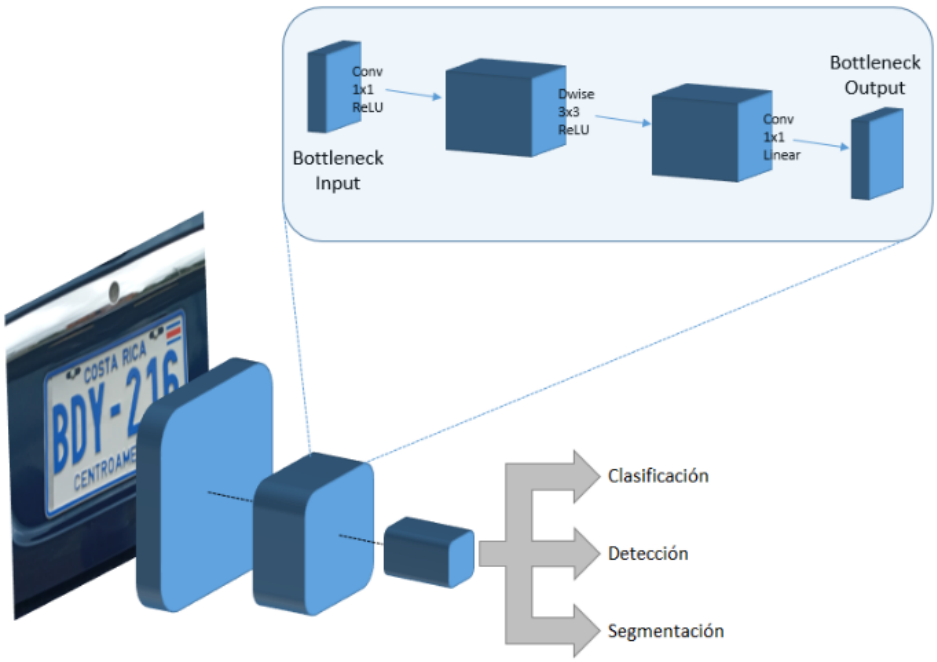

Figura 15. Arquitectura general de Mobilenet V2.

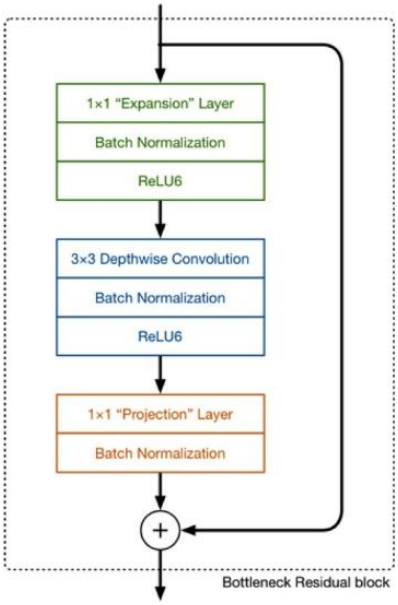

Figura 16. Detalle de los bloques de la arquitectura Mobilenet V2.

3.1.3 VGG16

VGG16 es un modelo de red neuronal convolucional propuesto por K. Simonyan y A. Zisserman de la Universidad de Oxford en [3]. El modelo alcanza el 92.7% de precisión de prueba de top 5 en ImageNet, que es un conjunto de datos de más de 14 millones de imágenes pertenecientes a 1000 clases. Fue uno de los famosos modelos presentados a ILSVRC-2014. Mejora con respecto a AlexNet al reemplazar los filtros grandes del tamaño del núcleo (11 y 5 en la primera y segunda capa convolucional, respectivamente) con múltiples filtros 3 × 3 del tamaño del núcleo uno tras otro, mostrado en la Figura 17. VGG16 fue entrenado durante semanas y estaba usando GPU NVIDIA Titan Black.

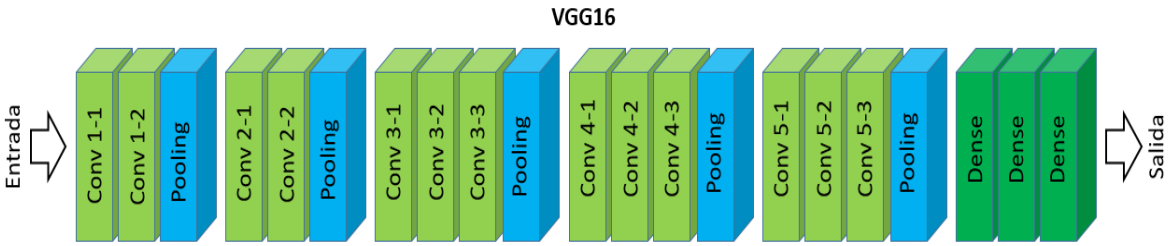

Figura 17. Arquitectura VGG16.

De la misma manera, en este proyecto se utilizan los pesos del mismo set de datos, ImageNet. Este es un conjunto de datos de más de 15 millones de imágenes de alta resolución etiquetadas que pertenecen a aproximadamente 22,000 categorías. Las imágenes fueron recopiladas de la web y etiquetadas por etiquetadoras humanas utilizando la herramienta de contratación colectiva Mechanical Turk de Amazon. El detalle de la arquitectura se puede ver en la Figura 18

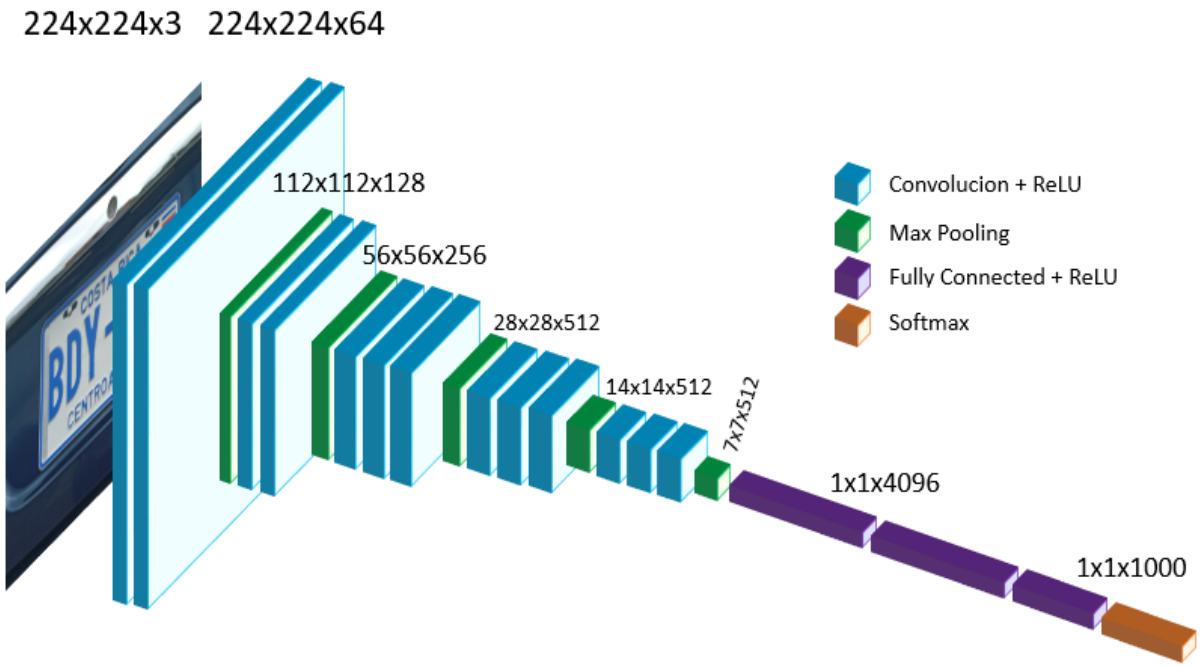

Figura 18. Arquitectura detallada de las capas de VGG16 [14].

La entrada a la capa conv1 es de imagen RGB de tamaño fijo 224 x 224. La imagen se pasa a través de una pila de capas convolucionales, donde los filtros se usaron con un campo receptivo muy pequeño: 3 × 3, que es el tamaño más pequeño para capturar la noción de izquierda / derecha, arriba / abajo, centro. En una de las configuraciones, también utiliza filtros de convolución 1 × 1, que pueden verse como una transformación lineal de los canales de entrada, seguida de la no linealidad. El stride o "zancada" de convolución se fija a 1 píxel el acolchado espacial de la capa de convolución. La entrada de capa es tal que la resolución espacial se conserva después de la convolución, es decir, el relleno es de 1 píxel para 3 × 3 capas de convolución. La agrupación espacial se lleva a cabo mediante cinco capas de agrupación máxima, que siguen algunas de las capas de convolución (no todas las capas de convolución van seguidas de una agrupación máxima). La agrupación máxima se realiza en una ventana de 2 × 2 píxeles, con zancada 2.

Todas las capas ocultas están equipadas con la no linealidad de rectificación (ReLU). También se observa que ninguna de las redes (excepto una) contiene Normalización de respuesta local (LRN), tal normalización no mejora el rendimiento en el conjunto de datos ILSVRC, pero conduce a una mayor memoria de ejecución y tiempo de cálculo.

3.2 Creación del set de datos

El set de datos de una red neuronal no es más que el conjunto de (en este caso) imágenes con las que se entrena el modelo. Pero no significa únicamente tener un directorio con un número determinado de imágenes. A partir de estas imágenes, la red neuronal convolucional obtendrá características específicas, aprendiendo así a diferenciarlas. Es por ello por lo que, conforme mayor sea el número de imágenes de entrenamiento que se tengan para entrenar la red, mejores serán los resultados.

Para el entrenamiento de los modelos, llámese este VGG16, Mobilenet V2 o Inception V3, es necesario también que el set de datos tenga el formato adecuado para la entrada al modelo. Generalmente un set de datos es un conjunto de datos tabulados correspondiente a los contenidos de una única tabla.

Para efectos de este proyecto, el set de datos está conformado por imágenes de vehículos en las cuales la placa de este ha sido debidamente anotada. Al hablar de la anotación de la placa, se hace referencia a las coordenadas dentro de la imagen en las cuales se encuentra el área de interés, es decir, la placa del vehículo.

Actualmente es posible encontrar varias herramientas gratuitas disponibles para efectuar las anotaciones en las imágenes, obteniendo como resultado una base de datos con la información de interés. La Figura 19 muestra un ejemplo de anotación en la herramienta “make sense” mencionada en [7].

La principal razón para el uso de esta herramienta sobre las demás es el hecho de que no es necesario el envío de las imágenes por internet, sino que todo el trabajo se hace en el equipo local. Esto es especialmente importante en este proyecto, ya que, para el recopilado de las imágenes, se informó a las personas que la información de las placas de sus vehículos, al ser información en cierta forma sensible, se mantendría confidencial y no circularía por ningún medio en Internet.

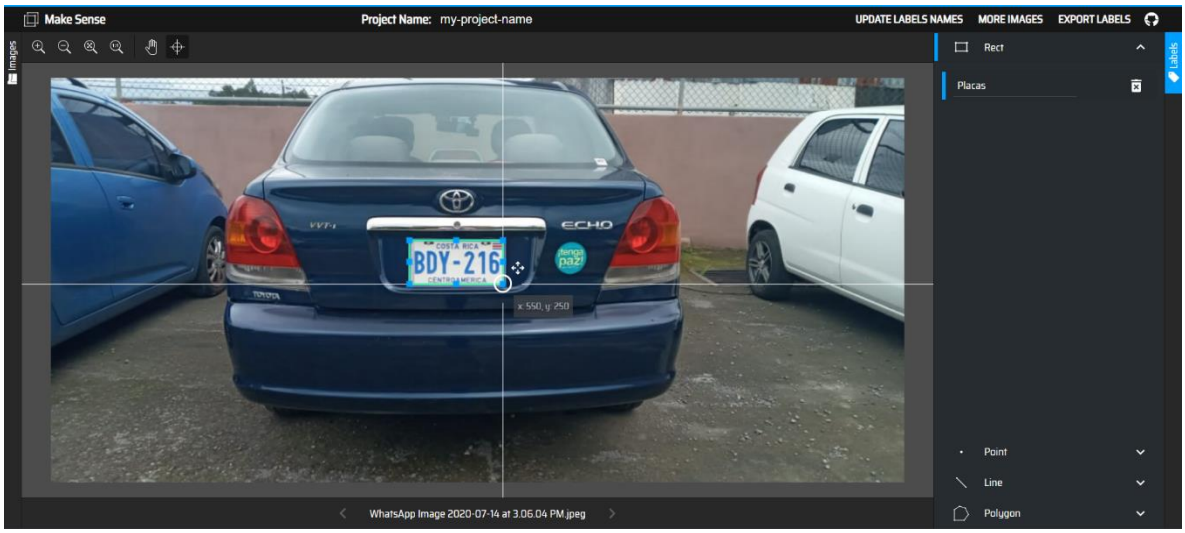

Figura 19. Interfaz de la aplicación para anotación de imágenes.

Para la creación del set de datos, se han recopilado fotografías de placas oficiales de vehículos costarricenses tanto en formato alfanumérico como en formato solamente numérico. Cabe mencionar que todas las fotografías utilizadas para la creación del set de datos fueron tomadas con autorización de los dueños de los respectivos vehículos ya que esto puede considerarse información sensible, motivo por el cual también se informó a las personas que contribuyeron con las fotografías, que las mismas serían utilizadas solamente con fin académico y no se harían circular por ninguna herramienta en Internet.

Para obtener resultados razonables, se obtuvieron 130 fotografías de placas similares a la presentada en la Figura 20. Muestra de imagen original para el set de datos.. Esto no significa que se fotografiaran 130 diferentes vehículos, sino más bien que se tomaron fotografías en

diferentes ángulos de la placa tanto trasera como delantera de un mismo vehículo. Entre los vehículos se encuentran motocicletas, vehículos livianos, vehículos de carga liviana y de transporte público, tanto taxis como autobuses.

Figura 20. Muestra de imagen original para el set de datos.

Una vez recopiladas las 130 fotografías, se utilizó un código sencillo de Python, haciendo uso de Python Image Library (PIL) de forma que fuera posible aplicar rotaciones de ángulo a las diferentes imágenes obtenidas. Se aplicaron rotaciones de 3°, 6° y 9° así como de 351°, 354° y 357° respectivamente a cada imagen para obtener el resultado mostrado en la Figura 21. Una vez aplicado este efecto, se lograron convertir las 130 fotografías originales en 910 fotografías (130 originales + (130 x 6 rotaciones)).

Figura 21. Imágenes resultantes luego de las rotaciones.

Una vez obtenidas las fotografías con rotaciones, se aplica también el efecto cambio del brillo tanto a las imágenes originales, como a sus respectivas rotaciones. Se aplica un aumento de brillo en factores de 15%, 20%, 25% y 30% y se aplica también una reducción del brillo en 5% y 2% resultando en imágenes similares a las mostradas en la Figura 22. Empíricamente se observa que estos valores son los más convenientes para la reducción y el aumento de brillos de manera que se puedan obtener fotografías resultantes que sean aun suficientemente claras como para ser consideradas válidas para el entrenamiento de los modelos. Aplicando estos efectos, podemos ahora contar con 6370 imágenes (910 + (910 x 6)).

Figura 22. Imágenes resultantes luego de los cambios de brillo.

Como paso adicional, se aplica otro efecto a las imágenes que ya se obtuvieron, de forma que se pueda ampliar el número de imágenes disponibles para el set de datos. En esta ocasión, se aplica el cambio de contraste. Es importante mencionar que al igual que el cambio de brillo, esto se aplica solamente a las 910 fotografías obtenidas luego de la rotación de las imágenes principales. Esto quiere decir que aplicamos aumentos en el contraste de la imagen de 2%, 4%, 6%, 8%, 10% y 12% sobre la imagen original. Esto nos da como resultado 5460 nuevas imágenes. Nótese que el número se reduce a 5460 ya que en esta ocasión no se cuentan las 910 imágenes originales, ya que estas fueron contadas en la aplicación del cambio de brillo, por lo cual la manera adecuada de calcular el número de nuevas imágenes obtenidas sería de la siguiente manera: 910 x 6 = 5460. Por último, como se observa en la Figura 23, con la aplicación de rotaciones, brillo y contraste, se obtiene un total de 11830 imágenes (5460 + 6370 = 11830).

Figura 23. Imágenes resultantes luego del cambio de contraste.

Una vez, obtenidas las 11830 imágenes, solamente por razones de estandarización y orden, se renombran las mismas con el formato “placa_cr###” donde “#” representa un contador, por lo que se obtiene un nombre de imagen, por ejemplo “placa_cr983.jpeg”.

Otro punto importante por mencionar es que debido a que las imágenes se obtuvieron de diferentes fuentes, enviadas por diferentes personas y diferentes dispositivos, el formato de la imagen no siempre fue el mismo. Se recibieron fotografías en formato jpg, jpeg, gif entre otras, razón por la cual se utilizó otro código sencillo en Python para convertir las imágenes a formato jpeg. Una vez convertidas las 11830 imágenes a formato jpeg, las mismas están listas para ser debidamente procesadas o ingresadas para el entrenamiento de los diferentes modelos.
Debido a los diferentes filtros y efectos aplicados a las imágenes, algunas de ellas resultaron muy oscuras, muy claras o brillantes, o simplemente el ángulo de rotación, sumado a la fotografía original, resulta en una imagen muy inclinada de forma que podría inducir a errores

en el modelo, por lo cual muchas de las fotografías se desecharon. En la Figura 24 se muestra un ejemplo de imagen desechada. Luego de descartar las imágenes, se obtuvieron 9600 imágenes válidas.

Figura 24. Imagen considerada inapropiada para el entrenamiento.

Con todo lo anterior, tenemos listas las imágenes que serán utilizadas para el entrenamiento de los modelos, sin embargo, también se debe hacer la anotación correspondiente en cada fotografía de forma que se "enseñe" al modelo qué es una placa y en donde está la misma ubicada en cada fotografía. Esta es la parte del trabajo que representa mayor consumo de tiempo debido a que esto es un trabajo que debe hacerse de forma manual como se muestra en la Figura 19.

Para la generación de las anotaciones, se utiliza la herramienta "make sense" mencionada en [7], la cual es un sitio web en el cual se puede "cargar" las fotografías y hacer las respectivas anotaciones, las cuales pueden ser luego descargadas en diferentes formatos según sea la necesidad. Es importante mencionar que las imágenes NO se envían a ningún servidor ni circulan por internet, tal como se prometió a quienes colaboraron con las mismas, ya que la aplicación es basada en web, sin embargo, ejecuta todo de forma local en la computadora.

Debido a la cantidad de tiempo que se necesita para anotar las 9600 imágenes y considerando que para los efectos de este proyecto el número de imágenes puede ser ligeramente menor, se decide anotar un total de 5069 imágenes.

Una vez terminada la anotación en las imágenes, se procede a descargar el archivo en formato json (JavaScript Object Notation), el cual contiene datos de la imagen tales como: nombre y ruta del archivo, etiqueta, coordenadas de la anotación, ancho y alto de la imagen. A continuación, se muestra un ejemplo de línea del archivo .json generado por la aplicación "make sense". Se genera una línea similar a la mostrada por cada imagen que contiene anotaciones, lo cual indica que se obtiene un archivo con 9600 diferentes líneas.

{"content":/placa_cr1.jpeg","annotation":[{"label":["Placas"],"notes":"","points":[{"x":0.417307692,"y":0.399898374},{"x":0.531971154,"y":0.514227642}],"imageWidth":4160,"imageHeight":1968}],"extras":null}

Para el entrenamiento es necesario ingresar la siguiente información a los diferentes modelos para su respectivo aprendizaje: Nombre de la imagen, ancho de la imagen, alto de la imagen y las coordenadas de la imagen donde se encuentra el rectángulo que rodea la placa en las anotaciones, similar a lo que se muestra en la Figura 25.

Figura 25. Ejemplo de coordenadas del rectángulo de anotación.

Con el fin de ingresar solamente la información que es necesaria para el entrenamiento, se filtra la información mediante la creación de un archivo del tipo csv (comma separated value), de forma que solamente se extrae del archivo .json la información relevante mencionada en el párrafo anterior. El archivo csv es un archivo mucho más simple que incluye solamente la información mostrada en el ejemplo a continuación.

placa_cr1,4160,1968,0.417307692,0.399898374,0.5319711540000001,0.514227642

El archivo csv se obtiene mediante otro sencillo código de Python, el cual lee el archivo .json obtenido mediante la herramienta de anotación y de este se extrae la información de interés, la cual es entonces almacenada en un archivo separado por comas. Cabe mencionar que este proceso solo es necesario realizarlo para el entrenamiento del primer modelo, ya que una vez obtenido el archivo csv, se puede omitir el paso de leer el archivo json y generar un nuevo archivo csv debido a que las imágenes y las anotaciones son las mismas para todos los modelos.

3.3 Entrenamiento de la red neuronal

El paso más importante para la creación del sistema completo es, sin duda, el entrenamiento de la Red Neuronal Convolucional. El algoritmo de aprendizaje de este tipo de redes permite extraer las características de cada clase a partir de un conjunto de datos de entrenamiento previamente clasificado. Para ello modifica los pesos de las neuronas que forman la red y sus valores se calculan iterativamente mediante el método backpropagation de aprendizaje supervisado. Este algoritmo de backpropagation o propagación hacia atrás, consta de dos etapas principales:

- Para cada elemento del conjunto de entrenamiento, se calcula la clase a la que pertenece según los valores que tienen los pesos de la red en ese momento. Entonces, el algoritmo determina que tan buena es dicha clasificación mediante la función de error, comparando la clasificación realizada con la clase a la que realmente pertenece dicho elemento.

- Una vez que se ha obtenido el error cometido, el algoritmo propaga hacia atrás las neuronas con pesos que aportan suficiente a la clasificación de la entrada. Con este proceso iterativo se irán actualizando los pesos para optimizar la función de error, mediante el algoritmo de descenso del gradiente. Este método actualiza los pesos de la red en la dirección opuesta del gradiente de la función de error.

Como parte del entrenamiento, además se hace una "transferencia de aprendizaje" o transfer learning, de forma que al modelo (ya sea Inception V3, Mobilenet V2 o VGG16) pre-entrenado se le cargan los pesos correspondientes. Una vez realizado esto, se agregan las capas adicionales y se entrena en "nuevo modelo" haciendo un "freeze" en las capas que ya habían sido previamente entrenadas en el modelo y se entrenan las nuevas capas agregadas, con las cuales entonces es posible identificar las placas oficiales utilizadas por los vehículos que circulan en Costa Rica.

3.4 Selección de la arquitectura

La selección de la arquitectura es uno de los aspectos más importantes para este proyecto, ya que debe cumplir con las expectativas de la Intersección sobre la Unión. Sin embargo, hay otros factores que también deben ser considerados, entre ellos uno muy importante es el tiempo de entrenamiento. En el próximo capítulo se entra más en detalle en cuanto a los términos de selección de la arquitectura, basado en los resultados obtenidos experimentalmente.

3.5 Evaluación con Intersección sobre unión

Con el fin de comprobar el funcionamiento adecuado de la red neuronal luego del entrenamiento, se hace una prueba con algunas imágenes aleatorias de forma que se pueda identificar la placa de un vehículo costarricense según lo esperado. Las imágenes de prueba utilizadas no son parte ni del set de entrenamiento ni de validación. Para obtener un resultado adecuado, es necesario calcular la intercepción sobre la unión (Intersection over unión, IoU) del área del rectángulo formado por las coordenadas esperadas y el área del rectángulo formado por la predicción del modelo. En el siguiente capítulo, en la sección *4.6 Cálculo de la intercepción sobre la unión* se detalla el resultado obtenido con esta actividad.

Debido a que el modelo identifica placas de vehículos, lo cual es una selección binaria (es una placa o no lo es) y no está haciendo una selección entre diferentes categorías es necesario implementar esta métrica adicional de precisión que se evalúa en este caso mediante la IoU.

La IoU es una métrica de evaluación que se utiliza para medir la precisión de un detector de objetos en un conjunto de datos en particular. Se decidió utilizar esta técnica ya que a menudo se utiliza en desafíos de detección de objetos, como el popular desafío PASCAL VOC y para evaluar el rendimiento de los detectores de objetos HOG + Linear SVM y los detectores de redes neuronales convolucionales (R-CNN, Faster R-CNN, YOLO, etc.).

Como requisito para aplicar la técnica de IoU, se necesitan dos factores fundamentales: los datos de las anotaciones (ground-truth bounding box) que se abarca en la sección 3.2 Creación del set de datos, y la predicción obtenida del modelo, lo cual genera un resultado similar al de la Figura 26 donde se resaltan los rectángulos rojo (para la predicción) y verde (etiqueta).

Figura 26. Ejemplo de imagen con rectángulo de predicción (rojo) y etiqueta (verde).

La intersección de la unión es calculada de la siguiente manera:

$$\text{IoU} = \text{Área de superposición} / \text{área de unión} \quad (3.1)$$

En la Figura 27, se muestra gráficamente lo que se define como superposición de dos áreas, así como lo que se entiende por unión de dos áreas para efectos del cálculo de la IoU.

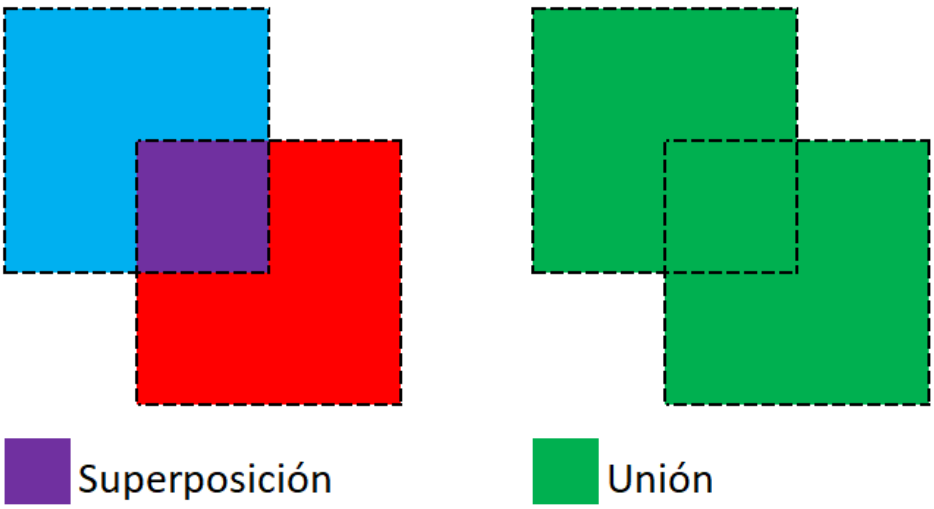

Figura 27. Demostración de la diferencia entre superposición y unión.

Nuevamente, la razón por la que es necesario utilizar esta métrica, se debe a que no se están prediciendo etiquetas de clase donde el modelo genera una sola etiqueta que es correcta o incorrecta. Este tipo de clasificación binaria simplifica la precisión informática; sin embargo, para la detección de objetos no es una manera funcional de calcular la precisión. En realidad, es extremadamente improbable que las coordenadas del rectángulo predicho por el modelo coincidan exactamente con las coordenadas del cuadro delimitador de la etiqueta.

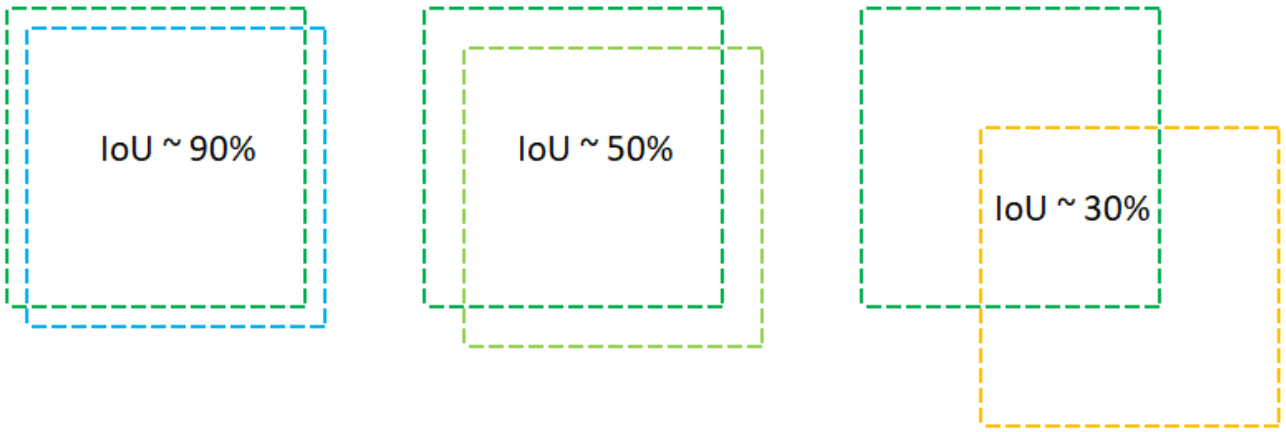

Figura 28. Diferentes ejemplos de Intercepciones de unión.

En la Figura 28, se muestra un ejemplo de los resultados de la intersección de uniones, donde según diferentes artículos [8] y de acuerdo a las pruebas experimentales, se determina que una IoU con un resultado de 50% es completamente aceptable y cumple con la función del modelo, logrando satisfactoriamente una detección adecuada de las placas de los vehículos. En la Figura 29, Figura 30 y Figura 31 se muestran ejemplos reales de diferentes valores de IoU.

Figura 29. Ejemplo de predicción con IoU = 92.98%

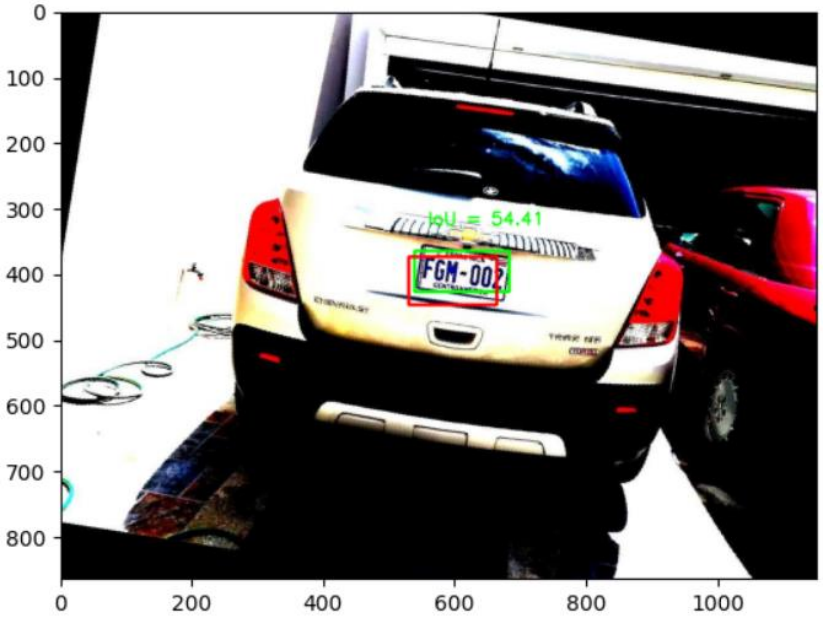

Figura 30. Ejemplo de predicción con IoU = 54.41%

Figura 31. Ejemplo de predicción con IoU = 29.87%

Capítulo 4. Resultados y análisis de la exploración

En esta sección del documento, se hará énfasis en los resultados obtenidos experimentalmente para cada arquitectura seleccionada. Se presentan los datos obtenidos, así como un análisis de estos y los pasos que conducen a la obtención de dichos resultados. Todas las gráficas, tablas y datos presentados se han obtenido mediante el entrenamiento de las redes neuronales.

Para mayor facilidad, los resultados se analizan por secciones individuales para cada arquitectura entrenada, teniendo como resultado tres secciones principales, cada una con subsecciones que explican con detalle cada paso y resultado obtenido y adicionalmente otras dos secciones principales que tratan los temas del set de datos y el hardware utilizado, ambos comunes para todas las arquitecturas.

4.1 Ambiente de desarrollo

Para el entrenamiento de las diferentes arquitecturas, se utiliza el mismo hardware y software, de forma que las comparaciones de tiempos y desempeño sean válidas y no dependan de factores como cantidad de memoria o capacidad de procesamiento entre otras variables que podrían alterar los resultados.

El hardware utilizado para las redes neuronales es una laptop HP Pavilion que cuenta con un procesador AMD A8-6410 APU con tarjeta gráfica AMD Radeon r5 x4. Se cuenta en el sistema con 4GB de memoria RAM y una partición de 180GB de disco duro.

El sistema operativo instalado en el hardware anteriormente mencionado es Ubuntu 18.04.4 LTS 64 bits. Además, es necesario el uso de diversas aplicaciones, entre ellas las más importantes mencionadas a continuación en la Tabla 1:

Tabla 1. Software utilizado para el entrenamiento de las diferentes arquitecturas.

Herramienta	Versión
Python	3.6.9
Keras	2.3.1
Pandas	1.0.4
OpenCV	4.1.1
Numpy	1.16.1
Tensorflow	1.15.3
Python Image Library	5.1.0

4.2 Inception V3

El primer modelo en ser entrenado es Inception V3. Para este modelo se han utilizado un total de 5069 imágenes que son distribuidas entre entrenamiento y validación. El modelo para utilizar como base es “sequential” aprovechando el hecho de que solo se tiene un tensor de entrada y un tensor de salida. A este modelo secuencial, se le agregan los pesos de Inception V3 que han sido pre-entrenados y cinco capas adicionales por entrenar que se muestran a continuación:

- model.add(Flatten())
- model.add(Dense(128, activation="relu"))
- model.add(Dense(64, activation="relu"))
- model.add(Dense(64, activation="relu"))
- model.add(Dense(4, activation="sigmoid"))

La forma de la arquitectura que se ha entrenado será la misma de la Figura 14 donde se muestra la arquitectura básica de modelo de Inception V3, sumándole las capas adicionales que se incorporaron para propósitos de este proyecto, lo cual resulta en lo mostrado en la Figura 32.

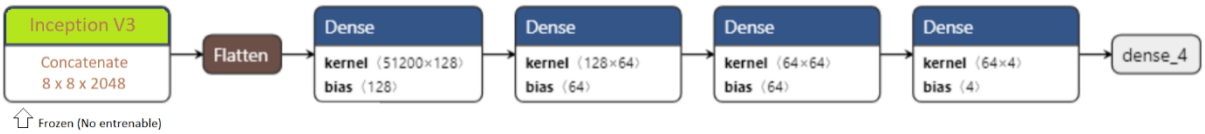

Figura 32. Capas adicionales agregadas al modelo Inception V3.

A continuación, se presenta en la Tabla 2 los valores totales de parámetros de la arquitectura:

Tabla 2. Parámetros del modelo (Inception V3).

Total de parámetros	28.37M
Parámetros entrenables	6.57M
Parámetros no entrenables	21.80M

Se han cargado los pesos del modelo pre-entrenado de Inception V3 con ImageNet, por esta razón se utiliza en el código esta línea que limita el entrenamiento a las últimas capas solamente:

```
model.layers[-6].trainable = False
```

Con esto se indica que solamente las ultimas 6 capas serán "entrenables". Además, se define el número de pasos de entrenamiento y validación de la siguiente manera:

```
STEP_SIZE_TRAIN = int(np.ceil(train_generator.n / train_generator.batch_size))
STEP_SIZE_VAL = int(np.ceil(validation_generator.n / validation_generator.batch_size))
```

Se obtiene entonces los resultados de los tamaños de validación y de entrenamiento mostrados a continuación en la Tabla 3.

Tabla 3. Tamaños de pasos de validación y entrenamiento (Inception V3).

Train step size	143
Validation step size	16

Una vez lista la información obtenida, está listo el sistema para iniciar con el entrenamiento. Durante el mismo, se monitorea por cada paso la perdida (loss) y la precisión (accurracy) de las pruebas. Al final del Epoch, se muestra la perdida y la precisión de entrenamiento, así como la perdida y la precisión de validación además del tiempo tomado en total por Epoch. La siguiente Tabla 4 muestra un resumen de los resultados del entrenamiento de la red con Inception V3. La razón por la que se utilizan 17 Epochs es porque se entrenó el modelo

inicialmente con 30 epochs y se observó que luego del epoch 17 la variación de la precisión no es significativa.

Tabla 4. Resultados del entrenamiento de la red con Inception V3.

Epoch	Time	Train Loss	Train Accurracy	Validation Loss	Validation Accurracy
1	3219s	0,1335	0,4167	0,1287	0,5996
2	3186s	0,1338	0,4185	0,1571	0,5996
3	3186s	0,1334	0,4185	0,1526	0,5996
4	3185s	0,1332	0,4185	0,177	0,5996
5	3185s	0,1338	0,4185	0,1362	0,5996
6	3190s	0,133	0,4185	0,1393	0,5996
7	3185s	0,1328	0,4185	0,1358	0,5996
8	3185s	0,1327	0,4185	0,145	0,5996
9	3184s	0,0724	0,4185	0,0483	0,5996
10	3185s	0,0139	0,7012	0,0057	0,8107
11	3194s	0,0016	0,8593	0,0041	0,6943
12	3183s	0,0012	0,8862	0,004	0,6391
13	3191s	9,80E-04	0,9055	0,0034	0
14	3190s	8,23E-04	0,9108	0,004	0,785
15	3185s	7,71E-04	0,9139	0,0051	0,6765
16	3186s	7,71E-04	0,9193	0,0048	0,7396
17	3186s	6,87E-04	0,9171	0,0028	0,7179

Se obtiene además al ejecutar el código el gráfico mostrado en la Figura 33 en el cual se compara la pérdida del modelo tanto en entrenamiento como en validación.

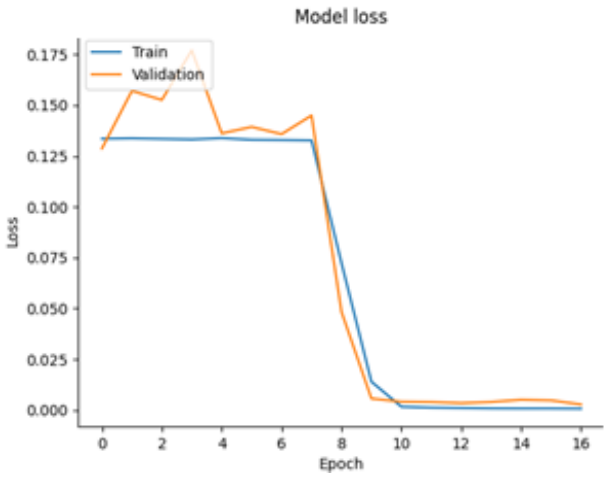

Figura 33. Gráfico de pérdida del modelo (Inception V3).

Como es de esperar, se obtiene una perdida final bastante baja, lo cual es importante para entender que tan buena es la red neuronal, mostrando esto que la misma ha aprendido y es capaz de cumplir con el objetivo de forma aceptable. Se observa claramente que luego del epoch 7 se da un descenso importante en la pérdida del modelo tanto en entrenamiento como en validación, y luego de ese instante se estabilizan los valores. En el Apéndice A – Código y Salida de consola de Inception V3 se puede encontrar el código del modelo entrenado.

Finalmente, a modo de resumen se muestra en la Tabla 5. Resumen del modelo Inception V3., los resultados para el modelo entrenado con base en la arquitectura de Inception V3.

Tabla 5. Resumen del modelo Inception V3.

Perdida	0.28%
Precisión	71.79%
Tiempo total	54205s (15h 3m 25s)

4.3 Mobilenet V2

El segundo modelo por entrenar tiene como base Mobilenet V2. Al igual que en el anterior, se han utilizado un total de 5069 imágenes que son distribuidas entre entrenamiento y validación, siendo 4562 utilizadas para entrenamiento y 507 para validación. El modelo por utilizar como base es también "sequential" aprovechando el hecho de que solo se tiene un tensor de entrada y un tensor de salida. A este modelo secuencial, se le agregan esta vez los pesos de Mobilenet V2 que han sido pre-entrenados con set de datos de ImageNet y cinco capas adicionales por entrenar que se muestran a continuación.

- model.add(Flatten())
- model.add(Dense(128, activation="relu"))
- model.add(Dense(64, activation="relu"))
- model.add(Dense(64, activation="relu"))
- model.add(Dense(4, activation="sigmoid"))

En este momento ya se puede obtener el resumen del modelo por entrenar, el cual tiene las características mostradas a continuación en la Figura 34. En esta se omite el diseño de Mobilenet V2, reemplazando el mismo por un cuadro representativo del modelo y se agregan las capas adicionales que se utilizan en este proyecto de forma que se pueda visualizar la arquitectura completa.

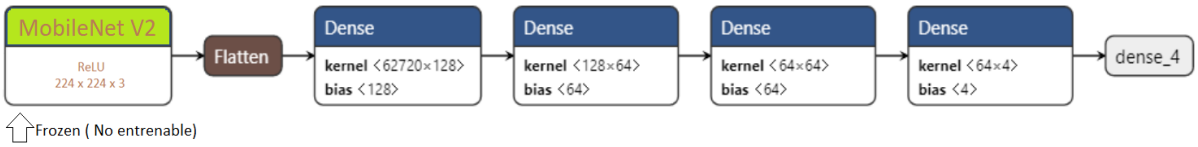

Figura 34. Resumen del modelo utilizado con Mobilenet V2.

Las funciones de activación de las diferentes capas son iguales a las que se explicaron en el modelo anterior, esto con el fin de mantener las características lo más similares posibles para poder hacer una comparación válida entre los diferentes modelos.

Cabe mencionar también que, como parte de los requisitos, se debe redimensionar las imágenes de su tamaño original, el cual no es una constante, sino que, por ejemplo, se recibieron imágenes algunas en formato horizontal y otras en vertical. Además, hay imágenes con mayor resolución que otras. El modelo necesita que se ingrese una imagen de 224x224x3, lo cual significa que la imagen debe tener un alto y un ancho de 224 pixeles y 3 canales (RGB).

En cuanto al tema de los parámetros, se tiene un cambio significativo en el número de parámetros, sin embargo, esto se debe a la diferencia en la cantidad de parámetros que se

obtienen de Inception V3 y de Mobilenet V2. A continuación se muestra en la Tabla 6 el resumen de los parámetros para Mobilenet V2.

Tabla 6. Parámetros del modelo (Mobilenet V2).

Total de parámetros	10.30M
Parámetros entrenables	8.04M
Parámetros no entrenables	2.26M

De la misma manera, no es necesario volver a entrenar el modelo pre-entrenado de Mobilenet V2, por lo cual solamente se entrenan las últimas capas que se agregaron. De la misma forma, se define el set de validación y de entrenamiento, resultando como se muestra en la Tabla 7.

Tabla 7. Tamaños de pasos de validación y entrenamiento (Mobilenet V2).

Train step size	143
Validation step size	16

Con esto, ya está listo el sistema para iniciar con el entrenamiento. Durante el mismo, se monitorea, al igual que en el caso de Inception V3 la perdida (loss) y la precisión (accurracy) de entrenamiento en cada paso. Al final del Epoch, se muestra también la perdida y la precisión de entrenamiento y de validación además del tiempo tomado en total por Epoch. La Tabla 8 muestra los resultados del entrenamiento de la red con Mobilenet V2.

Tabla 8. Resultados del entrenamiento de la red con Mobilenet V2.

Epoch	Time	Train Loss	Train Accurracy	Validation Loss	Validation Accurracy
1	1148s	0,0904	0,5741	0,0903	0,4004
2	1132s	0,0796	0,5815	0,0783	0,4004
3	1075s	0,0761	0,5815	0,0786	0,4004
4	1086s	0,0754	0,5815	0,097	0,4004
5	1065s	0,0751	0,5815	0,0854	0,4004
6	1062s	0,0751	0,5815	0,0838	0,4004
7	1062s	0,0749	0,5815	0,0778	0,4004
8	1062s	0,0749	0,5815	0,0787	0,4004
9	1064s	0,0749	0,5815	0,0798	0,4004
10	1062s	0,0748	0,5815	0,0883	0,4004
11	1070s	0,0463	0,5815	0,0396	0,4004
12	1060s	0,0342	0,587	0,0086	0,7594
13	1063s	0,0021	0,8505	0,004	0,7653
14	1061s	9,02E-04	0,9071	0,003	0,7732
15	1062s	6,73E-04	0,9174	0,0029	0,6607
16	1064s	5,74E-04	0,9272	0,004	0,7278
17	1057s	5,26E-04	0,9261	0,0047	0,7554

En el Apéndice B – Código y Salida de consola de Mobilenet V2 se puede además encontrar el código del modelo entrenado con Mobilenet V2. En la salida de consola, se puede comprobar los tiempos por paso, la perdida y precisión de entrenamiento y al final de cada epoch la pérdida y precisión de validación y el tiempo total que tomó el sistema en completar el epoch. Al final

del archivo, luego del último epoch se muestra el valor final de perdida y de precisión de entrenamiento.

En la Figura 35 se muestra el gráfico en el que se compara la perdida tanto en entrenamiento como en validación para el modelo que se ha entrenado con base a Mobilenet V2.

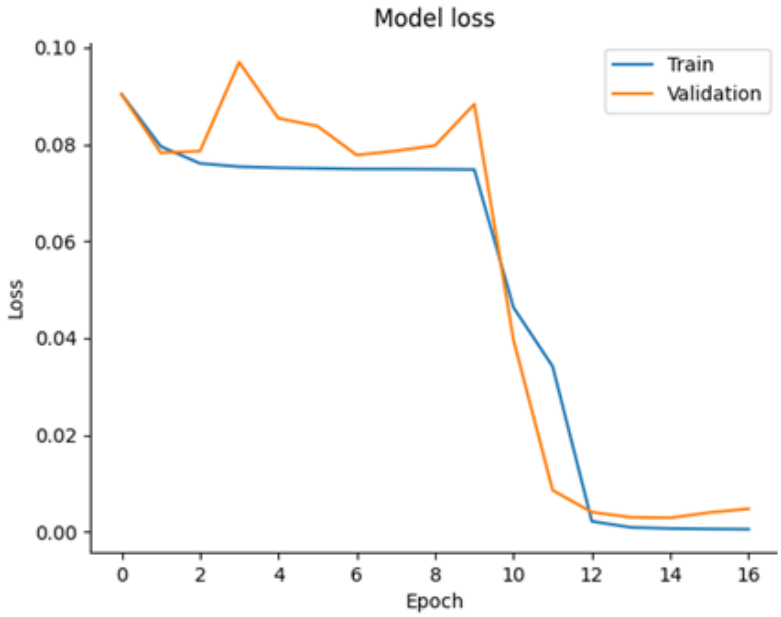

Figura 35. Gráfico de pérdida del modelo (Mobilenet V2).

Para finalizar, se tiene también para este modelo en la Tabla 9 un resumen del modelo que presenta algunas de las métricas de mayor interés, las cuales son tiempo total de entrenamiento, precisión y pérdida en validación.

Tabla 9. Resumen del modelo Mobilenet V2.

Pérdida	0.47%
Precisión	75.54%
Tiempo total	18255s (5h 4m 15s)

Es preciso recordar que Mobilenet V2 es una red convolucional que tiene una profundidad de 53 capas y en este caso, ha sido entrenada con más de un millón de imágenes que se encuentran en la base de datos de ImageNet. Esta red pre entrenada puede clasificar imágenes en más de 1000 diferentes categorías, como por ejemplo teclados, lápices, animales, entre otras. Como resultado de este entrenamiento extensivo, la red aprende una gran variedad de características de un amplio rango de imágenes.

La entrada de la red debe ser de 224 x 224, razón por la cual fue necesario el redimensionamiento, como ya se explicó anteriormente. En Mobilenet V2 hay dos diferentes tipos de bloques, uno residual con stride = 1 y otro de reducción con stride = 2. Hay tres capas para cada tipo de bloque, la primera convolucional de 1 x 1 con activación ReLU6, la segunda es de convolución profunda y la tercera es otra capa convolucional de 1 x 1 con no linealidad. Se comenta que, si se utiliza ReLU nuevamente, las redes ocultas solo tendrían la capacidad de clasificación lineal en la parte del dominio de salida diferente de cero.

4.4 VGG16

Al igual que en los modelos anteriores, para el entrenamiento de VGG16, se hace uso de 5069 imágenes que son distribuidas entre entrenamiento y validación, siendo 4562 utilizadas para entrenamiento y 507 para validación. El modelo por utilizar sigue siendo “sequential” y se le agregan esta vez los pesos de VGG16 que han sido pre-entrenados con set de datos de ImageNet y las mismas capas adicionales por entrenar que se muestran a continuación.

- model.add(Flatten())
- model.add(Dense(128, activation="relu"))
- model.add(Dense(64, activation="relu"))
- model.add(Dense(64, activation="relu"))
- model.add(Dense(4, activation="sigmoid"))

Al igual que los modelos anteriores, VGG16 también requiere una entrada de 224 x 224 x 3. En la Figura 36 a continuación se omite el modelo ya conocido de VGG16, simulándolo con un solo cuadro representativo y se muestran las capas adicionales que se han incorporado al modelo utilizado para este proyecto.

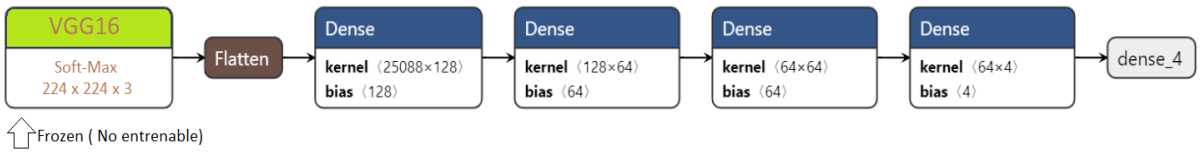

Figura 36. Capas adicionales agregadas al modelo VGG16.

La arquitectura de VGG16 está conformada por 16 capas, y al igual que Inception V3 y Mobilenet V2, esta ha sido entrenada con el set de datos de ImageNet, el cual tiene un tamaño de 528MB. Una de las principales desventajas de este modelo, es que es especialmente lento en cuanto a tiempo de entrenamiento.

De las 16 capas, las primeras dos tienen 64 canales con filtros de 3 x 3, posteriormente, una capa de max pool de stride (2, 2) seguida de otras dos capas de convolución y una capa más de max pooling idéntica a la primera. Estas son seguidas por tres bloques conformados cada uno por 3 capas convolucionales y una de max pooling, estas con filtros de 3x3. Algunas capas también tienen un padding de un píxel después de cada convolución para evitar el desplazamiento espacial de la imagen.

De la misma manera que en los casos de Inception V3 y Mobilenet V2, para VGG16 también se han utilizado las mismas capas y funciones de activación, con el fin de mantener las características de los modelos lo más similares posibles para hacer una comparación justa entre los mismos. El redimensionamiento de las imágenes de igual forma es de 224 x 224 x 3. En la Tabla 10 a continuación, se muestra el resumen de los parámetros para el modelo entrenado con base en VGG16.

Tabla 10. Parámetros del modelo (VGG16).

Total de parámetros	17.94M
Parámetros entrenables	3.23M
Parámetros no entrenables	14.71M

Se define además de la misma manera la cantidad de pasos de validación y entrenamiento, con el mismo resultado de los modelos anteriores, según se muestra en la Tabla 11.

Tabla 11. Tamaños de pasos de validación y entrenamiento (VGG16).

Train step size	143
Validation step size	16

Ya en este punto, inicia el entrenamiento, de donde podemos obtener la información de pérdida y precisión tanto por cada paso como por epoch. Nuevamente, se obtienen los datos de entrenamiento por cada paso y los de validación por cada epoch. A continuación, se muestra en la Tabla 12 los resultados por cada epoch tanto de entrenamiento como de validación para este modelo.

Tabla 12. Resultados del entrenamiento de la red con VGG16.

Epoch	Time	Test Loss	Test Accurracy	Validation Loss	Validation Accurracy
1	15709s	0,0031	0,8466	0,0013	0,9071
2	15677s	7,73E-04	0,9121	0,002	0,8953
3	15680s	5,36E-04	0,9305	0,0011	0,9111
4	15682s	4,60E-04	0,9325	6,48E-04	0,9348
5	15680s	3,76E-04	0,9492	8,37E-04	0,9071
6	15669s	3,19E-04	0,9494	7,74E-04	0,9091
7	15665s	2,95E-04	0,9527	0,0016	0,9328
8	15655s	2,50E-04	0,9518	0,0021	0,9249
9	15652s	2,54E-04	0,9562	0,0017	0,9387
10	15654s	2,54E-04	0,9522	0,0014	0,9328
11	15681s	2,33E-04	0,9546	0,0019	0,9427
12	15663s	1,83E-04	0,9599	7,25E-04	0,9111
13	15649s	1,84E-04	0,9584	0,0011	0,9605
14	15655s	1,91E-04	0,9566	0,002	0,9407
15	15650s	1,56E-04	0,9641	4,41E-04	0,9506
16	15651s	2,00E-04	0,9522	0,0029	0,9229
17	15685s	1,66E-04	0,9621	0,0025	0,9545

En el Apéndice C – Código y Salida de consola de VGG16 se puede encontrar el código del modelo entrenado con VGG16. En la salida de consola, se puede comprobar los tiempos, la perdida y precisión de entrenamiento y epoch, así como el tiempo total que tomó el sistema en completar el epoch. Al final del archivo, luego del último epoch se muestra el valor final de perdida y de precisión de entrenamiento y de validación.

En la Figura 37 se presenta el gráfico de comparación entre la perdida en entrenamiento y la pérdida en validación para el modelo que se ha entrenado con base a VGG16.

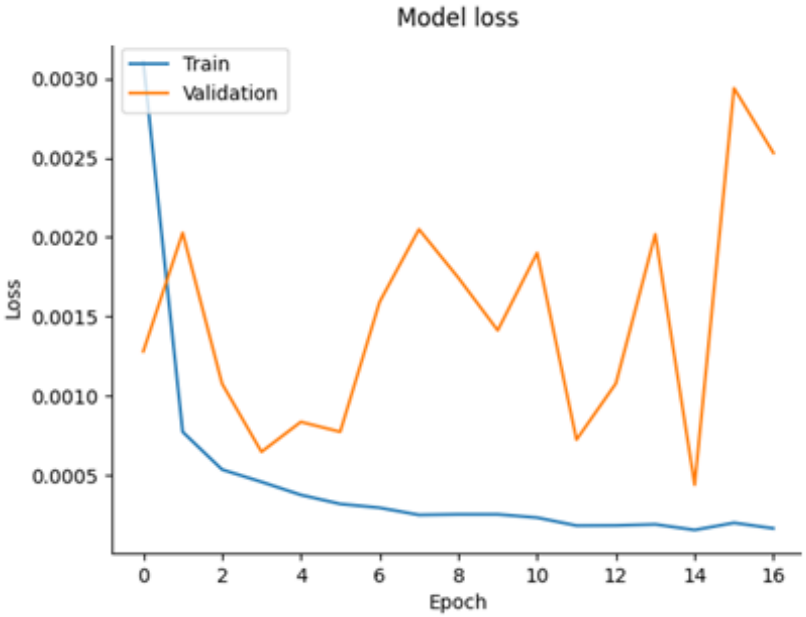

Figura 37. Gráfico de pérdida del modelo (VGG16).

Es interesante notar que, en esta comparación, se observa una fluctuación en la pérdida de validación mientras que la pérdida de entrenamiento se mantiene más estable. Recordando que la pérdida de validación es el error después de ejecutar el conjunto de datos de validación a través de la red capacitada, esto podría hacer creer que el modelo tiene algunos problemas o que no es suficientemente bueno, sin embargo, podemos ver más adelante que esto no es totalmente cierto. Dado que la selección de imágenes de validación se realiza de forma aleatoria, es posible que haya algunas imágenes que no son totalmente claras o adecuadas para que sean identificadas de forma correcta causando las fallas en la validación.

Para el entrenamiento de este modelo, fue necesario mantener el hardware encendido durante varios días de forma que pudiera terminar el entrenamiento con los mismos 143 pasos de los 17 epochs que se utilizaron en cada uno de los modelos. A continuación, se muestra en la Tabla 13 un resumen con el tiempo total necesario para entrenar el modelo y los valores de precisión y pérdida.

Tabla 13. Resumen del modelo VGG16.

Pérdida	0.25%
Precisión	95.45%
Tiempo total	266357s (3d 1h 59m 17s)

4.5 Comparación de los modelos

A continuación, se hace una comparación de los parámetros más importantes en los tres modelos entrenados mencionados anteriormente, el primero basado en Inception V3, el segundo basado en Mobilenet V2 y finalmente el que se basa en VGG16.

La primera comparación es en términos del tiempo necesario para completar el entrenamiento en cada modelo. Al comparar estos tiempos, se nota una duración mucho mayor en el modelo

entrenado con base en VGG16. Comparando la cantidad de tiempo que se necesitó para entrenar este modelo con respecto a Inception V3, VGG16 necesitó cerca de 5 veces (4.91) la cantidad de tiempo ocupada por Inception V3.

Al comparar el modelo de VGG16 con respecto a Mobilenet V2, que cabe mencionar fue el modelo más rápido, la diferencia es todavía más notoria, ya que el tiempo tomado por VGG16 fue casi 15 veces (14.59) superior al necesitado por Mobilenet V2.

Al comparar los modelos Mobilenet V2 e Inception V3 se observa una diferencia mucho menor entre ambos, donde Inception V3 necesitó casi 3 veces (2.97) más tiempo para completar el entrenamiento.

A continuación, se presenta en la Figura 38 una comparación entre los tiempos totales de entrenamiento.

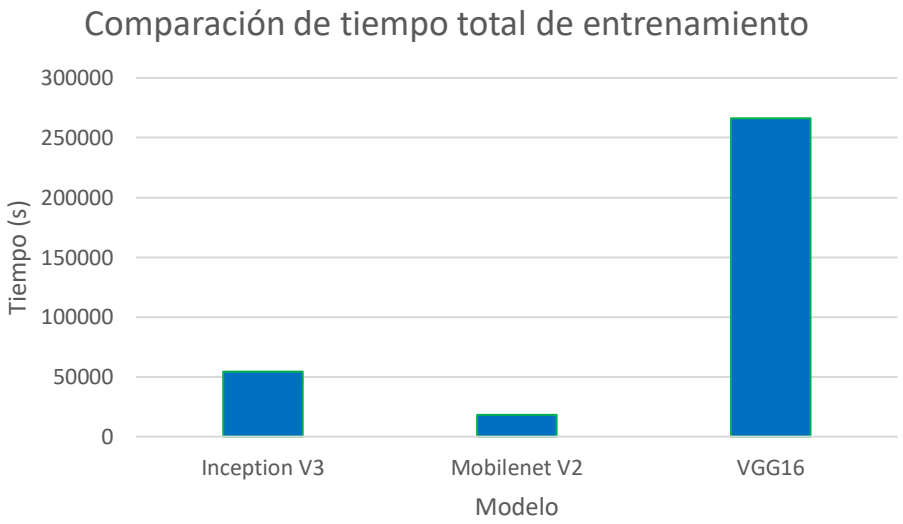

Figura 38. Comparación de tiempo total de entrenamiento de los modelos.

Con base en la información de los tiempos, es fácil demostrar que, en términos de rapidez de entrenamiento, el modelo basado en Mobilenet V2 es el más eficiente y el modelo más lento por una gran diferencia es el basado en VGG16.

A continuación, se comparan los valores de perdida en cada modelo para la parte de entrenamiento. Cuando se habla acerca de la pérdida, se hace referencia a la medida cuantitativa de la desviación o diferencia entre el resultado previsto y el resultado real esperado. Nos da la medida de los errores cometidos por la red al predecir la salida. En la Tabla 14 se muestran estos valores para cada modelo considerado para este proyecto.

Tabla 14. Valores de perdida en entrenamiento para los diferentes modelos.

Inception V3	Mobilenet V2	VGG16
0,00068662	0,00052561	**0,00016575**

A partir de la tabla, se puede observar que la pérdida en VGG16 es mucho menor que en los otros 2 modelos y la pérdida más grande se da en el modelo entrenado con base en Inception V3.

Con respecto a la pérdida en la etapa de validación, es importante realizar un análisis similar. Al igual que en la parte de entrenamiento, la perdida en la validación hace referencia que tanto se desvía el resultado obtenido del resultado esperado. En la Tabla 15 se muestran los valores de la perdida de cada modelo en la fase de validación.

Tabla 15. Valores de perdida en validación para los diferentes modelos.

Inception V3	Mobilenet V2	VGG16
0,0028	*0,0047*	**0,0025**

En términos de la pérdida en etapa de validación, los números son un poco más cercanos entre los modelos, sin embargo, sigue teniendo el mejor resultado el modelo entrenado con base en VGG16, con un valor mínimo de perdida en validación de 0.0004 y máximo de 0.0029, mientras que los modelos basados en Inception V3 y Mobilenet V2 respectivamente tuvieron valores mínimos de 0.0028 y 0.0029 y máximos de 0.1770 y 0.0970. Interesante notar que el máximo valor que se obtuvo con VGG16 fue el valor mínimo que se pudo conseguir con Mobilenet V2 y aunque Inception V3 y Mobilenet V2 tuvieron una gran disminución en sus valores, sigue sin ser suficiente para mejorar los resultados obtenidos con VGG16.

Otra métrica de gran importancia para valorar el comportamiento adecuado de los modelos es la precisión. A continuación, se hace la comparación de las precisiones de los diferentes modelos, primeramente, en entrenamiento y posteriormente en validación.

En la Tabla 16 se muestran los valores de precisión obtenidos experimentalmente para los tres modelos. Nótese que los valores de precisión se expresan en porcentajes, permitiendo esto una mejor comprensión de lo bien o mal que se comporta el modelo en cada etapa.

Tabla 16. Valores de precisión en entrenamiento para los diferentes modelos.

Inception V3	Mobilenet V2	VGG16
91,71%	92,61%	**96,21%**

Es fácil determinar a partir de los resultados mostrados que el modelo que mejor se desempeña, al menos en términos de precisión en la etapa de entrenamiento es VGG16 con una precisión superior al 95%, mientras que los otros dos modelos no superan el 93% de precisión.

En definitiva, hablando en términos de precisión en etapa de entrenamiento, el modelo con el mejor desempeño para el propósito de este proyecto es el basado en VGG16 ya que es el que alcanza una mayor precisión.

Finamente, es importante también comparar los resultados de precisión de cada modelo en su etapa de validación. En la Figura 39, mostrada a continuación, se presentan los valores experimentales obtenidos en cada modelo de la precisión de este en su etapa de validación todo esto expresado en porcentajes.

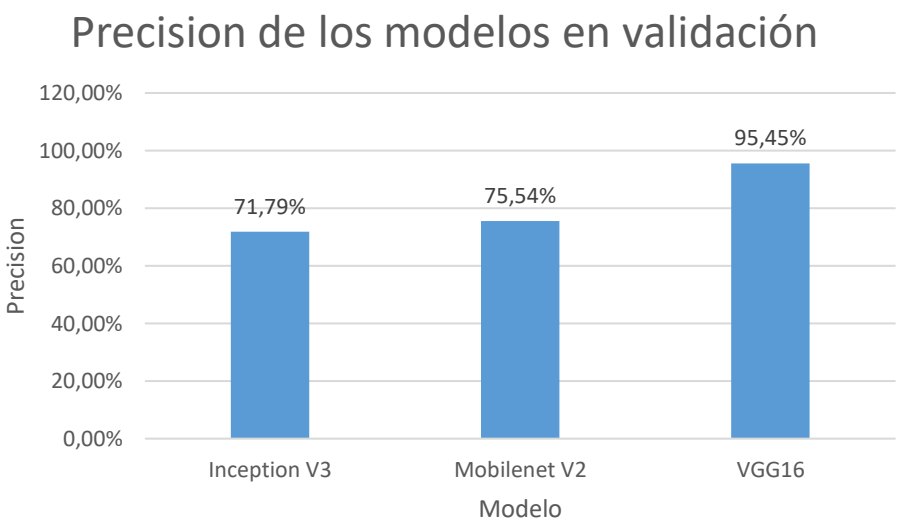

Figura 39. Precisión de los modelos en validación.

Puede observarse en los resultados de la tabla, que el mejor valor de precisión, esta vez en fase de validación, pertenece nuevamente al modelo VGG16, el cual indiscutiblemente logra un porcentaje mucho mayor, nuevamente superando el 95%, casi 20% por encima de lo que logró el segundo mejor modelo.

Puede además notarse una diferencia no tan marcada entre los modelos Inception V3 y Mobilenet V2, donde el peor porcentaje de precisión en validación se da para Inception V3, alcanzando apenas un 71.79% de precisión, habiendo una diferencia de casi un 4% entre ellos.

Habiendo comparado tanto la precisión como la pérdida de los tres modelos entrenados, tanto en su etapa de validación como en su etapa de prueba y teniendo como referencia el tiempo de entrenamiento de cada modelo, se puede hacer el resumen mostrado en la Tabla 17.

Tabla 17. Resumen de las principales métricas de los modelos.

	Inception V3	Mobilenet V2	VGG16
Tiempo de entrenamiento	54205s	**18255s**	*266357s*
Pérdida en entrenamiento	*0,000687*	0,000526	**0,000166**
Precisión en entrenamiento	*91,71%*	92,61%	**96,21%**
Pérdida en validación	0,0028	*0,0047*	**0,0025**
Precisión en validación	*71,79%*	75,54%	**95,45%**

4.6 Cálculo de la intercepción sobre la unión

El cálculo de la IoU se logra fácilmente mediante una función que se creó e incorporó al mismo código en el cual se ejecuta la predicción del modelo. En esta función se encuentran cuatro variables tx, ty, bx y by que corresponden a los valores de la etiqueta o "ground truth". Los valores obtenidos de la predicción del modelo se provienen de las variables xt, yt, xb y yb. Al ingresar estos valores a la función, se aplica la ecuación (3.1) lo cual da como resultado el valor de la IoU.

Con el fin de obtener el valor de la precisión de los modelos, se calculó la IoU para las 5074 imágenes. Una vez realizado esto, se obtuvo el porcentaje de imágenes con un valor igual o superior al 50% para la IoU, valor que se utilizó como referencia para definir la precisión de los diferentes modelos entrenados. Estos resultados se muestran en la Tabla 18.

Tabla 18. Precisión de los modelos mediante el cálculo de la IoU.

	IoU
Inception V3	72.08%
Mobilenet V2	*69.71%*
VGG16	***78.20%***

4.7 Selección del modelo

A pesar de que el entrenamiento del modelo VGG16 ha tardado 5 veces más que Inception V3 y casi 15 veces más que Mobilenet V2, esta tarea de entrenamiento se ejecuta una sola vez, ya que cuando se tiene el modelo entrenado, el mismo se guarda en dos formatos, primero se guarda en formato .h5 y luego en formato .pb. Al obtener estos archivos es posible entonces cargar el modelo entrenado, eliminando la necesidad de pasar nuevamente por el proceso de entrenamiento.

Además, uno de los factores más importantes es la precisión en la etapa de validación. Aunque es importante también que el valor de precisión en entrenamiento sea adecuado siendo este el que muestra el progreso del modelo, el valor de precisión en la etapa de validación es el que muestra verdaderamente que tan bien o mal se desempeña el modelo en la realidad.

Teniendo esto como referencia, es indudablemente el modelo entrenado con VGG16 el que mejor cumple con los requerimientos, especialmente el IoU, mostrando valores superiores a los de los otros dos modelos. VGG16 cumple con una IoU superior al 78%, mientras que los otros modelos no superan el 73% de IoU, por esta razón se selecciona el modelo VGG16 como el modelo con el que se continúa para la segunda parte del proyecto.

4.8 Identificación del último dígito de la placa

Como parte fundamental del proyecto, necesaria para poder completar con éxito la primera parte de este, se debe identificar el último dígito de la placa que se detecte en la fotografía. Esto quiere decir que, al ingresar una fotografía, el modelo debe ser capaz de identificar el lugar donde está la placa del vehículo y a través de esa identificación, lograr identificar el último

dígito de esta, pudiendo así determinar el día de la semana en que dicho vehículo tiene restricción de tránsito.

Con el fin de identificar este último dígito, el modelo produce un recorte de la imagen, en el cual solamente se mantiene la sección de la imagen en la que se ha detectado la placa y almacena temporalmente la imagen recortada. En la Figura 40 se muestra la fotografía original y en la Figura 41 la imagen recortada que muestra únicamente el recorte del número de la placa.

Figura 40. Imagen de muestra para la detección de caracteres.

Figura 41. Imagen obtenida al recortar el número de placa.

Una vez obtenida esta imagen, se aplica un reconocimiento óptico de caracteres, en el cual solamente es de interés los caracteres numéricos, ya que toda placa oficial de Costa Rica debe terminar en al menos 3 dígitos numéricos, sea esta del nuevo formato alfanumérico o del formato anterior que incluye solamente números.

La aplicación de este reconocimiento de caracteres se realiza a través de la siguiente línea de código, donde "last" es el nombre de la variable donde se guarda el resultado de la aplicación del reconocimiento de caracteres:

```
last = pytesseract.image_to_string(c_image, config='--psm 10 --oem 3 -c tessedit_char_whitelist=0123456789')
```

La función "pytesseract" cuenta con diversos modos de pagesegmode (--psm), en este caso se selecciona el modo 10 el cual corresponde a la configuración que trata la imagen como un solo carácter ya que solo es de interés el último de estos.

La opción de OCR Engine mode (--oem) se define como la opción 3, lo cual significa que utiliza el engine por defecto según lo que haya disponible en el sistema.

Al final del comando, se establece además una lista de caracteres de interés o “White list” donde se listan todos los números desde el 0 hasta el 9, que son los únicos caracteres que son de interés para este proyecto.

Una vez ejecutado este código, se obtiene en consola la información del último dígito de la placa y el día en que dicho vehículo tendrá restricción de tránsito. Un ejemplo de lo que se obtiene en la consola se muestra a continuación en la Figura 42.

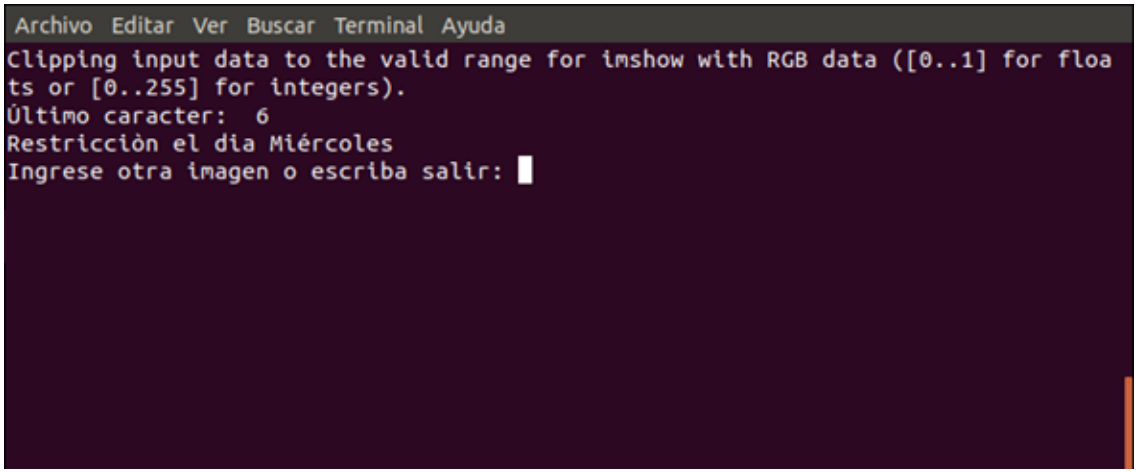

Figura 42. Salida de consola con reconocimiento de carácter y día de restricción.

Como se observa en la imagen, el modelo es capaz de cumplir con la función e identificar el último número de la placa de la imagen y mostrar el día de restricción vehicular para dicho número de placa.

Capítulo 5. Etapa II – Aplicación y comparación de optimizaciones de modelos para ejecución en GPUs embebidos

En esta sección se presenta con mayor detalle la aplicación de las optimizaciones al modelo seleccionado en la primera etapa, el cual se basa en la arquitectura VGG16.

Para el desarrollo de las actividades descritas en este capítulo, se utilizó como plataforma de hardware el kit para desarrolladores Jetson Nano, mostrado en la Figura 43. Este Kit consta de una GPU 128-core NVIDIA Maxwell y un procesador Quad-core ARM A57 con una frecuencia de 1.43 GHz. Además, cuenta con una memoria de 4GB 64-bit LPDDR4 25.6GB/s, todo esto en una pequeña tarjeta con las siguientes dimensiones: 100 mm x 80 mm x 29 mm [11].

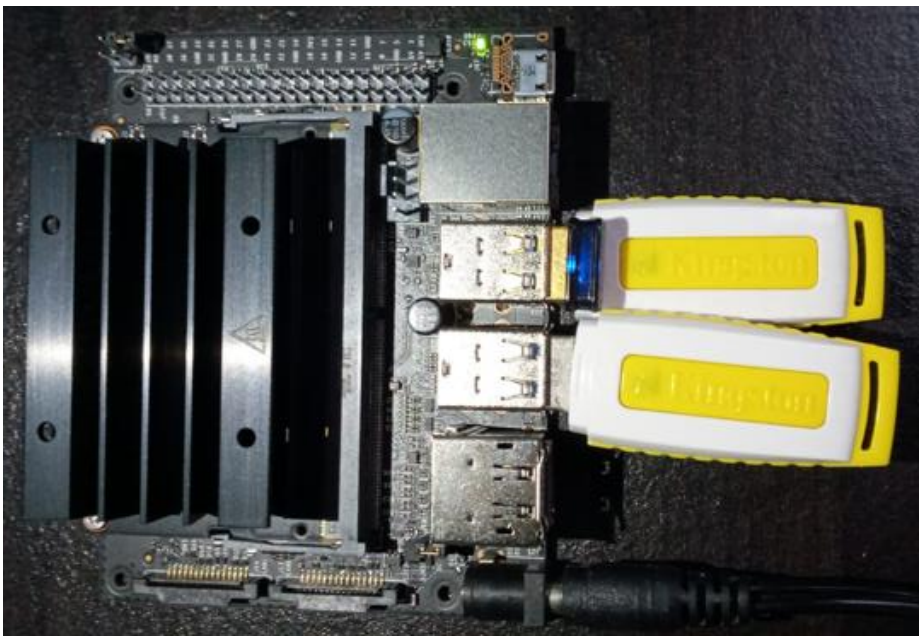

Figura 43. Hardware utilizado para el desarrollo del proyecto.

5.1 Tipos de optimizaciones

Aunque existen diversos tipos de optimizaciones, la aplicación de estos depende mucho del tipo de aplicación para la cual fue creado el modelo y lo que se desee lograr con la o las optimizaciones. A continuación, se describen brevemente las optimizaciones más populares actualmente.

5.1.1 Cuantización

La cuantización funciona reduciendo la precisión de los números utilizados para representar los parámetros de un modelo, que por defecto son números de punto flotante de 32 bits. Esto da como resultado un tamaño de modelo más pequeño y un cálculo más rápido. Para efectos

de este trabajo, se realizó cuantización post-entrenamiento de punto flotante de 16 bits (floating point 16 o FP16) y entero 8 bits (Integer 8 o INT8). La cuantización post-entrenamiento es una técnica de conversión que puede reducir el tamaño del modelo y al mismo tiempo mejorar su tiempo de ejecución. Con FP16 se obtiene una reducción del tamaño de un 50% mientras que con INT8 se obtiene una reducción del 75%, nuevamente, a cambio (teóricamente) de un sacrificio en la precisión del modelo.

5.1.2 Pruning

El Pruning funciona eliminando parámetros dentro de un modelo que solo tienen un impacto menor en sus predicciones, haciéndolo muy útil para reducir el tamaño del modelo. El pruning basado en la magnitud pone a cero los pesos de los modelos de forma gradual durante el proceso de entrenamiento. Esto permite lograr el mayor número de ceros en las matrices del modelo, lo que lo hace más fácil de comprimir, permitiendo además omitir los ceros durante la inferencia para mejorar su tiempo de ejecución, aunque esta técnica aún continúa en etapa de evaluación. Se ha visto mejoras hasta 6 veces mayores en la compresión del modelo con una mínima pérdida de precisión según la documentación actual [9].

5.2 Frameworks de Deep learning

Como parte de la investigación para esta segunda etapa, se decidió utilizar diferentes frameworks para la ejecución del modelo seleccionado en la primera etapa del proyecto. Entre las seleccionadas para realizar las pruebas, tenemos el modelo base, el cual es un modelo de Keras (.h5), el cual también fue convertido a protobuf (.pb) para utilizarse con TensorRT. Adicionalmente, el modelo también fue convertido a Tensorflow lite (.TFLite) y finalmente se compiló con Relay para su ejecución con TVM, los cuales se comentan brevemente a continuación. Cabe mencionar que Relay es una *representación intermedia* (código utilizado internamente por el compilador para representar el código fuente) de alto nivel para TVM cuyo objetivo es reemplazar este código interno basado en gráficos de cálculo con código más expresivo que pueda ser optimizado para diversos objetivos de una manera más eficiente [12].

5.2.1 Métricas

Para la comparación de las diferentes conversiones, se hicieron cuatro mediciones especificas en cada ejecución, de forma que permita la evaluación justa de cada plataforma. Estas mediciones son el tiempo de ejecución (de la predicción), la IoU del modelo, el consumo de la memoria durante la predicción y el tamaño del archivo.

Tiempo de ejecución:

Para la medición del tiempo de ejecución, se creó un código específico para cada plataforma de manera tal que se pueda calcular exactamente la parte en la que el modelo hace la predicción de las coordenadas en las que se detecta la placa, la manera en la que se realizó para cada plataforma se describe más adelante.

Intersection over unión (IoU):

Para lograr comparar las perdidas (o ganancias) en la IoU del modelo con las diferentes optimizaciones, al obtener las predicciones del modelo, se calculó la misma en todo el set de datos, es decir en las 5073 imágenes, de forma que se pudiera obtener un promedio lo más exacto posible para los diferentes frameworks y optimizaciones. La forma en que se calculó el IoU se define en la sección *4.6 Cálculo de la intercepción sobre la unión*. Una vez obtenido el valor de IoU, se obtuvo el número de predicciones que superan el 50%. Conociendo este número, se puede conocer el porcentaje de imágenes con respecto al total (5073) que superaron el valor mínimo aceptable, lo cual brinda el porcentaje que se define como la IoU del modelo para efectos de este proyecto, como lo muestra la ecuación 5.1.

$$\%\ de\ IoU = \left(\frac{\#\ de\ predicciones\ mayor\ a\ 50\%}{total\ de\ imagenes}\right) * 100 \qquad (5.1)$$

Memoria de ejecución:

Como tercera métrica se establece la medición de la memoria de ejecución utilizada para predecir los resultados, esto se logra utilizando el NVIDIA Visual Profiler (nvvp). Esta es una herramienta de generación de perfiles de rendimiento multiplataforma que ofrece a los desarrolladores comentarios vitales para optimizar las aplicaciones CUDA. Entre los parámetros que se pueden obtener se encuentra el uso y transferencias de memoria por parte del GPU al ejecutarse un proceso.

El nvvp se ejecuta de forma remota para hacer el Profiling en el dispositivo embebido. Para este proyecto, el sistema operativo que corre en dicho dispositivo es Ubuntu 18.04, por lo que la forma adecuada de ejecutar el nvvp (según la documentación de NVIDIA) es utilizando el comando "*nvvp -vm /usr/lib/jvm/java-8-openjdk-amd64/jre/bin/java*". Con la práctica se descubrió que también es necesario ejecutar el comando con privilegios de super usuario, por lo cual es necesario ejecutar con "*sudo*".

La herramienta nvvp se conecta con el dispositivo embebido por medio de SSH, sin embargo, también se descubrió con la práctica que es necesario ya sea utilizar un usuario con privilegios en el dispositivo remoto o habilitar e ingresar por SSH con usuario "root", de lo contrario nvvp tendrá errores de acceso y no logrará hacer el Profiling del sistema. En la Figura 44 se muestra un ejemplo de la ejecución de nvvp para la medición de la memoria del modelo de TensorRT con cuantización INT8.

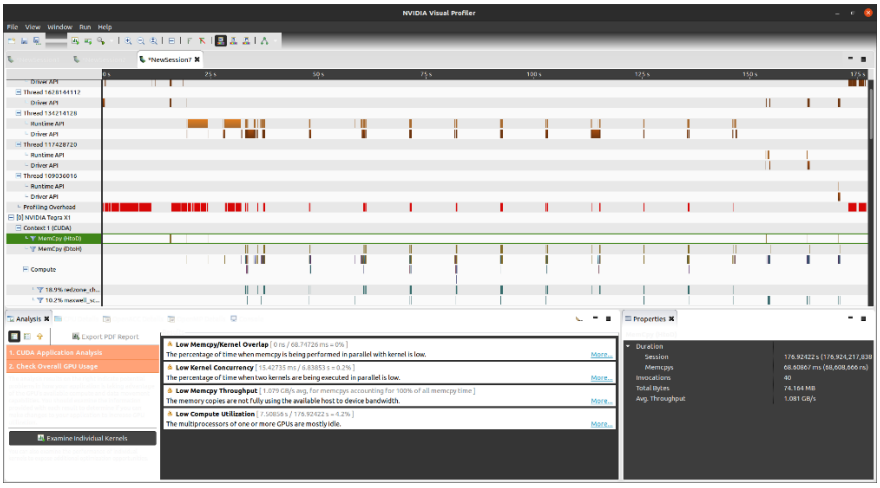

Figura 44. Ejemplo de ejecución de NVIDIA Visual Profiler.

Con el fin de determinar la cantidad de memoria utilizada por el proceso, luego de la ejecución y el análisis que realiza el nvvp y el chequeo de uso del GPU, se observa la sección de propiedades donde se muestran el número de Megabytes que fueron utilizados por el GPU en el proceso tal como se muestra en la Figura 45. El mismo procedimiento se realiza en todos los modelos y optimizaciones para la toma de mediciones de memoria.

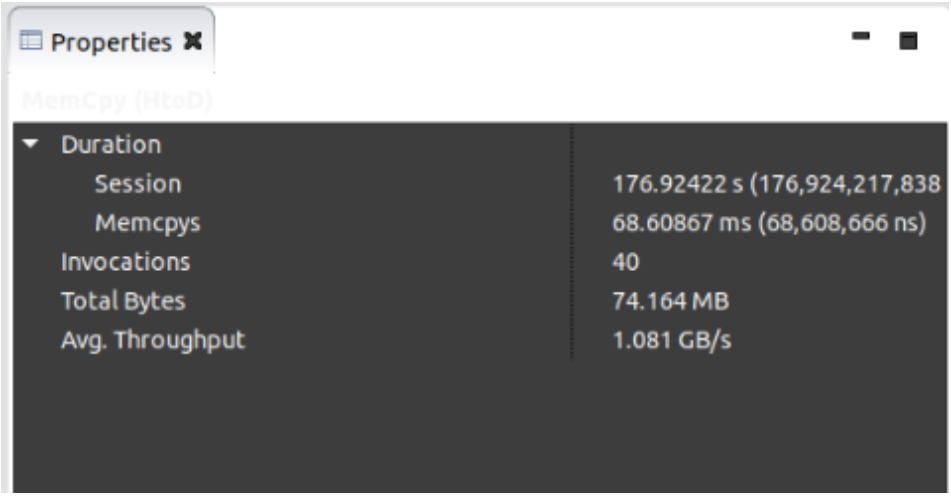

Figura 45. Memoria utilizada por el proceso, mostrada en nvvp.

Es importante también aclarar que la herramienta nvvp permite confirmar la ejecución del programa haciendo uso del GPU, ya que, de no ser así, al intentar hacer el chequeo de uso del GPU, claramente se obtiene un mensaje de error indicando que no hubo procesos que hicieran uso de este. Como puede observarse en la Figura 46, al hacer uso del GPU, se confirma mediante la línea que indica el número de dispositivo, así como su modelo y demás información de Profiling.

Figura 46. Ejemplo de proceso en ejecución con uso del GPU embebido.

Tamaño en disco:

Como última métrica, se compara el tamaño en disco de cada archivo, según las diferentes optimizaciones y los diferentes frameworks. En la sección
5.7 Comparación de resultados de las optimizaciones, se presenta una comparación de los tamaños para cada archivo obtenido.

Conociendo las métricas que se han establecido para la comparación, se procede a continuación con los detalles de cada uno de los frameworks y las optimizaciones, en las siguientes secciones se detalla la manera en la que se obtuvieron las diferentes métricas y el valor final obtenido para dicha optimización o framework.

5.3 Keras

Keras es un API diseñada para ser amigable con el programador, provee errores claros y concisos, así como una amplia documentación y extensas guías. Para este proyecto, este es el modelo base que se obtiene luego del entrenamiento, generando un archivo con la extensión .h5. Aunque Keras ha sido una parte integral de la API principal de TensorFlow, continúa desarrollándose como software independiente, ya que se diseñó como una interfaz para diversos frameworks.

Para obtener el modelo de Keras, simplemente se entrenó el modelo como se menciona en la sección 4.4 VGG16. Este entrenamiento resulta en la generación del archivo .h5, el cual es el modelo base para las demás conversiones que se detallan a continuación.

A continuación, se detallan en la Tabla 19 las métricas obtenidas al ejecutar este modelo con diferentes imágenes de prueba. Estos resultados son los valores considerados “base” para este proyecto, ya que son obtenidos directamente del modelo entrenado sin aplicar ninguna conversión ni optimización.

Tabla 19. Resultados de mediciones obtenidas para el modelo Keras.

Métrica	Keras
Tiempo de ejecución (ms)	350.79
IoU	78.20%
Memoria de ejecución (MB)	76.119
Tamaño (MB)	97.6

5.2.1 Pruning en modelo Keras

Como optimización adicional, se aplicó Pruning a cada uno de los modelos con los que se trabajó con el fin tanto de observar el impacto en el tamaño del archivo, como para verificar si hay algún impacto en la IoU y la ejecución del modelo al combinar el pruning con las cuantizaciones.

Modelo Base. Se aplicó pruning al modelo base y al modelo con cuantización que funcionan como base para todos los demás modelos, los cuales se basan en VGG16 con capas adicionales corriendo en Keras. Para esto, se utiliza el código descrito a continuación. Cabe mencionar que una vez generado el nuevo archivo .h5, se utiliza este nuevo archivo "Pruned" como base para generar los archivos correspondientes (.pb, ,TFLite) para las mediciones de pruning de las demás optimizaciones que se consideran en las secciones siguientes.

Se aplica pruning a todo el modelo configurando los parámetros iniciales de pruning para iniciar con un 20% de sparsity (20% de ceros en pesos) y terminar con un 80% y definiendo 2000 pasos. Una matriz en la que la mayoría de las entradas son 0 se denomina "sparse matrix" y puede almacenarse de manera más eficiente. Algunos cálculos también se pueden realizar de manera más eficiente en ellas siempre que la matriz sea lo suficientemente grande. Las redes neuronales pueden aprovechar la eficiencia obtenida del sparsity asumiendo que la mayoría de los pesos de conexión son iguales a 0.

Es importante mencionar que es necesario volver a entrenar el modelo para la aplicación del pruning. Se debe utilizar "tfmot.sparsity.keras.UpdatePruningStep" durante el proceso de entrenamiento, ya que este callback de Keras es el encargado de actualizar los wrappers de pruning con los pasos del optimizador. Además, se hace uso también de otro callback "tfmot.sparsity.keras.PruningSummaries" que ayuda con el proceso visual proporcionando registros para realizar un seguimiento del progreso y la depuración.

En la Tabla 20 se muestran los resultados de la aplicación de Pruning en el modelo de Keras. Se ejecuto el modelo para medir su IoU, el tiempo de ejecución y el tamaño de los archivos.

Tabla 20. Resultados de la aplicación de Pruning en el modelo Keras.

Métrica	Keras pruned
Tiempo de ejecución (ms)	171.83
IoU	79.81%
Memoria de ejecución (MB)	60.131
Tamaño (MB)	77.1

5.4 TensorRT

Es parte de la familia de NVIDIA basada en CUDA y proporciona optimizaciones INT8 y FP16 para implementaciones de aplicaciones de inferencia de deep learning.

Para poder utilizar TensorRT, es necesario primero poder convertir el modelo de Keras a Tensorflow. Básicamente se trata de generar un archivo del tipo protobuf (.pb). Con el fin de

obtener el archivo protobuf, no es necesario reentrenar el modelo existente del cual se obtuvo el archivo de Keras .h5, sino que más bien se realiza una conversión del modelo.
En primer lugar, se define la ruta al archivo original y a la ubicación en la que se desea guardar el archivo resultante. Posteriormente, se define una función "freeze_graph" para escribir el archivo. En esta función, se remueven los nodos que ya no son necesarios en el modelo, hay nodos como Identity y CheckNumerics que solo son útiles durante el entrenamiento y se pueden eliminar en gráficos que solo se usarán para inferencias.

Posteriormente, se reemplaza todas las variables con constantes de los mismos valores. Es conveniente convertirlas todas a operaciones continuas con los mismos valores ya que esto hace posible describir la red completamente con un solo archivo GraphDef y permite la eliminación de muchas operaciones relacionadas con la carga y el guardado de las variables, más detalles del código se encuentra en el Apéndice D.

5.4.1 Cuantización punto flotante 16 en TensorRT

Con el fin de observar el efecto de la cuantización de punto flotante de 16 bits (FP16), se aplicó al modelo base de Tensorflow esta optimización utilizando como plataforma TensorRT, que combina capas, optimiza la selección del kernel y también realiza la normalización y conversión a matemáticas matriciales optimizadas según la precisión especificada (en este caso FP16 o INT8) para mejorar la latencia, el rendimiento y la eficiencia.

La idea con esta cuantización es reducir el número de bits utilizados, lo cual evidentemente significa un posible sacrificio en la IoU del modelo. Por otra parte, FP16 solo ocupa 16 bits en la memoria en lugar de 32 bits, lo que indica menos espacio de almacenamiento, ancho de banda de memoria, consumo de energía, menor latencia de inferencia y mayor velocidad aritmética. La ventaja de la cuantización posterior al entrenamiento es que no necesita de afinamiento ni de volver a entrenar el modelo, lo cual representa un gran ahorro de tiempo.

Para realizar la conversión del modelo de FP32, que es el estándar utilizado por Tensorflow, y por ende, el utilizado en el modelo base, donde básicamente se cuantiza el modelo de punto flotante 32 bits a punto flotante 16 bits, lo cual ayuda con la reducción de la memoria de ejecución y además fusiona capas y tensores juntos, lo que optimiza aún más el uso de la memoria y el ancho de banda de la GPU, dando como resultado un graph de Tensorflow optimizado para correr en la Jetson Nano con TensorRT, el cual es guardado como un archivo .pb

Al aplicar un cuantización de FP16 en el modelo base de TensorRT, los resultados que se obtienen son realmente similares a los obtenidos con el modelo original, con una leve reducción del tiempo de ejecución y una IoU casi igual a la del modelo no cuantizado (apenas 0.02% menor).

En la Tabla 21 se presenta el resumen de las mediciones obtenidas al aplicar cuantización posterior al entrenamiento FP16 en el modelo base.

Tabla 21. Resultado de las mediciones del modelo TensorRT FP16.

Métrica	TensorRT FP16
Tiempo de ejecución (ms)	358.30
IoU	75.28%
Memoria de ejecución (MB)	76.572
Tamaño (MB)	71.8

Al aplicar pruning y generar el nuevo archivo de TensorRT FP16, se procedió con la toma de mediciones nuevamente, generando los resultados mostrados en la Tabla 22 a continuación.

Tabla 22. Resultado de las mediciones del modelo TensorRT FP16 pruned.

Métrica	TensorRT FP16 pruned
Tiempo de ejecución (ms)	164.67
IoU	77.30%
Memoria de ejecución (MB)	59.717
Tamaño (MB)	77.1

5.4.2 Cuantización Integer 8 (INT8) TensorRT

Como parte de las pruebas que se decidieron realizar en este proyecto, se incluyó la cuantización a Integer 8 bits (INT8). Para esto, nuevamente utilizando el modelo base de Tensorflow, se aplicó dicha cuantización con el fin de evaluar el desempeño del modelo.

Con esta cuantización se obtienen mejoras superiores de latencia, reducciones en el uso máximo de memoria y compatibilidad con aceleradores de hardware de solo enteros asegurándose de que todas las matemáticas del modelo se cuantifican en números enteros.

Al igual que en la cuantización FP16, se procedió a tomar las mediciones necesarias para evaluar las métricas en el modelo. En la Tabla 23 se presentan los resultados obtenidos de las mediciones para el modelo de TensorRT con cuantización INT8.

Tabla 23. Resultado de las mediciones del modelo TensorRT INT8.

Métrica	TensorRT INT8
Tiempo de ejecución (ms)	345.97
IoU	73.74%
Memoria de ejecución (MB)	74.164
Tamaño (MB)	71.8

Al igual que con la cuantización FP16, se aplicó pruning también al modelo cuantizado con INT8, lo cual da como resultado los valores mostrados en la Tabla 24.

Tabla 24. Resultado de las mediciones del modelo TensorRT INT8 pruned.

Métrica	TensorRT INT8 pruned
Tiempo de ejecución (ms)	144.86
IoU	72.36%
Memoria de ejecución (MB)	57.974
Tamaño (MB)	77.1

5.5 Tensorflow Lite

El conversor de TensorFlow Lite toma un modelo de TensorFlow y genera un modelo de TensorFlow Lite (un formato FlatBuffer optimizado que se puede identificar mediante la extensión de archivo .TFLite). Para obtener el modelo TFLite, se utiliza "*tf.lite.TFLiteConverter.from_keras_model()*" para convertir el modelo base (Keras .h5). La cuantización posterior al entrenamiento es una optimización que se usa con frecuencia en Tensorflow Lite y que puede reducir aún más la latencia y el tamaño del modelo con una pérdida mínima de la IoU. Para realizar la inferencia, se utiliza el intérprete de TensorFlow Lite, este está diseñado para ser ágil y rápido utilizando un orden de gráficos estáticos y un asignador de memoria personalizado (menos dinámico) para garantizar una carga mínima, inicialización y latencia de ejecución.

Para lograr la conversión del modelo de Keras a TFLite, se utilizó el código mostrado en el Anexo D. Debido a que se quiere analizar tanto el modelo base como las cuantizaciones, se inicia con la optimización por defecto de TFLite, la cual corresponde a float32.

Una vez ejecutado el código, se escribe el archivo, el cual tiene una extensión .TFLite. Este archivo es el que se utiliza para realizar las mediciones del modelo base de Tensorflow lite que se muestran en la Tabla 25.

Tabla 25. Resultado de las mediciones del modelo base (FP32) para TFLite.

Métrica	Tensorflow Lite FP32
Tiempo de ejecución (ms)	354.89
IoU	75.99%
Memoria de ejecución (MB)	76.032
Tamaño (MB)	71.8

Adicional a las mediciones del modelo base, se aplicó también pruning a este modelo, lo cual de como resultado los valores listados en la Tabla 26.

Tabla 26. Resultado de las mediciones del modelo base (FP32) para TFLite pruned.

Métrica	Tensorflow Lite pruned
Tiempo de ejecución (ms)	168.45
IoU	79.38%
Memoria de ejecución (MB)	59.198
Tamaño (MB)	77.1

5.5.1 Cuantización punto flotante 16 en Tensorflow Lite

De la misma manera que se hizo en TensorRT, también se realizaron cuantizaciones para TensorFlow Lite. En este caso, la conversión es realmente sencilla para pasar de la optimización por defecto de 32 bits punto flotante a la optimización de punto flotante de 16 bits.

El código solamente tiene una línea adicional con respecto a la conversión a TFLite FP32, en la cual se define la optimización a la cual se quiere convertir el modelo, para este caso, "*converter.target_spec.supported_types = [tf.float16]"*. Realizada esta cuantización, se obtienen los resultados que se muestran en la Tabla 27.

Tabla 27. Resultado de las mediciones del modelo TFLite FP16.

Métrica	Tensorflow Lite FP16
Tiempo de ejecución (ms)	275.49
IoU	74.51%
Memoria de ejecución (MB)	59.020
Tamaño (MB)	35.9

Similar a lo realizado anteriormente, se aplicó pruning al modelo de TFLite con FP16, lo cual brinda los resultados mostrados en la Tabla 28.

Tabla 28. Resultado de las mediciones del modelo TFLite FP16 pruned.

Métrica	Tensorflow Lite FP16 Pruned
Tiempo de ejecución (ms)	176.46
IoU	75.31%
Memoria de ejecución (MB)	43.557
Tamaño (MB)	38.6

5.5.2 Cuantización punto INT8 en Tensorflow Lite

El procedimiento para realizar la cuantización a 8 bits (INT8) es exactamente el mismo que el mostrado en la cuantización de FP16, con la única diferencia que la línea de código que define la optimización cambia a "*converter.target_spec.supported_types = [tf.int8]*". En la Tabla 29 se muestran las mediciones obtenidas con este modelo. La Tabla 30 muestra los resultados con el modelo optimizado con cuantización INT8 y pruning.

Tabla 29. Resultado de las mediciones del modelo TFLite INT8.

Métrica	Tensorflow Lite INT8
Tiempo de ejecución (ms)	186.23
IoU	71.93%
Memoria de ejecución (MB)	39.898
Tamaño (MB)	18.0

Tabla 30. Resultados de las mediciones del modelo TFLite INT8 pruned.

Métrica	Tensorflow Lite INT8 pruned
Tiempo de ejecución (ms)	102.96
IoU	72.93%
Memoria de ejecución (MB)	31.885
Tamaño (MB)	19.3

5.6 TVM

TVM es un compilador que proporciona portabilidad a través de diferentes hardware backends para deep learning. TVM automatiza la optimización de programas de bajo nivel para las características del hardware mediante modelado de costos basado en el aprendizaje para una exploración rápida de optimizaciones de código.

Para TVM, al igual como se menciona en las secciones anteriores, se realizaron diferentes optimizaciones. El modelo base para TVM fue el modelo de Tensorflow Lite, esto simplemente por la facilidad para la creación del código en comparación con los demás modelos.

Se utilizó además el código mostrado en el Anexo D, donde se muestra las definiciones de los parámetros del tensor, además de las líneas de código en las cuáles se carga el modelo de TFLite y además la compilación del modelo con Relay.

Para esto se hace uso de la importación fácil mediante "TFLite.Model.GetRootAsModel", esto importa las clases y funciones de un submódulo al módulo superior directamente evitando demasiadas importaciones por submódulo, permitiendo analizar los modelos TFLite en Python.

Posteriormente, "Relay.frontend.from_TFLite" convierte el modelo de TFLite a funciones compatibles con Relay. Esto provee el módulo de Relay para compilar y los parámetros para ser utilizados por Relay. Luego se establece el objetivo como "cuda" lo cual permite que se obtenga funciones ejecutables en CUDA.
Realizado lo anterior, se tiene "*with transform.PassContext(opt_level=3)*", esta es la base sobre la que se ejecuta una optimización de Relay. Cada contexto de paso contiene una cantidad de información auxiliar que se utiliza para ayudar a un pase de optimización. Dicha información incluye el reporte de errores para registrarlos durante la optimización.

Finalmente se tiene la función "*Relay.build*" que crea una función de Relay para ejecutarse en el graph executor de TVM y se guarda "la librería" para su despliegue en el dispositivo objetivo.

Al realizar las mediciones con TVM se obtuvieron los resultados mostrados en la Tabla 31 que se muestra a continuación.

Tabla 31. Resultado de las mediciones del modelo base FP32 ejecutado en TVM

Métrica	TVM FP32
Tiempo de ejecución (ms)	323.53
IoU	75.78%
Memoria de ejecución (MB)	69.311
Tamaño (MB)	71.8

De igual manera que en los frameworks anteriores, aplicando pruning al modelo, se obtienen los resultados compilados en la Tabla 32.

Tabla 32. Resultado de las mediciones del modelo FP32 pruned ejecutado en TVM.

Métrica	TVM FP32 pruned
Tiempo de ejecución (ms)	120.79
IoU	79.84%
Memoria de ejecución (MB)	62.478
Tamaño (MB)	77.1

5.6.1 Cuantización punto flotante 16 en TVM

Para la cuantización a 16 bits, se utiliza el modelo de Tensorflow lite FP16 y de la misma manera, se toman las mediciones tanto para el modelo FP16, como para el modelo FP16 con pruning aplicado. Los valores obtenidos en la ejecución de estos modelos se muestran respectivamente en la Tabla 33 y la Tabla 34.

Tabla 33. Resultado de las mediciones del modelo FP16 ejecutado en TVM.

Métrica	TVM FP16
Tiempo de ejecución (ms)	274.07
IoU	74.43%
Memoria de ejecución (MB)	58.716
Tamaño (MB)	35.9

Tabla 34. Resultado de las mediciones del modelo FP16 pruned ejecutado en TVM.

Métrica	TVM FP16 pruned
Tiempo de ejecución (ms)	120.69
IoU	77.85%
Memoria de ejecución (MB)	45.302
Tamaño (MB)	38.6

5.6.2 Cuantización INT8 en TVM

Finalmente, para continuar con las mediciones de la misma manera que en los frameworks anteriores, se procedió con la ejecución del modelo con cuantización INT8 y la aplicación de pruning a este modelo, lo cual brinda los resultados de la Tabla 35 y la Tabla 36 mostradas a continuación.

Tabla 35. Resultado de las mediciones del modelo INT8 ejecutado en TVM.

Métrica	TVM INT8
Tiempo de ejecución (ms)	244.33
IoU	73.95%
Memoria de ejecución (MB)	52.343
Tamaño (MB)	18

Tabla 36. Resultado de las mediciones del modelo INT8 pruned ejecutado en TVM.

Métrica	TVM INT8 pruned
Tiempo de ejecución (ms)	112.06
IoU	74.17%
Memoria de ejecución (MB)	37.949
Tamaño (MB)	19.3

5.7 Comparación de resultados de las optimizaciones

En esta sección se presenta la comparación de los resultados de las diferentes optimizaciones, con el fin de encontrar la más adecuada para este proyecto. Para iniciar, en la Tabla 37 se presenta un resumen de los resultados obtenidos en cada modelo ejecutado, tanto los modelos base (a) como los modelos a los que se les aplicó pruning (b).

Tabla 37. Resumen de los resultados obtenidos en los diferentes modelos.

Modelo base	t ejecución (ms)	Precisión	Tamaño (MB)	Memoria (MB)
Keras	350.79	78.20%	97.6	76.119
Tensorrt_16	358.30	75.28%	71.8	76.572
Tensorrt_8	345.97	73.74%	71.8	74.164
TFLite_32	354.89	75.99%	71.8	76.032
TFLite_16	275.49	74.51%	35.9	59.02
TFLite_8	186.23	71.93%	18.0	39.898
TVM	323.53	75.78%	71.8	69.311
TVM_16	274.07	74.43%	35.9	58.716
TVM_8	244.33	73.95%	18.0	52.343

a. Modelos base

Modelo Pruned	t ejecución (ms)	Precisión	Tamaño (MB)	Memoria (MB)
Keras	171.83	79.81%	77.1	60.131
Tensorrt_16	164.67	77.30%	77.1	59.717
Tensorrt_8	144.86	72.36%	77.1	57.974
TFLite_32	168.45	79.38%	77.1	59.198
TFLite_16	176.46	75.31%	38.6	43.557
TFLite_8	102.96	72.93%	19.3	31.885
TVM	120.79	79.84%	77.1	62.478
TVM_16	120.69	77.85%	38.6	45.302
TVM_8	112.06	74.17%	19.3	37.949

b. Modelos pruned

A continuación, en la Figura 47 se procede con la comparación gráfica de cada uno de los modelos pruned y sus diferentes metricas.

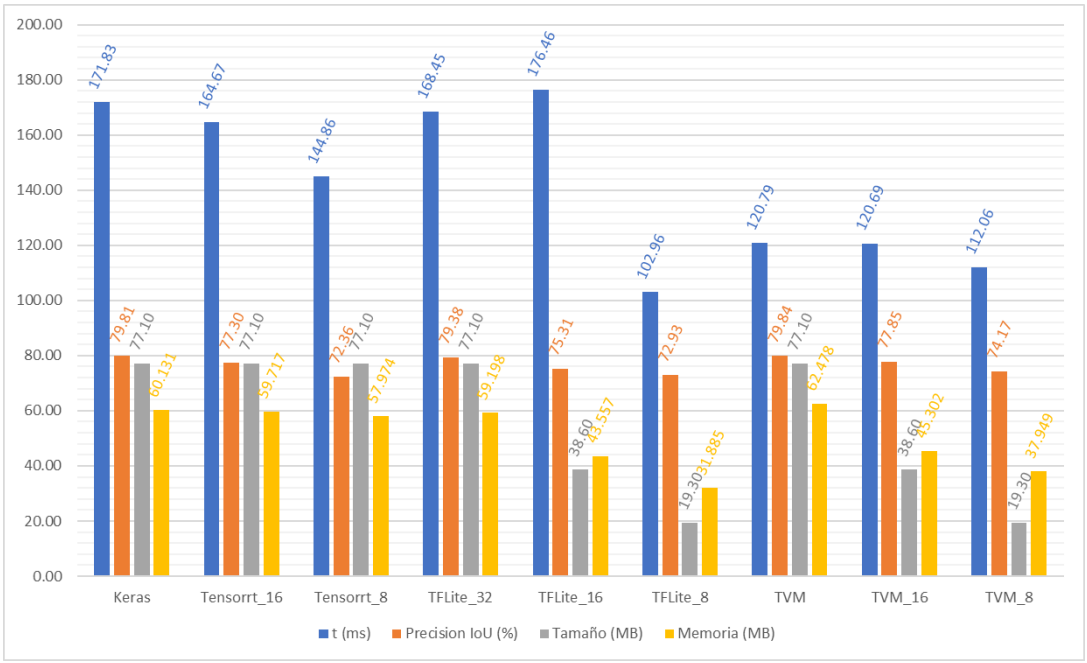

Figura 47. Métricas de los modelos pruned.

Siguiendo con el análisis de cada una de las métricas, se incorpora también en la Figura 48 los valores obtenidos con los modelos base con el fin de hacer notar la relación entre los modelos. Al comparar los tiempos de inferencia, se observa una reducción bastante pronunciada entre los modelos con pruning y los modelos a los que no se les aplicó esta optimización.

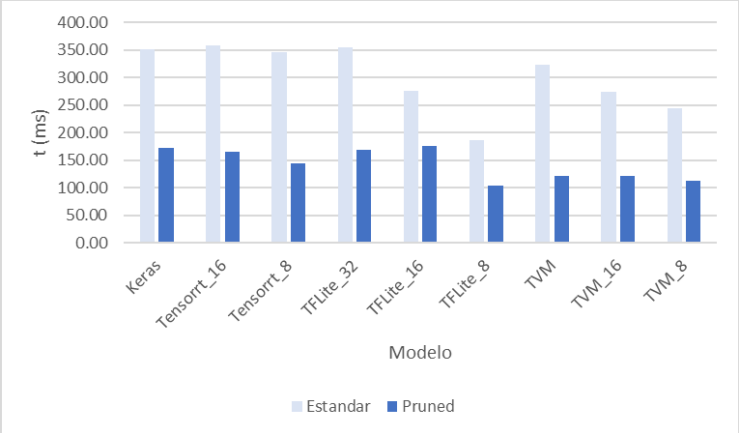

Figura 48. Comparación del tiempo de inferencia de los modelos base y pruned.

Es interesante notar que el modelo que tomó más tiempo al aplicar pruning, sigue siendo aún menor que el modelo más rápido de los modelos base. Nuevamente el modelo con el menor tiempo de inferencia es el TFLite INT8, el cual es un 40% menor al modelo de Keras pruned y un 71% menor que el modelo de Keras estándar. La reducción promedio en el tiempo de inferencia de los modelos pruned fue del 52% con respecto a los modelos a los que no se les aplicó pruning.

En términos de la precisión, se puede notar un leve aumento de la precisión para los modelos pruned. En la Figura 49 se observa que la mayor precisión se obtiene al aplicar pruning al

modelo de TVM, el cual alcanza apenas un 0.03% más que el modelo base de Keras con pruning. Con respecto a los modelos base, la mayor diferencia se da también en el modelo de TVM, donde la diferencia entre los modelos base y pruned es de un 4%. En promedio, la precisión de los modelos pruned fue un 2.4% menor a la de los modelos base.

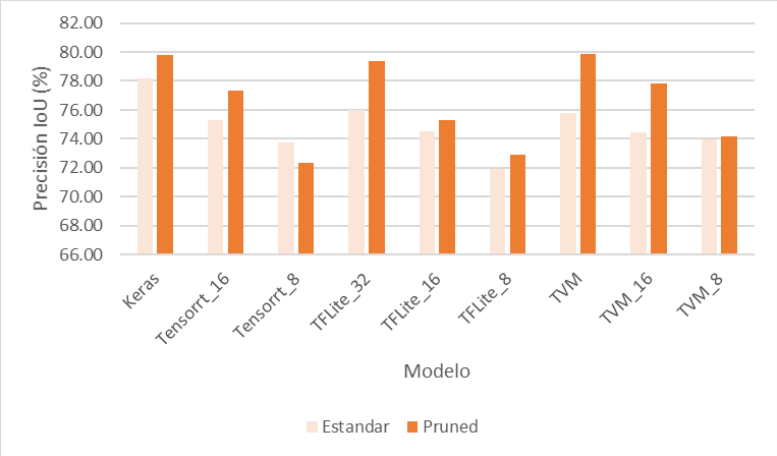

Figura 49. Comparación de la IoU de los modelos base y pruned.

Otro dato curioso con la comparación de precisión entre los modelos es que para el modelo de TensorRT con cuantización INT8, la precisión se ve afectada con respecto al modelo base. Al aplicar pruning a este modelo, se da una reducción en la precisión de casi un 2% con respecto al modelo base.

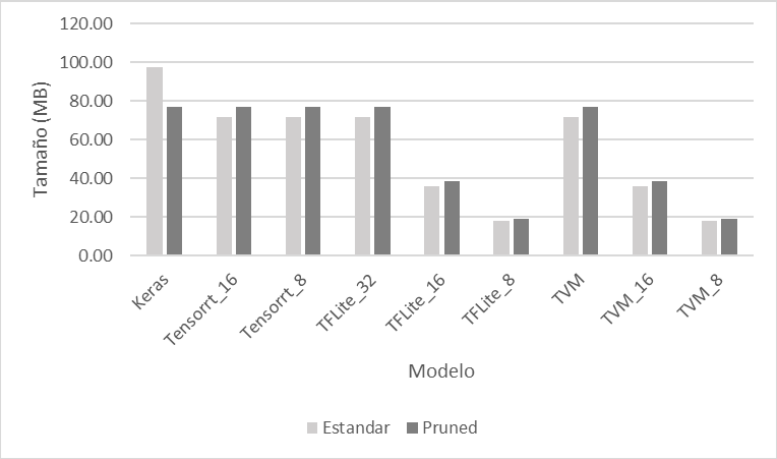

Figura 50. Comparación del tamaño del archivo de los modelos base y pruned.

Al comparar el tamaño del archivo, se puede observar claramente que al aplicar pruning, el cambio en el tamaño del archivo varía muy poco con respecto a los modelos en los que no se aplicó pruning. En la Figura 50 se puede observar que solamente en el caso de Keras se dio una reducción moderada del tamaño del archivo. En el resto de los modelos, no sobrepasa los 6MB de diferencia. Por otra parte, al igual que en los modelos sin pruning, el archivo de menor tamaño corresponde al modelo de TFLite INT8 que es igual que el obtenido con el modelo de TVM INT8. Cabe aclarar también nuevamente que TVM utiliza el modelo de TFLite como base, por lo que podemos concluir que no hay diferencia en el cambio del tamaño del modelo al convertir de TFLite a TVM.

Finalmente, al comparar la memoria, tal como se muestra en la Figura 51, hay una reducción del uso de la memoria al aplicar pruning, lo cual es el comportamiento esperado. La mayor reducción se presenta en el modelo de Tensorflow, el cual tiene una reducción de 17.29MB, lo que representa una reducción de casi un 22% del uso de memoria con respecto al modelo de Tensorflow sin pruning. En promedio, la reducción del uso de memoria en los modelos al aplicar pruning es de 21%.

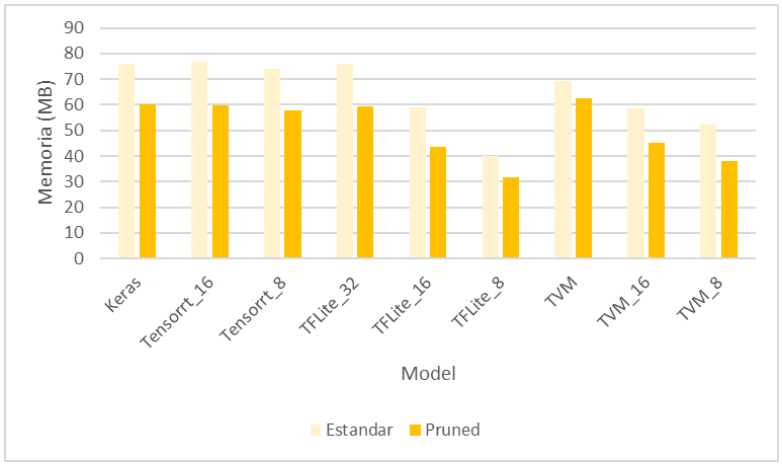

Figura 51. Comparación del uso de memoria de los modelos base y pruned.

Al realizar las comparaciones de las métricas según las optimizaciones aplicadas, se puede concluir que se debe definir las prioridades según la tarea que se desea ejecutar y el hardware que se tiene disponible. Sabiendo esto, entonces se puede dar prioridad a la precisión, a la memoria de ejecución, a la necesidad de espacio de almacenamiento para el modelo o al tiempo necesario para su ejecución.

Queda demostrado con las pruebas realizadas que el impacto en la precisión es bastante bajo, al menos para las necesidades de este proyecto, con respecto a las ganancias en espacio, memoria y tiempo de ejecución.

En general, se puede destacar lo siguiente de las pruebas realizadas:

- El mejor **tiempo de inferencia** se logra con el modelo **TFLite INT8 pruned.**

  Es posible observar que con el tiempo de inferencia logrado con TFLIite INT8 pruned, se obtiene un frame rate de 9.71 FPS, lo cual, para el tipo de aplicación especifico de este proyecto, es un valor razonable y funcional, especialmente al compararlo con el modelo base de Keras, el cual tiene un frame rate de 2.86 FPS.

- La mejor precisión en términos de **IoU** se obtiene con el modelo **TVM FP32 pruned**.

  Con respecto a la precisión IoU obtenida con TVM FP32 pruned que es igual a 79.84%, se obtiene una mejora de 1.64% con respecto al modelo base de Keras que obtuvo un IoU de 78.20%.

- El menor **tamaño de archivo** se consigue con el modelo **TFLite INT8.**

  El tamaño del archivo se logra reducir de 97.6MB del modelo base de Keras a apenas 19.3MB con TFLite INT8 pruned, lo cual representa una reducción de 78.3MB en el tamaño del archivo.

- La menor **memoria de ejecución** alcanzada fue con el modelo **TFLite INT8 pruned.**

  De manera similar, la memoria de ejecución se logra disminuir en 44.34MB con el modelo de TFLite INT8 pruned, el cual logra una memoria de ejecución de apenas 31.85MB, mientras que el modelo base de Keras tiene una memoria de ejecución de 76.19MB.

Por lo anterior, sumado a la facilidad para realizar la conversión del modelo, su ejecución y sus optimizaciones, se define el modelo de TFLite INT8 pruned como el modelo sugerido para ser seleccionado para las necesidades de este proyecto, logrando superar las expectativas de la tarea al identificar satisfactoriamente una placa oficial de Costa Rica para vehículos particulares, siendo ejecutado desde un GPU embebido.

Capítulo 6. Conclusiones

A través del desarrollo de la primera parte de este proyecto, se han identificado diversos aspectos interesantes de la creación, entrenamiento y desarrollo de una red neuronal que solamente con la práctica se logran comprender.

Se concluye que uno de los aspectos más importantes es el hecho de que la parte que demanda mayor trabajo y tiempo en el desarrollo de una red neuronal está en el set de datos. Si se tiene un set de datos de Internet, cosa que no es imposible, ya que hay diversos sitios que hacen disponibles muchos sets de datos, no siempre se puede obtener un set de datos adecuado a la necesidad específica que se presente. Es por esta razón que muchas veces será necesario crear el set de datos personalizado. No solo es laborioso, en este caso específico, obtener las imágenes suficientes para un set de datos de un tamaño "decente", sino también lograr anotar las imágenes con la información necesaria para ser utilizada por el modelo a entrenar.

Otra conclusión que se puede obtener a partir de las observaciones experimentales es que no necesariamente el modelo que tenga un aumento más grande entre los epochs será el que obtenga el mejor resultado, por ejemplo, en el desarrollo del proyecto se observó como Mobilenet V2 e Inception V3 tuvieron un cambio de casi 20% en su precisión desde el primer epoch hasta el final mientras que VGG16 aumento poco más de 5%. Aun así, este último modelo fue el que tuvo mejor precisión en validación. Al obtener una precisión en validación de más del 95% y ser muy superior a los otros modelos que obtuvieron precisiones menores al 80%, se determina que el mejor modelo para utilizar es el que se basa en VGG16.

Es posible también concluir según lo observado durante el desarrollo del proyecto que, aunque haya arquitecturas más recientes, más complejas o modernas, no siempre son las más adecuadas para las necesidades específicas del proyecto. En este caso, aunque VGG16 es del año 2003, Inception V3 de 2009 y Mobilenet V2 de 2018, para esta aplicación específica de identificación de un solo objeto en las imágenes, el mejor desempeño en cuanto a precisión se obtuvo con VGG16.

Se concluye además que el tiempo de entrenamiento en diferentes arquitecturas es significativamente diferente, aun cuando se utilice el mismo set de datos y el mismo hardware en el entrenamiento de estos. Experimentalmente pudo comprobarse este aspecto ya que el tiempo tomado para entrenar el modelo basado en VGG16 fue casi 15 veces mayor al tiempo necesario para entrenar el modelo basado en Mobilenet V2.

Cabe mencionar entre las conclusiones además el hecho de que, al utilizar modelos pre-entrenados, es importante ajustar la entrada que se le ingresa al modelo para que sea la adecuada al mismo. En el caso de este proyecto, la entrada de los tres modelos es la misma, imágenes de 224 (W) x 224 (H) x 3 (RGB) de forma que no haya errores ni inconvenientes cuando se ejecute o se intente entrenar el modelo.

Al identificar el modelo VGG16 como el más adecuado para utilizar en las optimizaciones en la segunda etapa, se implementa satisfactoriamente además un prototipo del sistema en el cual se logra no solamente identificar una placa de un vehículo de Costa Rica, sino que también se logra extraer el último número de esta y así identificar en consola el día en que este vehículo tiene restricción vehicular, cumpliendo así con la función con la que fue pensada la primera etapa.

Con respecto a la segunda etapa, se determina que, para este proyecto, lo mejor es convertir el modelo a TFLite y aplicar cuantización INT8, sin embargo, para alguien más, es posible que la idea de su red neuronal y las tareas a cumplir dependan más, por ejemplo, de la precisión que del tiempo de ejecución, por lo cual para esa necesidad específica, sería más útil una optimización diferente.

En la segunda etapa del proyecto, se evaluó la aplicación de pruning y cuantización en el modelo base con diferentes frameworks. Se hicieron mediciones del tiempo de ejecución, la IoU, el tamaño del archivo y la memoria de ejecución con cada optimización y cada framework.

Al evaluar los resultados obtenidos con cada framework y cada optimización, se sugiere la utilización del modelo de pruning con cuantización INT8 utilizando TFLite. Al comparar el modelo sugerido contra el modelo base de Keras, se observa que esta combinación de optimizaciones y framework permite una reducción en el tiempo de ejecución que permite pasar de 3 FPS a 10 FPS, una reducción de 44MB en el tamaño del archivo en disco, una reducción en la memoria de ejecución de 79.6MB y con una reducción de la precisión en términos de IoU de apenas 5.27%. Además, se considera que el frame rate obtenido es adecuado para cumplir exitosamente las expectativas del proyecto.

Como trabajo futuro, sería de interés aplicar autotuning a los diferentes modelos que se han desarrollado en este proyecto de forma que se pueda obtener otro set de resultados para ser comparados con los obtenidos. Esto permitiría observar el efecto al combinar un tercer método de optimización y evaluar si esto resulta en beneficio o perjuicio para el propósito de identificación de placas.

Bibliografía

[1] OpenCV team. [Visitado el 12 de Julio de 2020]. URL https://opencv.org/about/. 2020.

[2] Hubel, D. H.; Wiesel, T. N. Receptive fields of single neurones in the cat's striate cortex. The Journal of physiology 148 (3). arXiv:574–591.1959.

[3] Simonyan, Karen & Zisserman, Andrew. Very Deep Convolutional Networks for Large-Scale Image Recognition. arXiv: 1409.1556. 2014.

[4] AMLResearchProject. [Visitado el 14 de Julio de 2020]. URL https://github.com/AMLResearchProject/AML-ALL-Classifiers/blob/master/Python/_Movidius/NCS/Classes/inception_v3.py. 2020.

[5] Christian Szegedy, Vincent Vanhoucke, Sergey Ioffe, Jonathon Shlens, Zbigniew Wojna. Rethinking the Inception Architecture for Computer Vision. arXiv:1512.00567v3. 2015.

[6] Sandler M, Howard A, Zhu M, Zhmoginov A, Chen LC. MobileNetV2: Inverted Residuals and Linear Bottlenecks. arXiv:1801.04381, 2018.

[7] Make sense. [Visitado el 14 de Julio de 2020]. URL https://www.makesense.ai/. 2020.

[8] Hamid Rezatofighi, Nathan Tsoi, JunYoung Gwak, Amir Sadeghian, Ian Reid, Silvio Savarese. Generalized Intersection over Union: A Metric and A Loss for Bounding Box Regression. arXiv:1902.09630v2, 2019

[9] TensorFlow [Visitado el 6 de Julio de 2021].URL. https://www.tensorflow.org/model_optimization/guide/pruning

[10] TensorFlow [Visitado el 8 de Julio de 2021].URL. https://www.tensorflow.org/api_docs/python/tf/keras/optimizers/schedules/PolynomialDecay

[11] Nvidia.com [Visitado el 28 de Julio de 2021].URL. https://www.nvidia.com/es-la/autonomous-machines/embedded-systems/jetson-nano/

[12] TVM Relay [Visitado el 6 de Agosto de 2021].URL. https://tvm.apache.org/docs/api/python/relay/index.html

[13] Yasoubi, Ali & Hojabr, Reza & Modarressi, Mehdi. (2016). Power-Efficient Accelerator Design for Neural Networks Using Computation Reuse. IEEE Computer Architecture Letters. 16. 1-1. 10.1109/LCA.2016.2521654.

[14] Ferguson, Max & ak, Ronay & Lee, Yung-Tsun & Law, Kincho. (2017). Automatic localization of casting defects with convolutional neural networks. 1726-1735. 10.1109/BigData.2017.8258115.

Apéndice A – Código y Salida de consola de Inception V3

```
import pandas as pd
import urllib
import matplotlib.pyplot as plt
import numpy as np
import cv2
import glob
import os
import sys
import time
from PIL import Image
import tensorflow as tf
from keras.applications.inception_v3 import InceptionV3
from keras.layers import Flatten, Dense, Conv2D, MaxPooling2D, Input, Dropout
from keras.models import Model, Sequential
from keras.preprocessing.image import ImageDataGenerator
from keras.optimizers import Adam
#Fin de los imports

for dirname, _, filenames in os.walk('fotos_placas'):
        for filename in filenames:
                print(os.path.join(dirname, filename))

df = pd.read_json("placas.json", lines=True) #Lee el archivo .json
df.head()

os.mkdir("cr_plate_number")
dataset = dict()
dataset["image_name"] = list()
dataset["image_width"] = list()
dataset["image_height"] = list()
dataset["top_x"] = list()
dataset["top_y"] = list()
dataset["bottom_x"] = list()
dataset["bottom_y"] = list()
counter = 0
for index, row in df.iterrows():
        img = urllib.request.urlopen(row["content"])
        img = Image.open(img)
        img = img.convert('RGB') #Para asegurar el formato estándar en las imágenes
        img.save("cr_plate_number/placa_cr{}.jpeg".format(counter), "JPEG")
        dataset["image_name"].append("placa_cr{}".format(counter)) #Guarda las imágenes con un nombre
estándar genérico
        data = row["annotation"]
        dataset["image_width"].append(data[0]["imageWidth"])
        dataset["image_height"].append(data[0]["imageHeight"])
        dataset["top_x"].append(data[0]["points"][0]["x"])
        dataset["top_y"].append(data[0]["points"][0]["y"])
        dataset["bottom_x"].append(data[0]["points"][1]["x"])
        dataset["bottom_y"].append(data[0]["points"][1]["y"])
        counter += 1

df = pd.DataFrame(dataset)
df.head()
df.to_csv("test_cr_plates.csv", index=False) #Genera el archivo .csv

sys.stdout = open('output_InceptionV3_17_EPOCH.txt', 'w')
```

```
df = pd.read_csv("test_cr_plates.csv") #Lee el archivo CSV
df["image_name"] = df["image_name"] + ".jpeg"
df.drop(["image_width", "image_height"], axis=1, inplace=True)
df.head()

lucky_test_samples = np.random.randint(0, len(df), 5)
reduced_df = df.drop(lucky_test_samples, axis=0)
WIDTH = 224
HEIGHT = 224
CHANNEL = 3

def show_img(index):
        image = cv2.imread("cr_plate_number/" + df["image_name"].iloc[index])
        image = cv2.cvtColor(image, cv2.COLOR_BGR2RGB)
        image = cv2.resize(image, dsize=(WIDTH, HEIGHT))
        tx = int(df["top_x"].iloc[index] * WIDTH)
        ty = int(df["top_y"].iloc[index] * HEIGHT)
        bx = int(df["bottom_x"].iloc[index] * WIDTH)
        by = int(df["bottom_y"].iloc[index] * HEIGHT)
        image = cv2.rectangle(image, (tx, ty), (bx, by), (0, 0, 255), 1)
        plt.imshow(image)
        plt.show()

show_img(2020) # se escoge un número que yo definí aleatoriamente, puede reemplazarse por cualquier número
igual o menor al número total de imágenes, esto solamente como prueba.

from tensorflow.keras.preprocessing.image import ImageDataGenerator
datagen = ImageDataGenerator(rescale=1./255, validation_split=0.1)
train_generator = datagen.flow_from_dataframe(
        reduced_df,
        directory="cr_plate_number/",
        x_col="image_name",
        y_col=["top_x", "top_y", "bottom_x", "bottom_y"],
        target_size=(WIDTH, HEIGHT),
        batch_size=32,
        class_mode="other",
        subset="training")

validation_generator = datagen.flow_from_dataframe(
        reduced_df,
        directory="cr_plate_number/",
        x_col="image_name",
        y_col=["top_x", "top_y", "bottom_x", "bottom_y"],
        target_size=(WIDTH, HEIGHT),
        batch_size=32,
        class_mode="other",
        subset="validation")

#Definición del modelo
model = Sequential()
model.add(InceptionV3(weights="imagenet", include_top=False, input_shape=(HEIGHT, WIDTH,
CHANNEL)))
#Capas adicionales
model.add(Flatten())
model.add(Dense(128, activation="relu"))
model.add(Dense(64, activation="relu"))
model.add(Dense(64, activation="relu"))
model.add(Dense(4, activation="sigmoid"))

model.layers[-6].trainable = False
```

```
model.summary()

STEP_SIZE_TRAIN = int(np.ceil(train_generator.n / train_generator.batch_size))
STEP_SIZE_VAL = int(np.ceil(validation_generator.n / validation_generator.batch_size))

print("Train step size:", STEP_SIZE_TRAIN)
print("Validation step size:", STEP_SIZE_VAL)
train_generator.reset()
validation_generator.reset()

#############TRAINING############################

adam = Adam(lr=0.0005)
model.compile(optimizer=adam, loss="mse", metrics=["accuracy"])

history = model.fit_generator(train_generator,
   steps_per_epoch=STEP_SIZE_TRAIN,
   validation_data=validation_generator,
   validation_steps=STEP_SIZE_VAL,
   epochs=17)

plt.plot(history.history['loss'])
plt.plot(history.history['val_loss'])
plt.title('Model loss')
plt.ylabel('Loss')
plt.xlabel('Epoch')
plt.legend(['Train', 'Validation'], loc='upper left')
plt.show()

####################Guardando el .h5#######################

model.evaluate_generator(validation_generator, steps=STEP_SIZE_VAL)

score = model.evaluate_generator(validation_generator, steps=STEP_SIZE_VAL)
print(" - %s: %.2f%% \n - %s: %.2f%%" % (model.metrics_names[0], score[0]*100, model.metrics_names[1],
score[1]*100))
model.save("InceptionV3_cr.h5")
print("Model succesfully saved as h5 file")

####################Guardando el .PB#######################

from keras import backend as K
from keras.models import load_model

def freeze_session(session, keep_var_names=None, output_names=None, clear_devices=True):

        from tensorflow.python.framework.graph_util import convert_variables_to_constants
        graph = session.graph
        with graph.as_default():
                freeze_var_names = list(set(v.op.name for v in
tf.global_variables()).difference(keep_var_names or []))
                output_names = output_names or []
                output_names += [v.op.name for v in tf.global_variables()]
    # Graph -> GraphDef ProtoBuf
                input_graph_def = graph.as_graph_def()
                if clear_devices:
                        for node in input_graph_def.node:
                                node.device = ""
```

```
                frozen_graph = convert_variables_to_constants(session, input_graph_def, output_names,
freeze_var_names)
                return frozen_graph

frozen_graph = freeze_session(K.get_session(), output_names=[out.op.name for out in model.outputs])

tf.train.write_graph(frozen_graph, "./", "InceptionV3_cr.pb", as_text=False)
print("successfully saved model as InceptionV3_cr.pb")

####################Prueba#######################

img = cv2.resize(cv2.imread("placa2.jpg") / 255.0, dsize=(WIDTH, HEIGHT))
y_hat = model.predict(img.reshape(1, WIDTH, HEIGHT, 3)).reshape(-1) * WIDTH
xt, yt = y_hat[0], y_hat[1]
xb, yb = y_hat[2], y_hat[3]
img = cv2.cvtColor(img.astype(np.float32), cv2.COLOR_BGR2RGB)
image = cv2.rectangle(img, (xt, yt), (xb, yb), (0, 0, 255), 1)
plt.imshow(image)
plt.show()
####################file close#######################
sys.stdout.close()
```

Apéndice B – Código y Salida de consola de Mobilenet V2

```
import pandas as pd
import urllib
import matplotlib.pyplot as plt
import numpy as np
import cv2
import glob
import os
import sys
import time
from PIL import Image
import tensorflow as tf
from keras.applications.mobilenet_v2 import MobileNetV2
from keras.layers import Flatten, Dense, Conv2D, MaxPooling2D, Input, Dropout, GlobalAveragePooling2D,
multiply, LocallyConnected2D, Lambda
from keras.models import Model, Sequential
from keras.preprocessing.image import ImageDataGenerator
from keras.optimizers import Adam
#fin de los imports

sys.stdout = open('output_MobilenetV2_17_EPOCH.txt', 'w') # Archivo para guardar la salida de consola

########################################
#Como ya se había generado el .csv en Inception V3, simplemente se lee el csv directamente.
########################################

df = pd.read_csv("test_cr_plates.csv")
df["image_name"] = df["image_name"] + ".jpeg"
df.drop(["image_width", "image_height"], axis=1, inplace=True)
df.head()

lucky_test_samples = np.random.randint(0, len(df), 5)
reduced_df = df.drop(lucky_test_samples, axis=0)

WIDTH = 224
HEIGHT = 224
CHANNEL = 3

def show_img(index):
        image = cv2.imread("cr_plate_number/" + df["image_name"].iloc[index])
        image = cv2.cvtColor(image, cv2.COLOR_BGR2RGB)
        image = cv2.resize(image, dsize=(WIDTH, HEIGHT))
        tx = int(df["top_x"].iloc[index] * WIDTH)
        ty = int(df["top_y"].iloc[index] * HEIGHT)
        bx = int(df["bottom_x"].iloc[index] * WIDTH)
        by = int(df["bottom_y"].iloc[index] * HEIGHT)
        image = cv2.rectangle(image, (tx, ty), (bx, by), (0, 0, 255), 1)
        plt.imshow(image)
        plt.show()

show_img(936) # Número que yo definí, puede ser cualquiera igual o menor al total de imágenes

from tensorflow.keras.preprocessing.image import ImageDataGenerator
datagen = ImageDataGenerator(rescale=1./255, validation_split=0.1) # 10% de las imágenes van a ser utilizadas
para validación
train_generator = datagen.flow_from_dataframe(
        reduced_df,
        directory="cr_plate_number/",
```

```
        x_col="image_name",
        y_col=["top_x", "top_y", "bottom_x", "bottom_y"],
        target_size=(WIDTH, HEIGHT),
        batch_size=32,
        class_mode="other",
        subset="training")

validation_generator = datagen.flow_from_dataframe(
        reduced_df,
        directory="cr_plate_number/",
        x_col="image_name",
        y_col=["top_x", "top_y", "bottom_x", "bottom_y"],
        target_size=(WIDTH, HEIGHT),
        batch_size=32,
        class_mode="other",
        subset="validation")

#Definicion del modelo
model = Sequential()
model.add(MobileNetV2(weights="imagenet", include_top=False, input_shape=(HEIGHT, WIDTH,
CHANNEL)))

#Capas adicionales
model.add(Flatten())
model.add(Dense(128, activation="relu"))
model.add(Dense(64, activation="relu"))
model.add(Dense(64, activation="relu"))
model.add(Dense(4, activation="sigmoid"))

model.layers[-6].trainable = False
model.summary()

STEP_SIZE_TRAIN = int(np.ceil(train_generator.n / train_generator.batch_size))
STEP_SIZE_VAL = int(np.ceil(validation_generator.n / validation_generator.batch_size))

print("Train step size:", STEP_SIZE_TRAIN)
print("Validation step size:", STEP_SIZE_VAL)
train_generator.reset()
validation_generator.reset()

###################TRAINING#####################

adam = Adam(lr=0.0005)
model.compile(optimizer=adam, loss="mse", metrics=["accuracy"])
history = model.fit_generator(train_generator, steps_per_epoch=STEP_SIZE_TRAIN,
validation_data=validation_generator, validation_steps=STEP_SIZE_VAL, epochs=17)
#generación del gráfico
plt.plot(history.history['loss'])
plt.plot(history.history['val_loss'])
plt.title('Model loss')
plt.ylabel('Loss')
plt.xlabel('Epoch')
plt.legend(['Train', 'Validation'], loc='upper right')
plt.show()
###################Guardando el .H5#####################
model.evaluate_generator(validation_generator, steps=STEP_SIZE_VAL)
score = model.evaluate_generator(validation_generator, steps=STEP_SIZE_VAL)
print(" - %s: %.2f%% \n - %s: %.2f%%" % (model.metrics_names[0], score[0]*100, model.metrics_names[1],
score[1]*100))
model.save("MovilenetV2_cr.h5")
```

```
print("Model succesfully saved as h5 file")
####################Prueba#######################
img = cv2.resize(cv2.imread("placa2.jpg") / 255.0, dsize=(WIDTH, HEIGHT))
y_hat = model.predict(img.reshape(1, WIDTH, HEIGHT, 3)).reshape(-1) * WIDTH
xt, yt = y_hat[0], y_hat[1]
xb, yb = y_hat[2], y_hat[3]
img = cv2.cvtColor(img.astype(np.float32), cv2.COLOR_BGR2RGB)
image = cv2.rectangle(img, (xt, yt), (xb, yb), (0, 0, 255), 1)
plt.imshow(image)
plt.show()
####################Guardando el .PB#######################
from keras import backend as K
from keras.models import load_model
def freeze_session(session, keep_var_names=None, output_names=None, clear_devices=True):
        from tensorflow.python.framework.graph_util import convert_variables_to_constants
        graph = session.graph
        with graph.as_default():
                freeze_var_names = list(set(v.op.name for v in
tf.global_variables()).difference(keep_var_names or []))
                output_names = output_names or []
                output_names += [v.op.name for v in tf.global_variables()]
    # Graph -> GraphDef ProtoBuf
                input_graph_def = graph.as_graph_def()
                if clear_devices:
                        for node in input_graph_def.node:
                                node.device = ""
                frozen_graph = convert_variables_to_constants(session, input_graph_def, output_names,
freeze_var_names)
                return frozen_graph

frozen_graph = freeze_session(K.get_session(), output_names=[out.op.name for out in model.outputs])
tf.train.write_graph(frozen_graph, "./", "MovilenetV2_cr.pb", as_text=False)
print("successfully saved model as MovilenetV2_cr.pb")

sys.stdout.close() # cierre del archivo donde se guarda la salida de consola.
```

Apéndice C – Código y Salida de consola de VGG16

```
import pandas as pd
import urllib
import matplotlib.pyplot as plt
import numpy as np
import cv2
import glob
import os
import sys
import time
from PIL import Image
import tensorflow as tf
from keras.applications.vgg16 import VGG16
from keras.layers import Flatten, Dense, Conv2D, MaxPooling2D, Input, Dropout
from keras.models import Model, Sequential
from keras.preprocessing.image import ImageDataGenerator
from keras.optimizers import Adam
#Fin de los imports

sys.stdout = open('output_VGG16_17_EPOCH.txt', 'w') # Archivo para guardar la salida de consola

########################################
#Como ya se habia generado el .csv en Inception V3, simplemente se lee el csv directamente.
########################################

df = pd.read_csv("test_cr_plates.csv")
df["image_name"] = df["image_name"] + ".jpeg"
df.drop(["image_width", "image_height"], axis=1, inplace=True)
df.head()

lucky_test_samples = np.random.randint(0, len(df), 5)
reduced_df = df.drop(lucky_test_samples, axis=0)
WIDTH = 224
HEIGHT = 224
CHANNEL = 3

def show_img(index):
        image = cv2.imread("cr_plate_number/" + df["image_name"].iloc[index])
        image = cv2.cvtColor(image, cv2.COLOR_BGR2RGB)
        image = cv2.resize(image, dsize=(WIDTH, HEIGHT))
        tx = int(df["top_x"].iloc[index] * WIDTH)
        ty = int(df["top_y"].iloc[index] * HEIGHT)
        bx = int(df["bottom_x"].iloc[index] * WIDTH)
        by = int(df["bottom_y"].iloc[index] * HEIGHT)
        image = cv2.rectangle(image, (tx, ty), (bx, by), (0, 0, 255), 1)
        plt.imshow(image)
        plt.show()

show_img(1914) # se escoge un número que yo definí aleatoriamente, puede reemplazarse por cualquier número igual o menor al número total de imágenes

from tensorflow.keras.preprocessing.image import ImageDataGenerator
datagen = ImageDataGenerator(rescale=1./255, validation_split=0.1) # 10% de las imágenes van a ser utilizadas para validación
train_generator = datagen.flow_from_dataframe(
        reduced_df,
        directory="cr_plate_number/",
        x_col="image_name",
```

```
        y_col=["top_x", "top_y", "bottom_x", "bottom_y"],
        target_size=(WIDTH, HEIGHT),
        batch_size=32,
        class_mode="other",
        subset="training")

validation_generator = datagen.flow_from_dataframe(
        reduced_df,
        directory="cr_plate_number/",
        x_col="image_name",
        y_col=["top_x", "top_y", "bottom_x", "bottom_y"],
        target_size=(WIDTH, HEIGHT),
        batch_size=32,
        class_mode="other",
        subset="validation")

model = Sequential()
model.add(VGG16(weights="imagenet", include_top=False, input_shape=(HEIGHT, WIDTH, CHANNEL)))

model.add(Flatten())
model.add(Dense(128, activation="relu"))
model.add(Dense(64, activation="relu"))
model.add(Dense(64, activation="relu"))
model.add(Dense(4, activation="sigmoid"))

model.layers[-6].trainable = False
model.summary()

STEP_SIZE_TRAIN = int(np.ceil(train_generator.n / train_generator.batch_size))
STEP_SIZE_VAL = int(np.ceil(validation_generator.n / validation_generator.batch_size))

print("Train step size:", STEP_SIZE_TRAIN)
print("Validation step size:", STEP_SIZE_VAL)
train_generator.reset()
validation_generator.reset()

####################TRAINING#######################

adam = Adam(lr=0.0005)
model.compile(optimizer=adam, loss="mse", metrics=["accuracy"])

history = model.fit_generator(train_generator, steps_per_epoch=STEP_SIZE_TRAIN,
validation_data=validation_generator, validation_steps=STEP_SIZE_VAL, epochs=17)

plt.plot(history.history['loss'])
plt.plot(history.history['val_loss'])
plt.title('Model loss')
plt.ylabel('Loss')
plt.xlabel('Epoch')
plt.legend(['Train', 'Validation'], loc='upper left')
plt.show()

####################Guardando el .H5#######################

model.evaluate_generator(validation_generator, steps=STEP_SIZE_VAL)

score = model.evaluate_generator(validation_generator, steps=STEP_SIZE_VAL)
print(" - %s: %.2f%% \n - %s: %.2f%%" % (model.metrics_names[0], score[0]*100, model.metrics_names[1],
score[1]*100))
model.save("VGG16_cr.h5")
```

```
print("Model succesfully saved as h5 file")

####################Guardando el .PB#######################
from keras import backend as K
from keras.models import load_model

def freeze_session(session, keep_var_names=None, output_names=None, clear_devices=True):

        from tensorflow.python.framework.graph_util import convert_variables_to_constants
        graph = session.graph
        with graph.as_default():
                freeze_var_names = list(set(v.op.name for v in
tf.global_variables()).difference(keep_var_names or []))
                output_names = output_names or []
                output_names += [v.op.name for v in tf.global_variables()]
    # Graph -> GraphDef ProtoBuf
                input_graph_def = graph.as_graph_def()
                if clear_devices:
                        for node in input_graph_def.node:
                                node.device = ""
                frozen_graph = convert_variables_to_constants(session, input_graph_def, output_names,
freeze_var_names)
                return frozen_graph

frozen_graph = freeze_session(K.get_session(), output_names=[out.op.name for out in model.outputs])

tf.train.write_graph(frozen_graph, "./", "VGG16_cr.pb", as_text=False)
print("successfully saved model as VGG16_cr.pb")
####################Prueba#######################
img = cv2.resize(cv2.imread("placa4.jpg") / 255.0, dsize=(WIDTH, HEIGHT))
y_hat = model.predict(img.reshape(1, WIDTH, HEIGHT, 3)).reshape(-1) * WIDTH
xt, yt = y_hat[0], y_hat[1]
xb, yb = y_hat[2], y_hat[3]
img = cv2.cvtColor(img.astype(np.float32), cv2.COLOR_BGR2RGB)
image = cv2.rectangle(img, (xt, yt), (xb, yb), (0, 0, 255), 1)
plt.imshow(image)
plt.show()
####################Close file#######################
sys.stdout.close()
```

Apéndice D – Fragmentos de código de las optimizaciones

Fragmento de código utilizado para la aplicación del pruning al modelo Keras.

```
pruning_params = {
    'pruning_schedule': tfmot.sparsity.keras.PolynomialDecay(initial_sparsity=0.2,
                                                             final_sparsity=0.8,
                                                             begin_step=0,
                                                             end_step=2000)
    'block_size': (1, 1),
    'block_pooling_type': 'AVG'
}

model_for_pruning = tfmot.sparsity.keras.prune_low_magnitude(base_model, **pruning_params)

adam = Adam(lr=0.0001)
model_for_pruning.compile(
      loss="mse",
      optimizer='adam',
      metrics=['accuracy']
)

log_dir = tempfile.mkdtemp()
callbacks = [
    tfmot.sparsity.keras.UpdatePruningStep(),
    tfmot.sparsity.keras.PruningSummaries(log_dir=log_dir)
]

model_for_pruning.fit(train_generator,
                      validation_data=validation_generator,
                      callbacks=callbacks,
                      epochs=20)

model_for_pruning.save("Pruned_vgg16.h5")
_, keras_model_file = tempfile.mkstemp('.h5')
model_for_pruning.save(keras_model_file, include_optimizer=True)

with tfmot.sparsity.keras.prune_scope():
  loaded_model = tf.keras.models.load_model(keras_model_file)

model_for_export = tfmot.sparsity.keras.strip_pruning(model_for_pruning)
```

Se utiliza la función de “PolynomialDecay” ya que según Keras [10], “se observa que un learning rate que disminuye monótonamente da como resultado un modelo de mejor rendimiento”.

Código para generar el archivo que se utilizará con TensorRT.

```
#Definicion de los directorios de trabajo
save_pb_dir = './'
model_fname = './VGG16_cr.h5'

#Funcion para generar el Graph
def freeze_graph(graph, session, output, save_pb_dir='.', save_pb_name='trt_fp32.pb', save_pb_as_text=False):
    with graph.as_default():
        graphdef_inf = tf.graph_util.remove_training_nodes(graph.as_graph_def())
        graphdef_frozen = tf.graph_util.convert_variables_to_constants(session, graphdef_inf, output)
        graph_io.write_graph(graphdef_frozen, save_pb_dir, save_pb_name, as_text=save_pb_as_text)
        return graphdef_frozen

#Haciendo learning_phase = 0 --> El modelo ya fue entrenado
tf.keras.backend.set_learning_phase(0)

#Cargando el modelo
model = load_model(model_fname)
session = tf.keras.backend.get_session()
input_names = [t.op.name for t in model.inputs]
output_names = [t.op.name for t in model.outputs]

#Escribiendo el archivo .pb
frozen_graph = freeze_graph(session.graph,
                            session,
                            [out.op.name for out in model.outputs],
                            save_pb_dir=save_pb_dir
                            )
```

Se escribe el graph mediante graph_io.write_graph como un archivo protobuf, el cual es guardado como archivo del tipo .pb. Es importante mencionar que debido a que el modelo ya ha sido entrenado, se hace el learning pase = 0, de manera que se establece la fase de aprendizaje a un valor fijo. Una vez obtenido el archivo .pb se procede a tomar las mediciones de este, ya que este será el modelo base para las optimizaciones que se realizan en TensorRT.

Cuantización INT8.

```
import tensorflow.contrib.tensorrt as trt

trt_graph = trt.create_inference_graph(
    input_graph_def=frozen_graph,
    outputs=output_names,
    max_batch_size=1,
    max_workspace_size_bytes=1 << 25,
    precision_mode='INT8',
    minimum_segment_size=50
)

graph_io.write_graph(trt_graph, "./",
                     "trt_INT8_VGG16.pb", as_text=False)
```

Fragmento de código utilizado para generar el archivo de TensorRT con cuantización INT8.

Conversión de Keras a Tensorflow Lite.

```
import tensorflow as tf
converter = tf.compat.v1.lite.TFLiteConverter.from_keras_model_file("VGG16_cr.h5")
converter.optimizations = [tf.lite.Optimize.DEFAULT]
tflite_model = converter.convert()
open("base_model.tflite", "wb").write(tflite_model)
```

Fragmento de código utilizado para la conversión del modelo de Keras a tflite donde se inicia con “la optimización por defecto” correspondiente a float32.

Fragmento de código utilizado para el modelo de TVM.

```
#Nombre, forma y tipo de tensor
input_tensor = "vgg16_input"
input_shape = (1, 224, 224, 3)
input_dtype = "float32"

#Cargando el modelo de tflite
tflite_model_file = './base_model.tflite'
tflite_model_buf = open(tflite_model_file, "rb").read()
tflite_model = tflite.Model.GetRootAsModel(tflite_model_buf, 0)

#Compilando el modelo con Relay
mod, params = relay.frontend.from_tflite(tflite_model,
                                         shape_dict={input_tensor: input_shape},
                                         dtype_dict={input_tensor: input_dtype}
                                         )
target = "cuda"
with transform.PassContext(opt_level=3):
    lib = relay.build(mod, target, params=params)
lib.export_library('./compiled_fp32.so')
```

Fragmento de código donde se muestra las definiciones de los parámetros del tensor y las líneas de código en las cuáles se carga el modelo de TFLite, además de la compilación del modelo con Relay. También se crea la “librería” que será utilizada en la siguiente sección de código.

Código de ejecución de TVM.

```
def exec_tvm(image_data, lib):
    module = runtime.GraphModule(lib['default'](tvm.gpu(0)))
    module.set_input(input_tensor, tvm.nd.array(image_data))
    module.run()
    tvm_output = module.get_output(0).asnumpy()
    predictions = np.squeeze(tvm_output)
    return(predictions)
```

Sección de código con el cual se ejecuta el “modelo” en TVM. Esa función requiere como entrada la “librería” obtenida en el fragmento de código mostrado anteriormente. Además, “*image_data*” es la imagen pre procesada para ser utilizada por el modelo.